

Graph Algorithms

Contents

Articles

Introduction	1
Graph theory	1
Glossary of graph theory	9
Undirected graphs	20
Directed graphs	27
Directed acyclic graphs	30
Computer representations of graphs	35
Adjacency list	38
Adjacency matrix	40
Implicit graph	43
Graph exploration and vertex ordering	47
Depth-first search	47
Breadth-first search	53
Lexicographic breadth-first search	57
Iterative deepening depth-first search	61
Topological sorting	64
Application: Dependency graphs	68
Connectivity of undirected graphs	70
Connected components	70
Edge connectivity	72
Vertex connectivity	73
Menger's theorems on edge and vertex connectivity	74
Ear decomposition	76
Algorithms for 2-edge-connected components	80
Algorithms for 2-vertex-connected components	82
Algorithms for 3-vertex-connected components	85
Karger's algorithm for general vertex connectivity	89
Connectivity of directed graphs	94
Strongly connected components	94
Tarjan's strongly connected components algorithm	96
Path-based strong component algorithm	99

Kosaraju's strongly connected components algorithm	100
Reachability	102
Transitive closure	105
Transitive reduction	109
Application: 2-satisfiability	112
Shortest paths	122
Shortest path problem	122
Dijkstra's algorithm for single-source shortest paths with positive edge lengths	128
Bellman–Ford algorithm for single-source shortest paths allowing negative edge lengths	136
Johnson's algorithm for all-pairs shortest paths in sparse graphs	142
Floyd–Warshall algorithm for all-pairs shortest paths in dense graphs	145
Suurballe's algorithm for two shortest disjoint paths	150
Bidirectional search	153
A* search algorithm	157
Longest path problem	164
Widest path problem	166
Canadian traveller problem	170
K shortest path routing	173
Application: Centrality analysis of social networks	178
Application: Schulze voting system	186
Minimum spanning trees	201
Minimum spanning tree	201
Borůvka's algorithm	207
Kruskal's algorithm	210
Prim's algorithm	214
Edmonds's algorithm for directed minimum spanning trees	219
Degree-constrained spanning tree	222
Maximum-leaf spanning tree	223
K-minimum spanning tree	224
Capacitated minimum spanning tree	226
Application: Single-linkage clustering	227
Application: Maze generation	229
Cliques, independent sets, and coloring	235
Clique problem	235
Bron–Kerbosch algorithm for listing all maximal cliques	249

Independent set problem	253
Maximal independent set	258
Graph coloring	262
Bipartite graph	277
Greedy coloring	282
Application: Register allocation	284
Covering and domination	287
Vertex cover	287
Dominating set	292
Feedback vertex set	297
Feedback arc set	300
Tours	302
Eulerian path	302
Hamiltonian path	305
Hamiltonian path problem	309
Travelling salesman problem	310
Bottleneck traveling salesman problem	325
Christofides' heuristic for the TSP	326
Route inspection problem	328
Matching	330
Matching	330
Hopcroft–Karp algorithm for maximum matching in bipartite graphs	334
Edmonds's algorithm for maximum matching in non-bipartite graphs	339
Assignment problem	346
Hungarian algorithm for the assignment problem	348
FKT algorithm for counting matchings in planar graphs	353
Stable marriage problem	355
Stable roommates problem	359
Permanent	362
Computing the permanent	368
Network flow	372
Maximum flow problem	372
Max-flow min-cut theorem	378
Ford–Fulkerson algorithm for maximum flows	383
Edmonds–Karp algorithm for maximum flows	387

Dinic's algorithm for maximum flows	391
Push–relabel maximum flow algorithm	393
Closure problem	398
Minimum-cost flow problem	400
Graph drawing and planar graphs	403
Planar graph	403
Dual graph	410
Fáry's theorem	412
Steinitz's theorem	414
Planarity testing	416
Fraysseix–Rosenstiehl planarity criterion	418
Graph drawing	419
Force-directed graph drawing	425
Layered graph drawing	428
Upward planar drawing	430
Graph embedding	435
Application: Sociograms	437
Application: Concept maps	438
Special classes of graphs	443
Interval graph	443
Chordal graph	446
Perfect graph	450
Intersection graph	454
Unit disk graph	456
Line graph	459
Claw-free graph	468
Median graph	475
Graph isomorphism	486
Graph isomorphism	486
Graph isomorphism problem	488
Graph canonization	494
Subgraph isomorphism problem	495
Color-coding	498
Induced subgraph isomorphism problem	501
Maximum common subgraph isomorphism problem	502

Graph decomposition and graph minors	503
Graph partition	503
Kernighan–Lin algorithm	508
Tree decomposition	509
Branch-decomposition	512
Path decomposition	516
Planar separator theorem	529
Graph minors	542
Courcelle's theorem	548
Robertson–Seymour theorem	550
Bidimensionality	555

References

Article Sources and Contributors	558
Image Sources, Licenses and Contributors	566

Article Licenses

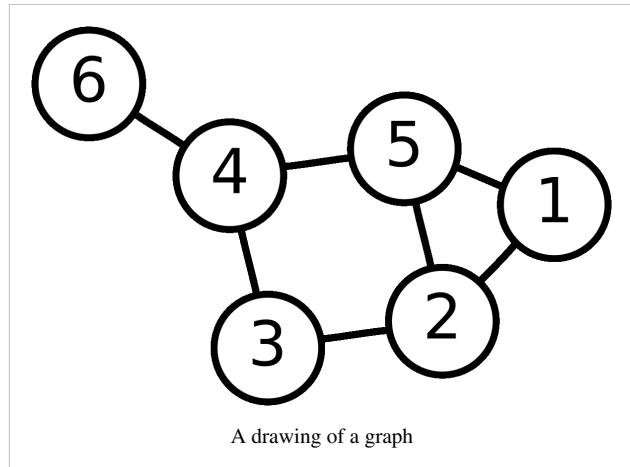
License	572
---------	-----

Introduction

Graph theory

This article is about sets of vertices connected by edges. For graphs of mathematical functions, see Graph of a function. For other uses, see Graph (disambiguation).

In mathematics and computer science, **graph theory** is the study of *graphs*, which are mathematical structures used to model pairwise relations between objects. A "graph" in this context is made up of "vertices" or "nodes" and lines called *edges* that connect them. A graph may be *undirected*, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be *directed* from one vertex to another; see graph (mathematics) for more detailed definitions and for other variations in the types of graph that are commonly considered. Graphs are one of the prime objects of study in discrete mathematics.



Refer to the glossary of graph theory for basic definitions in graph theory.

Definitions

Definitions in graph theory vary. The following are some of the more basic ways of defining graphs and related mathematical structures.

Graph

In the most common sense of the term,^[1] a **graph** is an ordered pair $G = (V, E)$ comprising a set V of **vertices** or **nodes** together with a set E of **edges** or **lines**, which are 2-element subsets of V (i.e., an edge is related with two vertices, and the relation is represented as an unordered pair of the vertices with respect to the particular edge). To avoid ambiguity, this type of graph may be described precisely as undirected and simple.

Other senses of *graph* stem from different conceptions of the edge set. In one more generalized notion,^[2] E is a set together with a relation of **incidence** that associates with each edge two vertices. In another generalized notion, E is a multiset of unordered pairs of (not necessarily distinct) vertices. Many authors call this type of object a multigraph or pseudograph.

All of these variants and others are described more fully below.

The vertices belonging to an edge are called the **ends**, **endpoints**, or **end vertices** of the edge. A vertex may exist in a graph and not belong to an edge.

V and E are usually taken to be finite, and many of the well-known results are not true (or are rather different) for **infinite graphs** because many of the arguments fail in the infinite case. The **order** of a graph is $|V|$ (the number of vertices). A graph's **size** is $|E|$, the number of edges. The **degree** of a vertex is the number of edges that connect to it, where an edge that connects to the vertex at both ends (a loop) is counted twice.

For an edge $\{u, v\}$, graph theorists usually use the somewhat shorter notation uv .

Applications

Graphs can be used to model many types of relations and processes in physical, biological, social and information systems. Many practical problems can be represented by graphs.

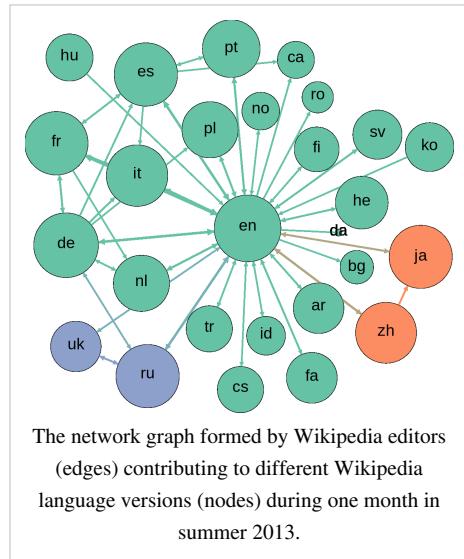
In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. The transformation of graphs is often formalized and represented by graph rewrite systems. Complementary to graph transformation systems focusing on rule-based in-memory manipulation of graphs are graph databases geared towards transaction-safe, persistent storing and querying of graph-structured data.

Graph-theoretic methods, in various forms, have proven particularly useful in linguistics, since natural language often lends itself well to discrete structure. Traditionally, syntax and compositional semantics follow tree-based structures, whose expressive power lies in the principle of compositionality, modeled in a hierarchical graph. More contemporary approaches such as head-driven phrase structure grammar model the syntax of natural language using typed feature structures, which are directed acyclic graphs. Within lexical semantics, especially as applied to computers, modeling word meaning is easier when a given word is understood in terms of related words; semantic networks are therefore important in computational linguistics. Still other methods in phonology (e.g. optimality theory, which uses lattice graphs) and morphology (e.g. finite-state morphology, using finite-state transducers) are common in the analysis of language as a graph. Indeed, the usefulness of this area of mathematics to linguistics has borne organizations such as TextGraphs^[3], as well as various 'Net' projects, such as WordNet, VerbNet, and others.

Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three-dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms. In chemistry a graph makes a natural model for a molecule, where vertices represent atoms and edges bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching. In statistical physics, graphs can represent local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems. Graphs are also used to represent the micro-scale channels of porous media, in which the vertices represent the pores and the edges represent the smaller channels connecting the pores.

Graph theory is also widely used in sociology as a way, for example, to measure actors' prestige or to explore rumor spreading, notably through the use of social network analysis software. Under the umbrella of social networks are many different types of graphs: Acquaintanceship and friendship graphs describe whether people know each other. Influence graphs model whether certain people can influence the behavior of others. Finally, collaboration graphs model whether two people work together in a particular way, such as acting in a movie together.

Likewise, graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist (or habitats) and the edges represent migration paths, or movement between the regions. This information is important when looking at breeding patterns or tracking the spread of disease, parasites or how changes to the movement can affect other species.



In mathematics, graphs are useful in geometry and certain parts of topology such as knot theory. Algebraic graph theory has close links with group theory.

A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pairwise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road.

History

The paper written by Leonhard Euler on the *Seven Bridges of Königsberg* and published in 1736 is regarded as the first paper in the history of graph theory. This paper, as well as the one written by Vandermonde on the *knight problem*, carried on with the *analysis situs* initiated by Leibniz. Euler's formula relating the number of edges, vertices, and faces of a convex polyhedron was studied and generalized by Cauchy and L'Huillier, and is at the origin of topology.

More than one century after Euler's paper on the bridges of Königsberg and while Listing introduced topology, Cayley was led by the study of particular analytical forms arising from differential calculus to study a particular class of graphs, the *trees*. This study had many implications in theoretical chemistry. The involved techniques mainly concerned the enumeration of graphs having particular properties. Enumerative graph theory then rose from the results of Cayley and the fundamental results published by Pólya between 1935 and 1937 and the generalization of these by De Bruijn in 1959. Cayley linked his results on trees with the contemporary studies of chemical composition. The fusion of the ideas coming from mathematics with those coming from chemistry is at the origin of a part of the standard terminology of graph theory.

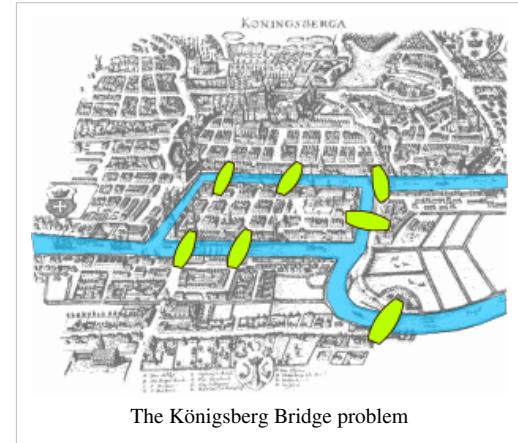
In particular, the term "graph" was introduced by Sylvester in a paper published in 1878 in *Nature*, where he draws an analogy between "quantic invariants" and "co-variants" of algebra and molecular diagrams:^[4]

"[...] Every invariant and co-variant thus becomes expressible by a *graph* precisely identical with a Kekuléan diagram or chemicograph. [...] I give a rule for the geometrical multiplication of graphs, *i.e.* for constructing a *graph* to the product of in- or co-variants whose separate graphs are given. [...]" (italics as in the original).

The first textbook on graph theory was written by Dénes Kőnig, and published in 1936. Another book by Frank Harary, published in 1969, was "considered the world over to be the definitive textbook on the subject", and enabled mathematicians, chemists, electrical engineers and social scientists to talk to each other. Harary donated all of the royalties to fund the Pólya Prize.

One of the most famous and stimulating problems in graph theory is the four color problem: "Is it true that any map drawn in the plane may have its regions colored with four colors, in such a way that any two regions having a common border have different colors?" This problem was first posed by Francis Guthrie in 1852 and its first written record is in a letter of De Morgan addressed to Hamilton the same year. Many incorrect proofs have been proposed, including those by Cayley, Kempe, and others. The study and the generalization of this problem by Tait, Heawood, Ramsey and Hadwiger led to the study of the colorings of the graphs embedded on surfaces with arbitrary genus. Tait's reformulation generated a new class of problems, the *factorization problems*, particularly studied by Petersen and Kőnig. The works of Ramsey on colorations and more specially the results obtained by Turán in 1941 was at the origin of another branch of graph theory, *extremal graph theory*.

The four color problem remained unsolved for more than a century. In 1969 Heinrich Heesch published a method for solving the problem using computers.^[5] A computer-aided proof produced in 1976 by Kenneth Appel and Wolfgang



Haken makes fundamental use of the notion of "discharging" developed by Heesch. The proof involved checking the properties of 1,936 configurations by computer, and was not fully accepted at the time due to its complexity. A simpler proof considering only 633 configurations was given twenty years later by Robertson, Seymour, Sanders and Thomas.

The autonomous development of topology from 1860 and 1930 fertilized graph theory back through the works of Jordan, Kuratowski and Whitney. Another important factor of common development of graph theory and topology came from the use of the techniques of modern algebra. The first example of such a use comes from the work of the physicist Gustav Kirchhoff, who published in 1845 his Kirchhoff's circuit laws for calculating the voltage and current in electric circuits.

The introduction of probabilistic methods in graph theory, especially in the study of Erdős and Rényi of the asymptotic probability of graph connectivity, gave rise to yet another branch, known as *random graph theory*, which has been a fruitful source of graph-theoretic results.

Drawing graphs

Main article: Graph drawing

Graphs are represented visually by drawing a dot or circle for every vertex, and drawing an arc between two vertices if they are connected by an edge. If the graph is directed, the direction is indicated by drawing an arrow.

A graph drawing should not be confused with the graph itself (the abstract, non-visual structure) as there are several ways to structure the graph drawing. All that matters is which vertices are connected to which others by how many edges and not the exact layout. In practice it is often difficult to decide if two drawings represent the same graph. Depending on the problem domain some layouts may be better suited and easier to understand than others.

The pioneering work of W. T. Tutte was very influential in the subject of graph drawing. Among other achievements, he introduced the use of linear algebraic methods to obtain graph drawings.

Graph drawing also can be said to encompass problems that deal with the crossing number and its various generalizations. The crossing number of a graph is the minimum number of intersections between edges that a drawing of the graph in the plane must contain. For a planar graph, the crossing number is zero by definition.

Drawings on surfaces other than the plane are also studied.

Graph-theoretic data structures

Main article: Graph (abstract data type)

There are different ways to store graphs in a computer system. The data structure used depends on both the graph structure and the algorithm used for manipulating the graph. Theoretically one can distinguish between list and matrix structures but in concrete applications the best structure is often a combination of both. List structures are often preferred for sparse graphs as they have smaller memory requirements. Matrix structures on the other hand provide faster access for some applications but can consume huge amounts of memory.

List structures include the incidence list, an array of pairs of vertices, and the adjacency list, which separately lists the neighbors of each vertex: Much like the incidence list, each vertex has a list of which vertices it is adjacent to.

Matrix structures include the incidence matrix, a matrix of 0's and 1's whose rows represent vertices and whose columns represent edges, and the adjacency matrix, in which both the rows and columns are indexed by vertices. In both cases a 1 indicates two adjacent objects and a 0 indicates two non-adjacent objects. The Laplacian matrix is a modified form of the adjacency matrix that incorporates information about the degrees of the vertices, and is useful in some calculations such as Kirchhoff's theorem on the number of spanning trees of a graph. The distance matrix, like the adjacency matrix, has both its rows and columns indexed by vertices, but rather than containing a 0 or a 1 in each cell it contains the length of a shortest path between two vertices.

Problems in graph theory

Enumeration

There is a large literature on graphical enumeration: the problem of counting graphs meeting specified conditions. Some of this work is found in Harary and Palmer (1973).

Subgraphs, induced subgraphs, and minors

A common problem, called the subgraph isomorphism problem, is finding a fixed graph as a subgraph in a given graph. One reason to be interested in such a question is that many graph properties are *hereditary* for subgraphs, which means that a graph has the property if and only if all subgraphs have it too. Unfortunately, finding maximal subgraphs of a certain kind is often an NP-complete problem.

- Finding the largest complete graph is called the clique problem (NP-complete).

A similar problem is finding induced subgraphs in a given graph. Again, some important graph properties are hereditary with respect to induced subgraphs, which means that a graph has a property if and only if all induced subgraphs also have it. Finding maximal induced subgraphs of a certain kind is also often NP-complete. For example,

- Finding the largest edgeless induced subgraph, or independent set, called the independent set problem (NP-complete).

Still another such problem, the *minor containment problem*, is to find a fixed graph as a minor of a given graph. A minor or **subcontraction** of a graph is any graph obtained by taking a subgraph and contracting some (or no) edges. Many graph properties are hereditary for minors, which means that a graph has a property if and only if all minors have it too. A famous example:

- A graph is planar if it contains as a minor neither the complete bipartite graph $K_{3,3}$ (See the Three-cottage problem) nor the complete graph K_5 .

Another class of problems has to do with the extent to which various species and generalizations of graphs are determined by their *point-deleted subgraphs*, for example:

- The reconstruction conjecture.

Graph coloring

Many problems have to do with various ways of coloring graphs, for example:

- The four-color theorem
- The strong perfect graph theorem
- The Erdős–Faber–Lovász conjecture (unsolved)
- The total coloring conjecture, also called Behzad's conjecture (unsolved)
- The list coloring conjecture (unsolved)
- The Hadwiger conjecture (graph theory) (unsolved)

Subsumption and unification

Constraint modeling theories concern families of directed graphs related by a partial order. In these applications, graphs are ordered by specificity, meaning that more constrained graphs—which are more specific and thus contain a greater amount of information—are subsumed by those that are more general. Operations between graphs include evaluating the direction of a subsumption relationship between two graphs, if any, and computing graph unification. The unification of two argument graphs is defined as the most general graph (or the computation thereof) that is consistent with (i.e. contains all of the information in) the inputs, if such a graph exists; efficient unification algorithms are known.

For constraint frameworks which are strictly compositional, graph unification is the sufficient satisfiability and combination function. Well-known applications include automatic theorem proving and modeling the elaboration of linguistic structure.

Route problems

- Hamiltonian path and cycle problems
- Minimum spanning tree
- Route inspection problem (also called the "Chinese Postman Problem")
- Seven Bridges of Königsberg
- Shortest path problem
- Steiner tree
- Three-cottage problem
- Traveling salesman problem (NP-hard)

Network flow

There are numerous problems arising especially from applications that have to do with various notions of flows in networks, for example:

- Max flow min cut theorem

Visibility graph problems

- Museum guard problem

Covering problems

Covering problems in graphs are specific instances of subgraph-finding problems, and they tend to be closely related to the clique problem or the independent set problem.

- Set cover problem
- Vertex cover problem

Decomposition problems

Decomposition, defined as partitioning the edge set of a graph (with as many vertices as necessary accompanying the edges of each part of the partition), has a wide variety of question. Often, it is required to decompose a graph into subgraphs isomorphic to a fixed graph; for instance, decomposing a complete graph into Hamiltonian cycles. Other problems specify a family of graphs into which a given graph should be decomposed, for instance, a family of cycles, or decomposing a complete graph K_n into $n - 1$ specified trees having, respectively, 1, 2, 3, ..., $n - 1$ edges.

Some specific decomposition problems that have been studied include:

- Arboricity, a decomposition into as few forests as possible
- Cycle double cover, a decomposition into a collection of cycles covering each edge exactly twice
- Edge coloring, a decomposition into as few matchings as possible
- Graph factorization, a decomposition of a regular graph into regular subgraphs of given degrees

Graph classes

Many problems involve characterizing the members of various classes of graphs. Some examples of such questions are below:

- Enumerating the members of a class
- Characterizing a class in terms of forbidden substructures
- Ascertaining relationships among classes (e.g., does one property of graphs imply another)
- Finding efficient algorithms to decide membership in a class
- Finding representations for members of a class.

Notes

[1] See, for instance, Iyanaga and Kawada, **69 J**, p. 234 or Biggs, p. 4.

[2] See, for instance, Graham et al., p. 5.

[3] <http://www.textgraphs.org>

[4] John Joseph Sylvester (1878), *Chemistry and Algebra*. Nature, volume 17, page 284. . Online version (<https://archive.org/stream/nature15unknog#page/n312/mode/1up>). Retrieved 2009-12-30.

[5] Heinrich Heesch: Untersuchungen zum Vierfarbenproblem. Mannheim: Bibliographisches Institut 1969.

References

- Berge, Claude (1958), *Théorie des graphes et ses applications*, Collection Universitaire de Mathématiques **II**, Paris: Dunod. English edition, Wiley 1961; Methuen & Co, New York 1962; Russian, Moscow 1961; Spanish, Mexico 1962; Roumanian, Bucharest 1969; Chinese, Shanghai 1963; Second printing of the 1962 first English edition, Dover, New York 2001.
- Biggs, N.; Lloyd, E.; Wilson, R. (1986), *Graph Theory, 1736–1936*, Oxford University Press.
- Bondy, J.A.; Murty, U.S.R. (2008), *Graph Theory*, Springer, ISBN 978-1-84628-969-9.
- Bondy, Riordan, O.M (2003), *Mathematical results on scale-free random graphs in "Handbook of Graphs and Networks"* (S. Bornholdt and H.G. Schuster (eds)), Wiley VCH, Weinheim, 1st ed..
- Chartrand, Gary (1985), *Introductory Graph Theory*, Dover, ISBN 0-486-24775-9.
- Gibbons, Alan (1985), *Algorithmic Graph Theory*, Cambridge University Press.
- Reuven Cohen, Shlomo Havlin (2010), *Complex Networks: Structure, Robustness and Function*, Cambridge University Press
- Golumbic, Martin (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press.
- Harary, Frank (1969), *Graph Theory*, Reading, MA: Addison-Wesley.
- Harary, Frank; Palmer, Edgar M. (1973), *Graphical Enumeration*, New York, NY: Academic Press.
- Mahadev, N.V.R.; Peled, Uri N. (1995), *Threshold Graphs and Related Topics*, North-Holland.
- Mark Newman (2010), *Networks: An Introduction*, Oxford University Press.

External links

Online textbooks

- Phase Transitions in Combinatorial Optimization Problems, Section 3: Introduction to Graphs (<http://arxiv.org/pdf/cond-mat/0602129.pdf>) (2006) by Hartmann and Weigt
- Digraphs: Theory Algorithms and Applications (<http://www.cs.rhul.ac.uk/books/dbook/>) 2007 by Jorgen Bang-Jensen and Gregory Gutin
- Graph Theory, by Reinhard Diestel (<http://diestel-graph-theory.com/index.html>)

Other resources

- Graph theory with examples (<http://scanftree.com/Graph-Theory/>)
- Hazewinkel, Michiel, ed. (2001), "Graph theory" (<http://www.encyclopediaofmath.org/index.php?title=p/g045010>), *Encyclopedia of Mathematics*, Springer, ISBN 978-1-55608-010-4
- Graph theory tutorial (<http://www.utm.edu/departments/math/graph/>)
- A searchable database of small connected graphs (<http://www.gfredericks.com/main/sandbox/graphs>)
- Image gallery: graphs (<https://web.archive.org/web/20060206155001/http://www.nd.edu/~networks/gallery.htm>) at the Wayback Machine (archived February 6, 2006)
- Concise, annotated list of graph theory resources for researchers (<http://www.babelgraph.org/links.html>)
- rocs (<http://www.kde.org/applications/education/rocs/>) — a graph theory IDE
- The Social Life of Routers (<http://www.orgnet.com/SocialLifeOfRouters.pdf>) — non-technical paper discussing graphs of people and computers
- Graph Theory Software (<http://graphtheorysoftware.com/>) — tools to teach and learn graph theory
- Online books (<http://tools.wmflabs.org/ftl/cgi-bin/ftl?st=&su=Graph+theory&library=OLBP>), and library resources in your library (<http://tools.wmflabs.org/ftl/cgi-bin/ftl?st=&su=Graph+theory>) and in other libraries (<http://tools.wmflabs.org/ftl/cgi-bin/ftl?st=&su=Graph+theory&library=0CHOOSE0>) about graph theory

Glossary of graph theory



Look up *Appendix:Glossary of graph theory* in Wiktionary, the free dictionary.

Graph theory is a growing area in mathematical research, and has a large specialized vocabulary. Some authors use the same word with different meanings. Some authors use different words to mean the same thing. This page attempts to describe the majority of current usage.

Basics

A **graph** G consists of two types of elements, namely *vertices* and *edges*. Every edge has two *endpoints* in the set of vertices, and is said to **connect** or **join** the two endpoints. An edge can thus be defined as a set of two vertices (or an ordered pair, in the case of a **directed graph** - see Section Direction). The two endpoints of an edge are also said to be **adjacent** to each other.

Alternative models of graphs exist; e.g., a graph may be thought of as a Boolean binary function over the set of vertices or as a square $(0,1)$ -matrix.

A **vertex** is simply drawn as a *node* or a *dot*. The **vertex set** of G is usually denoted by $V(G)$, or V when there is no danger of confusion. The **order** of a graph is the number of its vertices, i.e. $|V(G)|$.

An **edge** (a set of two elements) is drawn as a *line* connecting two vertices, called **endpoints** or (less often) **endvertices**. An edge with endvertices x and y is denoted by xy (without any symbol in between). The **edge set** of G is usually denoted by $E(G)$, or E when there is no danger of confusion. An edge xy is called **incident** to a vertex when this vertex is one of the endpoints x or y .

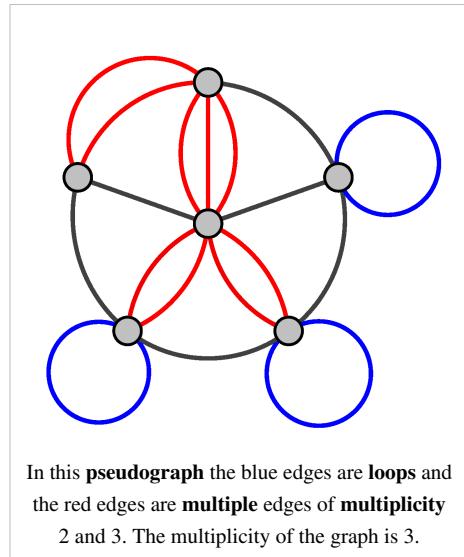
The **size** of a graph is the number of its edges, i.e. $|E(G)|$.

A **loop** is an edge whose endpoints are the same vertex. A **link** has two distinct endvertices. An edge is **multiple** if there is another edge with the same endvertices; otherwise it is **simple**. The **multiplicity of an edge** is the number of multiple edges sharing the same end vertices; the **multiplicity of a graph**, the maximum multiplicity of its edges. A graph is a **simple graph** if it has no multiple edges or loops, a **multigraph** if it has multiple edges, but no loops, and a **multigraph** or **pseudograph** if it contains both multiple edges and loops (the literature is highly inconsistent).

When stated without any qualification, a graph is usually assumed to be simple, except in the literature of **category theory**, where it refers to a **quiver**.

Graphs whose edges or vertices have names or labels are known as **labeled**, those without as **unlabeled**. Graphs with labeled vertices only are **vertex-labeled**, those with labeled edges only are **edge-labeled**. The difference between a labeled and an unlabeled graph is that the latter has no specific set of vertices or edges; it is regarded as another way to look upon an isomorphism type of graphs. (Thus, this usage distinguishes between graphs with identifiable vertex or edge sets on the one hand, and isomorphism types or classes of graphs on the other.)

(**Graph labeling** usually refers to the assignment of labels (usually natural numbers, usually distinct) to the edges and vertices of a graph, subject to certain rules depending on the situation. This should not be confused with a graph's merely having distinct labels or names on the vertices.)



In this **pseudograph** the blue edges are **loops** and the red edges are **multiple edges** of **multiplicity** 2 and 3. The multiplicity of the graph is 3.

A **hyperedge** is an edge that is allowed to take on any number of vertices, possibly more than 2. A graph that allows any hyperedge is called a **hypergraph**. A simple graph can be considered a special case of the hypergraph, namely the 2-uniform hypergraph. However, when stated without any qualification, an edge is always assumed to consist of at most 2 vertices, and a graph is never confused with a hypergraph.

A **non-edge** (or **anti-edge**) is an edge that is not present in the graph. More formally, for two vertices u and v , $\{u, v\}$ is a non-edge in a graph G whenever $\{u, v\}$ is not an edge in G . This means that there is either no edge between the two vertices or (for directed graphs) at most one of (u, v) and (v, u) from v is an arc in G .

Occasionally the term **cotriangle** or **anti-triangle** is used for a set of three vertices none of which are connected.

The **complement** \bar{G} of a graph G is a graph with the same vertex set as G but with an edge set such that xy is an edge in \bar{G} if and only if xy is not an edge in G .

An **edgeless graph** or **empty graph** or **null graph** is a graph with zero or more vertices, but no edges. The **empty graph** or **null graph** may also be the graph with no vertices and no edges. If it is a graph with no edges and any number n of vertices, it may be called the **null graph on n vertices**. (There is no consistency at all in the literature.)

A graph is **infinite** if it has infinitely many vertices or edges or both; otherwise the graph is **finite**. An infinite graph where every vertex has finite *degree* is called **locally finite**. When stated without any qualification, a graph is usually assumed to be finite. See also continuous graph.

Two graphs G and H are said to be **isomorphic**, denoted by $G \sim H$, if there is a one-to-one correspondence, called an **isomorphism**, between the vertices of the graph such that two vertices are adjacent in G if and only if their corresponding vertices are adjacent in H . Likewise, a graph G is said to be **homomorphic** to a graph H if there is a mapping, called a **homomorphism**, from $V(G)$ to $V(H)$ such that if two vertices are adjacent in G then their corresponding vertices are adjacent in H .

Subgraphs

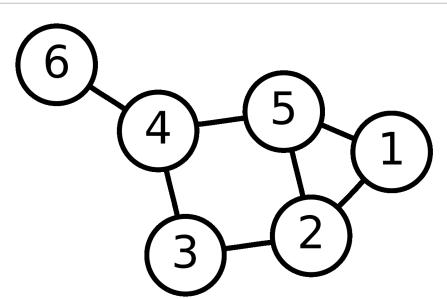
A **subgraph** of a graph G is a graph whose vertex set is a subset of that of G , and whose adjacency relation is a subset of that of G restricted to this subset. In the other direction, a **supergraph** of a graph G is a graph of which G is a subgraph. We say a graph G **contains** another graph H if some subgraph of G is H or is isomorphic to H .

A subgraph H is a **spanning subgraph**, or **factor**, of a graph G if it has the same vertex set as G . We say H spans G .

A subgraph H of a graph G is said to be **induced** (or **full**) if, for any pair of vertices x and y of H , xy is an edge of H if and only if xy is an edge of G . In other words, H is an induced subgraph of G if it has exactly the edges that appear in G over the same vertex set. If the vertex set of H is the subset S of $V(G)$, then H can be written as $G[S]$ and is said to be **induced by S** .

A graph G is **minimal** with some property P provided that G has property P and no proper subgraph of G has property P . In this definition, the term *subgraph* is usually understood to mean "induced subgraph." The notion of maximality is defined dually: G is **maximal** with P provided that $P(G)$ and G has no proper supergraph H such that $P(H)$.

A graph that does *not* contain H as an induced subgraph is said to be **H -free**, and more generally if \mathcal{F} is a family of graphs then the graphs that do not contain any induced subgraph isomorphic to a member of \mathcal{F} are called \mathcal{F} -free. For example the triangle-free graphs are the graphs that do not have a triangle graph as an induced subgraph. Many important classes of graphs can be defined by sets of forbidden subgraphs, the graphs that are not in the class and are



A labeled simple graph with vertex set $V = \{1, 2, 3, 4, 5, 6\}$ and edge set $E = \{\{1,2\}, \{1,5\}, \{2,3\}, \{2,5\}, \{3,4\}, \{4,5\}, \{4,6\}\}$.

minimal with respect to subgraphs, induced subgraphs, or graph minors.

A **universal graph** in a class K of graphs is a simple graph in which every element in K can be embedded as a subgraph.

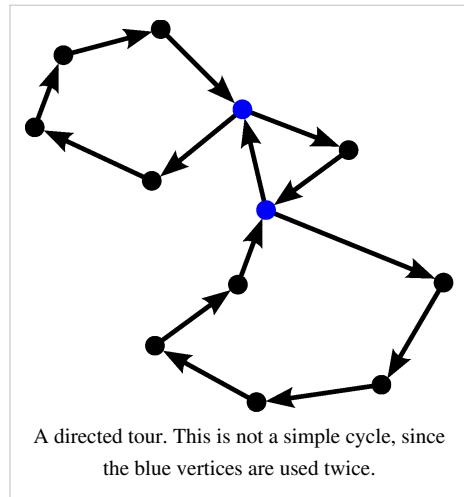
Walks

A **walk** is an alternating sequence of vertices and edges, beginning and ending with a vertex, where the vertices that precede and follow an edge in the sequence are just the two end vertices of that edge. A walk is **closed** if its first and last vertices are the same, and **open** if they are different.

The **length** l of a walk is the number of edges that it uses. For an open walk, $l = n-1$, where n is the number of vertices visited (a vertex is counted each time it is visited). For a closed walk, $l = n$ (the start/end vertex is listed twice, but is not counted twice). In the example graph, $(1, 2, 5, 1, 2, 3)$ is an open walk with length 5, and $(4, 5, 2, 1, 5, 4)$ is a closed walk of length 5.

A **trail** is a walk in which all the edges are distinct. A closed trail has been called a **tour** or **circuit**, but these are not universal, and the latter is often reserved for a regular subgraph of degree two.

Traditionally, a **path** referred to what is now usually known as an *open walk*. Nowadays, when stated without any qualification, a path is usually understood to be **simple**, meaning that no vertices (and thus no edges) are repeated. (The term **chain** has also been used to refer to a walk in which all vertices and edges are distinct.) In the example graph, $(5, 2, 1)$ is a path of length 2. The closed equivalent to this type of walk, a walk that starts and ends at the same vertex but otherwise has no repeated vertices or edges, is called a **cycle**. Like *path*, this term traditionally referred to any closed walk, but now is usually understood to be simple by definition. In the example graph, $(1, 5, 2, 1)$ is a cycle of length 3. (A cycle, unlike a path, is not allowed to have length 0.) Paths and cycles of n vertices are often denoted by P_n and C_n , respectively. (Some authors use the length instead of the number of vertices, however.)



C_1 is a **loop**, C_2 is a **digon** (a pair of parallel undirected edges in a multigraph, or a pair of antiparallel edges in a directed graph), and C_3 is called a **triangle**.

A cycle that has odd length is an **odd cycle**; otherwise it is an **even cycle**. One theorem is that a graph is bipartite if and only if it contains no odd cycles. (See complete bipartite graph.)

A graph is **acyclic** if it contains no cycles; **unicyclic** if it contains exactly one cycle; and **pancyclic** if it contains cycles of every possible length (from 3 to the order of the graph).

A **wheel graph** is a graph with n vertices ($n \geq 4$), formed by connecting a single vertex to all vertices of C_{n-1} .

The **girth** of a graph is the length of a shortest (simple) cycle in the graph; and the **circumference**, the length of a longest (simple) cycle. The girth and circumference of an acyclic graph are defined to be infinity ∞ .

A path or cycle is **Hamiltonian** (or *spanning*) if it uses all vertices exactly once. A graph that contains a Hamiltonian path is **traceable**; and one that contains a Hamiltonian path for any given pair of (distinct) end vertices is a **Hamiltonian connected graph**. A graph that contains a Hamiltonian cycle is a **Hamiltonian graph**.

A trail or circuit (or cycle) is **Eulerian** if it uses all edges precisely once. A graph that contains an Eulerian trail is **traversable**. A graph that contains an Eulerian circuit is an **Eulerian graph**.

Two paths are **internally disjoint** (some people call it *independent*) if they do not have any vertex in common, except the first and last ones.

A **theta graph** is the union of three internally disjoint (simple) paths that have the same two distinct end vertices.^[1]

A **theta₀ graph** has seven vertices which can be arranged as the vertices of a regular hexagon plus an additional vertex in the center. The eight edges are the perimeter of the hexagon plus one diameter.

Trees

A **tree** is a connected acyclic simple graph. For directed graphs, each vertex has at most one incoming edge. A vertex of degree 1 is called a **leaf**, or *pendant vertex*. An edge incident to a leaf is a **leaf edge**, or *pendant edge*. (Some people define a leaf edge as a *leaf* and then define a *leaf vertex* on top of it. These two sets of definitions are often used interchangeably.) A non-leaf vertex is an **internal vertex**. Sometimes, one vertex of the tree is distinguished, and called the **root**; in this case, the tree is called **rooted**. Rooted trees are often treated as **directed acyclic graphs** with the edges pointing away from the root.

A **subtree** of the tree T is a connected subgraph of T .

A **forest** is an acyclic simple graph. For directed graphs, each vertex has at most one incoming edge. (That is, a tree with the connectivity requirement removed; a graph containing multiple disconnected trees.)

A **subforest** of the forest F is a subgraph of F .

A **spanning tree** is a spanning subgraph that is a tree. Every graph has a spanning forest. But only a connected graph has a spanning tree.

A special kind of tree called a **star** is $K_{1,k}$. An induced star with 3 edges is a **claw**.

A **caterpillar** is a tree in which all non-leaf nodes form a single path.

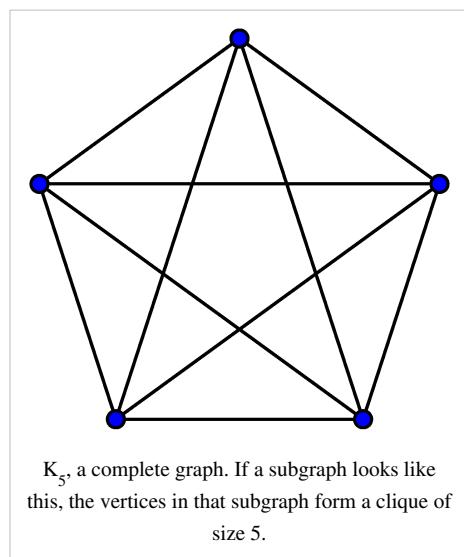
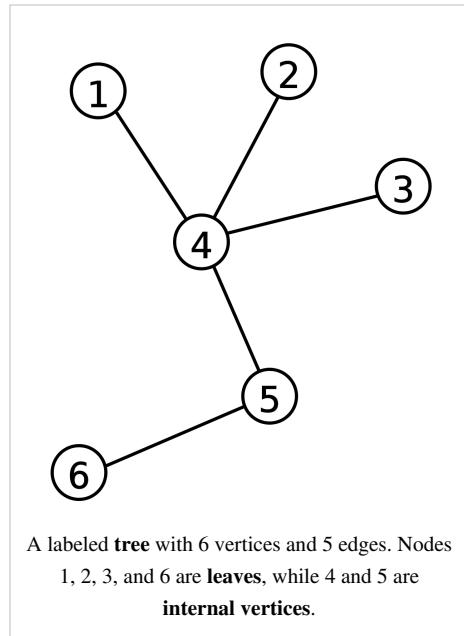
A **k -ary tree** is a rooted tree in which every internal vertex has no more than k *children*. A 1-ary tree is just a path. A 2-ary tree is also called a **binary tree**.

Cliques

The **complete graph** K_n of order n is a simple graph with n vertices in which every vertex is adjacent to every other. The example graph to the right is complete. The complete graph on n vertices is often denoted by K_n . It has $n(n-1)/2$ edges (corresponding to all possible choices of pairs of vertices).

A **clique** in a graph is a set of pairwise adjacent vertices. Since any subgraph induced by a clique is a complete subgraph, the two terms and their notations are usually used interchangeably. A **k -clique** is a clique of order k . In the example graph above, vertices 1, 2 and 5 form a 3-clique, or a *triangle*. A maximal clique is a clique that is not a subset of any other clique (some authors reserve the term clique for maximal cliques).

The **clique number** $\omega(G)$ of a graph G is the order of a largest clique in G .



Strongly connected component

A related but weaker concept is that of a *strongly connected component*. Informally, a strongly connected component of a directed graph is a subgraph where all nodes in the subgraph are reachable by all other nodes in the subgraph. Reachability between nodes is established by the existence of a *path* between the nodes.

A directed graph can be decomposed into strongly connected components by running the depth-first search (DFS) algorithm twice: first, on the graph itself and next on the *transpose graph* in decreasing order of the finishing times of the first DFS. Given a directed graph G , the transpose G_T is the graph G with all the edge directions reversed.

Hypercubes

A **hypercube graph** Q_n is a regular graph with 2^n vertices, $2^{n-1}n$ edges, and n edges touching each vertex. It can be obtained as the one-dimensional skeleton of the geometric **hypercube**.

Knots

A **knot** in a directed graph is a collection of vertices and edges with the property that every vertex in the knot has outgoing edges, and all outgoing edges from vertices in the knot terminate at other vertices in the knot. Thus it is impossible to leave the knot while following the directions of the edges.

Minors

A *minor* $G_2 = (V_2, E_2)$ of $G_1 = (V_1, E_1)$ is an injection from V_2 to V_1 such that every edge in E_2 corresponds to a path (disjoint from all other such paths) in G_1 such that every vertex in V_1 is in one or more paths, or is part of the injection from V_2 to V_1 . This can alternatively be phrased in terms of *contractions*, which are operations which collapse a path and all vertices on it into a single edge (see Minor (graph theory)).

Embedding

An *embedding* $G_2 = (V_2, E_2)$ of $G_1 = (V_1, E_1)$ is an injection from V_2 to V_1 such that every edge in E_2 corresponds to a path in G_1 .

Adjacency and degree

In graph theory, degree, especially that of a vertex, is usually a *measure of immediate adjacency*.

An edge connects two vertices; these two vertices are said to be **incident** to that edge, or, equivalently, that edge incident to those two vertices. All degree-related concepts have to do with adjacency or incidence.

The **degree**, or *valency*, $d_G(v)$ of a vertex v in a graph G is the number of edges incident to v , with loops being counted twice. A vertex of degree 0 is an **isolated vertex**. A vertex of degree 1 is a leaf. In the labelled simple graph example, vertices 1 and 3 have a degree of 2, vertices 2, 4 and 5 have a degree of 3, and vertex 6 has a degree of 1. If E is finite, then the total sum of vertex degrees is equal to twice the number of edges.

The **total degree** of a graph is the sum of the degrees of all its vertices. Thus, for a graph without loops, it is equal to the number of incidences between graphs and edges. The handshaking lemma states that the total degree is always equal to two times the number of edges, loops included. This means that for a simple graph with 3 vertices with each vertex having a degree of two (i.e. a triangle) the total degree would be six (e.g. $3 \times 2 = 6$).

A **degree sequence** is a list of degrees of a graph in non-increasing order (e.g. $d_1 \geq d_2 \geq \dots \geq d_n$). A sequence of non-increasing integers is **realizable** if it is a degree sequence of some graph.

Two vertices u and v are called **adjacent** if an edge exists between them. We denote this by $u \sim v$ or $u \downarrow v$. In the above graph, vertices 1 and 2 are adjacent, but vertices 2 and 4 are not. The set of **neighbors** of v , that is, vertices adjacent to v not including v itself, forms an induced subgraph called the **(open) neighborhood** of v and denoted

$N_G(v)$. When v is also included, it is called a **closed neighborhood** and denoted by $N_G[v]$. When stated without any qualification, a neighborhood is assumed to be open. The subscript G is usually dropped when there is no danger of confusion; the same neighborhood notation may also be used to refer to sets of adjacent vertices rather than the corresponding induced subgraphs. In the example graph, vertex 1 has two neighbors: vertices 2 and 5. For a simple graph, the number of neighbors that a vertex has coincides with its degree.

A **dominating set** of a graph is a vertex subset whose closed neighborhood includes all vertices of the graph. A vertex v **dominates** another vertex u if there is an edge from v to u . A vertex subset V **dominates** another vertex subset U if every vertex in U is adjacent to some vertex in V . The minimum size of a dominating set is the **domination number** $\gamma(G)$.

In computers, a finite, directed or undirected graph (with n vertices, say) is often represented by its **adjacency matrix**: an n -by- n matrix whose entry in row i and column j gives the number of edges from the i -th to the j -th vertex.

Spectral graph theory studies relationships between the properties of a graph and its adjacency matrix or other matrices associated with the graph.

The **maximum degree** $\Delta(G)$ of a graph G is the largest degree over all vertices; the **minimum degree** $\delta(G)$, the smallest.

A graph in which every vertex has the same degree is **regular**. It is **k -regular** if every vertex has degree k . A 0-regular graph is an independent set. A 1-regular graph is a matching. A 2-regular graph is a vertex disjoint union of cycles. A 3-regular graph is said to be **cubic**, or *trivalent*.

A **k -factor** is a k -regular spanning subgraph. A 1-factor is a **perfect matching**. A partition of edges of a graph into k -factors is called a **k -factorization**. A **k -factorable graph** is a graph that admits a k -factorization.

A graph is **biregular** if it has unequal maximum and minimum degrees and every vertex has one of those two degrees.

A **strongly regular graph** is a regular graph such that any adjacent vertices have the same number of common neighbors as other adjacent pairs and that any nonadjacent vertices have the same number of common neighbors as other nonadjacent pairs.

Independence

In graph theory, the word *independent* usually carries the connotation of *pairwise disjoint* or *mutually nonadjacent*. In this sense, independence is a form of *immediate nonadjacency*. An **isolated vertex** is a vertex not incident to any edges. An **independent set**, or *co clique*, or *stable set* or *staset*, is a set of vertices of which no pair is adjacent. Since the graph induced by any independent set is an empty graph, the two terms are usually used interchangeably. In the example at the top of this page, vertices 1, 3, and 6 form an independent set; and 2 and 4 form another one.

Two subgraphs are **edge disjoint** if they have no edges in common. Similarly, two subgraphs are **vertex disjoint** if they have no vertices (and thus, also no edges) in common. Unless specified otherwise, a set of **disjoint subgraphs** are assumed to be pairwise vertex disjoint.

The **independence number** $\alpha(G)$ of a graph G is the size of the largest independent set of G .

A graph can be **decomposed** into independent sets in the sense that the entire vertex set of the graph can be partitioned into pairwise disjoint independent subsets. Such independent subsets are called **partite sets**, or simply *parts*.

A graph that can be decomposed into two partite sets **bipartite**; three sets, **tripartite**; k sets, **k -partite**; and an unknown number of sets, **multipartite**. An 1-partite graph is the same as an independent set, or an empty graph. A 2-partite graph is the same as a bipartite graph. A graph that can be decomposed into k partite sets is also said to be **k -colourable**.

A **complete multipartite** graph is a graph in which vertices are adjacent if and only if they belong to different partite sets. A *complete bipartite graph* is also referred to as a **biclique**; if its partite sets contain n and m vertices, respectively, then the graph is denoted $K_{n,m}$.

A k -partite graph is **semiregular** if each of its partite sets has a uniform degree; **equipartite** if each partite set has the same size; and **balanced k -partite** if each partite set differs in size by at most 1 with any other.

The **matching number** $\alpha'(G)$ of a graph G is the size of a largest **matching**, or pairwise vertex disjoint edges, of G .

A *spanning matching*, also called a **perfect matching** is a matching that covers all vertices of a graph.

Complexity

Complexity of a graph denotes the quantity of information that a graph contained, and can be measured in several ways. For example, by counting the number of its spanning trees, or the value of a certain formula involving the number of vertices, edges, and proper paths in a graph.

Connectivity

Connectivity extends the concept of adjacency and is essentially a form (and measure) of *concatenated adjacency*.

If it is possible to establish a path from any vertex to any other vertex of a graph, the graph is said to be **connected**; otherwise, the graph is **disconnected**. A graph is **totally disconnected** if there is no path connecting any pair of vertices. This is just another name to describe an empty graph or independent set.

A **cut vertex**, or *articulation point*, is a vertex whose removal disconnects the remaining subgraph. A **cut set**, or *vertex cut* or *separating set*, is a set of vertices whose removal disconnects the remaining subgraph. A *bridge* is an analogous edge (see below).

If it is always possible to establish a path from any vertex to every other even after removing any $k - 1$ vertices, then the graph is said to be **k -vertex-connected** or **k -connected**. Note that a graph is k -connected if and only if it contains k internally disjoint paths between any two vertices. The example graph above is connected (and therefore 1-connected), but not 2-connected. The **vertex connectivity** or **connectivity** $\kappa(G)$ of a graph G is the minimum number of vertices that need to be removed to disconnect G . The complete graph K_n has connectivity $n - 1$ for $n > 1$; and a disconnected graph has connectivity 0.

In network theory, a **giant component** is a connected subgraph that contains a majority of the entire graph's nodes.

A **bridge**, or *cut edge* or *isthmus*, is an edge whose removal disconnects a graph. (For example, all the edges in a tree are bridges.) A *cut vertex* is an analogous vertex (see above). A **disconnecting set** is a set of edges whose removal increases the number of components. An **edge cut** is the set of all edges which have one vertex in some proper vertex subset S and the other vertex in $V(G) \setminus S$. Edges of K_3 form a disconnecting set but not an edge cut. Any two edges of K_3 form a minimal disconnecting set as well as an edge cut. An edge cut is necessarily a disconnecting set; and a minimal disconnecting set of a nonempty graph is necessarily an edge cut. A **bond** is a minimal (but not necessarily minimum), nonempty set of edges whose removal disconnects a graph.

A graph is **k -edge-connected** if any subgraph formed by removing any $k - 1$ edges is still connected. The **edge connectivity** $\kappa'(G)$ of a graph G is the minimum number of edges needed to disconnect G . One well-known result is that $\kappa(G) \leq \kappa'(G) \leq \delta(G)$.

A **component** is a maximally connected subgraph. A **block** is either a maximally 2-connected subgraph, a bridge (together with its vertices), or an isolated vertex. A **biconnected component** is a 2-connected component.

An **articulation point** (also known as a *separating vertex*) of a graph is a vertex whose removal from the graph increases its number of connected components. A biconnected component can be defined as a subgraph induced by a maximal set of nodes that has no separating vertex.

Distance

The **distance** $d_G(u, v)$ between two (not necessary distinct) vertices u and v in a graph G is the length of a shortest path between them. The subscript G is usually dropped when there is no danger of confusion. When u and v are identical, their distance is 0. When u and v are unreachable from each other, their distance is defined to be infinity ∞ .

The **eccentricity** $\varepsilon_G(v)$ of a vertex v in a graph G is the maximum distance from v to any other vertex. The **diameter** $\text{diam}(G)$ of a graph G is the maximum eccentricity over all vertices in a graph; and the **radius** $\text{rad}(G)$, the minimum. When there are two components in G , $\text{diam}(G)$ and $\text{rad}(G)$ defined to be infinity ∞ . Trivially, $\text{diam}(G) \leq 2 \text{ rad}(G)$. Vertices with maximum eccentricity are called **peripheral vertices**. Vertices of minimum eccentricity form the **center**. A tree has at most two center vertices.

The **Wiener index of a vertex** v in a graph G , denoted by $W_G(v)$ is the sum of distances between v and all others. The **Wiener index of a graph** G , denoted by $W(G)$, is the sum of distances over all pairs of vertices. An undirected graph's **Wiener polynomial** is defined to be $\sum q^{d(u,v)}$ over all unordered pairs of vertices u and v . Wiener index and Wiener polynomial are of particular interest to mathematical chemists.

The **k -th power** G^k of a graph G is a supergraph formed by adding an edge between all pairs of vertices of G with distance at most k . A **second power** of a graph is also called a **square**.

A **k -spanner** is a spanning subgraph, S , in which every two vertices are at most k times as far apart on S than on G . The number k is the **dilation**. k -spanner is used for studying geometric network optimization.

Genus

A **crossing** is a pair of intersecting edges. A graph is **embeddable** on a surface if its vertices and edges can be arranged on it without any crossing. The **genus** of a graph is the lowest genus of any surface on which the graph can embed.

A **planar graph** is one which *can be* drawn on the (Euclidean) plane without any crossing; and a **plane graph**, one which *is* drawn in such fashion. In other words, a planar graph is a graph of genus 0. The example graph is planar; the complete graph on n vertices, for $n > 4$, is not planar. Also, a tree is necessarily a planar graph.

When a graph is drawn without any crossing, any cycle that surrounds a region without any edges reaching from the cycle into the region forms a **face**. Two faces on a plane graph are **adjacent** if they share a common edge. A **dual**, or **planar dual** when the context needs to be clarified, G^* of a plane graph G is a graph whose vertices represent the faces, including any outerface, of G and are adjacent in G^* if and only if their corresponding faces are adjacent in G . The dual of a planar graph is always a planar *pseudograph* (e.g. consider the dual of a triangle). In the familiar case of a 3-connected simple planar graph G (isomorphic to a convex polyhedron P), the dual G^* is also a 3-connected simple planar graph (and isomorphic to the dual polyhedron P^*).

Furthermore, since we can establish a sense of "inside" and "outside" on a plane, we can identify an "outermost" region that contains the entire graph if the graph does not cover the entire plane. Such outermost region is called an **outer face**. An **outerplanar graph** is one which *can be* drawn in the planar fashion such that its vertices are all adjacent to the outer face; and an **outerplane graph**, one which *is* drawn in such fashion.

The minimum number of crossings that must appear when a graph is drawn on a plane is called the **crossing number**.

The minimum number of planar graphs needed to cover a graph is the **thickness** of the graph.

Weighted graphs and networks

A **weighted graph** associates a label (**weight**) with every edge in the graph. Weights are usually real numbers. They may be restricted to rational numbers or integers. Certain algorithms require further restrictions on weights; for instance, Dijkstra's algorithm works properly only for positive weights. The **weight of a path** or the **weight of a tree** in a weighted graph is the sum of the weights of the selected edges. Sometimes a non-edge is labeled by a special weight representing infinity. Sometimes the word **cost** is used instead of weight. When stated without any qualification, a graph is always assumed to be unweighted. In some writing on graph theory the term **network** is a synonym for a **weighted graph**. A network may be directed or undirected, it may contain special vertices (nodes), such as **source** or **sink**. The classical network problems include:

- minimum cost spanning tree,
- shortest paths,
- maximal flow (and the max-flow min-cut theorem)

Direction

Main article: Digraph (mathematics)

A **directed arc**, or *directed edge*, is an ordered pair of endvertices that can be represented graphically as an arrow drawn between the endvertices. In such an ordered pair the first vertex is called the *initial vertex* or **tail**; the second one is called the *terminal vertex* or **head** (because it appears at the arrow head). An **undirected edge** disregards any sense of direction and treats both endvertices interchangeably. A **loop** in a digraph, however, keeps a sense of direction and treats both head and tail identically. A set of arcs are **multiple**, or *parallel*, if they share the same head and the same tail. A pair of arcs are **anti-parallel** if one's head/tail is the other's tail/head. A **digraph**, or *directed graph*, or **oriented graph**, is analogous to an undirected graph except that it contains only arcs. A **mixed graph** may contain both directed and undirected edges; it generalizes both directed and undirected graphs. When stated without any qualification, a graph is almost always assumed to be undirected.

A digraph is called **simple** if it has no loops and at most one arc between any pair of vertices. When stated without any qualification, a digraph is usually assumed to be simple. A **quiver** is a directed graph which is specifically allowed, but not required, to have loops and more than one arc between any pair of vertices.

In a digraph Γ , we distinguish the **out degree** $d_{\Gamma}^+(v)$, the number of edges leaving a vertex v , and the **in degree** $d_{\Gamma}^-(v)$, the number of edges entering a vertex v . If the graph is oriented, the degree $d_{\Gamma}(v)$ of a vertex v is equal to the sum of its out- and in- degrees. When the context is clear, the subscript Γ can be dropped. Maximum and minimum out degrees are denoted by $\Delta^+(\Gamma)$ and $\delta^+(\Gamma)$; and maximum and minimum in degrees, $\Delta^-(\Gamma)$ and $\delta^-(\Gamma)$.

An **out-neighborhood**, or *successor set*, $N_{\Gamma}^+(v)$ of a vertex v is the set of heads of arcs going from v . Likewise, an **in-neighborhood**, or *predecessor set*, $N_{\Gamma}^-(v)$ of a vertex v is the set of tails of arcs going into v .

A **source** is a vertex with 0 in-degree; and a **sink**, 0 out-degree.

A vertex v **dominates** another vertex u if there is an arc from v to u . A vertex subset S is **out-dominating** if every vertex not in S is dominated by some vertex in S ; and **in-dominating** if every vertex in S is dominated by some vertex not in S .

A **kernel** in a (possibly directed) graph G is an independent set S such that every vertex in $V(G) \setminus S$ dominates some vertex in S . In undirected graphs, kernels are maximal independent sets.^[2] A digraph is **kernel perfect** if every induced sub-digraph has a kernel.^[3]

An **Eulerian digraph** is a digraph with equal in- and out-degrees at every vertex.

The **zweieck** of an undirected edge $e = (u, v)$ is the pair of diedges (u, v) and (v, u) which form the simple dicircuit.

An **orientation** is an assignment of directions to the edges of an undirected or partially directed graph. When stated without any qualification, it is usually assumed that all undirected edges are replaced by a directed one in an orientation. Also, the underlying graph is usually assumed to be undirected and simple.

A **tournament** is a digraph in which each pair of vertices is connected by exactly one arc. In other words, it is an oriented complete graph.

A **directed path**, or just a *path* when the context is clear, is an oriented simple path such that all arcs go the same direction, meaning all internal vertices have in- and out-degrees 1. A vertex v is **reachable** from another vertex u if there is a directed path that starts from u and ends at v . Note that in general the condition that u is reachable from v does not imply that v is also reachable from u .

If v is reachable from u , then u is a **predecessor** of v and v is a **successor** of u . If there is an arc from u to v , then u is a **direct predecessor** of v , and v is a **direct successor** of u .

A digraph is **strongly connected** if every vertex is reachable from every other following the directions of the arcs. On the contrary, a digraph is **weakly connected** if its underlying undirected graph is connected. A weakly connected graph can be thought of as a digraph in which every vertex is "reachable" from every other but not necessarily following the directions of the arcs. A strong orientation is an orientation that produces a strongly connected digraph.

A **directed cycle**, or just a *cycle* when the context is clear, is an oriented simple cycle such that all arcs go the same direction, meaning all vertices have in- and out-degrees 1. A digraph is **acyclic** if it does not contain any directed cycle. A finite, acyclic digraph with no isolated vertices necessarily contains at least one source and at least one sink.

An **arborescence**, or *out-tree* or *branching*, is an oriented tree in which all vertices are reachable from a single vertex. Likewise, an *in-tree* is an oriented tree in which a single vertex is reachable from every other one.

Directed acyclic graphs

Main article: directed acyclic graph

The partial order structure of directed acyclic graphs (or DAGs) gives them their own terminology.

If there is a directed edge from u to v , then we say u is a **parent** of v and v is a **child** of u . If there is a directed path from u to v , we say u is an **ancestor** of v and v is a **descendant** of u .

The **moral graph** of a DAG is the undirected graph created by adding an (undirected) edge between all parents of the same node (sometimes called *marrying*), and then replacing all directed edges by undirected edges. A DAG is **perfect** if, for each node, the set of parents is complete (i.e. no new edges need to be added when forming the moral graph).

Colouring

Main article: Graph colouring

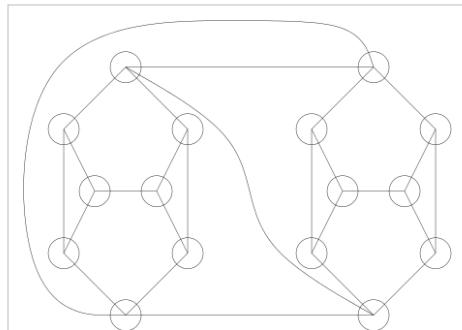
Vertices in graphs can be given **colours** to identify or label them. Although they may actually be rendered in diagrams in different colours, working mathematicians generally pencil in numbers or letters (usually numbers) to represent the colours.

Given a graph $G(V, E)$ a **k -colouring** of G is a map $\phi : V \rightarrow \{1, \dots, k\}$ with the property that $(u, v) \in E \Rightarrow \phi(u) \neq \phi(v)$ - in other words, every vertex is assigned a colour with the condition that adjacent vertices cannot be assigned the same colour.

The **chromatic number** $\chi(G)$ is the smallest k for which G has a k -colouring.

Given a graph and a colouring, the **colour classes** of the graph are the sets of vertices given the same colour.

A graph is called **k -critical** if its chromatic number is k but all of its proper subgraphs have chromatic number less than k . An odd cycle is 3-critical, and the complete graph on k vertices is k -critical.



This graph is an example of a 4-critical graph. Its chromatic number is 4 but all of its proper subgraphs have a chromatic number less than 4.

This graph is also planar

Various

A **graph invariant** is a property of a graph G , usually a number or a polynomial, that depends only on the isomorphism class of G . Examples are the order, genus, chromatic number, and chromatic polynomial of a graph.

References

- [1] ; .
- [2] Bondy, J.A., Murty, U.S.R., *Graph Theory*, p. 298
- [3] Béla Bollobás, *Modern Graph theory*, p. 298
- Bollobás, Béla (1998). *Modern Graph Theory*. Graduate Texts in Mathematics **184**. New York: Springer-Verlag. ISBN 0-387-98488-7. Zbl 0902.05016 (<http://www.zentralblatt-math.org/zmath/en/search/?format=complete&q=an:0902.05016>). [Packed with advanced topics followed by a historical overview at the end of each chapter.]
- Brandstädt, Andreas; Le, Van Bang; Spinrad, Jeremy P. (1999). *Graph classes: a survey*. SIAM Monographs on Discrete Mathematics. and Applications **3**. Philadelphia, PA: Society for Industrial and Applied Mathematics. ISBN 978-0-898714-32-6. Zbl 0919.05001 (<http://www.zentralblatt-math.org/zmath/en/search/?format=complete&q=an:0919.05001>).
- Diestel, Reinhard (2005). *Graph Theory* (<http://www.math.uni-hamburg.de/home/diestel/books/graph-theory/>). Graduate Texts in Mathematics **173** (3rd ed.). Springer-Verlag. ISBN 3-540-26182-6. Zbl 1086.05001 (<http://www.zentralblatt-math.org/zmath/en/search/?format=complete&q=an:1086.05001>). [Standard textbook, most basic material and some deeper results, exercises of various difficulty and notes at the end of each chapter; known for being quasi error-free.]
- West, Douglas B. (2001). *Introduction to Graph Theory* (2ed). Upper Saddle River: Prentice Hall. ISBN 0-13-014400-2. [Tons of illustrations, references, and exercises. The most complete introductory guide to the subject.]
- Weisstein, Eric W., "Graph" (<http://mathworld.wolfram.com/Graph.html>), *MathWorld*.
- Zaslavsky, Thomas. Glossary of signed and gain graphs and allied areas. *Electronic Journal of Combinatorics*, Dynamic Surveys in Combinatorics, # DS 8. <http://www.combinatorics.org/Surveys/>

Undirected graphs

This article is about sets of vertices connected by edges. For graphs of mathematical functions, see Graph of a function. For other uses, see Graph (disambiguation).

In mathematics, and more specifically in graph theory, a **graph** is a representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by mathematical abstractions called *vertices*, and the links that connect some pairs of vertices are called *edges*. Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. Graphs are one of the objects of study in discrete mathematics.

The edges may be directed or undirected. For example, if the vertices represent people at a party, and there is an edge between two people if they shake hands, then this is an undirected graph, because if person A shook hands with person B, then person B also shook hands with person A. In contrast, if there is an edge from person A to person B when person A knows of person B, then this graph is directed, because knowledge of someone is not necessarily a symmetric relation (that is, one person knowing another person does not necessarily imply the reverse; for example, many fans may know of a celebrity, but the celebrity is unlikely to know of all their fans). This latter type of graph is called a *directed graph* and the edges are called *directed edges* or *arcs*.

Vertices are also called *nodes* or *points*, and **edges** are also called *arcs* or *lines*. Graphs are the basic subject studied by graph theory. The word "graph" was first used in this sense by J.J. Sylvester in 1878.^[1]

Definitions

Definitions in graph theory vary. The following are some of the more basic ways of defining graphs and related mathematical structures.

Graph

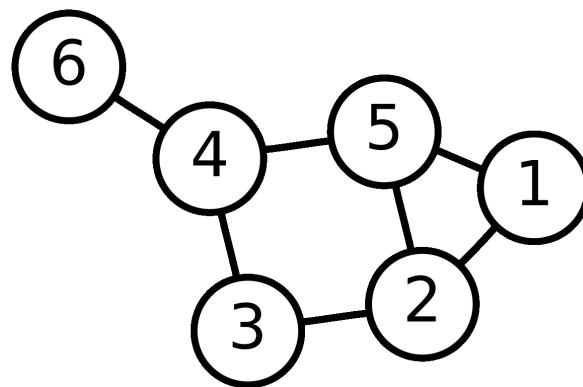
In the most common sense of the term,^[2] a **graph** is an ordered pair $G = (V, E)$ comprising a set V of **vertices** or **nodes** together with a set E of **edges** or **lines**, which are 2-element subsets of V (i.e., an edge is related with two vertices, and the relation is represented as an unordered pair of the vertices with respect to the particular edge). To avoid ambiguity, this type of graph may be described precisely as undirected and simple.

Other senses of *graph* stem from different conceptions of the edge set. In one more generalized notion,^[3] E is a set together with a relation of **incidence** that associates with each edge two vertices. In another generalized notion, E is a multiset of unordered pairs of (not necessarily distinct) vertices. Many authors call this type of object a multigraph or pseudograph.

All of these variants and others are described more fully below.

The vertices belonging to an edge are called the **ends**, **endpoints**, or **end vertices** of the edge. A vertex may exist in a graph and not belong to an edge.

V and E are usually taken to be finite, and many of the well-known results are not true (or are rather different) for **infinite graphs** because many of the arguments fail in the infinite case. The **order** of a graph is $|V|$ (the number of vertices). A graph's **size** is $|E|$, the number of edges. The **degree** of a vertex is the number of edges that connect to



A drawing of a labeled graph on 6 vertices and 7 edges.

it, where an edge that connects to the vertex at both ends (a loop) is counted twice.

For an edge $\{u, v\}$, graph theorists usually use the somewhat shorter notation uv .

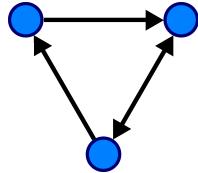
Adjacency relation

The edges E of an undirected graph G induce a symmetric binary relation \sim on V that is called the **adjacency** relation of G . Specifically, for each edge $\{u, v\}$ the vertices u and v are said to be **adjacent** to one another, which is denoted $u \sim v$.

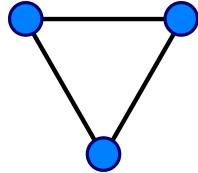
Types of graphs

Distinction in terms of the main definition

As stated above, in different contexts it may be useful to define the term *graph* with different degrees of generality. Whenever it is necessary to draw a strict distinction, the following terms are used. Most commonly, in modern texts in graph theory, unless stated otherwise, *graph* means "undirected simple finite graph" (see the definitions below).



A directed graph.



A simple undirected graph with three vertices and three edges. Each vertex has degree two, so this is also a regular graph.

Undirected graph

An undirected graph is one in which edges have no orientation. The edge (a, b) is identical to the edge (b, a) , i.e., they are not ordered pairs, but sets $\{u, v\}$ (or 2-multisets) of vertices. The maximum number of edges in an undirected graph without a self-loop is $n(n - 1)/2$.

Directed graph

Main article: Directed graph

A **directed graph** or **digraph** is an ordered pair $D = (V, A)$ with

- V a set whose elements are called **vertices** or **nodes**, and
- A a set of ordered pairs of vertices, called **arcs**, **directed edges**, or **arrows**.

An arc $a = (x, y)$ is considered to be directed **from** x **to** y ; y is called the **head** and x is called the **tail** of the arc; y is said to be a **direct successor** of x , and x is said to be a **direct predecessor** of y . If a path leads from x to y , then y is said to be a **successor** of x and **reachable** from x , and x is said to be a **predecessor** of y . The arc (y, x) is called the arc (x, y) **inverted**.

A directed graph D is called **symmetric** if, for every arc in D , the corresponding inverted arc also belongs to D . A symmetric loopless directed graph $D = (V, A)$ is equivalent to a simple undirected graph $G = (V, E)$, where the pairs of inverse arcs in A correspond 1-to-1 with the edges in E ; thus the edges in G number $|E| = |A|/2$, or half the number

of arcs in D .

An **oriented graph** is a directed graph in which at most one of (x, y) and (y, x) may be arcs.

Mixed graph

Main article: Mixed graph

A **mixed graph** G is a graph in which some edges may be directed and some may be undirected. It is written as an ordered triple $G = (V, E, A)$ with V , E , and A defined as above. Directed and undirected graphs are special cases.

Multigraph

A loop is an edge (directed or undirected) which starts and ends on the same vertex; these may be permitted or not permitted according to the application. In this context, an edge with two different ends is called a **link**.

The term "multigraph" is generally understood to mean that multiple edges (and sometimes loops) are allowed. Where graphs are defined so as to *allow* loops and multiple edges, a multigraph is often defined to mean a graph *without* loops,^[4] however, where graphs are defined so as to *disallow* loops and multiple edges, the term is often defined to mean a "graph" which can have both multiple edges *and* loops,^[5] although many use the term "pseudograph" for this meaning.^[6]

Quiver

A **quiver** or "multidigraph" is a directed graph which may have more than one arrow from a given source to a given target. A quiver may also have directed loops in it.

Simple graph

As opposed to a multigraph, a simple graph is an undirected graph that has no loops (edges connected at both ends to the same vertex) and no more than one edge between any two different vertices. In a simple graph the edges of the graph form a set (rather than a multiset) and each edge is a *distinct* pair of vertices. In a simple graph with n vertices every vertex has a degree that is less than n (the converse, however, is not true — there exist non-simple graphs with n vertices in which every vertex has a degree smaller than n).

Weighted graph

A graph is a weighted graph if a number (weight) is assigned to each edge. Such weights might represent, for example, costs, lengths or capacities, etc. depending on the problem at hand. Some authors call such a graph a network. Weighted correlation networks can be defined by soft-thresholding the pairwise correlations among variables (e.g. gene measurements).

Half-edges, loose edges

In certain situations it can be helpful to allow edges with only one end, called **half-edges**, or no ends (**loose edges**); see for example signed graphs and biased graphs.

Important graph classes

Regular graph

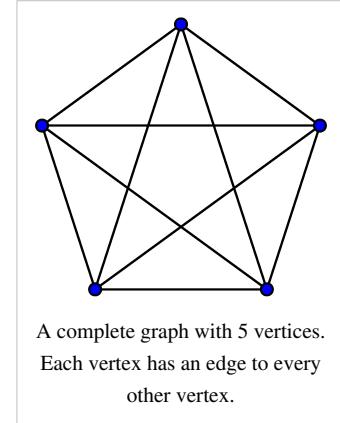
Main article: Regular graph

A regular graph is a graph where each vertex has the same number of neighbours, i.e., every vertex has the same degree or valency. A regular graph with vertices of degree k is called a k -regular graph or regular graph of degree k .

Complete graph

Main article: Complete graph

Complete graphs have the feature that each pair of vertices has an edge connecting them.



Finite and infinite graphs

A finite graph is a graph $G = (V, E)$ such that V and E are finite sets. An infinite graph is one with an infinite set of vertices or edges or both.

Most commonly in graph theory it is implied that the graphs discussed are finite. If the graphs are infinite, that is usually specifically stated.

Graph classes in terms of connectivity

Main article: Connectivity (graph theory)

In an undirected graph G , two vertices u and v are called **connected** if G contains a path from u to v . Otherwise, they are called **disconnected**. A graph is called **connected** if every pair of distinct vertices in the graph is connected; otherwise, it is called **disconnected**.

A graph is called **k -vertex-connected** or **k -edge-connected** if no set of k -1 vertices (respectively, edges) exists that, when removed, disconnects the graph. A k -vertex-connected graph is often called simply **k -connected**.

A directed graph is called **weakly connected** if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. It is **strongly connected** or **strong** if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v .

Category of all graphs

The category of all graphs is the slice category $\mathbf{Set} \downarrow D$ where $D : \mathbf{Set} \rightarrow \mathbf{Set}$ is the functor taking a set s to $s \times s$.

Properties of graphs

See also: Glossary of graph theory and Graph property

Two edges of a graph are called **adjacent** if they share a common vertex. Two arrows of a directed graph are called **consecutive** if the head of the first one is at the tail of the second one. Similarly, two vertices are called **adjacent** if they share a common edge (**consecutive** if they are at the tail and at the head of an arrow), in which case the common edge is said to **join** the two vertices. An edge and a vertex on that edge are called **incident**.

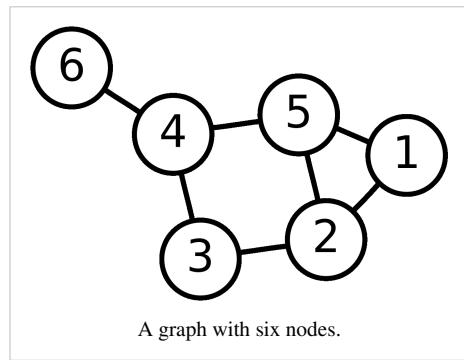
The graph with only one vertex and no edges is called the **trivial graph**. A graph with only vertices and no edges is known as an **edgeless graph**. The graph with no vertices and no edges is sometimes called the **null graph** or **empty graph**, but the terminology is not consistent and not all mathematicians allow this object.

In a **weighted** graph or digraph, each edge is associated with some value, variously called its *cost*, *weight*, *length* or other term depending on the application; such graphs arise in many contexts, for example in optimal routing problems such as the traveling salesman problem.

Normally, the vertices of a graph, by their nature as elements of a set, are distinguishable. This kind of graph may be called **vertex-labeled**. However, for many questions it is better to treat vertices as indistinguishable; then the graph may be called **unlabeled**. (Of course, the vertices may be still distinguishable by the properties of the graph itself, e.g., by the numbers of incident edges). The same remarks apply to edges, so graphs with labeled edges are called **edge-labeled** graphs. Graphs with labels attached to edges or vertices are more generally designated as **labeled**. Consequently, graphs in which vertices are indistinguishable and edges are indistinguishable are called *unlabeled*. (Note that in the literature the term *labeled* may apply to other kinds of labeling, besides that which serves only to distinguish different vertices or edges.)

Examples

- The diagram at right is a graphic representation of the following graph:
 $V = \{1, 2, 3, 4, 5, 6\}$
 $E = \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}.$
- In category theory a small category can be represented by a directed multigraph in which the objects of the category represented as vertices and the morphisms as directed edges. Then, the functors between categories induce some, but not necessarily all, of the digraph morphisms of the graph.
- In computer science, directed graphs are used to represent knowledge (e.g., Conceptual graph), finite state machines, and many other discrete structures.
- A binary relation R on a set X defines a directed graph. An element x of X is a direct predecessor of an element y of X iff xRy .
- A directed edge can model information networks such as Twitter, with one user following another ^[7]



Important graphs

Basic examples are:

- In a complete graph, each pair of vertices is joined by an edge; that is, the graph contains all possible edges.
- In a bipartite graph, the vertex set can be partitioned into two sets, W and X , so that no two vertices in W are adjacent and no two vertices in X are adjacent. Alternatively, it is a graph with a chromatic number of 2.
- In a complete bipartite graph, the vertex set is the union of two disjoint sets, W and X , so that every vertex in W is adjacent to every vertex in X but there are no edges within W or X .
- In a *linear graph* or path graph of length n , the vertices can be listed in order, v_0, v_1, \dots, v_n , so that the edges are $v_{i-1}v_i$ for each $i = 1, 2, \dots, n$. If a linear graph occurs as a subgraph of another graph, it is a path in that graph.
- In a cycle graph of length $n \geq 3$, vertices can be named v_1, \dots, v_n so that the edges are $v_{i-1}v_i$ for each $i = 2, \dots, n$ in addition to v_nv_1 . Cycle graphs can be characterized as connected 2-regular graphs. If a cycle graph occurs as a subgraph of another graph, it is a *cycle* or *circuit* in that graph.
- A planar graph is a graph whose vertices and edges can be drawn in a plane such that no two of the edges intersect (i.e., *embedded* in a plane).
- A tree is a connected graph with no cycles.
- A *forest* is a graph with no cycles (i.e. the disjoint union of one or more *trees*).

More advanced kinds of graphs are:

- The Petersen graph and its generalizations
- Perfect graphs
- Cographs
- Chordal graphs
- Other graphs with large automorphism groups: vertex-transitive, arc-transitive, and distance-transitive graphs.
- Strongly regular graphs and their generalization distance-regular graphs.

Operations on graphs

Main article: Operations on graphs

There are several operations that produce new graphs from old ones, which might be classified into the following categories:

- Elementary operations, sometimes called "editing operations" on graphs, which create a new graph from the original one by a simple, local change, such as addition or deletion of a vertex or an edge, merging and splitting of vertices, etc.
- Graph rewrite operations replacing the occurrence of some pattern graph within the host graph by an instance of the corresponding replacement graph.
- Unary operations, which create a significantly new graph from the old one. Examples:
 - Line graph
 - Dual graph
 - Complement graph
- Binary operations, which create new graph from two initial graphs. Examples:
 - Disjoint union of graphs
 - Cartesian product of graphs
 - Tensor product of graphs
 - Strong product of graphs
 - Lexicographic product of graphs

Generalizations

In a hypergraph, an edge can join more than two vertices.

An undirected graph can be seen as a simplicial complex consisting of 1-simplices (the edges) and 0-simplices (the vertices). As such, complexes are generalizations of graphs since they allow for higher-dimensional simplices.

Every graph gives rise to a matroid.

In model theory, a graph is just a structure. But in that case, there is no limitation on the number of edges: it can be any cardinal number, see continuous graph.

In computational biology, power graph analysis introduces power graphs as an alternative representation of undirected graphs.

In geographic information systems, geometric networks are closely modeled after graphs, and borrow many concepts from graph theory to perform spatial analysis on road networks or utility grids.

Notes

[1] See:

- J. J. Sylvester (February 7, 1878) "Chemistry and algebra," (<http://books.google.com/books?id=KcoKAAAAYAAJ&vq=Sylvester&pg=PA284#v=onepage&q&f=false>) *Nature*, **17** : 284. From page 284: "Every invariant and covariant thus becomes expressible by a graph precisely identical with a Kekul an diagram or chemicograph."
- J. J. Sylvester (1878) "On an application of the new atomic theory to the graphical representation of the invariants and covariants of binary quantics, — with three appendices." (<http://books.google.com/books?id=1q0EAAAAYAAJ&pg=PA64#v=onepage&q&f=false>) *American Journal of Mathematics, Pure and Applied*, **1** (1) : 64-90. The term "graph" first appears in this paper on page 65.

[2] See, for instance, Iyanaga and Kawada, **69 J**, p. 234 or Biggs, p. 4.

[3] See, for instance, Graham et al., p. 5.

[4] For example, see Balakrishnan, p. 1, Gross (2003), p. 4, and Zwillinger, p. 220.

[5] For example, see Bollob s, p. 7 and Diestel, p. 25.

[6] Gross (1998), p. 3, Gross (2003), p. 205, Harary, p.10, and Zwillinger, p. 220.

[7] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Bosagh Zadeh WTF: The who-to-follow system at Twitter (<http://dl.acm.org/citation.cfm?id=2488433>), Proceedings of the 22nd international conference on World Wide Web

References

- Balakrishnan, V. K. (1997-02-01). *Graph Theory* (1st ed.). McGraw-Hill. ISBN 0-07-005489-4.
- Berge, Claude (1958). *Th orie des graphes et ses applications* (in French). Dunod, Paris: Collection Universitaire de Math matiques, II. pp. viii+277. Translation: -. Dover, New York: Wiley. 2001 [1962].
- Biggs, Norman (1993). *Algebraic Graph Theory* (2nd ed.). Cambridge University Press. ISBN 0-521-45897-8.
- Bollob s, B la (2002-08-12). *Modern Graph Theory* (1st ed.). Springer. ISBN 0-387-98488-7.
- Bang-Jensen, J.; Gutin, G. (2000). *Digraphs: Theory, Algorithms and Applications* (<http://www.cs.rhul.ac.uk/books/dbook/>). Springer.
- Diestel, Reinhard (2005). *Graph Theory* (<http://diestel-graph-theory.com/GrTh.html>) (3rd ed.). Berlin, New York: Springer-Verlag. ISBN 978-3-540-26183-4..
- Graham, R.L., Gr tschel, M., and Lov sz, L, ed. (1995). *Handbook of Combinatorics*. MIT Press. ISBN 0-262-07169-X.
- Gross, Jonathan L.; Yellen, Jay (1998-12-30). *Graph Theory and Its Applications*. CRC Press. ISBN 0-8493-3982-0.
- Gross, Jonathan L., & Yellen, Jay, ed. (2003-12-29). *Handbook of Graph Theory*. CRC. ISBN 1-58488-090-2.
- Harary, Frank (January 1995). *Graph Theory*. Addison Wesley Publishing Company. ISBN 0-201-41033-8.
- Iyanaga, Sh kichi; Kawada, Yukiyosi (1977). *Encyclopedic Dictionary of Mathematics*. MIT Press. ISBN 0-262-09016-3.
- Zwillinger, Daniel (2002-11-27). *CRC Standard Mathematical Tables and Formulae* (31st ed.). Chapman & Hall/CRC. ISBN 1-58488-291-3.

Further reading

- Trudeau, Richard J. (1993). *Introduction to Graph Theory* (<http://store.doverpublications.com/0486678709.html>) (Corrected, enlarged republication. ed.). New York: Dover Publications. ISBN 978-0-486-67870-2. Retrieved 8 August 2012.

External links

Library resources about Graph(mathematics)
<ul style="list-style-type: none"> • Resources in your library (http://tools.wmflabs.org/ftl/cgi-bin/ftl?st=wp&su=Graph+(mathematics))

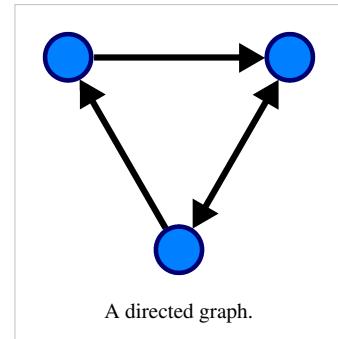
- Weisstein, Eric W., "Graph" (<http://mathworld.wolfram.com/Graph.html>), *MathWorld*.

Directed graphs

In mathematics, and more specifically in graph theory, a **directed graph** (or **digraph**) is a graph, or set of nodes connected by edges, where the edges have a direction associated with them. In formal terms, a digraph is a pair $G = (V, A)$ (sometimes $G = (V, E)$) of:^[1]

- a set V , whose elements are called *vertices* or *nodes*,
- a set A of ordered pairs of vertices, called *arcs*, *directed edges*, or *arrows* (and sometimes simply *edges* with the corresponding set named E instead of A).

It differs from an ordinary or undirected graph, in that the latter is defined in terms of unordered pairs of vertices, which are usually called edges.



A digraph is called "simple" if it has no loops, and no multiple arcs (arcs with same starting and ending nodes). A *directed multigraph*, in which the arcs constitute a multiset, rather than a set, of ordered pairs of vertices may have loops (that is, "self-loops" with same starting and ending node) and multiple arcs. Some but not all texts allow a digraph, without the qualification simple, to have self loops, multiple arcs, or both.

Basic terminology

An arc $e = (x, y)$ is considered to be directed *from x to y* ; y is called the *head* and x is called the *tail* of the arc; y is said to be a *direct successor* of x , and x is said to be a *direct predecessor* of y . If a path made up of one or more successive arcs leads from x to y , then y is said to be a *successor* of x , and x is said to be a *predecessor* of y . The arc (y, x) is called the arc (x, y) *inverted*.

An orientation of a simple undirected graph is obtained by assigning a direction to each edge. Any directed graph constructed this way is called an "oriented graph". A directed graph is an oriented simple graph if and only if it has neither self-loops nor 2-cycles.^[2]

A *weighted digraph* is a digraph with weights assigned to its arcs, similar to a weighted graph. In the context of graph theory a digraph with weighted edges is called a *network*.

The adjacency matrix of a digraph (with loops and multiple arcs) is the integer-valued matrix with rows and columns corresponding to the nodes, where a nondiagonal entry a_{ij} is the number of arcs from node i to node j , and the diagonal entry a_{ii} is the number of loops at node i . The adjacency matrix of a digraph is unique up to identical permutation of rows and columns.

Another matrix representation for a digraph is its incidence matrix.

See direction for more definitions.

Indegree and outdegree

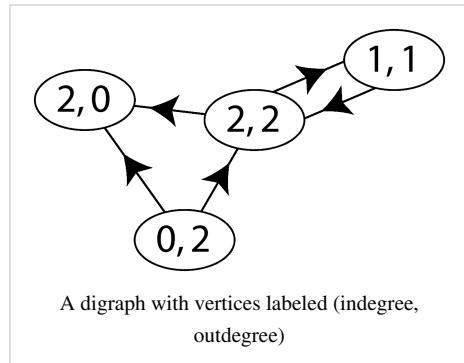
For a node, the number of head endpoints adjacent to a node is called the *indegree* of the node and the number of tail endpoints adjacent to a node is its *outdegree* (called "branching factor" in trees).

The indegree is denoted $\deg^-(v)$ and the outdegree as $\deg^+(v)$. A vertex with $\deg^-(v) = 0$ is called a *source*, as it is the origin of each of its incident edges. Similarly, a vertex with $\deg^+(v) = 0$ is called a *sink*.

The *degree sum formula* states that, for a directed graph,

$$\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = |A|.$$

If for every node $v \in V$, $\deg^+(v) = \deg^-(v)$, the graph is called a *balanced digraph*.^[3]



Degree sequence

The degree sequence of a directed graph is the list of its indegree and outdegree pairs; for the above example we have degree sequence ((2,0),(2,2),(0,2),(1,1)). The degree sequence is a directed graph invariant so isomorphic directed graphs have the same degree sequence. However, the degree sequence does not, in general, uniquely identify a graph; in some cases, non-isomorphic graphs have the same degree sequence.

The digraph realization problem is the problem of finding a digraph with the degree sequence being a given sequence of positive integer pairs. (Trailing pairs of zeros may be ignored since they are trivially realized by adding an appropriate number of isolated vertices to the digraph.) A sequence which is the degree sequence of some digraph, i.e. for which the digraph realization problem has a solution, is called a *digraphic* or *digraphical* sequence. This problem can either be solved by the Kleitman–Wang algorithm or by the Fulkerson–Chen–Anstee theorem.

Digraph connectivity

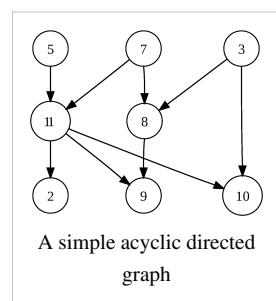
Main article: Connectivity (graph theory)

A digraph G is called *weakly connected* (or just *connected*)^[4] if the undirected *underlying graph* obtained by replacing all directed edges of G with undirected edges is a connected graph. A digraph is *strongly connected* or *strong* if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v . The *strong components* are the maximal strongly connected subgraphs.

Classes of digraphs

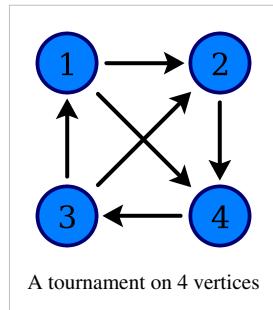
A directed graph G is called **symmetric** if, for every arc that belongs to G , the corresponding reversed arc also belongs to G . A symmetric, loopless directed graph is equivalent to an undirected graph with the edges replaced by pairs of inverse arcs; thus the number of edges is equal to the number of arcs halved.

An **acyclic** directed graph, acyclic digraph, or directed acyclic graph is a directed graph with no directed cycles. Special cases of acyclic directed graphs include multitrees (graphs in which no two directed paths from a single starting node meet back at the same ending node), oriented trees or polytrees (digraphs formed by orienting the edges of undirected acyclic graphs), and rooted trees (oriented trees in which all edges of the underlying undirected tree are directed away from the roots).



A **tournament** is an oriented graph obtained by choosing a direction for each edge in an undirected complete graph.

In the theory of Lie groups, a **quiver** Q is a directed graph serving as the domain of, and thus characterizing the shape of, a *representation* V defined as a functor, specifically an object of the functor category $\mathbf{FinVct}_K^{F(Q)}$ where $F(Q)$ is the free category on Q consisting of paths in Q and \mathbf{FinVct}_K is the category of finite-dimensional vector spaces over a field K . Representations of a quiver label its vertices with vector spaces and its edges (and hence paths) compatibly with linear transformations between them, and transform via natural transformations.



A tournament on 4 vertices

Notes

- [1] . , Section 1.10. , Section 10.
- [2] , Section 1.10.
- [3] ; .
- [4] p. 19 in the 2007 edition; p. 20 in the 2nd edition (2009).

References

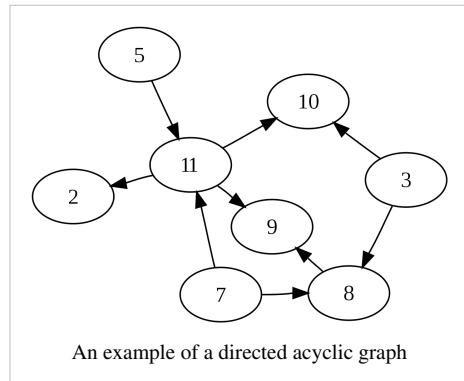
- Bang-Jensen, Jørgen; Gutin, Gregory (2000), *Digraphs: Theory, Algorithms and Applications* (<http://www.cs.rhul.ac.uk/books/dbook/>), Springer, ISBN 1-85233-268-9
(the corrected 1st edition of 2007 is now freely available on the authors' site; the 2nd edition appeared in 2009 ISBN 1-84800-997-6).
- Bondy, John Adrian; Murty, U. S. R. (1976), *Graph Theory with Applications* (<http://www.ecp6.jussieu.fr/pageperso/bondy/books/gtwa/gtwa.html>), North-Holland, ISBN 0-444-19451-7.
- Diestel, Reinhard (2005), *Graph Theory* (<http://www.math.uni-hamburg.de/home/diestel/books/graph-theory/>) (3rd ed.), Springer, ISBN 3-540-26182-6 (the electronic 3rd edition is freely available on author's site).
- Harary, Frank; Norman, Robert Z.; Cartwright, Dorwin (1965), *Structural Models: An Introduction to the Theory of Directed Graphs*, New York: Wiley.
- *Number of directed graphs (or digraphs) with n nodes.* (<http://oeis.org/A000273>)

Directed acyclic graphs

In mathematics and computer science, a **directed acyclic graph (DAG)** (pronounced /'dæg/), is a directed graph with no directed cycles. That is, it is formed by a collection of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again.

DAGs may be used to model many different kinds of information. The reachability relation in a DAG forms a partial order, and any finite partial order may be represented by a DAG using reachability. A collection of tasks that must be ordered into a sequence, subject to constraints that certain tasks must be performed earlier than others, may be represented as a DAG with a vertex for each task and an edge for each constraint; algorithms for topological ordering may be used to generate a valid sequence. Additionally, DAGs may be used as a space-efficient representation of a collection of sequences with overlapping subsequences. DAGs are also used to represent systems of events or potential events and the causal relationships between them. DAGs may also be used to model processes in which data flows in a consistent direction through a network of processors.

The corresponding concept for undirected graphs is a forest, an undirected graph without cycles. Choosing an orientation for a forest produces a special kind of directed acyclic graph called a polytree. However there are many other kinds of directed acyclic graph that are not formed by orienting the edges of an undirected acyclic graph, and every undirected graph has an acyclic orientation, an assignment of a direction for its edges that makes it into a directed acyclic graph. For this reason it may be more accurate to call directed acyclic graphs **acyclic directed graphs** or **acyclic digraphs**.

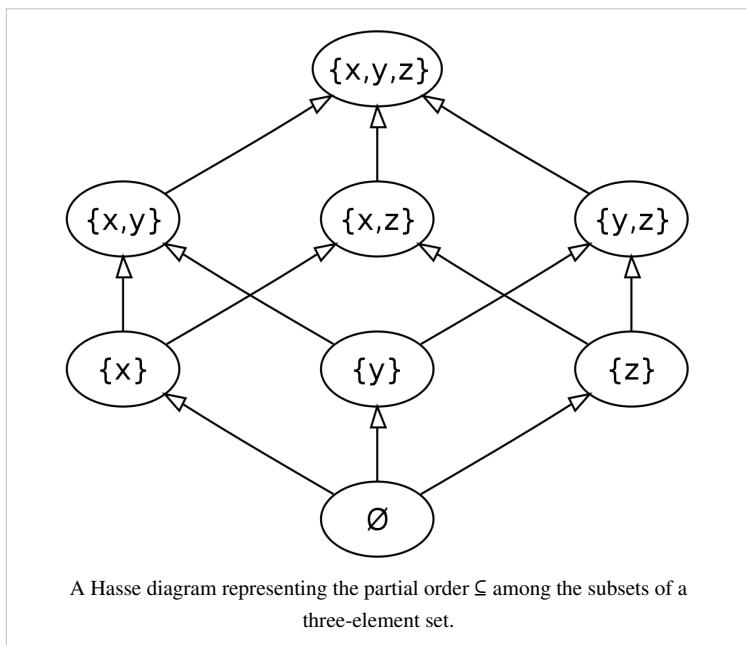


An example of a directed acyclic graph

Mathematical properties

Reachability, transitive closure, and transitive reduction

Each directed acyclic graph gives rise to a partial order \leq on its vertices, where $u \leq v$ exactly when there exists a directed path from u to v in the DAG. However, many different DAGs may give rise to this same reachability relation: for example, the DAG with two edges $a \rightarrow b$ and $b \rightarrow c$ has the same reachability as the graph with three edges $a \rightarrow b$, $b \rightarrow c$, and $a \rightarrow c$. If G is a DAG, its transitive reduction is the graph with the fewest edges that represents the same reachability as G , and its transitive closure is the graph with the most edges that represents the same reachability. The transitive reduction and transitive closure



A Hasse diagram representing the partial order \subseteq among the subsets of a three-element set.

are both uniquely defined for DAGs; in contrast, for a directed graph that is not acyclic, there can be more than one minimal subgraph with the same reachability relation.

The transitive closure of G has an edge $u \rightarrow v$ for every related pair $u \leq v$ of distinct elements in the reachability relation of G , and may therefore be thought of as a direct translation of the reachability relation \leq into graph-theoretic terms: every partially ordered set may be translated into a DAG in this way. If a DAG G represents a partial order \leq , then the transitive reduction of G is a subgraph of G with an edge $u \rightarrow v$ for every pair in the covering relation of \leq ; transitive reductions are useful in visualizing the partial orders they represent, because they have fewer edges than other graphs representing the same orders and therefore lead to simpler graph drawings. A Hasse diagram of a partial order is a drawing of the transitive reduction in which the orientation of each edge is shown by placing the starting vertex of the edge in a lower position than its ending vertex.

Topological ordering

Every directed acyclic graph has a topological ordering, an ordering of the vertices such that the starting endpoint of every edge occurs earlier in the ordering than the ending endpoint of the edge. In general, this ordering is not unique; a DAG has a unique topological ordering if and only if it has a directed path containing all the vertices, in which case the ordering is the same as the order in which the vertices appear in the path. The family of topological orderings of a DAG is the same as the family of linear extensions of the reachability relation for the DAG, so any two graphs representing the same partial order have the same set of topological orders.

Combinatorial enumeration

The graph enumeration problem of counting directed acyclic graphs was studied by Robinson (1973).^[1] The number of DAGs on n labeled nodes, for $n = 0, 1, 2, 3, \dots$, (allowing these numbers to appear in any order in a topological ordering of the DAG) is

1, 1, 3, 25, 543, 29281, 3781503, ... (sequence A003024 in OEIS).

These numbers may be computed by the recurrence relation

$$a_n = \sum_{k=1}^n (-1)^{k-1} \binom{n}{k} 2^{k(n-k)} a_{n-k}.$$

Eric W. Weisstein conjectured, and McKay et al. (2004) proved,^[2] that the same numbers count the (0,1) matrices in which all eigenvalues are positive real numbers. The proof is bijective: a matrix A is an adjacency matrix of a DAG if and only if $A + I$ is a (0,1) matrix with all eigenvalues positive, where I denotes the identity matrix. Because a DAG cannot have self-loops, its adjacency matrix must have a zero diagonal, so adding I preserves the property that all matrix coefficients are 0 or 1.

Related families of graphs

A polytree is a directed graph formed by orienting the edges of a free tree. Every polytree is a DAG. In particular, this is true of the arborescences formed by directing all edges outwards from the root of a tree. A multitree (also called a strongly unambiguous graph or a mangrove) is a directed graph in which there is at most one directed path (in either direction) between any two nodes; equivalently, it is a DAG in which, for every node v , the set of nodes reachable from v forms a tree.

Computational problems

Topological sorting and recognition

Main article: Topological sorting

Topological sorting is the algorithmic problem of finding topological orderings; it can be solved in linear time.^[3] Kahn's algorithm for topological sorting builds the vertex ordering directly, by maintaining a list of vertices that have no edges connecting them to vertices that have not already been listed, and repeatedly adding one such vertex to the end of the list that is being built. Alternatively, a topological ordering may be constructed by reversing a postorder numbering of a depth-first search graph traversal.

It is also possible to check whether a given directed graph is a DAG in linear time, either by attempting to find a topological ordering and then testing for each edge whether the resulting ordering is valid^[4] or alternatively, for some topological sorting algorithms, by verifying that the algorithm successfully orders all the vertices without meeting an error condition.^[5]

Construction from cyclic graphs

Any undirected graph may be made into a DAG by choosing a total order for its vertices and orienting every edge from the earlier endpoint in the order to the later endpoint. However, different total orders may lead to the same acyclic orientation. The number of acyclic orientations is equal to $|\chi(-1)|$, where χ is the chromatic polynomial of the given graph.

Any directed graph may be made into a DAG by removing a feedback vertex set or a feedback arc set. However, the smallest such set is NP-hard to find.^[5] An arbitrary directed graph may also be transformed into a DAG, called its condensation, by contracting each of its strongly connected components into a single supervertex. When the graph is already acyclic, its smallest feedback vertex sets and feedback arc sets are empty, and its condensation is the graph itself.

Transitive closure and transitive reduction

The transitive closure of a given DAG, with n vertices and m edges, may be constructed in time $O(mn)$ by using either breadth-first search or depth-first search to test reachability from each vertex.^[6] Alternatively, it can be solved in time $O(n^\omega)$ where $\omega < 2.373$ is the exponent for fast matrix multiplication algorithms; this is a theoretical improvement over the $O(mn)$ bound for dense graphs.^[7]

In all of these transitive closure algorithms, it is possible to distinguish pairs of vertices that are reachable by at least one path of length two or more from pairs that can only be connected by a length-one path. The transitive reduction consists of the edges that form length-one paths that are the only paths connecting their endpoints. Therefore, the transitive reduction can be constructed in the same asymptotic time bounds as the transitive closure.^[8]

Closure problem

Main article: Closure problem

The closure problem takes as input a directed acyclic graph with weights on its vertices and seeks the minimum (or maximum) weight of a closure, a set of vertices with no outgoing edges. (The problem may be formulated for directed graphs without the assumption of acyclicity, but with no greater generality, because in this case it is equivalent to the same problem on the condensation of the graph.) It may be solved in polynomial time using a reduction to the maximum flow problem.

Applications

Path algorithms

Some algorithms become simpler when used on DAGs instead of general graphs, based on the principle of topological ordering. For example, it is possible to find shortest paths and longest paths from a given starting vertex in DAGs in linear time by processing the vertices in a topological order, and calculating the path length for each vertex to be the minimum or maximum length obtained via any of its incoming edges.^[9] In contrast, for arbitrary graphs the shortest path may require slower algorithms such as Dijkstra's algorithm or the Bellman–Ford algorithm,^[10] and longest paths in arbitrary graphs are NP-hard to find.^[11]

Scheduling

DAG representations of partial orderings have many applications in scheduling problems for systems of tasks with ordering constraints.^[12] For instance, a DAG may be used to describe the dependencies between cells of a spreadsheet: if one cell is computed by a formula involving the value of a second cell, draw a DAG edge from the second cell to the first one. If the input values to the spreadsheet change, all of the remaining values of the spreadsheet may be recomputed with a single evaluation per cell, by topologically ordering the cells and re-evaluating each cell in this order. Similar problems of task ordering arise in makefiles for program compilation, instruction scheduling for low-level computer program optimization, and PERT scheduling for management of large human projects. Dependency graphs without circular dependencies form directed acyclic graphs.

Data processing networks

A directed graph may be used to represent a network of processing elements; in this formulation, data enters a processing element through its incoming edges and leaves the element through its outgoing edges. Examples of this include the following:

- In electronic circuit design, a combinational logic circuit is an acyclic system of logic gates that computes a function of an input, where the input and output of the function are represented as individual bits.
- Dataflow programming languages describe systems of values that are related to each other by a directed acyclic graph. When one value changes, its successors are recalculated; each value is evaluated as a function of its predecessors in the DAG.
- In compilers, straight line code (that is, sequences of statements without loops or conditional branches) may be represented by a DAG describing the inputs and outputs of each of the arithmetic operations performed within the code; this representation allows the compiler to perform common subexpression elimination efficiently.
- In most spreadsheet systems, the dependency graph that connects one cell to another if the first cell stores a formula that uses the value in the second cell must be a directed acyclic graph. Cycles of dependencies are disallowed because they cause the cells involved in the cycle to not have a well-defined value. Additionally, requiring the dependencies to be acyclic allows a topological sort to be used to schedule the recalculations of cell values when the spreadsheet is changed.

Causal structures

Graphs that have vertices representing events, and edges representing causal relations between events, are often acyclic – arranging the vertices in linear order of time, all arrows point in the same direction as time, from parent to child (due to causality affecting the future, not the past), and thus have no loops.

For instance, a Bayesian network represents a system of probabilistic events as nodes in a directed acyclic graph, in which the likelihood of an event may be calculated from the likelihoods of its predecessors in the DAG. In this context, the moral graph of a DAG is the undirected graph created by adding an (undirected) edge between all parents of the same node (sometimes called *marrying*), and then replacing all directed edges by undirected edges.

Another type of graph with a similar causal structure is an influence diagram, the nodes of which represent either decisions to be made or unknown information, and the edges of which represent causal influences from one node to another. In epidemiology, for instance, these diagrams are often used to estimate the expected value of different choices for intervention.

Genealogy and version history

Family trees may also be seen as directed acyclic graphs, with a vertex for each family member and an edge for each parent-child relationship. Despite the name, these graphs are not necessarily trees because of the possibility of marriages between distant relatives (so a child has a common ancestor on both the mother's and father's side) causing pedigree collapse. (The graphs of matrilineal descent ("mother" relationships between women) and patrilineal descent ("father" relationships between men) are trees within this graph.) Because no one can become their own ancestor, these graphs are acyclic.

For the same reason, the version history of a distributed revision control system generally has the structure of a directed acyclic graph, in which there is a vertex for each revision and an edge connecting pairs of revisions that were directly derived from each other; these are not trees in general due to merges.

In many randomized algorithms in computational geometry, the algorithm maintains a *history DAG* representing the version history of a geometric structure over the course of a sequence of changes to the structure. For instance in a randomized incremental algorithm for Delaunay triangulation, the triangulation changes by replacing one triangle by three smaller triangles when each point is added, and by "flip" operations that replace pairs of triangles by a different pair of triangles. The history DAG for this algorithm has a vertex for each triangle constructed as part of the algorithm, and edges from each triangle to the two or three other triangles that replace it. Tracing a path through this DAG representing the sequence of triangles that contain an individual point allows point location queries to be answered efficiently.

Data compression

Another type of application of directed acyclic graphs arises in the concise representation of a set of sequences as paths in a graph. For example, the directed acyclic word graph is a data structure in computer science formed by a directed acyclic graph with a single source and with edges labeled by letters or symbols; the paths from the source to the sinks in this graph represent a set of strings, such as English words. Any set of sequences can be represented as paths in a tree, by forming a tree node for every prefix of a sequence and making the parent of one of these nodes represent the sequence with one fewer element; the tree formed in this way for a set of strings is called a trie. A directed acyclic word graph saves space over a trie by allowing paths to diverge and rejoin, so that a set of words with the same possible suffixes can be represented by a single tree node.

The same idea of using a DAG to represent a family of paths occurs in the binary decision diagram, a DAG-based data structure for representing binary functions. In a binary decision diagram, each non-sink vertex is labeled by the name of a binary variable, and each sink and each edge is labeled by a 0 or 1. The function value for any truth assignment to the variables is the value at the sink found by following a path, starting from the single source vertex,

that at each non-sink vertex follows the outgoing edge labeled with the value of that vertex's variable. Just as directed acyclic word graphs can be viewed as a compressed form of tries, binary decision diagrams can be viewed as compressed forms of decision trees that save space by allowing paths to rejoin when they agree on the results of all remaining decisions.

References

- [1] . See also .
- [2] , Article 04.3.3.
- [3] Section 22.4, Topological sort, pp. 549–552.
- [4] For depth-first search based topological sorting algorithm, this validity check can be interleaved with the topological sorting algorithm itself; see e.g. .
- [5] , Problems GT7 and GT8, pp. 191–192.
- [6] , p. 495.
- [7] , p. 496.
- [8] , p. 38.
- [9] Cormen et al. 2001, Section 24.2, Single-source shortest paths in directed acyclic graphs, pp. 592–595.
- [10] Cormen et al. 2001, Sections 24.1, The Bellman–Ford algorithm, pp. 588–592, and 24.3, Dijkstra's algorithm, pp. 595–601.
- [11] Cormen et al. 2001, p. 966.
- [12] , p. 469.

External links

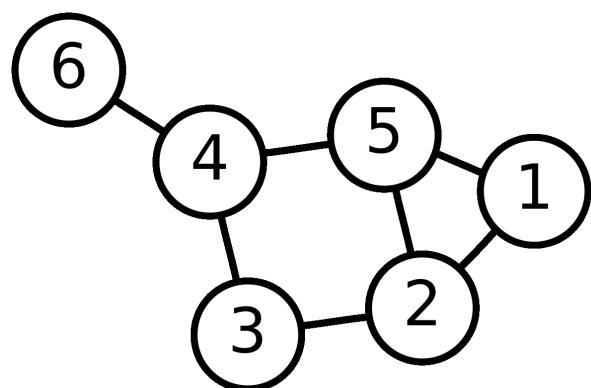
- Weisstein, Eric W., "Acyclic Digraph" (<http://mathworld.wolfram.com/AcyclicDigraph.html>), *MathWorld*.

Computer representations of graphs

In computer science, a **graph** is an abstract data type that is meant to implement the graph and hypergraph concepts from mathematics.

A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (x,y) is said to **point** or **go from x to y** . The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).



A labeled graph of 6 vertices and 7 edges.

Algorithms

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd–Warshall algorithm.

Operations

The basic operations provided by a graph data structure G usually include:

- $\text{adjacent}(G, x, y)$: tests whether there is an edge from node x to node y .
- $\text{neighbors}(G, x)$: lists all nodes y such that there is an edge from x to y .
- $\text{add}(G, x, y)$: adds to G the edge from x to y , if it is not there.
- $\text{delete}(G, x, y)$: removes the edge from x to y , if it is there.
- $\text{get_node_value}(G, x)$: returns the value associated with the node x .
- $\text{set_node_value}(G, x, a)$: sets the value associated with the node x to a .

Structures that associate values to the edges usually also provide:

- $\text{get_edge_value}(G, x, y)$: returns the value associated to the edge (x,y) .
- $\text{set_edge_value}(G, x, y, v)$: sets the value associated to the edge (x,y) to v .

Representations

Different data structures for the representation of graphs are used in practice:

Adjacency list

Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

Adjacency matrix

A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

Incidence matrix

A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

The following table gives the time complexity cost of performing various operations on graphs, for each of these representations.[Wikipedia:Citation needed](#) In the matrix representations, the entries encode the cost of following an edge. The cost of edges that are not present are assumed to be ∞ .

	Adjacency list	Adjacency matrix	Incidence matrix
Storage	$O(V + E)$	$O(V ^2)$	$O(V \cdot E)$
Add vertex	$O(1)$	$O(V ^2)$	$O(V \cdot E)$
Add edge	$O(1)$	$O(1)$	$O(V \cdot E)$
Remove vertex	$O(E)$	$O(V ^2)$	$O(V \cdot E)$
Remove edge	$O(E)$	$O(1)$	$O(V \cdot E)$
Query: are vertices u, v adjacent? (Assuming that the storage positions for u, v are known)	$O(V)$	$O(1)$	$O(E)$
Remarks	When removing edges or vertices, need to find all vertices or edges	Slow to add or remove vertices, because matrix must be resized/copied	Slow to add or remove vertices and edges, because matrix must be resized/copied

Adjacency lists are generally preferred because they efficiently represent sparse graphs. An adjacency matrix is preferred if the graph is dense, that is the number of edges $|E|$ is close to the number of vertices squared, $|V|^2$, or if one must be able to quickly look up if there is an edge connecting two vertices.^[1]

References

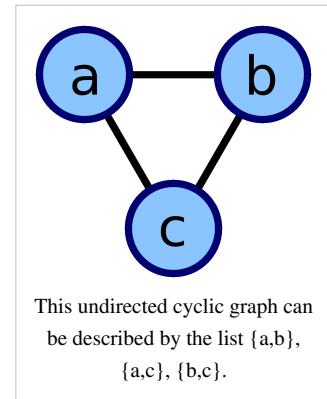
- [1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-53196-8.

External links

- Boost Graph Library: a powerful C++ graph library (<http://www.boost.org/libs/graph>)
- Networkx: a Python graph library (<http://networkx.github.com/>)

Adjacency list

In graph theory and computer science, an **adjacency list** representation of a graph is a collection of unordered lists, one for each vertex in the graph. Each list describes the set of neighbors of its vertex. See Storing a sparse matrix for alternatives.



Implementation details

The graph pictured above has this adjacency list representation:		
a	adjacent to	b,c
b	adjacent to	a,c
c	adjacent to	a,b

An adjacency list representation for a graph associates each vertex in the graph with the collection of its neighboring vertices or edges. There are many variations of this basic idea, differing in the details of how they implement the association between vertices and collections, in how they implement the collections, in whether they include both vertices and edges or only vertices as first class objects, and in what kinds of objects are used to represent the vertices and edges.

- An implementation suggested by Guido van Rossum uses a hash table to associate each vertex in a graph with an array of adjacent vertices. In this representation, a vertex may be represented by any hashable object. There is no explicit representation of edges as objects.
- Cormen et al. suggest an implementation in which the vertices are represented by index numbers. Their representation uses an array indexed by vertex number, in which the array cell for each vertex points to a singly linked list of the neighboring vertices of that vertex. In this representation, the nodes of the singly linked list may be interpreted as edge objects; however, they do not store the full information about each edge (they only store one of the two endpoints of the edge) and in undirected graphs there will be two different linked list nodes for each edge (one within the lists for each of the two endpoints of the edge).
- The object oriented **incidence list** structure suggested by Goodrich and Tamassia has special classes of vertex objects and edge objects. Each vertex object has an instance variable pointing to a collection object that lists the neighboring edge objects. In turn, each edge object points to the two vertex objects at its endpoints. This version of the adjacency list uses more memory than the version in which adjacent vertices are listed directly, but the existence of explicit edge objects allows it extra flexibility in storing additional information about edges.

Operations

The main operation performed by the adjacency list data structure is to report a list of the neighbors of a given vertex. Using any of the implementations detailed above, this can be performed in constant time per neighbor. In other words, the total time to report all of the neighbors of a vertex v is proportional to the degree of v .

It is also possible, but not as efficient, to use adjacency lists to test whether an edge exists or does not exist between two specified vertices. In an adjacency list in which the neighbors of each vertex are unsorted, testing for the existence of an edge may be performed in time proportional to the degree of one of the two given vertices, by using a sequential search through the neighbors of this vertex. If the neighbors are represented as a sorted array, binary search may be used instead, taking time proportional to the logarithm of the degree.

Trade-offs

The main alternative to the adjacency list is the adjacency matrix, a matrix whose rows and columns are indexed by vertices and whose cells contain a Boolean value that indicates whether an edge is present between the vertices corresponding to the row and column of the cell. For a sparse graph (one in which most pairs of vertices are not connected by edges) an adjacency list is significantly more space-efficient than an adjacency matrix (stored as an array): the space usage of the adjacency list is proportional to the number of edges and vertices in the graph, while for an adjacency matrix stored in this way the space is proportional to the square of the number of vertices. However, it is possible to store adjacency matrices more space-efficiently, matching the linear space usage of an adjacency list, by using a hash table indexed by pairs of vertices rather than an array.

The other significant difference between adjacency lists and adjacency matrices is in the efficiency of the operations they perform. In an adjacency list, the neighbors of each vertex may be listed efficiently, in time proportional to the degree of the vertex. In an adjacency matrix, this operation takes time proportional to the number of vertices in the graph, which may be significantly higher than the degree. On the other hand, the adjacency matrix allows testing whether two vertices are adjacent to each other in constant time; the adjacency list is slower to support this operation.

Data Structures

For use as a data structure, the main alternative to the adjacency matrix is the adjacency list. Because each entry in the adjacency matrix requires only one bit, it can be represented in a very compact way, occupying only $|V|^2/8$ bytes of contiguous space. Besides avoiding wasted space, this compactness encourages locality of reference. However, for a sparse graph, adjacency lists require less storage space, because they do not waste any space to represent edges that are not present. Using a naïve array implementation on a 32-bit computer, an adjacency list for an undirected graph requires about $8|E|$ bytes of storage.

Noting that a simple graph can have at most $|V|^2$ edges, allowing loops, we can let $d = |E|/|V|^2$ denote the density of the graph. Then, $8|E| > |V|^2/8$, or the adjacency list representation occupies more space precisely when $d > 1/64$. Thus a graph must be sparse indeed to justify an adjacency list representation. Besides the space tradeoff, the different data structures also facilitate different operations. Finding all vertices adjacent to a given vertex in an adjacency list is as simple as reading the list. With an adjacency matrix, an entire row must instead be scanned, which takes $O(|V|)$ time. Whether there is an edge between two given vertices can be determined at once with an adjacency matrix, while requiring time proportional to the minimum degree of the two vertices with the adjacency list.

References

Additional reading

- David Eppstein (1996). "ICS 161 Lecture Notes: Graph Algorithms" (<http://www.ics.uci.edu/~eppstein/161/960201.html>).

External links

- The Boost Graph Library implements an efficient adjacency list (http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/index.html)
- Open Data Structures - Section 12.2 - AdjacencyList: A Graph as a Collection of Lists (http://opendatastructures.org/versions/edition-0.1e/ods-java/12_2_AdjacencyLists_Graph_a.html)

Adjacency matrix

In mathematics and computer science, an **adjacency matrix** is a means of representing which vertices (or nodes) of a graph are adjacent to which other vertices. Another matrix representation for a graph is the incidence matrix.

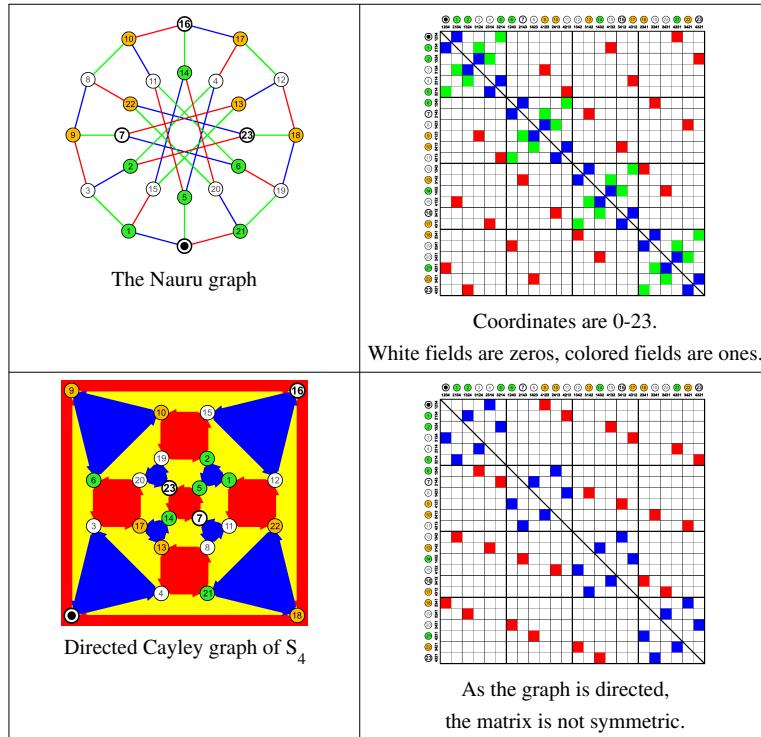
Specifically, the adjacency matrix of a finite graph G on n vertices is the $n \times n$ matrix where the non-diagonal entry a_{ij} is the number of edges from vertex i to vertex j , and the diagonal entry a_{ii} , depending on the convention, is either once or twice the number of edges (loops) from vertex i to itself. Undirected graphs often use the latter convention of counting loops twice, whereas directed graphs typically use the former convention. There exists a unique adjacency matrix for each isomorphism class of graphs (up to permuting rows and columns), and it is not the adjacency matrix of any other isomorphism class of graphs. In the special case of a finite simple graph, the adjacency matrix is a $(0,1)$ -matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric.

The relationship between a graph and the eigenvalues and eigenvectors of its adjacency matrix is studied in spectral graph theory.

Examples

The convention followed here is that an adjacent edge counts 1 in the matrix for an undirected graph.

Labeled graph	Adjacency matrix
<pre> graph LR 1((1)) --- 2((2)) 1 --- 5((5)) 2 --- 3((3)) 2 --- 5 3 --- 4((4)) 4 --- 6((6)) 1 --> 1 </pre>	$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ <p>Coordinates are 1-6.</p>



- The adjacency matrix of a complete graph contains all ones except along the diagonal where there are only zeros.
- The adjacency matrix of an empty graph is a zero matrix.

Adjacency matrix of a bipartite graph

The adjacency matrix A of a bipartite graph whose parts have r and s vertices has the form

$$A = \begin{pmatrix} 0_{r,r} & B \\ B^T & 0_{s,s} \end{pmatrix},$$

where B is an $r \times s$ matrix, and 0 represents the zero matrix. Clearly, the matrix B uniquely represents the bipartite graphs. It is sometimes called the biadjacency matrix. Formally, let $G = (U, V, E)$ be a bipartite graph with parts $U = u_1, \dots, u_r$ and $V = v_1, \dots, v_s$. The **biadjacency matrix** is the $r \times s$ 0-1 matrix B in which $b_{i,j} = 1$ iff $(u_i, v_j) \in E$.

If G is a bipartite multigraph or weighted graph then the elements $b_{i,j}$ are taken to be the number of edges between the vertices or the weight of the edge (u_i, v_j) , respectively.

Properties

The adjacency matrix of an undirected simple graph is symmetric, and therefore has a complete set of real eigenvalues and an orthogonal eigenvector basis. The set of eigenvalues of a graph is the **spectrum** of the graph.

Suppose two directed or undirected graphs G_1 and G_2 with adjacency matrices A_1 and A_2 are given. G_1 and G_2 are isomorphic if and only if there exists a permutation matrix P such that

$$PA_1P^{-1} = A_2.$$

In particular, A_1 and A_2 are similar and therefore have the same minimal polynomial, characteristic polynomial, eigenvalues, determinant and trace. These can therefore serve as isomorphism invariants of graphs. However, two graphs may possess the same set of eigenvalues but not be isomorphic.^[1]

If A is the adjacency matrix of the directed or undirected graph G , then the matrix A^n (i.e., the matrix product of n copies of A) has an interesting interpretation: the entry in row i and column j gives the number of (directed or undirected) walks of length n from vertex i to vertex j . This implies, for example, that the number of triangles in an

undirected graph G is exactly the trace of A^3 divided by 6.

The main diagonal of every adjacency matrix corresponding to a graph without loops has all zero entries. Note that here 'loops' means, for example $A \rightarrow A$, not 'cycles' such as $A \rightarrow B \rightarrow A$.

For (d) -regular graphs, d is also an eigenvalue of A for the vector $v = (1, \dots, 1)$, and G is connected if and only if the multiplicity of d is 1. It can be shown that $-d$ is also an eigenvalue of A if G is a connected bipartite graph. The above are results of Perron–Frobenius theorem.

Variations

An (a, b, c) -adjacency matrix A of a simple graph has $A_{ij} = a$ if ij is an edge, b if it is not, and c on the diagonal. The Seidel adjacency matrix is a **(-1,1,0)-adjacency matrix**. This matrix is used in studying strongly regular graphs and two-graphs.

The **distance matrix** has in position (i,j) the distance between vertices v_i and v_j . The distance is the length of a shortest path connecting the vertices. Unless lengths of edges are explicitly provided, the length of a path is the number of edges in it. The distance matrix resembles a high power of the adjacency matrix, but instead of telling only whether or not two vertices are connected (i.e., the connection matrix, which contains boolean values), it gives the exact distance between them.

Data structures

For use as a data structure, the main alternative to the adjacency matrix is the adjacency list. Because each entry in the adjacency matrix requires only one bit, it can be represented in a very compact way, occupying only $n^2/8$ bytes of contiguous space, where n is the number of vertices. Besides avoiding wasted space, this compactness encourages locality of reference.

However, for a sparse graph, adjacency lists require less storage space, because they do not waste any space to represent edges that are *not* present. Using a naïve array implementation on a 32-bit computer, an adjacency list for an undirected graph requires about $8e$ bytes of storage, where e is the number of edges.

Noting that a simple graph can have at most n^2 edges, allowing loops, we can let $d = e/n^2$ denote the *density* of the graph. Then, $8e > n^2/8$, or the adjacency list representation occupies more space precisely when $d > 1/64$.

Thus a graph must be sparse indeed to justify an adjacency list representation.

Besides the space tradeoff, the different data structures also facilitate different operations. Finding all vertices adjacent to a given vertex in an adjacency list is as simple as reading the list. With an adjacency matrix, an entire row must instead be scanned, which takes $O(n)$ time. Whether there is an edge between two given vertices can be determined at once with an adjacency matrix, while requiring time proportional to the minimum degree of the two vertices with the adjacency list.

References

[1] Godsil, Chris; Royle, Gordon *Algebraic Graph Theory*, Springer (2001), ISBN 0-387-95241-1, p.164

Further reading

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 22.1: Representations of graphs". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw-Hill. pp. 527–531. ISBN 0-262-03293-7.
- Godsil, Chris; Royle, Gordon (2001). *Algebraic Graph Theory*. New York: Springer. ISBN 0-387-95241-1.

External links



Wikimedia Commons has media related to *Adjacency matrices of graphs*.

- Fluffschack (<http://www.x2d.org/java/projects/fluffschack.jnlp>) — an educational Java web start game demonstrating the relationship between adjacency matrices and graphs.
- Open Data Structures - Section 12.1 - AdjacencyMatrix: Representing a Graph by a Matrix (http://opendatastructures.org/versions/edition-0.1e/ods-java/12_1_AdjacencyMatrix_Repres.html)
- McKay, Brendan. "Description of graph6 and sparse6 encodings" (<http://cs.anu.edu.au/~bdm/data/formats.txt>).
- Café math : Adjacency Matrices of Graphs (<http://www.cafemath.fr/mathblog/article.php?page=GoodWillHunting.php>) : Application of the adjacency matrices to the computation generating series of walks.

Implicit graph

In the study of graph algorithms, an **implicit graph representation** (or more simply **implicit graph**) is a graph whose vertices or edges are not represented as explicit objects in a computer's memory, but rather are determined algorithmically from some more concise input.

Neighborhood representations

The notion of an implicit graph is common in various search algorithms which are described in terms of graphs. In this context, an implicit graph may be defined as a set of rules to define all neighbors for any specified vertex. This type of implicit graph representation is analogous to an adjacency list, in that it provides easy access to the neighbors of each vertex. For instance, in searching for a solution to a puzzle such as Rubik's Cube, one may define an implicit graph in which each vertex represents one of the possible states of the cube, and each edge represents a move from one state to another. It is straightforward to generate the neighbors of any vertex by trying all possible moves in the puzzle and determining the states reached by each of these moves; however, an implicit representation is necessary, as the state space of Rubik's Cube is too large to allow an algorithm to list all of its states.

In computational complexity theory, several complexity classes have been defined in connection with implicit graphs, defined as above by a rule or algorithm for listing the neighbors of a vertex. For instance, PPA is the class of problems in which one is given as input an undirected implicit graph (in which vertices are n -bit binary strings, with a polynomial time algorithm for listing the neighbors of any vertex) and a vertex of odd degree in the graph, and must find a second vertex of odd degree. By the handshaking lemma, such a vertex exists; finding one is a problem in NP, but the problems that can be defined in this way may not necessarily be NP-complete, as it is unknown

whether PPA = NP. PPAD is an analogous class defined on implicit directed graphs that has attracted attention in algorithmic game theory because it contains the problem of computing a Nash equilibrium. The problem of testing reachability of one vertex to another in an implicit graph may also be used to characterize space-bounded nondeterministic complexity classes including NL (the class of problems that may be characterized by reachability in implicit directed graphs whose vertices are $O(\log n)$ -bit bitstrings), SL (the analogous class for undirected graphs), and PSPACE (the class of problems that may be characterized by reachability in implicit graphs with polynomial-length bitstrings). In this complexity-theoretic context, the vertices of an implicit graph may represent the states of a nondeterministic Turing machine, and the edges may represent possible state transitions, but implicit graphs may also be used to represent many other types of combinatorial structure. PLS, another complexity class, captures the complexity of finding local optima in an implicit graph.

Implicit graph models have also been used as a form of relativization in order to prove separations between complexity classes that are stronger than the known separations for non-relativized models. For instance, Childs et al. used neighborhood representations of implicit graphs to define a graph traversal problem that can be solved in polynomial time on a quantum computer but that requires exponential time to solve on any classical computer.

Adjacency labeling schemes

In the context of efficient representations of graphs, J. H. Muller defined a *local structure* or *adjacency labeling scheme* for a graph G in a given family F of graphs to be an assignment of an $O(\log n)$ -bit identifier to each vertex of G , together with an algorithm (that may depend on F but is independent of the individual graph G) that takes as input two vertex identifiers and determines whether or not they are the endpoints of an edge in G . That is, this type of implicit representation is analogous to an adjacency matrix: it is straightforward to check whether two vertices are adjacent but finding the neighbors of any vertex requires a search through all possible vertices.

Graph families with adjacency labeling schemes include:

- **Sparse graphs.** If every vertex in G has at most d neighbors, one may number the vertices of G from 1 to n and let the identifier for a vertex be the $(d + 1)$ -tuple of its own number and the numbers of its neighbors. Two vertices are adjacent when the first numbers in their identifiers appear later in the other vertex's identifier. More generally, the same approach can be used to provide an implicit representation for graphs with bounded arboricity or bounded degeneracy, including the planar graphs and the graphs in any minor-closed graph family.
- **Intersection graphs.** An interval graph is the intersection graph of a set of line segments in the real line. It may be given an adjacency labeling scheme in which the points that are endpoints of line segments are numbered from 1 to $2n$ and each vertex of the graph is represented by the numbers of the two endpoints of its corresponding interval. With this representation, one may check whether two vertices are adjacent by comparing the numbers that represent them and verifying that these numbers define overlapping intervals. The same approach works for other geometric intersection graphs including the graphs of bounded boxicity and the circle graphs, and subfamilies of these families such as the distance-hereditary graphs and cographs. However, a geometric intersection graph representation does not always imply the existence of an adjacency labeling scheme, because it may require more than a logarithmic number of bits to specify each geometric object; for instance, representing a graph as a unit disk graph may require exponentially many bits for the coordinates of the disk centers.
- **Low-dimensional comparability graphs.** The comparability graph for a partially ordered set has a vertex for each set element and an edge between two set elements that are related by the partial order. The order dimension of a partial order is the minimum number of linear orders whose intersection is the given partial order. If a partial order has bounded order dimension, then an adjacency labeling scheme for the vertices in its comparability graph may be defined by labeling each vertex with its position in each of the defining linear orders, and determining that two vertices are adjacent if each corresponding pair of numbers in their labels has the same order relation as each other pair. In particular, this allows for an adjacency labeling scheme for the chordal comparability graphs, which come from partial orders of dimension at most four.

Not all graph families have local structures. For some families, a simple counting argument proves that adjacency labeling schemes do not exist: only $O(n \log n)$ bits may be used to represent an entire graph, so a representation of this type can only exist when the number of n -vertex graphs in the given family F is at most $2^{O(n \log n)}$. Graph families that have larger numbers of graphs than this, such as the bipartite graphs or the triangle-free graphs, do not have adjacency labeling schemes. However, even families of graphs in which the number of graphs in the family is small might not have an adjacency labeling scheme; for instance, the family of graphs with fewer edges than vertices has $2^{O(n \log n)}$ n -vertex graphs but does not have an adjacency labeling scheme, because one could transform any given graph into a larger graph in this family by adding a new isolated vertex for each edge, without changing its labelability. Kannan et al. asked whether having a forbidden subgraph characterization and having at most $2^{O(n \log n)}$ n -vertex graphs are together enough to guarantee the existence of an adjacency labeling scheme; this question, which Spinrad restated as a conjecture, remains open.

If a graph family F has an adjacency labeling scheme, then the n -vertex graphs in F may be represented as induced subgraphs of a common universal graph of polynomial size, the graph consisting of all possible vertex identifiers. Conversely, if a universal graph of this type can be constructed, then the identities of its vertices may be used as labels in an adjacency labeling scheme. For this application of implicit graph representations, it is important that the labels use as few bits as possible, because the number of bits in the labels translates directly into the number of vertices in the universal graph. Alstrup and Rauhe showed that any tree has an adjacency labeling scheme with $\log_2 n + O(\log^* n)$ bits per label, from which it follows that any graph with arboricity k has a scheme with $k \log_2 n + O(\log^* n)$ bits per label and a universal graph with $n^k 2^{O(\log^* n)}$ vertices. In particular, planar graphs have arboricity at most three, so they have universal graphs with a nearly-cubic number of vertices. For the family of planar graphs, Gavoille and Labourel showed a labeling scheme with $2\log(n) + O(\log \log(n))$ bits per label.

Evasiveness

The Aanderaa–Karp–Rosenberg conjecture concerns implicit graphs given as a set of labeled vertices with a black-box rule for determining whether any two vertices are adjacent; this differs from an adjacency labeling scheme in that the rule may be specific to a particular graph rather than being a generic rule that applies to all graphs in a family. This difference allows every graph to have an implicit representation: for instance, the rule could be to look up the pair of vertices in a separate adjacency matrix. However, an algorithm that is given as input an implicit graph of this type must operate on it only through the implicit adjacency test, without reference to the implementation of that test.

A *graph property* is the question of whether a graph belongs to a given family of graphs; the answer must remain invariant under any relabeling of the vertices. In this context, the question to be determined is how many pairs of vertices must be tested for adjacency, in the worst case, before the property of interest can be determined to be true or false for a given implicit graph. Rivest and Vuillemin proved that any deterministic algorithm for any nontrivial graph property must test a quadratic number of pairs of vertices; the full Aanderaa–Karp–Rosenberg conjecture is that any deterministic algorithm for a monotonic graph property (one that remains true if more edges are added to a graph with the property) must in some cases test every possible pair of vertices. Several cases of the conjecture have been proven to be true—for instance, it is known to be true for graphs with a prime number of vertices—but the full conjecture remains open. Variants of the problem for randomized algorithms and quantum algorithms have also been studied.

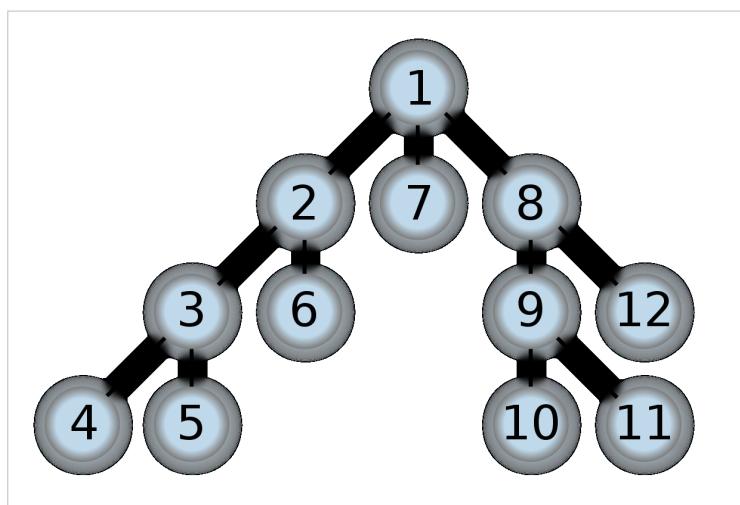
Bender and Ron have shown that, in the same model used for the evasiveness conjecture, it is possible in only constant time to distinguish directed acyclic graphs from graphs that are very far from being acyclic. In contrast, such a fast time is not possible in neighborhood-based implicit graph models,

References

Graph exploration and vertex ordering

Depth-first search

Depth-first search



Order in which the nodes are visited

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(E)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor b searched to depth d
Worst case space complexity	$O(V)$ if entire graph is traversed without repetition, O(longest path length searched) for implicit graphs without elimination of duplicate nodes

Graph and tree search algorithms

- $\alpha-\beta$
- A*
- B*
- Backtracking
- Beam
- Bellman–Ford
- Best-first
- Bidirectional
- Borůvka
- Branch & bound
- BFS
- British Museum
- D*

• DFS	
• Depth-limited	
Dijkstra	
Edmonds	
• Floyd–Warshall	
• Fringe search	
• Hill climbing	
IDA*	
• Iterative deepening	
Johnson	
• Jump point	
• Kruskal	
• Lexicographic BFS	
• Prim	
• SMA*	
• Uniform-cost	
Listings	
• <i>Graph algorithms</i>	
• <i>Search algorithms</i>	
• <i>List of graph algorithms</i>	
Related topics	
• Dynamic programming	
• Graph traversal	
• Tree traversal	
• Search games	
• v	
• t	
• e [1]	

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux^[2] as a strategy for solving mazes.

Properties

The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time $O(|E|)$, linear in the size of the graph. In these applications it also uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the time and space bounds are the same as for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

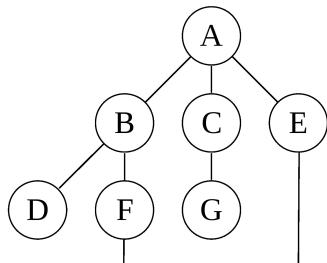
For applications of DFS in relation to specific domains, such as searching for solutions in artificial intelligence or web-crawling, the graph to be traversed is often either too large to visit in its entirety or infinite (DFS may suffer from non-termination). In such cases, search is only performed to a limited depth; due to limited resources, such as memory or disk space, one typically does not use data structures to keep track of the set of all previously visited vertices. When search is performed to a limited depth, the time is still linear in terms of the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be

searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, and as a result, is much smaller than the space needed for searching to the same depth using breadth-first search. For such applications, DFS also lends itself much better to heuristic methods for choosing a likely-looking branch. When an appropriate depth limit is not known *a priori*, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits. In the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

DFS may also be used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree.

Example

For the following graph:



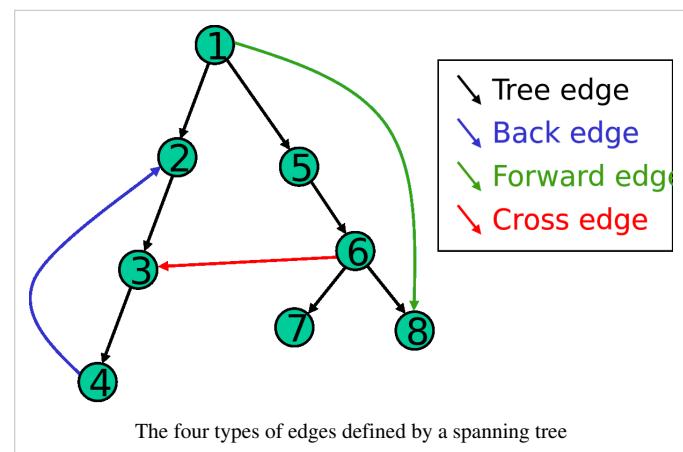
a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening is one technique to avoid this infinite loop and would reach all nodes.

Output of a depth-first search

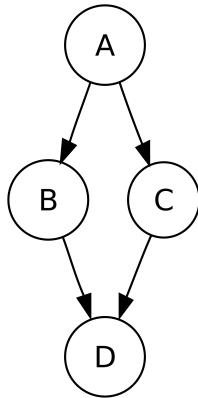
A convenient description of a depth first search of a graph is in terms of a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: **forward edges**, which point from a node of the tree to one of its descendants, **back edges**, which point from a node to one of its ancestors, and **cross edges**, which do neither. Sometimes **tree edges**, edges which belong to the spanning tree itself, are classified separately from forward edges. If the original graph is undirected then all of its edges are tree edges or back edges.



Vertex orderings

It is also possible to use the depth-first search to linearly order the vertices of the original graph (or tree). There are three common ways of doing this:

- A **preordering** is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation.
- A **postordering** is a list of the vertices in the order that they were *last* visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.
- A **reverse postordering** is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. Reverse postordering is not the same as preordering. For example, when searching the directed graph in pre-order



beginning at node A, one visits the nodes in sequence, to produce lists either A B D B A C A, or A C D C A B A (depending upon whether the algorithm chooses to visit B or C first). Note that repeat visits in the form of backtracking to a node, to check if it has still unvisited neighbours, are included here (even if it is found to have none). Thus the possible preorderings are A B D C and A C D B (order by node's leftmost occurrence in above list), while the possible reverse postorderings are A C B D and A B C D (order by node's rightmost occurrence in above list). Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flows. The graph above might represent the flow of control in a code fragment like

```

if (A) then {
    B
} else {
    C
}
D
  
```

and it is natural to consider this code in the order A B C D or A C B D, but not natural to use the order A B D C or A C D B.

Pseudocode

Input: A graph G and a vertex v of G

Output: All vertices reachable from v labeled as discovered

A recursive implementation of DFS:^[3]

```

1  procedure DFS( $G, v$ ) :
2      label  $v$  as discovered
3      for all edges from  $v$  to  $w$  in  $G.\text{adjacentEdges}(v)$  do
4          if vertex  $w$  is not labeled as discovered then
5              recursively call DFS( $G, w$ )

```

A non-recursive implementation of DFS:^[4]

```

1  procedure DFS-iterative( $G, v$ ) :
2      let  $S$  be a stack
3       $S.\text{push}(v)$ 
4      while  $S$  is not empty
5           $v \leftarrow S.\text{pop}()$ 
6          if  $v$  is not labeled as discovered:
7              label  $v$  as discovered
8              for all edges from  $v$  to  $w$  in  $G.\text{adjacentEdges}(v)$  do
9                   $S.\text{push}(w)$ 

```

These two variations of DFS visit the neighbors of each vertex in the opposite order from each other: the first neighbor of v visited by the recursive variation is the first one in the list of adjacent edges, while in the iterative variation the first visited neighbor is the last one in the list of adjacent edges. The non-recursive implementation is similar to breadth-first search but differs from it in two ways: it uses a stack instead of a queue, and it delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before pushing the vertex.

Applications

Algorithms that use depth-first search as a building block include:

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding 3-(edge or vertex)-connected components.
- Finding the bridges of a graph.
- Generating words in order to plot the Limit Set of a Group.
- Finding strongly connected components.
- Planarity testing
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)



Randomized algorithm similar to depth-first search used in generating a maze.

- Maze generation may use a randomized depth-first search.
- Finding biconnectivity in graphs.

Notes

- [1] http://en.wikipedia.org/w/index.php?title=Template:Graph_search_algorithm&action=edit
- [2] Charles Pierre Trémaux (1859–1882) École Polytechnique of Paris (X:1876), French engineer of the telegraph
in Public conference, December 2, 2010 – by professor Jean Pelletier-Thibert in Académie de Macon (Burgundy – France) – (Abstract published in the Annals academic, March 2011 – ISSN: 0980-6032)
- [3] Goodrich and Tamassia; Cormen, Leiserson, Rivest, and Stein
- [4] Kleinberg and Tardos

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.
- Goodrich, Michael T.; Tamassia, Roberto (2001), *Algorithm Design: Foundations, Analysis, and Internet Examples*, Wiley, ISBN 0-471-38365-1
- Kleinberg, Jon; Tardos, Éva (2006), *Algorithm Design*, Addison Wesley, pp. 92–94
- Knuth, Donald E. (1997), *The Art Of Computer Programming Vol 1. 3rd ed* (<http://www-cs-faculty.stanford.edu/~knuth/taocp.html>), Boston: Addison-Wesley, ISBN 0-201-89683-4, OCLC 155842391 (<http://www.worldcat.org/oclc/155842391>)

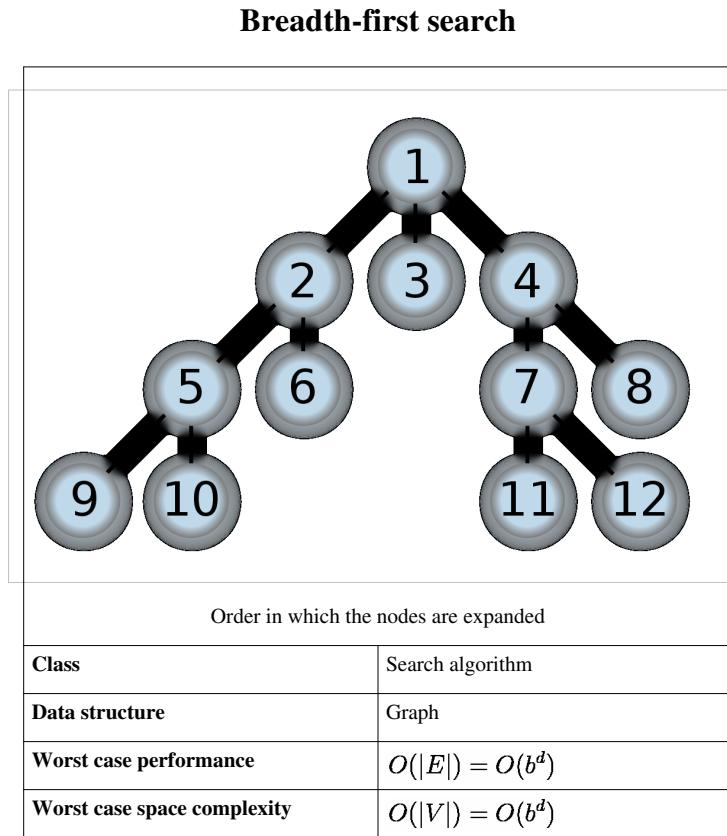
External links



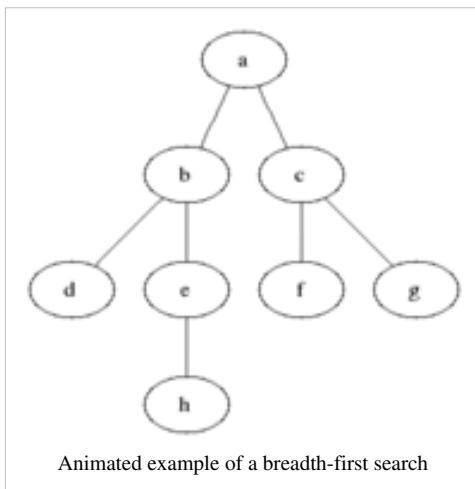
Wikimedia Commons has media related to **Depth-first search**.

- Open Data Structures - Section 12.3.2 - Depth-First-Search (http://opendatastructures.org/versions/edition-0.1e/ods-java/12_3_Graph_Traversal.html#SECTION0015320000000000000000)
- Depth-First Explanation and Example (<http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch14.html>)
- C++ Boost Graph Library: Depth-First Search (http://www.boost.org/libs/graph/doc/depth_first_search.html)
- Depth-First Search Animation (for a directed graph) (<http://www.cs.duke.edu/csed/jawaa/DFSanim.html>)
- Depth First and Breadth First Search: Explanation and Code (http://www.kirupa.com/developer/actionscript/depth_breadth_search.htm)
- QuickGraph ([http://quickgraph.codeplex.com/Wiki/View.aspx?title=Depth First Search Example](http://quickgraph.codeplex.com/Wiki/View.aspx?title=Depth%20First%20Search%20Example)), depth first search example for .Net
- Depth-first search algorithm illustrated explanation (Java and C++ implementations) (http://www.algoist.net/Algorithms/Graph_algorithms/Undirected/Depth-first_search)
- YAGSBPL – A template-based C++ library for graph search and planning (<http://code.google.com/p/yagsbpl/>)

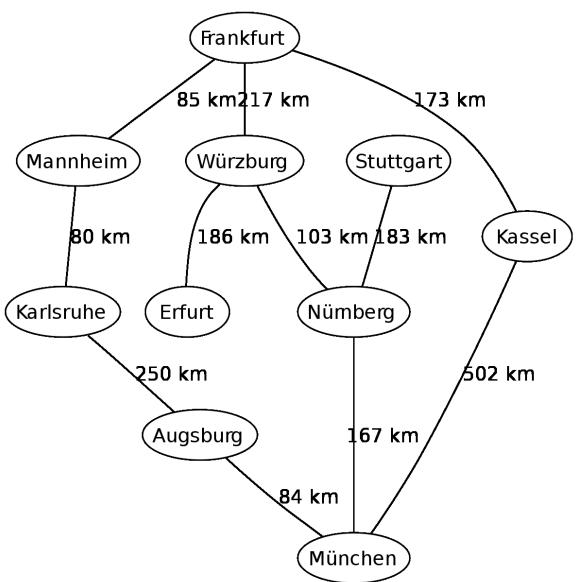
Breadth-first search



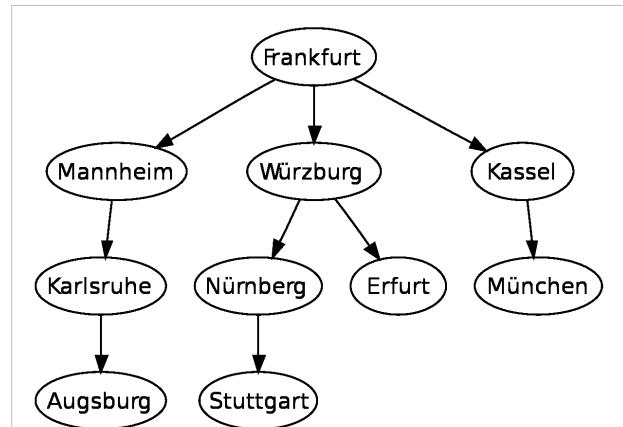
In graph theory, **breadth-first search (BFS)** is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. Compare BFS with the equivalent, but more memory-efficient Iterative deepening depth-first search and contrast with depth-first search.



Algorithm



An example map of Germany with some connections between cities



The breadth-first tree obtained when running BFS on the given map and starting in Frankfurt

Graph and tree search algorithms

- $\alpha-\beta$
- A*
- B*
- Backtracking
- Beam
- Bellman–Ford
- Best-first
- Bidirectional
- Borůvka
- Branch & bound
- BFS
- British Museum
- D*
- DFS
- Depth-limited
- Dijkstra
- Edmonds
- Floyd–Warshall
- Fringe search
- Hill climbing
- IDA*
- Iterative deepening
- Johnson
- Jump point
- Kruskal
- Lexicographic BFS

• Prim
• SMA*
• Uniform-cost
Listings
• <i>Graph algorithms</i>
• <i>Search algorithms</i>
• <i>List of graph algorithms</i>
Related topics
• Dynamic programming
• Graph traversal
• Tree traversal
• Search games
• v
• t
• e [1]

The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node and examine it
 - If the element sought is found in this node, quit the search and return a result.
 - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

Pseudocode

Input: A graph G and a root v of G

```

1  procedure BFS( $G, v$ ) is
2      create a queue  $Q$ 
3      create a set  $V$ 
4      add  $v$  to  $V$ 
5      enqueue  $v$  onto  $Q$ 
6      while  $Q$  is not empty loop
7           $t \leftarrow Q.\text{dequeue}()$ 
8          if  $t$  is what we are looking for then
9              return  $t$ 
10         end if
11         for all edges  $e$  in  $G.\text{adjacentEdges}(t)$  loop
12              $u \leftarrow G.\text{adjacentVertex}(t, e)$ 
13             if  $u$  is not in  $V$  then
14                 add  $u$  to  $V$ 
15                 enqueue  $u$  onto  $Q$ 
16             end if
17         end loop
18     end loop
19     return none
20 end BFS

```

Features

Space complexity

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$ where $|V|$ is the cardinality of the set of vertices. If the graph is represented by an Adjacency list it occupies $\Theta(|V| + |E|)$ ^[1] space in memory, while an Adjacency matrix representation occupies $\Theta(|V|^2)$.^[2]

Time complexity

The time complexity can be expressed as $O(|V| + |E|)$ ^[3] since every vertex and every edge will be explored in the worst case. Note: $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$, depending on how sparse the input graph is.

Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

- Finding all nodes within one connected component
- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes u and v (with path length measured by number of edges)
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.
- Construction of the *failure function* of the Aho-Corasick pattern matcher.

Finding connected components

The set of nodes reached by a BFS (breadth-first search) form the connected component containing the starting node.

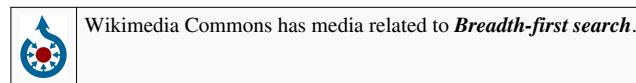
Testing bipartiteness

BFS can be used to test bipartiteness, by starting the search at any vertex and giving alternating labels to the vertices visited during the search. That is, give label 0 to the starting vertex, 1 to all its neighbours, 0 to those neighbours' neighbours, and so on. If at any step a vertex has (visited) neighbours with the same label as itself, then the graph is not bipartite. If the search ends without such a situation occurring, then the graph is bipartite.

References

- [1] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. p.590
- [2] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. p.591
- [3] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. p.597
- Knuth, Donald E. (1997), *The Art Of Computer Programming Vol 1. 3rd ed.* (<http://www-cs-faculty.stanford.edu/~knuth/taocp.html>), Boston: Addison-Wesley, ISBN 0-201-89683-4

External links



- Breadth-First Explanation and Example (<http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch14.html#QQ1-46-92>)
- Open Data Structures - Section 12.3.1 - Breadth-First Search (http://opendatastructures.org/versions/edition-0.1e/ods-java/12_3_Graph_Traversal.html#SECTION0015310000000000000000)

Lexicographic breadth-first search

Graph and tree search algorithms	
•	$\alpha-\beta$
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall
•	Fringe search
•	Hill climbing
•	IDA*
•	Iterative deepening
•	Johnson
•	Jump point
•	Kruskal
•	Lexicographic BFS

• Prim
• SMA*
• Uniform-cost
Listings
• <i>Graph algorithms</i>
• <i>Search algorithms</i>
• <i>List of graph algorithms</i>
Related topics
• Dynamic programming
• Graph traversal
• Tree traversal
• Search games
• v
• t
• e [1]

In computer science, **lexicographic breadth-first search** or Lex-BFS is a linear time algorithm for ordering the vertices of a graph. The algorithm is different from breadth first search, but it produces an ordering that is consistent with breadth-first search.

The lexicographic breadth-first search algorithm is based on the idea of partition refinement and was first developed by Donald J. Rose, Robert E. Tarjan, and George S. Lueker (1976). A more detailed survey of the topic is presented by Corneil (2004). It has been used as a subroutine in other graph algorithms including the recognition of chordal graphs, and optimal coloring of distance-hereditary graphs.

Algorithm

The lexicographic breadth-first search algorithm replaces the queue of vertices of a standard breadth-first search with an ordered sequence of sets of vertices. The sets in the sequence form a partition of the remaining vertices. At each step, a vertex v from the first set in the sequence is removed from that set, and if that removal causes the set to become empty then the set is removed from the sequence. Then, each set in the sequence is replaced by two subsets: the neighbors of v and the non-neighbors of v . The subset of neighbors is placed earlier in the sequence than the subset of non-neighbors. In pseudocode, the algorithm can be expressed as follows:

- Initialize a sequence Σ of sets, to contain a single set containing all vertices.
- Initialize the output sequence of vertices to be empty.
- While Σ is non-empty:
 - Find and remove a vertex v from the first set in Σ
 - If the first set in Σ is now empty, remove it from Σ
 - Add v to the end of the output sequence.
 - For each edge $v-w$ such that w still belongs to a set S in Σ :
 - If the set S containing w has not yet been replaced while processing v , create a new empty replacement set T and place it prior to S in the sequence; otherwise, let T be the set prior to S .
 - Move w from S to T , and if this causes S to become empty remove S from Σ .

Each vertex is processed once, each edge is examined only when its two endpoints are processed, and (with an appropriate representation for the sets in Σ that allows items to be moved from one set to another in constant time) each iteration of the inner loop takes only constant time. Therefore, like simpler graph search algorithms such as breadth-first search and depth first search, this algorithm takes linear time.

The algorithm is called lexicographic breadth-first search because the order it produces is an ordering that could also have been produced by a breadth-first search, and because if the ordering is used to index the rows and columns of an adjacency matrix of a graph then the algorithm sorts the rows and columns into Lexicographical order.

Applications

Chordal graphs

A graph G is defined to be chordal if its vertices have a *perfect elimination ordering*, an ordering such that for any vertex v the neighbors that occur later in the ordering form a clique. In a chordal graph, the reverse of a lexicographic ordering is always a perfect elimination ordering. Therefore, one can test whether a graph is chordal in linear time by the following algorithm:

- Use lexicographic breadth-first search to find a lexicographic ordering of G
- Reverse this ordering
- For each vertex v :
 - Let w be the neighbor of v occurring prior to v in the reversed sequence, as close to v in the sequence as possible
 - (Continue to the next vertex v if there is no such w)
 - If the set of earlier neighbors of v (excluding w itself) is not a subset of the set of earlier neighbors of w , the graph is not chordal
- If the loop terminates without showing that the graph is not chordal, then it is chordal.

This application was the original motivation that led Rose, Tarjan & Lueker (1976) to develop the lexicographic breadth first search algorithm.^[1]

Graph coloring

A graph G is said to be *perfectly orderable* if there is a sequence of its vertices with the property that, for any induced subgraph of G , a greedy coloring algorithm that colors the vertices in the induced sequence ordering is guaranteed to produce an optimal coloring.

For a chordal graph, a perfect elimination ordering is a perfect ordering: the number of the color used for any vertex is the size of the clique formed by it and its earlier neighbors, so the maximum number of colors used is equal to the size of the largest clique in the graph, and no coloring can use fewer colors. An induced subgraph of a chordal graph is chordal and the induced subsequence of its perfect elimination ordering is a perfect elimination ordering on the subgraph, so chordal graphs are perfectly orderable, and lexicographic breadth-first search can be used to optimally color them.

The same property is true for a larger class of graphs, the distance-hereditary graphs: distance-hereditary graphs are perfectly orderable, with a perfect ordering given by the reverse of a lexicographic ordering, so lexicographic breadth-first search can be used in conjunction with greedy coloring algorithms to color them optimally in linear time.^[2]

Other applications

Bretscher et al. (2008) describe an extension of lexicographic breadth-first search that breaks any additional ties using the complement graph of the input graph. As they show, this can be used to recognize cographs in linear time. Habib et al. (2000) describe additional applications of lexicographic breadth-first search including the recognition of comparability graphs and interval graphs.

Notes

- [1] Corneil (2004).
- [2] , Theorem 5.2.4, p. 71.

References

- Brandstädt, Andreas; Le, Van Bang; Spinrad, Jeremy (1999), *Graph Classes: A Survey*, SIAM Monographs on Discrete Mathematics and Applications, ISBN 0-89871-432-X.
- Bretscher, Anna; Corneil, Derek; Habib, Michel; Paul, Christophe (2008), "A simple linear time LexBFS cograph recognition algorithm" (<http://www.liafa.jussieu.fr/~habib/Documents/cograph.ps>), *SIAM Journal on Discrete Mathematics* **22** (4): 1277–1296, doi: 10.1137/060664690 (<http://dx.doi.org/10.1137/060664690>).
- Corneil, Derek G. (2004), "Lexicographic breadth first search – a survey", *Graph-Theoretic Methods in Computer Science: 30th International Workshop, WG 2004, Bad Honnef, Germany, June 21-23, 2004, Revised Papers*, Lecture Notes in Computer Science **3353**, Springer-Verlag, pp. 1–19, doi: 10.1007/978-3-540-30559-0_1 (http://dx.doi.org/10.1007/978-3-540-30559-0_1).
- Habib, Michel; McConnell, Ross; Paul, Christophe; Viennot, Laurent (2000), "Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing" (<http://www.cecm.sfu.ca/~cchauve/MATH445/PROJECTS/MATH445-TCS-234-59.pdf>), *Theoretical Computer Science* **234** (1–2): 59–84, doi: 10.1016/S0304-3975(97)00241-7 ([http://dx.doi.org/10.1016/S0304-3975\(97\)00241-7](http://dx.doi.org/10.1016/S0304-3975(97)00241-7)).
- Rose, D. J.; Tarjan, R. E.; Lueker, G. S. (1976), "Algorithmic aspects of vertex elimination on graphs", *SIAM Journal on Computing* **5** (2): 266–283, doi: 10.1137/0205021 (<http://dx.doi.org/10.1137/0205021>).

Iterative deepening depth-first search

Graph and tree search algorithms	
•	α - β
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall
•	Fringe search
•	Hill climbing
•	IDA*
•	Iterative deepening
•	Johnson
•	Jump point
•	Kruskal
•	Lexicographic BFS
•	Prim
•	SMA*
•	Uniform-cost
Listings	
•	<i>Graph algorithms</i>
•	<i>Search algorithms</i>
•	<i>List of graph algorithms</i>
Related topics	
•	Dynamic programming
•	Graph traversal
•	Tree traversal
•	Search games
•	v
•	t
•	e ^[1]

Iterative deepening depth-first search (IDDFS) is a state space search strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches d , the depth of the shallowest goal state. IDDFS is equivalent to breadth-first search, but uses much less memory; on each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

Properties

IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness (when the branching factor is finite). It is optimal when the path cost is a non-decreasing function of the depth of the node.

The space complexity of IDDFS is $O(bd)$, where b is the branching factor and d is the depth of shallowest goal.

Since iterative deepening visits states multiple times, it may seem wasteful, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level, so it does not matter much if the upper levels are visited multiple times.

The main advantage of IDDFS in game tree searching is that the earlier searches tend to improve the commonly used heuristics, such as the killer heuristic and alpha-beta pruning, so that a more accurate estimate of the score of various nodes at the final depth search can occur, and the search completes more quickly since it is done in a better order. For example, alpha-beta pruning is most efficient if it searches the best moves first.

A second advantage is the responsiveness of the algorithm. Because early iterations use small values for d , they execute extremely quickly. This allows the algorithm to supply early indications of the result almost immediately, followed by refinements as d increases. When used in an interactive setting, such as in a chess-playing program, this facility allows the program to play at any time with the current best move found in the search it has completed so far. This can be phrased as each depth of the search *corecursively* producing a better approximation of the solution, though the work done at each step is recursive. This is not possible with a traditional depth-first search, which does not produce intermediate results.

The time complexity of IDDFS in well-balanced trees works out to be the same as Depth-first search: $O(b^d)$.

In an iterative deepening search, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree, which is expanded $d + 1$ times. So the total number of expansions in an iterative deepening search is

$$(d)b + (d - 1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d$$

$$\sum_{i=1}^d (d + 1 - i)b^i$$

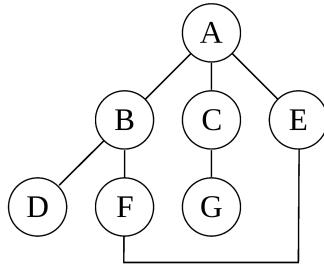
For $b = 10$ and $d = 5$ the number is

$$50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

All together, an iterative deepening search from depth 1 to depth d expands only about 11% more nodes than a single breadth-first or depth-limited search to depth d , when $b = 10$. The higher the branching factor, the lower the overhead of repeatedly expanded states, but even when the branching factor is 2, iterative deepening search only takes about twice as long as a complete breadth-first search. This means that the time complexity of iterative deepening is still $O(b^d)$, and the space complexity is $O(d)$ like a regular depth-first search. In general, iterative deepening is the preferred search method when there is a large search space and the depth of the solution is not known.

Example

For the following graph:



a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening prevents this loop and will reach the following nodes on the following depths, assuming it proceeds left-to-right as above:

- 0: A
- 1: A (repeated), B, C, E

(Note that iterative deepening has now seen C, when a conventional depth-first search did not.)

- 2: A, B, D, F, C, G, E, F

(Note that it still sees C, but that it came later. Also note that it sees E via a different path, and loops back to F twice.)

- 3: A, B, D, F, E, C, G, E, F, B

For this graph, as more depth is added, the two cycles "ABFE" and "AEFB" will simply get longer before the algorithm gives up and tries another branch.

Algorithm

The following pseudocode shows IDDFS implemented in terms of a recursive depth-limited DFS (called DLS).

```

IDDFS(root, goal)
{
  for(i=1, i<10, i++)
  {
    DLS(limit=i)
  }
}
  
```

Related algorithms

Similar to iterative deepening is a search strategy called iterative lengthening search that works with increasing path-cost limits instead of depth-limits. It expands nodes in the order of increasing path cost; therefore the first goal it encounters is the one with the cheapest path cost. But iterative lengthening incurs substantial overhead that makes it less useful than iterative deepening.

Notes

Topological sorting

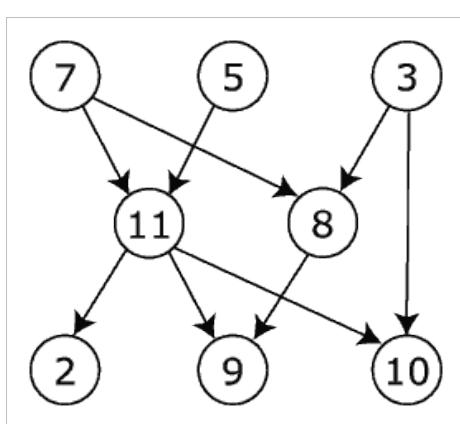
"Dependency resolution" redirects here. For other uses, see Dependency (disambiguation).

In computer science, a **topological sort** (sometimes abbreviated **topsort** or **toposort**) or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v , u comes before v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Examples

The canonical application of topological sorting (topological order) is in scheduling a sequence of jobs or tasks based on their dependencies; topological sorting algorithms were first studied in the early 1960s in the context of the PERT technique for scheduling in project management (Jarnagin 1960). The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started (for example, when washing clothes, the washing machine must finish before we put the clothes to dry). Then, a topological sort gives an order in which to perform the jobs.

In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers. It is also used to decide in which order to load tables with foreign keys in databases.



The graph shown to the left has many valid topological sorts, including:

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Algorithms

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, asymptotically, $O(|V| + |E|)$.

One of these algorithms, first described by Kahn (1962), works by choosing vertices in the same order as the eventual topological sort. First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist in an acyclic graph. Then:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

If the graph is a DAG, a solution will be contained in the list L (the solution is not necessarily unique). Otherwise, the graph must have at least one cycle and therefore a topological sorting is impossible.

Reflecting the non-uniqueness of the resulting sort, the structure S can be simply a set or a queue or a stack. Depending on the order that nodes n are removed from set S, a different solution is created. A variation of Kahn's algorithm that breaks ties lexicographically forms a key component of the Coffman–Graham algorithm for parallel scheduling and layered graph drawing.

An alternative algorithm for topological sorting is based on depth-first search. The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort:

```
L ← Empty list that will contain the sorted nodes
while there are unmarked nodes do
    select an unmarked node n
    visit(n)
function visit(node n)
    if n has a temporary mark then stop (not a DAG)
    if n is not marked (i.e. has not been visited yet) then
        mark n temporarily
        for each node m with an edge from n to m do
            visit(m)
        mark n permanently
        unmark n temporarily
        add n to head of L
```

Each node n gets *prepended* to the output list L only after considering all other nodes on which n depends (all ancestral nodes of n in the graph). Specifically, when the algorithm adds node n , we are guaranteed that all nodes on which n depends are already in the output list L: they were added to L either by the preceding recursive call to visit(),

or by an earlier call to `visit()`. Since each edge and node is visited once, the algorithm runs in linear time. This depth-first-search-based algorithm is the one described by Cormen et al. (2001); it seems to have been first described in print by Tarjan (1976).

Complexity

The computational complexity of the problem of computing a topological ordering of a directed acyclic graph is NC^2 ; that is, it can be computed in $O(\log^2 n)$ time on a parallel computer using a polynomial number $O(n^k)$ of processors, for some constant k (Cook 1985). One method for doing this is to repeatedly square the adjacency matrix of the given graph, logarithmically many times, using min-plus matrix multiplication with maximization in place of minimization. The resulting matrix describes the longest path distances in the graph. Sorting the vertices by the lengths of their longest incoming paths produces a topological ordering (Dekel, Nassimi & Sahni 1981).

Uniqueness

If a topological sort has the property that all pairs of consecutive vertices in the sorted order are connected by edges, then these edges form a directed Hamiltonian path in the DAG. If a Hamiltonian path exists, the topological sort order is unique; no other order respects the edges of the path. Conversely, if a topological sort does not form a Hamiltonian path, the DAG will have two or more valid topological orderings, for in this case it is always possible to form a second valid ordering by swapping two consecutive vertices that are not connected by an edge to each other. Therefore, it is possible to test in linear time whether a unique ordering exists, and whether a Hamiltonian path exists, despite the NP-hardness of the Hamiltonian path problem for more general directed graphs (Vernet & Markenzon 1997).

Relation to partial orders

Topological orderings are also closely related to the concept of a linear extension of a partial order in mathematics.

A partially ordered set is just a set of objects together with a definition of the " \leq " inequality relation, satisfying the axioms of reflexivity ($x \leq x$), antisymmetry (if $x \leq y$ and $y \leq x$ then $x = y$) and transitivity (if $x \leq y$ and $y \leq z$, then $x \leq z$). A total order is a partial order in which, for every two objects x and y in the set, either $x \leq y$ or $y \leq x$. Total orders are familiar in computer science as the comparison operators needed to perform comparison sorting algorithms. For finite sets, total orders may be identified with linear sequences of objects, where the " \leq " relation is true whenever the first object precedes the second object in the order; a comparison sorting algorithm may be used to convert a total order into a sequence in this way. A linear extension of a partial order is a total order that is compatible with it, in the sense that, if $x \leq y$ in the partial order, then $x \leq y$ in the total order as well.

One can define a partial ordering from any DAG by letting the set of objects be the vertices of the DAG, and defining $x \leq y$ to be true, for any two vertices x and y , whenever there exists a directed path from x to y ; that is, whenever y is reachable from x . With these definitions, a topological ordering of the DAG is the same thing as a linear extension of this partial order. Conversely, any partial ordering may be defined as the reachability relation in a DAG. One way of doing this is to define a DAG that has a vertex for every object in the partially ordered set, and an edge xy for every pair of objects for which $x \leq y$. An alternative way of doing this is to use the transitive reduction of the partial ordering; in general, this produces DAGs with fewer edges, but the reachability relation in these DAGs is still the same partial order. By using these constructions, one can use topological ordering algorithms to find linear extensions of partial orders.

References

- Cook, Stephen A. (1985), "A Taxonomy of Problems with Fast Parallel Algorithms", *Information and Control* **64** (1–3): 2–22, doi:10.1016/S0019-9958(85)80041-3 ^[1].
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "Section 22.4: Topological sort", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 549–552, ISBN 0-262-03293-7.
- Dekel, Eliezer; Nassimi, David; Sahni, Sartaj (1981), "Parallel matrix and graph algorithms", *SIAM Journal on Computing* **10** (4): 657–675, doi:10.1137/0210049 ^[2], MR 635424 ^[3].
- Jarnagin, M. P. (1960), *Automatic machine methods of testing PERT networks for consistency*, Technical Memorandum No. K-24/60, Dahlgren, Virginia: U. S. Naval Weapons Laboratory.
- Kahn, Arthur B. (1962), "Topological sorting of large networks", *Communications of the ACM* **5** (11): 558–562, doi:10.1145/368996.369025 ^[4].
- Tarjan, Robert E. (1976), "Edge-disjoint spanning trees and depth-first search", *Acta Informatica* **6** (2): 171–185, doi:10.1007/BF00268499 ^[5].
- Vernet, Oswaldo; Markenzon, Lilian (1997), "Hamiltonian problems for reducible flowgraphs", *Proc. 17th International Conference of the Chilean Computer Science Society (SCCC '97)*, pp. 264–267, doi:10.1109/SCCC.1997.637099 ^[6].

External links

- NIST Dictionary of Algorithms and Data Structures: topological sort ^[7]
- Weisstein, Eric W., "TopologicalSort" ^[8], *MathWorld*.

References

- [1] <http://dx.doi.org/10.1016%2FS0019-9958%2885%2980041-3>
- [2] <http://dx.doi.org/10.1137%2F0210049>
- [3] <http://www.ams.org/mathscinet-getitem?mr=635424>
- [4] <http://dx.doi.org/10.1145%2F368996.369025>
- [5] <http://dx.doi.org/10.1007%2FBF00268499>
- [6] <http://dx.doi.org/10.1109%2FSCCC.1997.637099>
- [7] <http://www.nist.gov/dads/HTML/topologicalSort.html>
- [8] <http://mathworld.wolfram.com/TopologicalSort.html>

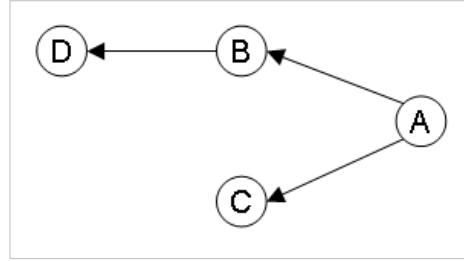
Application: Dependency graphs

In mathematics, computer science and digital electronics, a **dependency graph** is a directed graph representing dependencies of several objects towards each other. It is possible to derive an evaluation order or the absence of an evaluation order that respects the given dependencies from the dependency graph.

Definition

Given a set of objects S and a transitive relation $R \subseteq S \times S$ with $(a, b) \in R$ modeling a dependency "a needs b evaluated first", the dependency graph is a graph $G = (S, T)$ with $T \subseteq R$ and R being the transitive closure of T .

For example, assume a simple calculator. This calculator supports assignment of constant values to variables and assigning the sum of exactly 2 variables to a third variable. Given several equations like " $A = B+C; B = 5+D; C=4; D=2;$ ", then $S = A, B, C, D$ and $R = (A, B), (A, C), (B, D)$. You can derive this relation directly: A depends on B and C , because you can add two variables if and only if you know the values of both variables. Thus, B and C must be calculated before A can be calculated. However, D 's value is known immediately, because it is a number literal.



Recognizing impossible evaluations

In a dependency graph, the cycles of dependencies (also called **circular dependencies**) lead to a situation in which no valid evaluation order exists, because none of the objects in the cycle may be evaluated first. If a dependency graph does not have any circular dependencies, it forms a directed acyclic graph, and an evaluation order may be found by topological sorting. Most topological sorting algorithms are also capable of detecting cycles in their inputs, however, it may be desirable to perform cycle detection separately from topological sorting in order to provide appropriate handling for the detected cycles.

Assume the simple calculator from before. The equation system " $A=B; B=D+C; C=D+A; D=12;$ " contains a circular dependency formed by A , B and C , as B must be evaluated before A , C must be evaluated before B and A must be evaluated before C .

Deriving an evaluation order

A **correct evaluation order** is a numbering $n : S \rightarrow \mathbb{N}$ of the objects that form the nodes of the dependency graph so that the following equation holds: $n(a) < n(b) \Rightarrow (a, b) \notin R$ with $a, b \in S$. This means, if the numbering orders two elements a and b so that a will be evaluated before b , then a must not depend on b . Furthermore, there can be more than a single correct evaluation order. In fact, a correct numbering is a topological order, and any topological order is a correct numbering. Thus, any algorithm that derives a correct topological order derives a correct evaluation order.

Assume the simple calculator from above once more. Given the equation system " $A = B+C; B = 5+D; C=4; D=2;$ ", a correct evaluation order would be (D, C, B, A) . However, (C, D, B, A) is a correct evaluation order as well.

Examples

Dependency graphs are used in:

- Automated software installers. They walk the graph looking for software packages that are required but not yet installed. The dependency is given by the coupling of the packages.
- Software build scripts such as Unix Make, Node npm install, Twitter bower install, or Apache Ant. They need to know what files have changed so only the correct files need to be recompiled.
- In Compiler technology and formal language implementation:
 - Instruction Scheduling. Dependency graphs are computed for the operands of assembly or intermediate instructions and used to determine an optimal order for the instructions.
 - Dead code elimination. If no side effected operation depends on a variable, this variable is considered dead and can be removed.
- Spreadsheet calculators. They need to derive a correct calculation order similar to that one in the example used in this article.
- Web Forms standards such as XForms to know what visual elements to update if data in the model changes.

Dependency graphs are one aspect of:

- Manufacturing Plant Types. Raw materials are processed into products via several dependent stages.
- Job Shop Scheduling. A collection of related theoretical problems in computer science.

References

- Balmas, Francoise (2001) *Displaying dependence graphs: a hierarchical approach* [1], [2] were, p. 261, Eighth Working Conference on Reverse Engineering (WCRE'01)

References

- [1] <http://www.ai.univ-paris8.fr/~fb/version-ps/pdep.ps>
[2] <http://doi.ieeecomputersociety.org/10.1109/WCRE.2001.957830>

Connectivity of undirected graphs

Connected components

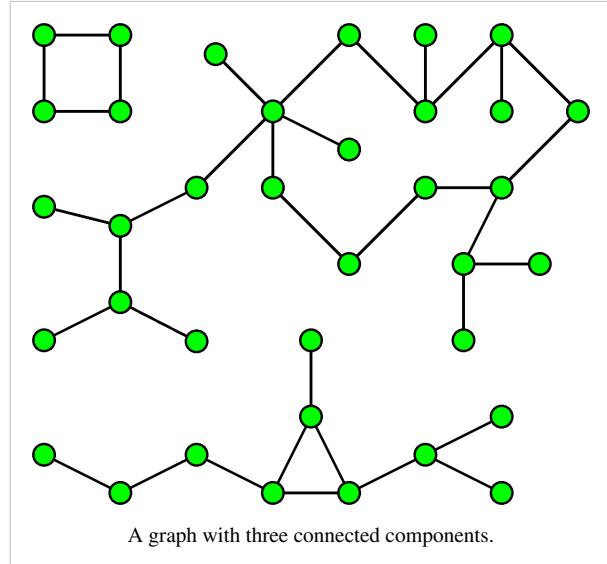
In graph theory, a **connected component** (or just **component**) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. For example, the graph shown in the illustration on the right has three connected components. A graph that is itself connected has exactly one connected component, consisting of the whole graph.

An equivalence relation

An alternative way to define connected components involves the equivalence classes of an equivalence relation that is defined on the vertices of the graph. In an undirected graph, a vertex v is *reachable* from a vertex u if there is a path from u to v . In this definition, a single vertex is counted as a path of length zero, and the same vertex may occur more than once within a path. Reachability is an equivalence relation, since:

- It is reflexive: There is a trivial path of length zero from any vertex to itself.
- It is symmetric: If there is a path from u to v , the same edges form a path from v to u .
- It is transitive: If there is a path from u to v and a path from v to w , the two paths may be concatenated together to form a path from u to w .

The connected components are then the induced subgraphs formed by the equivalence classes of this relation.



The number of connected components

The number of connected components is an important topological invariant of a graph. In topological graph theory it can be interpreted as the zeroth Betti number of the graph. In algebraic graph theory it equals the multiplicity of 0 as an eigenvalue of the Laplacian matrix of the graph. It is also the index of the first nonzero coefficient of the chromatic polynomial of a graph. Numbers of connected components play a key role in the Tutte theorem characterizing graphs that have perfect matchings, and in the definition of graph toughness.

Algorithms

It is straightforward to compute the connected components of a graph in linear time (in terms of the numbers of the vertices and edges of the graph) using either breadth-first search or depth-first search. In either case, a search that begins at some particular vertex v will find the entire connected component containing v (and no more) before returning. To find all the connected components of a graph, loop through its vertices, starting a new breadth first or depth first search whenever the loop reaches a vertex that has not already been included in a previously found connected component. Hopcroft and Tarjan (1973) describe essentially this algorithm, and state that at that point it was "well known".

There are also efficient algorithms to dynamically track the connected components of a graph as vertices and edges are added, as a straightforward application of disjoint-set data structures. These algorithms require amortized $O(\alpha(n))$ time per operation, where adding vertices and edges and determining the connected component in which a vertex falls are both operations, and $\alpha(n)$ is a very slow-growing inverse of the very quickly growing Ackermann function. A related problem is tracking connected components as all edges are deleted from a graph, one by one; an algorithm exists to solve this with constant time per query, and $O(|V||E|)$ time to maintain the data structure; this is an amortized cost of $O(|V|)$ per edge deletion. For forests, the cost can be reduced to $O(q + |V| \log |V|)$, or $O(\log |V|)$ amortized cost per edge deletion.^[1]

Researchers have also studied algorithms for finding connected components in more limited models of computation, such as programs in which the working memory is limited to a logarithmic number of bits (defined by the complexity class L). Lewis & Papadimitriou (1982) asked whether it is possible to test in logspace whether two vertices belong to the same connected component of an undirected graph, and defined a complexity class SL of problems logspace-equivalent to connectivity. Finally Reingold (2008) succeeded in finding an algorithm for solving this connectivity problem in logarithmic space, showing that $L = SL$.

References

- [1] Shiloach, Y. and Even, S. 1981. An On-Line Edge-Deletion Problem. *Journal of the ACM*: 28, 1 (Jan. 1981), pp.1-4.
- Lewis, Harry R.; Papadimitriou, Christos H. (1982), "Symmetric space-bounded computation", *Theoretical Computer Science* **19** (2): 161–187, doi: 10.1016/0304-3975(82)90058-5 ([http://dx.doi.org/10.1016/0304-3975\(82\)90058-5](http://dx.doi.org/10.1016/0304-3975(82)90058-5)).
- Reingold, Omer (2008), "Undirected connectivity in log-space", *Journal of the ACM* **55** (4): Article 17, 24 pages, doi: 10.1145/1391289.1391291 (<http://dx.doi.org/10.1145/1391289.1391291>).

External links

- MATLAB code to find connected components in undirected graphs (<http://www.mathworks.com/matlabcentral/fileexchange/42040-find-network-components>), MATLAB File Exchange.
- Connected components (<http://www.cs.sunysb.edu/~algorith/files/dfs-bfs.shtml>), Steven Skiena, The Stony Brook Algorithm Repository

Edge connectivity

In graph theory, a graph is **k -edge-connected** if it remains connected whenever fewer than k edges are removed.

The **edge-connectivity** of a graph is the largest k for which the graph is k -edge-connected.

Formal definition

Let $G = (V, E)$ be an arbitrary graph. If subgraph $G' = (V, E \setminus X)$ is connected for all $X \subseteq E$ where $|X| < k$, then G is k -edge-connected.

Relation to minimum vertex degree

Minimum vertex degree gives a trivial upper bound on edge-connectivity. That is, if a graph $G = (V, E)$ is k -edge-connected then it is necessary that $k \leq \delta(G)$, where $\delta(G)$ is the minimum degree of any vertex $v \in V$. Obviously, deleting all edges incident to a vertex, v , would then disconnect v from the graph.

Computational aspects

There is a polynomial-time algorithm to determine the largest k for which a graph G is k -edge-connected. A simple algorithm would, for every pair (u, v) , determine the maximum flow from u to v with the capacity of all edges in G set to 1 for both directions. A graph is k -edge-connected if and only if the maximum flow from u to v is at least k for any pair (u, v) , so k is the least u - v -flow among all (u, v) .

If n is the number of vertices in the graph, this simple algorithm would perform $O(n^2)$ iterations of the Maximum flow problem, which can be solved in $O(n^3)$ time. Hence the complexity of the simple algorithm described above is $O(n^5)$ in total.

An improved algorithm will solve the maximum flow problem for every pair (u, v) where u is arbitrarily fixed while v varies over all vertices. This reduces the complexity to $O(n^4)$ and is sound since, if a cut of capacity less than k exists, it is bound to separate u from some other vertex. It can be further improved by an algorithm of Gabow that runs in worst case $O(n^3)$ time.^[1]

A related problem: finding the minimum k -edge-connected subgraph of G (that is: select as few as possible edges in G that your selection is k -edge-connected) is NP-hard for $k \geq 2$.^[2]

References

- [1] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259–273, 1995.
- [2] M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.

Vertex connectivity

In graph theory, a graph G is said to be **k -vertex-connected** (or **k -connected**) if it has more than k vertices and remains connected whenever fewer than k vertices are removed.

The **vertex-connectivity**, or just **connectivity**, of a graph is the largest k for which the graph is k -vertex-connected.

Definitions

A graph (other than a complete graph) has connectivity k if k is the size of the smallest subset of vertices such that the graph becomes disconnected if you delete them. Complete graphs are not included in this version of the definition since they cannot be disconnected by deleting vertices. The complete graph with n vertices has connectivity $n - 1$, as implied by the first definition.

An equivalent definition is that a graph with at least two vertices is k -connected if, for every pair of its vertices, it is possible to find k vertex-disjoint paths connecting these vertices; see Menger's theorem (Diestel 2005, p. 55). This definition produces the same answer, $n - 1$, for the connectivity of the complete graph K_n .

A 1-connected graph is called connected; a 2-connected graph is called biconnected. A 3-connected graph is called triconnected.

Applications

Polyhedral Combinatorics

The 1-skeleton of any k -dimensional convex polytope forms a k -vertex-connected graph (Balinski's theorem, Balinski 1961). As a partial converse, Steinitz's theorem states that any 3-vertex-connected planar graph forms the skeleton of a convex polyhedron.

Computational complexity

The vertex-connectivity of an input graph G can be computed in polynomial time in the following way^[1]: consider all possible pairs (s, t) of nonadjacent nodes to disconnect, using Menger's theorem to justify that the minimal-size separator for (s, t) is the number of pairwise vertex-independent paths between them, encode the input by doubling each vertex as an edge to reduce to a computation of the number of pairwise edge-independent paths, and compute the maximum number of such paths by computing the maximum flow in the graph between s and t with capacity 1 to each edge, noting that a flow of k in this graph corresponds, by the integral flow theorem, to k pairwise edge-independent paths from s to t .

Notes

- [1] *The algorithm design manual*, p 506, and *Computational discrete mathematics: combinatorics and graph theory with Mathematica*, p. 290-291

References

- Balinski, M. L. (1961), "On the graph structure of convex polyhedra in n -space" (<http://www.projecteuclid.org/Dienst/UI/1.0/Summarize/euclid.pjm/1103037323>), *Pacific Journal of Mathematics* **11** (2): 431–434, doi: 10.2140/pjm.1961.11.431 (<http://dx.doi.org/10.2140/pjm.1961.11.431>).
- Diestel, Reinhard (2005), *Graph Theory* (<http://www.math.uni-hamburg.de/home/diestel/books/graph-theory/>) (3rd ed.), Berlin, New York: Springer-Verlag, ISBN 978-3-540-26183-4.

Menger's theorems on edge and vertex connectivity

In the mathematical discipline of graph theory and related areas, **Menger's theorem** is a characterization of the connectivity in finite undirected graphs in terms of the minimum number of disjoint paths that can be found between any pair of vertices. It was proved for edge-connectivity and vertex-connectivity by Karl Menger in 1927. The edge-connectivity version of Menger's theorem was later generalized by the max-flow min-cut theorem.

Edge connectivity

The **edge-connectivity** version of Menger's theorem is as follows:

Let G be a finite undirected graph and x and y two distinct vertices. Then the theorem states that the size of the minimum edge cut for x and y (the minimum number of edges whose removal disconnects x and y) is equal to the maximum number of pairwise edge-independent paths from x to y .

Extended to subgraphs: a maximal subgraph disconnected by no less than a k -edge cut is identical to a maximal subgraph with a minimum number k of edge-independent paths between any x, y pairs of nodes in the subgraph.

Vertex connectivity

The **vertex-connectivity** statement of Menger's theorem is as follows:

Let G be a finite undirected graph and x and y two nonadjacent vertices. Then the theorem states that the size of the minimum vertex cut for x and y (the minimum number of vertices whose removal disconnects x and y) is equal to the maximum number of pairwise vertex-independent paths from x to y .

Extended to subgraphs: a maximal subgraph disconnected by no less than a k -vertex cut is identical to a maximal subgraph with a minimum number k of vertex-independent paths between any x, y pairs of nodes in the subgraph.

Infinite graphs

Menger's theorem holds for infinite graphs, and in that context it applies to the minimum cut between any two elements that are either vertices or ends of the graph (Halin 1974). The following result of Ron Aharoni and Eli Berger was originally a conjecture proposed by Paul Erdős, and before being proved was known as the **Erdős–Menger conjecture**. It is equivalent to Menger's theorem when the graph is finite.

Let A and B be sets of vertices in a (possibly infinite) digraph G . Then there exists a family P of disjoint A - B -paths and a separating set which consists of exactly one vertex from each path in P .

References

- Menger, Karl (1927). "Zur allgemeinen Kurventheorie". *Fund. Math.* **10**: 96–115.
- Aharoni, Ron and Berger, Eli (2009). "Menger's Theorem for infinite graphs" ^[1]. *Inventiones Mathematicae* **176**: 1–62. doi:10.1007/s00222-008-0157-3 ^[2].
- Halin, R. (1974), "A note on Menger's theorem for infinite locally finite graphs", *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* **40**: 111–114, doi:10.1007/BF02993589 ^[3], MR 0335355 ^[4].

External links

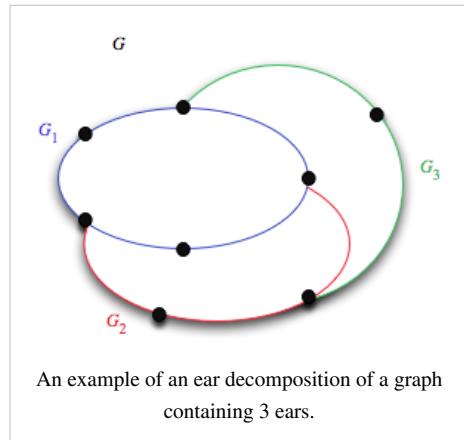
- A Proof of Menger's Theorem ^[5]
- Menger's Theorems and Max-Flow-Min-Cut ^[6]
- Network flow ^[7]
- Max-Flow-Min-Cut ^[8]

References

- [1] <http://www.springerlink.com/content/267k231365284lr6/?p=ddccdd0319b24e53958e286488757ca7&pi=0>
- [2] <http://dx.doi.org/10.1007%2Fs00222-008-0157-3>
- [3] <http://dx.doi.org/10.1007%2FBF02993589>
- [4] <http://www.ams.org/mathscinet-getitem?mr=0335355>
- [5] http://www.math.unm.edu/~loring/links/graph_s05/Menger.pdf
- [6] <http://www.math.fau.edu/locke/Menger.htm>
- [7] <http://gepard.bioinformatik.uni-saarland.de/teaching/ws-2008-09/bioinformatik-3/lectures/V12-NetworkFlow.pdf>
- [8] <http://gepard.bioinformatik.uni-saarland.de/teaching/ws-2008-09/bioinformatik-3/lectures/V13-MaxFlowMinCut.pdf>

Ear decomposition

In graph theory, an **ear** of an undirected graph G is a path P where the two endpoints of the path may coincide, but where otherwise no repetition of edges or vertices is allowed, so every internal vertex of P has degree two in P . An **ear decomposition** of an undirected graph G is a partition of its set of edges into a sequence of ears, such that the one or two endpoints of each ear belong to earlier ears in the sequence and such that the internal vertices of each ear do not belong to any earlier ear. Additionally, in most cases the first ear in the sequence must be a cycle. An **open ear decomposition** or a **proper ear decomposition** is an ear decomposition in which the two endpoints of each ear after the first are distinct from each other.



Ear decompositions may be used to characterize several important graph classes, and as part of efficient graph algorithms. They may also be generalized from graphs to matroids.

Characterizing graph classes

Several important classes of graphs may be characterized as the graphs having certain types of ear decompositions.

Graph connectivity

A graph is k -vertex-connected if the removal of any $(k - 1)$ vertices leaves a connected subgraph, and k -edge-connected if the removal of any $(k - 1)$ edges leaves a connected subgraph.

The following result is due to Hassler Whitney (1932):

A graph $G = (V, E)$ with $|V| \geq 2$ is 2-vertex-connected if and only if it has an open ear decomposition.

The following result is due to Herbert Robbins (1939):

A graph is 2-edge-connected if and only if it has an ear decomposition.

In both cases the number of ears is necessarily equal to the circuit rank of the given graph. Robbins introduced the ear decomposition of 2-edge-connected graphs as a tool for proving the Robbins theorem, that these are exactly the graphs that may be given a strongly connected orientation. Because of the pioneering work of Whitney and Robbins on ear decompositions, an ear decomposition is sometimes also called a **Whitney–Robbins synthesis** (Gross & Yellen 2006).

A **non-separating ear decomposition** is an open ear decomposition with two additional properties. Firstly, for each ear, the subset of vertices that are interior to later ears in the sequence must induce a connected graph. And secondly, each interior vertex of each ear must have a neighbor in a later ear. This type of ear decomposition may be used to generalize Whitney's result:

A graph $G = (V, E)$ with $|V| \geq 2$ is 3-vertex-connected if and only if it has a non-separating ear decomposition.

This result was first stated explicitly by Cherian & Maheshwari (1988), but as Schmidt (2013b) describes, it is equivalent to a result in the 1971 Ph.D. thesis of Lee Mondschein. Structures closely related to non-separating ear decompositions of maximal planar graphs, called canonical orderings, are also a standard tool in graph drawing.

Strong connectivity of directed graphs

The above definitions can also be applied to directed graphs. An **ear** would then be a directed path where all internal vertices have indegree and outdegree equal to 1. A directed graph is strongly connected if it contains a directed path from every vertex to every other vertex. Then we have the following theorem:

A directed graph is strongly connected if and only if it has an ear decomposition.

Similarly, a directed graph is biconnected if, for every two vertices, there exists a simple cycle in the graph containing both of them. Then

A directed graph is biconnected if and only if it has an open ear decomposition.

Factor-critical graphs

An ear decomposition is *odd* if each of its ears uses an odd number of edges. A factor-critical graph is a graph with an odd number of vertices, such that for each vertex v , if v is removed from the graph then the remaining vertices have a perfect matching. László Lovász (1972) found that:

A graph G is factor-critical if and only if G has an odd ear decomposition.

More generally, a result of Frank (1993) makes it possible to find in any graph G the ear decomposition with the fewest even ears.

Series-parallel graphs

A *tree* ear decomposition is a proper ear decomposition in which the first ear is a single edge and for each subsequent ear P_i , there is a single ear P_j , $i > j$, such that both endpoints of P_i lie on P_j (Khuller 1989). A *nested* ear decomposition is a tree ear decomposition such that, within each ear P_j , the set of pairs of endpoints of other ears P_i that lie within P_j form a set of nested intervals. A series-parallel graph is a graph with two designated terminals s and t that can be formed recursively by combining smaller series-parallel graphs in one of two ways: series composition (identifying one terminal from one smaller graph with one terminal from the other smaller graph, and keeping the other two terminals as the terminals of the combined graph) and parallel composition (identifying both pairs of terminals from the two smaller graphs).

The following result is due to David Eppstein (1992):

A 2-vertex-connected graph is series-parallel if and only if it has a nested ear decomposition.

Moreover, any open ear decomposition of a 2-vertex-connected series-parallel graph must be nested. The result may be extended to series-parallel graphs that are not 2-vertex-connected by using open ear decompositions that start with a path between the two terminals.

Matroids

The concept of an ear decomposition can be extended from graphs to matroids. An ear decomposition of a matroid is defined to be a sequence of circuits of the matroid, with two properties:

- each circuit in the sequence has a nonempty intersection with the previous circuits, and
- each circuit in the sequence remains a circuit even if all previous circuits in the sequence are contracted.

When applied to the graphic matroid of a graph G , this definition of an ear decomposition coincides with the definition of a proper ear decomposition of G : improper decompositions are excluded by the requirement that each circuit include at least one edge that also belongs to previous circuits. Using this definition, a matroid may be defined as factor-critical when it has an ear decomposition in which each circuit in the sequence has an odd number of new elements (Szegedy & Szegedy 2006).

Algorithms

On classical computers, ear decompositions of 2-edge-connected graphs and open ear decompositions of 2-vertex-connected graphs may be found by greedy algorithms that find each ear one at a time. A simple greedy approach that computes at the same time ear decompositions, open ear decompositions, st-numberings and -orientations in linear time (if exist) is given in Schmidt (2013a). The approach is based on computing a special ear decomposition named chain decomposition by one path-generating rule. Schmidt (2013b) shows that non-separating ear decompositions may also be constructed in linear time.

Lovász (1985), Maon, Schieber & Vishkin (1986), and Miller & Ramachandran (1986) provided efficient parallel algorithms for constructing ear decompositions of various types. For instance, to find an ear decomposition of a 2-edge-connected graph, the algorithm of Maon, Schieber & Vishkin (1986) proceeds according to the following steps:

1. Find a spanning tree of the given graph and choose a root for the tree.
2. Determine, for each edge uv that is not part of the tree, the distance between the root and the lowest common ancestor of u and v .
3. For each edge uv that is part of the tree, find the corresponding "master edge", a non-tree edge wx such that the cycle formed by adding wx to the tree passes through uv and such that, among such edges, w and x have a lowest common ancestor that is as close to the root as possible (with ties broken by edge identifiers).
4. Form an ear for each non-tree edge, consisting of it and the tree edges for which it is the master, and order the ears by their master edges' distance from the root (with the same tie-breaking rule).

These algorithms may be used as subroutines for other problems including testing connectivity, recognizing series-parallel graphs, and constructing st-numberings of graphs (an important subroutine in planarity testing).

An ear decomposition of a given matroid, with the additional constraint that every ear contains the same fixed element of the matroid, may be found in polynomial time given access to an independence oracle for the matroid (Coullard & Hellerstein 1996).

References

- Cheriyan, J.; Maheshwari, S. N. (1988), "Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs", *Journal of Algorithms* **9** (4): 507–537, doi:10.1016/0196-6774(88)90015-6 ^[1], MR 970192 ^[2].
- Coullard, Collette R.; Hellerstein, Lisa (1996), "Independence and port oracles for matroids, with an application to computational learning theory", *Combinatorica* **16** (2): 189–208, doi:10.1007/BF01844845 ^[3], MR 1401892 ^[4].
- Eppstein, D. (1992), "Parallel recognition of series-parallel graphs", *Information & Computation* **98** (1): 41–55, doi:10.1016/0890-5401(92)90041-D ^[5], MR 1161075 ^[6].
- Frank, András (1993), "Conservative weightings and ear-decompositions of graphs", *Combinatorica* **13** (1): 65–81, doi:10.1007/BF01202790 ^[7], MR 1221177 ^[8].
- Gross, Jonathan L.; Yellen, Jay (2006), "Characterization of strongly orientable graphs" ^[9], *Graph theory and its applications*, Discrete Mathematics and its Applications (Boca Raton) (2nd ed.), Chapman & Hall/CRC, Boca Raton, FL, pp. 498–499, ISBN 978-1-58488-505-4, MR 2181153 ^[10].
- Khuller, Samir (1989), "Ear decompositions" ^[11], *SIGACT News* **20** (1): 128.
- Lovász, László (1972), "A note on factor-critical graphs", *Studia Sci. Math. Hung.* **7**: 279–280, MR 0335371 ^[12].
- Lovász, László (1985), "Computing ears and branchings in parallel", *26th Annual Symposium on Foundations of Computer Science*, pp. 464–467, doi:10.1109/SFCS.1985.16 ^[13].
- Maon, Y.; Schieber, B.; Vishkin, U. (1986), "Parallel ear decomposition search (EDS) and ST-numbering in graphs", *Theoretical Computer Science* **47** (3), doi:10.1016/0304-3975(86)90153-2 ^[14], MR 0882357 ^[15].

- Miller, G.; Ramachandran, V. (1986), *Efficient parallel ear decomposition with applications*, Unpublished manuscript.
- Robbins, H. E. (1939), "A theorem on graphs, with an application to a problem of traffic control", *American Mathematical Monthly* **46**: 281–283, doi:10.2307/2303897^[16].
- Schmidt, Jens M. (2013a), "A Simple Test on 2-Vertex- and 2-Edge-Connectivity", *Information Processing Letters* **113** (7): 241–244, doi:10.1016/j.ipl.2013.01.016^[17].
- Schmidt, Jens M. (2013b), *The Mondschein sequence*, arXiv:1311.0750^[18].
- Schrijver, Alexander (2003), *Combinatorial Optimization. Polyhedra and efficiency. Vol A*, Springer-Verlag, ISBN 978-3-540-44389-6.
- Szegedy, Balázs; Szegedy, Christian (2006), "Symplectic spaces and ear-decomposition of matroids", *Combinatorica* **26** (3): 353–377, doi:10.1007/s00493-006-0020-3^[19], MR 2246153^[20].
- Whitney, H. (1932), "Non-separable and planar graphs", *Transactions of the American Mathematical Society* **34**: 339–362, doi:10.1090/S0002-9947-1932-1501641-2^[21], JSTOR 1989545^[22].

References

- [1] <http://dx.doi.org/10.1016%2F0196-6774%2888%2990015-6>
- [2] <http://www.ams.org/mathscinet-getitem?mr=970192>
- [3] <http://dx.doi.org/10.1007%2FBF01844845>
- [4] <http://www.ams.org/mathscinet-getitem?mr=1401892>
- [5] <http://dx.doi.org/10.1016%2F0890-5401%2892%2990041-D>
- [6] <http://www.ams.org/mathscinet-getitem?mr=1161075>
- [7] <http://dx.doi.org/10.1007%2FBF01202790>
- [8] <http://www.ams.org/mathscinet-getitem?mr=1221177>
- [9] http://books.google.com/books?id=unEloQ_sYmkC&pg=PA498
- [10] <http://www.ams.org/mathscinet-getitem?mr=2181153>
- [11] <http://portalparts.acm.org/70000/65780/bm/backmatter.pdf>
- [12] <http://www.ams.org/mathscinet-getitem?mr=0335371>
- [13] <http://dx.doi.org/10.1109%2FSFCS.1985.16>
- [14] <http://dx.doi.org/10.1016%2F0304-3975%2886%2990153-2>
- [15] <http://www.ams.org/mathscinet-getitem?mr=0882357>
- [16] <http://dx.doi.org/10.2307%2F2303897>
- [17] <http://dx.doi.org/10.1016%2Fj.ipl.2013.01.016>
- [18] <http://arxiv.org/abs/1311.0750>
- [19] <http://dx.doi.org/10.1007%2Fs00493-006-0020-3>
- [20] <http://www.ams.org/mathscinet-getitem?mr=2246153>
- [21] <http://dx.doi.org/10.1090%2FS0002-9947-1932-1501641-2>
- [22] <http://www.jstor.org/stable/1989545>

Algorithms for 2-edge-connected components

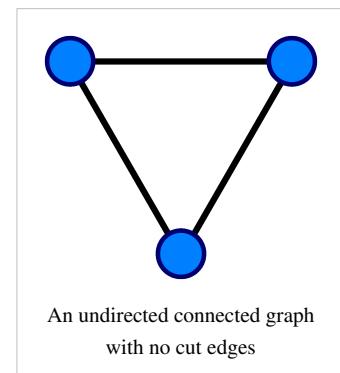
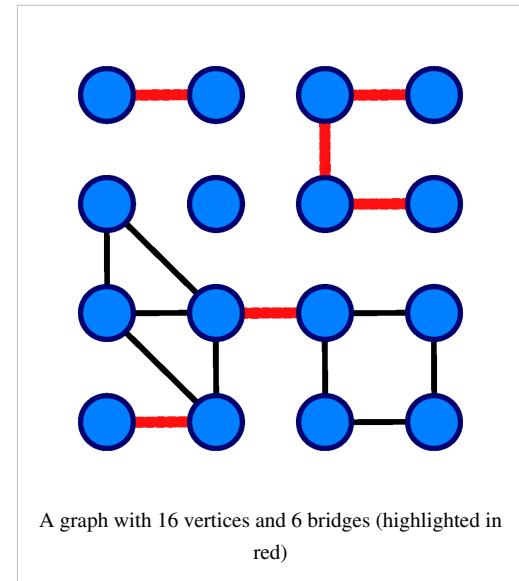
In graph theory, a **bridge**, **isthmus**, **cut-edge**, or **cut arc** is an edge of a graph whose deletion increases its number of connected components. Equivalently, an edge is a bridge if and only if it is not contained in any cycle. A graph is said to be **bridgeless** or **isthmus-free** if it contains no bridges.

Another meaning of "bridge" appears in the term **bridge of a subgraph**. If H is a subgraph of G , a **bridge of H in G** is a maximal subgraph of G that is not contained in H and is not separated by H .

Trees and forests

A graph with n nodes can contain at most $n - 1$ bridges, since adding additional edges must create a cycle. The graphs with exactly $n - 1$ bridges are exactly the trees, and the graphs in which every edge is a bridge are exactly the forests.

In every undirected graph, there is an equivalence relation on the vertices according to which two vertices are related to each other whenever there are two edge-disjoint paths connecting them. (Every vertex is related to itself via two length-zero paths, which are identical but nevertheless edge-disjoint.) The equivalence classes of this relation are called **2-edge-connected components**, and the bridges of the graph are exactly the edges whose endpoints belong to different components. The **bridge-block tree** of the graph has a vertex for every nontrivial component and an edge for every bridge.



Relation to vertex connectivity

Bridges are closely related to the concept of articulation vertices, vertices that belong to every path between some pair of other vertices. The two endpoints of a bridge are articulation vertices unless they have a degree of 1, although it may also be possible for a non-bridge edge to have two articulation vertices as endpoints. Analogously to bridgeless graphs being 2-edge-connected, graphs without articulation vertices are 2-vertex-connected.

In a cubic graph, every cut vertex is an endpoint of at least one bridge.

Bridgeless graphs

A **bridgeless graph** is a graph that does not have any bridges. Equivalent conditions are that each connected component of the graph has an open ear decomposition, that each connected component is 2-edge-connected, or (by Robbins' theorem) that every connected component has a strong orientation.

An important open problem involving bridges is the cycle double cover conjecture, due to Seymour and Szekeres (1978 and 1979, independently), which states that every bridgeless graph admits a set of simple cycles which contains each edge exactly twice.

Tarjan's Bridge-finding algorithm

The first linear time algorithm for finding the bridges in a graph was described by Robert Tarjan in 1974. It performs the following steps:

- Find a spanning forest of G
- Create a rooted forest F from the spanning tree
- Traverse the forest F in preorder and number the nodes. Parent nodes in the forest now have lower numbers than child nodes.
- For each node v in preorder, do:
 - Compute the number of forest descendants $ND(v)$ for this node, by adding one to the sum of its children's descendants.
 - Compute $L(v)$, the lowest preorder label reachable from v by a path for which all but the last edge stays within the subtree rooted at v . This is the minimum of the set consisting of the values of $L(w)$ at child nodes of v and of the preorder labels of nodes reachable from v by edges that do not belong to F .
 - Similarly, compute $H(v)$, the highest preorder label reachable by a path for which all but the last edge stays within the subtree rooted at v . This is the maximum of the set consisting of the values of $H(w)$ at child nodes of v and of the preorder labels of nodes reachable from v by edges that do not belong to F .
 - For each node w with parent node v , if $L(w) = w$ and $H(w) < w + ND(w)$ then the edge from v to w is a bridge.

Bridge-Finding with Chain Decompositions

A very simple bridge-finding algorithm uses chain decompositions. Chain decompositions do not only allow to compute all bridges of a graph, they also allow to *read off* every cut vertex of G (and the block-cut tree of G), giving a general framework for testing 2-edge- and 2-vertex-connectivity (which extends to linear-time 3-edge- and 3-vertex-connectivity tests).

Chain decompositions are special ear decompositions depending on a DFS-tree T of G and can be computed very simply: Let every vertex be marked as unvisited. For each vertex v in ascending DFS-numbers $1 \dots n$, traverse every backedge (i.e. every edge not in the DFS tree) that is incident to v and follow the path of tree-edges back to the root of T , stopping at the first vertex that is marked as visited. During such a traversal, every traversed vertex is marked as visited. Thus, a traversal stops at the latest at v and forms either a directed path or cycle, beginning with v ; we call this path or cycle a *chain*. The i th chain found by this procedure is referred to as C_i . $C = C_1, C_2, \dots$ is then a *chain decomposition* of G .

The following characterizations then allow to *read off* several properties of G from C efficiently, including all bridges of G . Let C be a chain decomposition of a simple connected graph $G = (V, E)$.

1. G is 2-edge-connected if and only if the chains in C partition E .
2. An edge e in G is a bridge if and only if e is not contained in any chain in C .
3. If G is 2-edge-connected, C is an ear decomposition.
4. G is 2-vertex-connected if and only if G has minimum degree 2 and C_1 is the only cycle in C .

5. A vertex v in a 2-edge-connected graph G is a cut vertex if and only if v is the first vertex of a cycle in $C - C_I$.
6. If G is 2-vertex-connected, C is an open ear decomposition.

Notes

Algorithms for 2-vertex-connected components

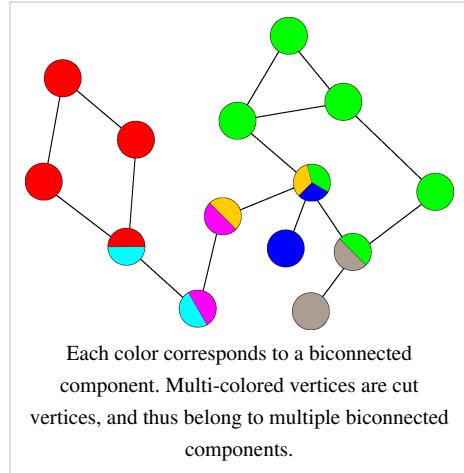
In graph theory, a **biconnected component** (or **2-connected component**) is a maximal biconnected subgraph. Any connected graph decomposes into a tree of biconnected components called the **block tree** of the graph. The blocks are attached to each other at shared vertices called **cut vertices** or **articulation points**. Specifically, a **cut vertex** is any vertex whose removal increases the number of connected components.

Algorithm

The classic sequential algorithm for computing biconnected components in a connected undirected graph is due to John Hopcroft and Robert Tarjan (1973). It runs in linear time, and is based on depth-first search. This algorithm is also outlined as Problem 22-2 of Introduction to Algorithms (both 2nd and 3rd editions).

The idea is to run a depth-first search while maintaining the following information:

1. the depth of each vertex in the depth-first-search tree (once it gets visited), and
2. for each vertex v , the lowest depth of neighbors of all descendants of v in the depth-first-search tree, called the **lowpoint**.



The depth is standard to maintain during a depth-first search. The lowpoint of v can be computed after visiting all descendants of v (i.e., just before v gets popped off the depth-first-search stack) as the minimum of the depth of v , the depth of all neighbors of v (other than the parent of v in the depth-first-search tree) and the lowpoint of all children of v in the depth-first-search tree.

The key fact is that a nonroot vertex v is a cut vertex (or articulation point) separating two biconnected components if and only if there is a child y of v such that $\text{lowpoint}(y) \geq \text{depth}(v)$. This property can be tested once the depth-first search returned from every child of v (i.e., just before v gets popped off the depth-first-search stack), and if true, v separates the graph into different biconnected components. This can be represented by computing one biconnected component out of every such y (a component which contains y will contain the subtree of y , plus v), and then erasing the subtree of y from the tree.

The root vertex must be handled separately: it is a cut vertex if and only if it has at least two children. Thus, it suffices to simply build one component out of each child subtree of the root (including the root).

A non-recursive example implementation in C++11 is

```
// Graph is a class with the following prototype
// class Graph{
//     range_of_arc_type out(int vertex) const; // Returns a range of all outgoing arcs.
//     int head(arc_type arc) const; // Maps an arc onto its head.
//     int node_count() const; // Counts the number of nodes in the graph.
```

```

// }

template<class Graph>
std::vector<int> find_all_articulation_points(const Graph&g) {
    // The empty graph contains no articulation points
    if(g.node_count() == 0)
        return {};

    typedef typename std::decay<decltype(std::begin(g.out(0)))>::type Iter;

    std::vector<bool>
        visited(g.node_count(), false),
        is_articulation_point(g.node_count(), false);

    std::vector<int>
        parent(g.node_count(), -1),
        depth(g.node_count(), std::numeric_limits<int>::max()),
        min_succ_depth(g.node_count(), std::numeric_limits<int>::max());
    std::vector<Iter>next_out_arc(g.node_count());

    const int root = 0;

    std::stack<int>s;
    s.push(root);
    depth[root] = 0;

    while(!s.empty()) {
        int x = s.top();
        s.pop();

        if(!visited[x]) {
            // x is visited for the first time.
            visited[x] = true;
            next_out_arc[x] = std::begin(g.out(x));
            min_succ_depth[x] = depth[x];
        } else {
            // A child of x has been fully processed, continue with the next child.
            auto xy = *next_out_arc[x];
            auto y = g.head(xy);
            if(min_succ_depth[y] >= depth[x] && !is_articulation_point[x]) {
                // As removing x would disconnect y and parent[x] we know that x is articulation point.
                is_articulation_point[x] = true;
                articulation_point_list.push_back(x);
            }
            min_succ_depth[x] = std::min(min_succ_depth[x], min_succ_depth[y]);
            ++next_out_arc[x];
        }
        auto end = std::end(g.out(x));
    }
}

```

```

        while(next_out_arc[x] != end) {
            auto xy = *next_out_arc[x];
            auto y = g.head(xy);
            if(visited[y]){
                if(parent[x] != y)
                    min_succ_depth[x] = std::min(min_succ_depth[x], depth[y]);
                ++next_out_arc[x];
            }else{
                auto xy = *next_out_arc[x];
                auto y = g.head(xy);
                s.push(x); // push x so that it is visited a second time on the way up
                s.push(y);
                parent[y] = x;
                depth[y] = depth[x]+1;
                break;
            }
        }
    }

    int root_child_count = 0;
    for(auto xy:g.out(root)){
        int y = g.head(xy);
        if(parent[y] == root)
            ++root_child_count;
    }

    if(root_child_count >= 2)
        articulation_point_list.push_back(root);
    return std::move(articulation_point_list);
}

```

Other algorithms

A simple alternative to the above algorithm uses chain decompositions, which are special ear decompositions depending on DFS-trees. Chain decompositions can be computed in linear time by this traversing rule. Let C be a chain decomposition of G . Then G is 2-vertex-connected if and only if G has minimum degree 2 and C_j is the only cycle in C . This gives immediately a linear-time 2-connectivity test and can be extended to list all cut vertices of G in linear time using the following statement: A vertex v in a connected graph G (with minimum degree 2) is a cut vertex if and only if v is incident to a bridge or v is the first vertex of a cycle in $C - C_j$. The list of cut vertices can be used to create the block-cut tree of G in linear time.

In the online version of the problem, vertices and edges are added (but not removed) dynamically, and a data structure must maintain the biconnected components. Jeffery Westbrook and Robert Tarjan (1992) developed an efficient data structure for this problem based on disjoint-set data structures. Specifically, it processes n vertex additions and m edge additions in $O(m \alpha(m, n))$ total time, where α is the inverse Ackermann function. This time bound is proved to be optimal.

Uzi Vishkin and Robert Tarjan (1985) designed a parallel algorithm on CRCW PRAM that runs in $O(\log n)$ time with $n + m$ processors. Guojing Cong and David A. Bader (2005) developed an algorithm that achieves a speedup of

5 with 12 processors on SMPs. Speedups exceeding 30 based on the original Tarjan-Vishkin algorithm were reported by James A. Edwards and Uzi Vishkin (2012).

Notes

References

- Eugene C. Freuder (1985). "A Sufficient Condition for Backtrack-Bounded Search". *Journal of the Association for Computing Machinery* **32** (4): 755–761. doi: 10.1145/4221.4225 (<http://dx.doi.org/10.1145/4221.4225>).

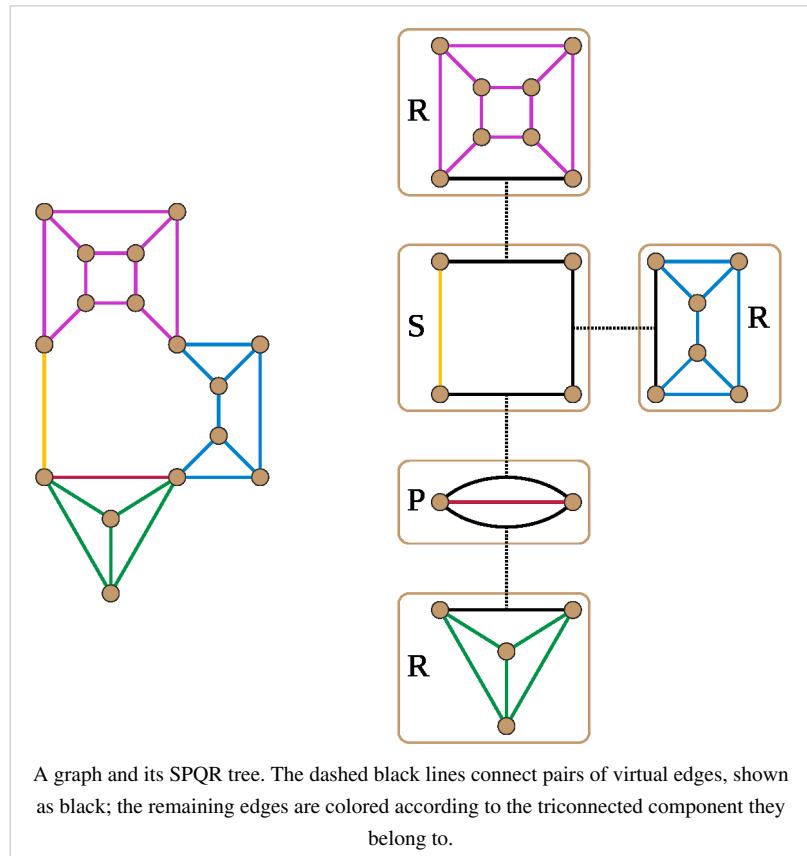
External links

- The tree of the biconnected components Java implementation (<http://code.google.com/p/jbpt/>) in the jBPT library (see BCTree class).

Algorithms for 3-vertex-connected components

In graph theory, a branch of mathematics, the **triconnected components** of a biconnected graph are a system of smaller graphs that describe all of the 2-vertex cuts in the graph. An **SPQR tree** is a tree data structure used in computer science, and more specifically graph algorithms, to represent the triconnected components of a graph. The SPQR tree of a graph may be constructed in linear time^[1] and has several applications in dynamic graph algorithms and graph drawing.

The basic structures underlying the SPQR tree, the triconnected components of a graph, and the connection between this decomposition and the planar embeddings of a planar graph, were first investigated by Saunders Mac Lane (1937); these structures were used in efficient algorithms by several other researchers^[2] prior to their formalization as the SPQR tree by Di Battista and Tamassia (1989, 1990, 1996).



Structure

An SPQR tree takes the form of an unrooted tree in which for each node x there is associated an undirected graph or multigraph G_x . The node, and the graph associated with it, may have one of four types, given the initials SPQR:

- In an S node, the associated graph is a cycle graph with three or more vertices and edges. This case is analogous to series composition in series-parallel graphs; the S stands for "series".
- In a P node, the associated graph is a dipole graph, a multigraph with two vertices and three or more edges, the planar dual to a cycle graph. This case is analogous to parallel composition in series-parallel graphs; the P stands for "parallel".
- In a Q node, the associated graph has a single real edge. This trivial case is necessary to handle the graph that has only one edge. In some works on SPQR trees, this type of node does not appear in the SPQR trees of graphs with more than one edge; in other works, all non-virtual edges are required to be represented by Q nodes with one real and one virtual edge, and the edges in the other node types must all be virtual.
- In an R node, the associated graph is a 3-connected graph that is not a cycle or dipole. The R stands for "rigid": in the application of SPQR trees in planar graph embedding, the associated graph of an R node has a unique planar embedding.

Each edge xy between two nodes of the SPQR tree is associated with two directed *virtual edges*, one of which is an edge in G_x and the other of which is an edge in G_y . Each edge in a graph G_x may be a virtual edge for at most one SPQR tree edge.

An SPQR tree T represents a 2-connected graph G_T , formed as follows. Whenever SPQR tree edge xy associates the virtual edge ab of G_x with the virtual edge cd of G_y , form a single larger graph by merging a and c into a single supervertex, merging b and d into another single supervertex, and deleting the two virtual edges. That is, the larger graph is the 2-clique-sum of G_x and G_y . Performing this gluing step on each edge of the SPQR tree produces the graph G_T ; the order of performing the gluing steps does not affect the result. Each vertex in one of the graphs G_x may be associated in this way with a unique vertex in G_T , the supervertex into which it was merged.

Typically, it is not allowed within an SPQR tree for two S nodes to be adjacent, nor for two P nodes to be adjacent, because if such an adjacency occurred the two nodes could be merged into a single larger node. With this assumption, the SPQR tree is uniquely determined from its graph. When a graph G is represented by an SPQR tree with no adjacent P nodes and no adjacent S nodes, then the graphs G_x associated with the nodes of the SPQR tree are known as the triconnected components of G .

Construction

The SPQR tree of a given 2-vertex-connected graph can be constructed in linear time.

The problem of constructing the triconnected components of a graph was first solved in linear time by Hopcroft & Tarjan (1973). Based on this algorithm, Di Battista & Tamassia (1996) suggested that the full SPQR tree structure, and not just the list of components, should be constructible in linear time. After an implementation of a slower algorithm for SPQR trees was provided as part of the GDToolkit library, Gutwenger & Mutzel (2001) provided the first linear-time implementation. As part of this process of implementing this algorithm, they also corrected some errors in the earlier work of Hopcroft & Tarjan (1973).

The algorithm of Gutwenger & Mutzel (2001) includes the following overall steps.

1. Sort the edges of the graph by the pairs of numerical indices of their endpoints, using a variant of radix sort that makes two passes of bucket sort, one for each endpoint. After this sorting step, parallel edges between the same two vertices will be adjacent to each other in the sorted list and can be split off into a P-node of the eventual SPQR tree, leaving the remaining graph simple.
2. Partition the graph into split components; these are graphs that can be formed by finding a pair of separating vertices, splitting the graph at these two vertices into two smaller graphs (with a linked pair of virtual edges

having the separating vertices as endpoints), and repeating this splitting process until no more separating pairs exist. The partition found in this way is not uniquely defined, because the parts of the graph that should become S-nodes of the SPQR tree will be subdivided into multiple triangles.

3. Label each split component with a P (a two-vertex split component with multiple edges), an S (a split component in the form of a triangle), or an R (any other split component). While there exist two split components that share a linked pair of virtual edges, and both components have type S or both have type P, merge them into a single larger component of the same time.

To find the split components, Gutwenger & Mutzel (2001) use depth-first search to find a structure that they call a palm tree; this is a depth-first search tree with its edges oriented away from the root of the tree, for the edges belonging to the tree, and towards the root for all other edges. They then find a special preorder numbering of the nodes in the tree, and use certain patterns in this numbering to identify pairs of vertices that can separate the graph into smaller components. When a component is found in this way, a stack data structure is used to identify the edges that should be part of the new component.

Usage

Finding 2-vertex cuts

With the SPQR tree of a graph G (without Q nodes) it is straightforward to find every pair of vertices u and v in G such that removing u and v from G leaves a disconnected graph, and the connected components of the remaining graphs:

- The two vertices u and v may be the two endpoints of a virtual edge in the graph associated with an R node, in which case the two components are represented by the two subtrees of the SPQR tree formed by removing the corresponding SPQR tree edge.
- The two vertices u and v may be the two vertices in the graph associated with a P node that has two or more virtual edges. In this case the components formed by the removal of u and v are represented by subtrees of the SPQR tree, one for each virtual edge in the node.
- The two vertices u and v may be two vertices in the graph associated with an S node such that either u and v are not adjacent, or the edge uv is virtual. If the edge is virtual, then the pair (u,v) also belongs to a node of type P and R and the components are as described above. If the two vertices are not adjacent then the two components are represented by two paths of the cycle graph associated with the S node and with the SPQR tree nodes attached to those two paths.

Representing all embeddings of planar graphs

If a planar graph is 3-connected, it has a unique planar embedding up to the choice of which face is the outer face and of orientation of the embedding: the faces of the embedding are exactly the nonseparating cycles of the graph. However, for a planar graph (with labeled vertices and edges) that is 2-connected but not 3-connected, there may be greater freedom in finding a planar embedding. Specifically, whenever two nodes in the SPQR tree of the graph are connected by a pair of virtual edges, it is possible to flip the orientation of one of the nodes relative to the other one. Additionally, in a P node of the SPQR tree, the different parts of the graph connected to virtual edges of the P node may be arbitrarily permuted. All planar representations may be described in this way.^[3]

Notes

- [1] ; .
- [2] E.g., and , both of which are cited as precedents by Di Battista and Tamassia.
- [3] Mac Lane (1937).

References

- Bienstock, Daniel; Monma, Clyde L. (1988), "On the complexity of covering vertices by faces in a planar graph", *SIAM Journal on Computing* **17** (1): 53–76, doi: 10.1137/0217004 (<http://dx.doi.org/10.1137/0217004>).
- Di Battista, Giuseppe; Tamassia, Roberto (1989), "Incremental planarity testing", *Proc. 30th Annual Symposium on Foundations of Computer Science*, pp. 436–441, doi: 10.1109/SFCS.1989.63515 (<http://dx.doi.org/10.1109/SFCS.1989.63515>).
- Di Battista, Giuseppe; Tamassia, Roberto (1990), "On-line graph algorithms with SPQR-trees", *Proc. 17th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **443**, Springer-Verlag, pp. 598–611, doi: 10.1007/BFb0032061 (<http://dx.doi.org/10.1007/BFb0032061>).
- Di Battista, Giuseppe; Tamassia, Roberto (1996), "On-line planarity testing" (<http://cs.brown.edu/research/pubs/pdfs/1996/DiBattista-1996-OPT.pdf>), *SIAM Journal on Computing* **25** (5): 956–997, doi: 10.1137/S0097539794280736 (<http://dx.doi.org/10.1137/S0097539794280736>).
- Gutwenger, Carsten; Mutzel, Petra (2001), "A linear time implementation of SPQR-trees", *Proc. 8th International Symposium on Graph Drawing (GD 2000)*, Lecture Notes in Computer Science **1984**, Springer-Verlag, pp. 77–90, doi: 10.1007/3-540-44541-2_8 (http://dx.doi.org/10.1007/3-540-44541-2_8).
- Hopcroft, John; Tarjan, Robert (1973), "Dividing a graph into triconnected components", *SIAM Journal on Computing* **2** (3): 135–158, doi: 10.1137/0202012 (<http://dx.doi.org/10.1137/0202012>).
- Mac Lane, Saunders (1937), "A structural characterization of planar combinatorial graphs", *Duke Mathematical Journal* **3** (3): 460–472, doi: 10.1215/S0012-7094-37-00336-3 (<http://dx.doi.org/10.1215/S0012-7094-37-00336-3>).

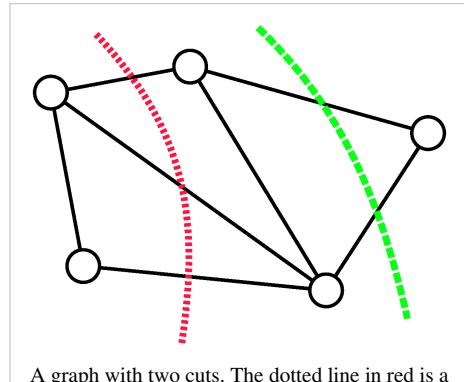
External links

- SPQR tree implementation (http://www.ogdf.net/doc-ogdf/classogdf_1_1_s_p_q_r_tree.html) in the Open Graph Drawing Framework.
- The tree of the triconnected components Java implementation (<http://code.google.com/p/jbpt/>) in the jBPT library (see TCTree class).

Karger's algorithm for general vertex connectivity

In computer science and graph theory, **Karger's algorithm** is a randomized algorithm to compute a minimum cut of a connected graph. It was invented by David Karger and first published in 1993.

The idea of the algorithm is based on the concept of contraction of an edge (u, v) in an undirected graph $G = (V, E)$. Informally speaking, the contraction of an edge merges the nodes u and v into one, reducing the total number of nodes of the graph by one. All other edges connecting either u or v are "reattached" to the merged node, effectively producing a multigraph. Karger's basic algorithm iteratively contracts randomly chosen edges until only two nodes remain; those nodes represent a cut in the original graph. By iterating this basic algorithm a sufficient number of times, a minimum cut can be found with high probability.



A graph with two cuts. The dotted line in red is a cut with three crossing edges. The dashed line in green is a min-cut of this graph, crossing only two edges.

The global minimum cut problem

Main article: Minimum cut

A *cut* (S, T) in an undirected graph $G = (V, E)$ is a partition of the vertices V into two non-empty, disjoint sets $S \cup T = V$. The *cutset* of a cut consists of the edges $\{uv \in E : u \in S, v \in T\}$ between the two parts. The *size* (or *weight*) of a cut in an unweighted graph is the cardinality of the cutset, i.e., the number of edges between the two parts,

$$w(S, T) = |\{uv \in E : u \in S, v \in T\}|.$$

There are $2^{|V|}$ ways of choosing for each vertex whether it belongs to S or to T , but two of these choices make S or T empty and do not give rise to cuts. Among the remaining choices, swapping the roles of S and T does not change the cut, so each cut is counted twice; therefore, there are $2^{|V|-1} - 1$ distinct cuts. The *minimum cut problem* is to find a cut of smallest size among these cuts.

For weighted graphs with positive edge weights $w : E \rightarrow \mathbf{R}^+$ the weight of the cut is the sum of the weights of edges between vertices in each part

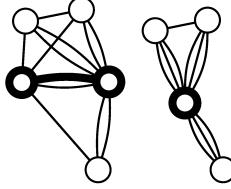
$$w(S, T) = \sum_{uv \in E: u \in S, v \in T} w(uv),$$

which agrees with the unweighted definition for $w = 1$.

A cut is sometimes called a “global cut” to distinguish it from an “ s - t cut” for a given pair of vertices, which has the additional requirement that $s \in S$ and $t \in T$. Every global cut is an s - t cut for some $s, t \in V$. Thus, the minimum cut problem can be solved in polynomial time by iterating over all choices of $s, t \in V$ and solving the resulting minimum s - t cut problem using the max-flow min-cut theorem and a polynomial time algorithm for maximum flow, such as the Ford–Fulkerson algorithm, though this approach is not optimal. There is a deterministic algorithm for the minimum cut problem with running time $O(mn + n^2 \log n)$.

Contraction algorithm

The fundamental operation of Karger's algorithm is a form of edge contraction. The result of contracting the edge $e = \{u, v\}$ is new node uv . Every edge $\{w, u\}$ or $\{w, v\}$ for $w \notin \{u, v\}$ to the endpoints of the contracted edge is replaced by an edge $\{w, uv\}$ to the new node. Finally, the contracted nodes u and v with all their incident edges are removed. In particular, the resulting graph contains no self-loops. The result of contracting edge e is denoted G/e .



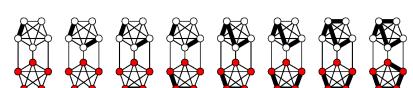
The contraction algorithm repeatedly contracts random edges in the graph, until only two nodes remain, at which point there is only a single cut.



Successful run of Karger's algorithm on a 10-vertex graph. The minimum cut has size 3.

```
procedure contract( $G = (V, E)$ ) :
while  $|V| > 2$ 
    choose  $e \in E$  uniformly at random
     $G \leftarrow G/e$ 
return the only cut in  $G$ 
```

When the graph is represented using adjacency lists or an adjacency matrix, a single edge contraction operation can be implemented with a linear number of updates to the data structure, for a total running time of $O(|V|^2)$. Alternatively, the procedure can be viewed as an execution of Kruskal's algorithm for constructing the minimum spanning tree in a graph where the edges have weights $w(e_i) = \pi(i)$ according to a random permutation π . Removing the heaviest edge of this tree results in two components that describe a cut. In this way, the contraction procedure can be implemented like Kruskal's algorithm in time $O(|E| \log |E|)$.



The random edge choices in Karger's algorithm correspond to an execution of Kruskal's algorithm on a graph with random edge ranks until only two components remain.

The best known implementations use $O(|E|)$ time and space, or $O(|E| \log |E|)$ time and $O(|V|)$ space, respectively.

Success probability of the contraction algorithm

In a graph $G = (V, E)$ with $n = |V|$ vertices, the contraction algorithm returns a minimum cut with polynomially small probability $\binom{n}{2}^{-1}$. Every graph has $2^{n-1} - 1$ cuts, among which at most $\binom{n}{2}$ can be minimum cuts. Therefore, the success probability for this algorithm is much better than the probability for picking a cut at random, which is at most $\binom{n}{2}/(2^{n-1} - 1)$

For instance, the cycle graph on n vertices has exactly $\binom{n}{2}$ minimum cuts, given by every choice of 2 edges. The contraction procedure finds each of these with equal probability.

To establish the bound on the success probability in general, let C denote the edges of a specific minimum cut of size k . The contraction algorithm returns C if none of the random edges belongs to the cutset of C . In particular, the first edge contraction avoids C , which happens with probability $1 - k/|E|$. The minimum degree of G is at least k (otherwise a minimum degree vertex would induce a smaller cut), so $|E| \geq nk/2$. Thus, the probability that the contraction algorithm picks an edge from C is

$$\frac{k}{|E|} \leq \frac{k}{nk/2} = \frac{2}{n}.$$

The probability p_n that the contraction algorithm on an n -vertex graph avoids C satisfies the recurrence $p_n \geq (1 - \frac{2}{n})p_{n-1}$, with $p_2 = 1$, which can be expanded as

$$p_n \geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right) = \prod_{i=0}^{n-3} \frac{n-i-2}{n-i} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} = \binom{n}{2}^{-1}.$$

Repeating the contraction algorithm

By repeating the contraction algorithm $T = \binom{n}{2} \ln n$ times with

independent random choices and returning the smallest cut, the probability of not finding a minimum cut is

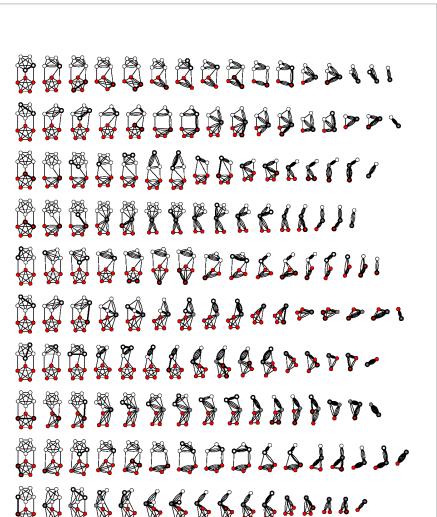
$$\left[1 - \binom{n}{2}^{-1}\right]^T \leq \frac{1}{e^{\ln n}} = \frac{1}{n}.$$

The total running time for T repetitions for a graph with n vertices and m edges is $O(Tm) = O(n^2 m \log n)$.

Karger–Stein algorithm

An extension of Karger's algorithm due to David Karger and Clifford Stein achieves an order of magnitude improvement.

The basic idea is to perform the contraction procedure until the graph reaches t vertices.



10 repetitions of the contraction procedure. The 5th repetition finds the minimum cut of size 3.

```
procedure contract( $G = (V, E)$ ,  $t$ ):
  while  $|V| > t$ 
    choose  $e \in E$  uniformly at random
     $G \leftarrow G/e$ 
```

```
return  $G$ 
```

The probability $p_{n,t}$ that this contraction procedure avoids a specific cut C in an n -vertex graph is

$$p_{n,t} \geq \prod_{i=0}^{n-t-1} \left(1 - \frac{2}{n-i}\right) = \binom{t}{2} / \binom{n}{2}.$$

This expression is $\Omega(t^2/n^2)$ becomes less than $\frac{1}{2}$ around $t = \lceil 1 + n/\sqrt{2} \rceil$. In particular, the probability that

an edge from C is contracted grows towards the end. This motivates the idea of switching to a slower algorithm after a certain number of contraction steps.

```
procedure fastmincut( $G = (V, E)$ ) :
  if  $|V| < 6$ :
    return mincut( $V$ )
  else:
     $t \leftarrow \lceil 1 + |V|/\sqrt{2} \rceil$ 
     $G_1 \leftarrow \text{contract}(G, t)$ 
     $G_2 \leftarrow \text{contract}(G, t)$ 
    return min {fastmincut( $G_1$ ), fastmincut( $G_2$ )}
```

Analysis

The probability $P(n)$ the algorithm finds a specific cutset C is given by the recurrence relation

$$P(n) = 1 - \left(1 - \frac{1}{2}P\left(\lceil 1 + \frac{n}{\sqrt{2}} \rceil\right)\right)^2$$

with solution $P(n) = O\left(\frac{1}{\log n}\right)$. The running time of fastmincut satisfies

$$T(n) = 2T\left(\lceil 1 + \frac{n}{\sqrt{2}} \rceil\right) + O(n^2)$$

with solution $T(n) = O(n^2 \log n)$. To achieve error probability $O(1/n)$, the algorithm can be repeated $O(\log n/P(n))$ times, for an overall running time of $T(n) \cdot \frac{\log n}{P(n)} = O(n^2 \log^3 n)$. This is an order of magnitude improvement over Karger's original algorithm.

Finding all min-cuts

Theorem: With high probability we can find all min cuts in the running time of $O(n^2 \ln^3 n)$.

Proof: Since we know that $P(n) = O\left(\frac{1}{\ln n}\right)$, therefore after running this algorithm $O(\ln^2 n)$ times The probability of missing a specific min-cut is

$$\Pr[\text{miss a specific min-cut}] = (1-P(n))^{O(\ln^2 n)} \leq \left(1 - \frac{c}{\ln n}\right)^{\frac{3\ln^2 n}{c}} \leq e^{-3\ln n} = \frac{1}{n^3}.$$

And there are at most $\binom{n}{2}$ min-cuts, hence the probability of missing any min-cut is

$$\Pr[\text{miss any min-cut}] \leq \binom{n}{2} \cdot \frac{1}{n^3} = O\left(\frac{1}{n}\right).$$

The probability of failures is considerably small when n is large enough. \square

Improvement bound

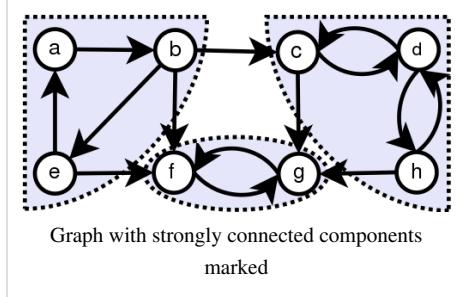
To determine a min-cut, one has to touch every edge in the graph at least once, which is $O(n^2)$ time in a dense graph. The Karger–Stein's min-cut algorithm takes the running time of $O(n^2 \ln^{O(1)} n)$, which is very close to that.

References

Connectivity of directed graphs

Strongly connected components

In the mathematical theory of directed graphs, a graph is said to be **strongly connected** if every vertex is reachable from every other vertex. The **strongly connected components** of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.



Definitions

A directed graph is called **strongly connected** if there is a path in each direction between each pair of vertices of the graph. In a directed graph G that may not itself be strongly connected, a pair of vertices u and v are said to be strongly connected to each other if there is a path in each direction between them.

The binary relation of being strongly connected is an equivalence relation, and the induced subgraphs of its equivalence classes are called **strongly connected components**. Equivalently, a **strongly connected component** of a directed graph G is a subgraph that is strongly connected, and is maximal with this property: no additional edges or vertices from G can be included in the subgraph without breaking its property of being strongly connected. The collection of strongly connected components forms a partition of the set of vertices of G .

If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, the **condensation** of G . A directed graph is acyclic if and only if it has no strongly connected subgraphs with more than one vertex, because a directed cycle is strongly connected and every nontrivial strongly connected component contains at least one directed cycle.

Algorithms

Several algorithms can compute strongly connected components in linear time.

- Kosaraju's algorithm uses two passes of depth first search. The first, in the original graph, is used to choose the order in which the outer loop of the second depth first search tests vertices for having been visited already and recursively explores them if not. The second depth first search is on the transpose graph of the original graph, and each recursive exploration finds a single new strongly connected component.^[1] It is named after S. Rao Kosaraju, who described it (but did not publish his results) in 1978; Micha Sharir later published it in 1981.^[2]
- Tarjan's strongly connected components algorithm, published by Robert Tarjan in 1972, performs a single pass of depth first search. It maintains a stack of vertices that have been explored by the search but not yet assigned to a component, and calculates "low numbers" of each vertex (an index number of the highest ancestor reachable in one step from a descendant of the vertex) which it uses to determine when a set of vertices should be popped off the stack into a new component.
- The path-based strong component algorithm uses a depth first search, like Tarjan's algorithm, but with two stacks. One of the stacks is used to keep track of the vertices not yet assigned to components, while the other keeps track

of the current path in the depth first search tree. The first linear time version of this algorithm was published by Edsger W. Dijkstra in 1976.

Although Kosaraju's algorithm is conceptually simple, Tarjan's and the path-based algorithm are favoured in practice since they require only one depth-first search rather than two.

Applications

Algorithms for finding strongly connected components may be used to solve 2-satisfiability problems (systems of Boolean variables with constraints on the values of pairs of variables): as Aspvall, Plass & Tarjan (1979) showed, a 2-satisfiability instance is unsatisfiable if and only if there is a variable v such that v and its complement are both contained in the same strongly connected component of the implication graph of the instance.

Strongly connected components are also used to compute the Dulmage–Mendelsohn decomposition, a classification of the edges of a bipartite graph, according to whether or not they can be part of a perfect matching in the graph.

Related results

A directed graph is strongly connected if and only if it has an ear decomposition, a partition of the edges into a sequence of directed paths and cycles such that the first subgraph in the sequence is a cycle, and each subsequent subgraph is either a cycle sharing one vertex with previous subgraphs, or a path sharing its two endpoints with previous subgraphs.

According to Robbins' theorem, an undirected graph may be oriented in such a way that it becomes strongly connected, if and only if it is 2-edge-connected. One way to prove this result is to find an ear decomposition of the underlying undirected graph and then orient each ear consistently.

References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.5, pp. 552–557.
- [2] Micha Sharir. A strong connectivity algorithm and its applications to data flow analysis. *Computers and Mathematics with Applications* 7(1):67–72, 1981.

External links

- Java implementation for computation of strongly connected components (<http://code.google.com/p/jbpt/>) in the jBPT library (see StronglyConnectedComponents class).

Tarjan's strongly connected components algorithm

Tarjan's strongly connected components algorithm

<p style="text-align: center;">Tarjan's algorithm animation</p>	
Data structure	Graph
Worst case performance	$O(V + E)$

Tarjan's Algorithm (named for its discoverer, Robert Tarjan) is a graph theory algorithm for finding the strongly connected components of a graph. Although it precedes it chronologically, it can be seen as an improved version of Kosaraju's algorithm, and is comparable in efficiency to the path-based strong component algorithm.

Overview

The algorithm takes a directed graph as input, and produces a partition of the graph's vertices into the graph's strongly connected components. Each vertex of the graph appears in exactly one of the strongly connected components. Any vertex that is not on a directed cycle forms a strongly connected component all by itself: for example, a vertex whose in-degree or out-degree is 0, or any vertex of an acyclic graph.

The basic idea of the algorithm is this: a depth-first search begins from an arbitrary start node (and subsequent depth-first searches are conducted on any nodes that have not yet been found). As usual with depth-first search, the search visits every node of the graph exactly once, declining to revisit any node that has already been explored. Thus, the collection of search trees is a spanning forest of the graph. The strongly connected components will be recovered as certain subtrees of this forest. The roots of these subtrees are called the "roots" of the strongly connected components. Any node of a strongly connected component might serve as the root, if it happens to be the first node of the component that is discovered by the search.

Stack invariant

The nodes are placed on a stack in the order in which they are visited. When the depth-first search recursively explores a node v and its descendants, those nodes are not all necessarily popped from the stack before this recursive call returns. The crucial invariant property is that a node remains on the stack after exploration if and only if it has a path to some node earlier on the stack.

At the end of the call that explores v and its descendants, we know whether v itself has a path to any node earlier on the stack. If so, the call returns, leaving v on the stack to preserve the invariant. If not, then v must be the root of its strongly connected component, which consists of v together with any later nodes on the stack (such nodes all have paths back to v but not to any earlier node, because if they had paths to earlier nodes then v would also have paths to earlier nodes which is false). This entire component is then popped from the stack and returned, again preserving the invariant.

Bookkeeping

Each node v is assigned a unique integer $v.\text{index}$, which numbers the nodes consecutively in the order in which they are discovered. It also maintains a value $v.\text{lowlink}$ that represents (roughly speaking) the smallest index of any node known to be reachable from v , including v itself. Therefore v must be left on the stack if $v.\text{lowlink} < v.\text{index}$, whereas v must be removed as the root of a strongly connected component if $v.\text{lowlink} == v.\text{index}$. The value $v.\text{lowlink}$ is computed during the depth-first search from v , as this finds the nodes that are reachable from v .

The algorithm in pseudocode

```

algorithm tarjan is
    input: graph  $G = (V, E)$ 
    output: set of strongly connected components (sets of vertices)

    index := 0
     $S$  := empty
    for each  $v$  in  $V$  do
        if ( $v.\text{index}$  is undefined) then
            strongconnect( $v$ )
        end if
    end for

    function strongconnect( $v$ )
        // Set the depth index for  $v$  to the smallest unused index
         $v.\text{index}$  := index
         $v.\text{lowlink}$  := index
        index := index + 1
         $S.\text{push}(v)$ 

        // Consider successors of  $v$ 
        for each ( $v$ ,  $w$ ) in  $E$  do
            if ( $w.\text{index}$  is undefined) then
                // Successor  $w$  has not yet been visited; recurse on it
                strongconnect( $w$ )
                 $v.\text{lowlink}$  := min( $v.\text{lowlink}$ ,  $w.\text{lowlink}$ )
            else if ( $w$  is in  $S$ ) then
                // Successor  $w$  is in stack  $S$  and hence in the current SCC
                 $v.\text{lowlink}$  := min( $v.\text{lowlink}$ ,  $w.\text{index}$ )
            end if
        end for

        // If  $v$  is a root node, pop the stack and generate an SCC
        if ( $v.\text{lowlink} = v.\text{index}$ ) then
            start a new strongly connected component
            repeat
                 $w := S.\text{pop}()$ 
                add  $w$  to current strongly connected component
            repeat
        end if
    end function

```

```

until ( $w = v$ )
    output the current strongly connected component
end if
end function

```

The `index` variable is the depth-first search node number counter. `S` is the node stack, which starts out empty and stores the history of nodes explored but not yet committed to a strongly connected component. Note that this is not the normal depth-first search stack, as nodes are not popped as the search returns up the tree; they are only popped when an entire strongly connected component has been found.

The outermost loop searches each node that has not yet been visited, ensuring that nodes which are not reachable from the first node are still eventually traversed. The function `strongconnect` performs a single depth-first search of the graph, finding all successors from the node `v`, and reporting all strongly connected components of that subgraph.

When each node finishes recursing, if its `lowlink` is still set to its `index`, then it is the root node of a strongly connected component, formed by all of the nodes above it on the stack. The algorithm pops the stack up to and including the current node, and presents all of these nodes as a strongly connected component.

Remarks

- Complexity: The Tarjan procedure is called once for each node; the `forall` statement considers each edge at most twice. The algorithm's running time is therefore linear in the number of edges and nodes in G , i.e. $O(|V| + |E|)$.
- The test for whether `w` is on the stack should be done in constant time, for example, by testing a flag stored on each node that indicates whether it is on the stack.
- While there is nothing special about the order of the nodes within each strongly connected component, one useful property of the algorithm is that no strongly connected component will be identified before any of its successors. Therefore, the order in which the strongly connected components are identified constitutes a reverse topological sort of the DAG formed by the strongly connected components.
- Tarjan's algorithm was mentioned as one of his favorite implementations by Knuth appearing in his book *The Stanford GraphBase*, pages 512–519. He considered this as one of the most beautiful algorithms with a quote
The data structures that he devised for this problem fit together in an amazingly beautiful way, so that the quantities you need to look at while exploring a directed graph are always magically at your fingertips. And his algorithm also does topological sorting as a byproduct.

References

External links

- Description of Tarjan's Algorithm (<http://www.ics.uci.edu/~eppstein/161/960220.html#sca>)
- Implementation of Tarjan's Algorithm in .NET (<http://stackoverflow.com/questions/6643076/tarjan-cycle-detection-help-c#sca>)
- Implementation of Tarjan's Algorithm in .NET (GitHub) (<https://github.com/danielrbradley/CycleDetection>)
- Another implementation of Tarjan's Algorithm in Python (<https://github.com/bwesterb/py-tarjan/>)
- Implementation of Tarjan's Algorithm in Javascript (<https://gist.github.com/1440602>)
- Implementation of Tarjan's Algorithm in Clojure (<http://clj-me.cgrand.net/2013/03/18/tarjans-strongly-connected-components-algorithm/>)

Path-based strong component algorithm

In graph theory, the strongly connected components of a directed graph may be found using an algorithm that uses depth-first search in combination with two stacks, one to keep track of the vertices in the current component and the second to keep track of the current search path. Versions of this algorithm have been proposed by Purdom (1970), Munro (1971), Dijkstra (1976), Cheriyan & Mehlhorn (1996), and Gabow (2000); of these, Dijkstra's version was the first to achieve linear time.^[1]

Description

The algorithm performs a depth-first search of the given graph G , maintaining as it does two stacks S and P . Stack S contains all the vertices that have not yet been assigned to a strongly connected component, in the order in which the depth-first search reaches the vertices. Stack P contains vertices that have not yet been determined to belong to different strongly connected components from each other. It also uses a counter C of the number of vertices reached so far, which it uses to compute the preorder numbers of the vertices.

When the depth-first search reaches a vertex v , the algorithm performs the following steps:

1. Set the preorder number of v to C , and increment C .
2. Push v onto S and also onto P .
3. For each edge from v to a neighboring vertex w :
 - If the preorder number of w has not yet been assigned, recursively search w ;
 - Otherwise, if w has not yet been assigned to a strongly connected component:
 - Repeatedly pop vertices from P until the top element of P has a preorder number less than or equal to the preorder number of w .
4. If v is the top element of P :
 - Pop vertices from S until v has been popped, and assign the popped vertices to a new component.
 - Pop v from P .

The overall algorithm consists of a loop through the vertices of the graph, calling this recursive search on each vertex that does not yet have a preorder number assigned to it.

Related algorithms

Like this algorithm, Tarjan's strongly connected components algorithm also uses depth first search together with a stack to keep track of vertices that have not yet been assigned to a component, and moves these vertices into a new component when it finishes expanding the final vertex of its component. However, in place of the second stack, Tarjan's algorithm uses a vertex-indexed array of preorder numbers, assigned in the order that vertices are first visited in the depth-first search. The preorder array is used to keep track of when to form a new component.

Notes

- [1] History of Path-based DFS for Strong Components (<http://www.cs.colorado.edu/~hal/Papers/DFS/pbDFShistory.html>), Hal Gabow, accessed 2012-04-24.

References

- Cheriyan, J.; Mehlhorn, K. (1996), "Algorithms for dense graphs and networks on the random access computer", *Algorithmica* **15**: 521–549, doi: 10.1007/BF01940880 (<http://dx.doi.org/10.1007/BF01940880>).
- Dijkstra, Edsger (1976), *A Discipline of Programming*, NJ: Prentice Hall, Ch. 25.
- Gabow, Harold N. (2000), "Path-based depth-first search for strong and biconnected components", *Information Processing Letters* **74** (3-4): 107–114, doi: 10.1016/S0020-0190(00)00051-X ([http://dx.doi.org/10.1016/S0020-0190\(00\)00051-X](http://dx.doi.org/10.1016/S0020-0190(00)00051-X)), MR 1761551 (<http://www.ams.org/mathscinet-getitem?mr=1761551>).
- Munro, Ian (1971), "Efficient determination of the transitive closure of a directed graph", *Information Processing Letters* **1**: 56–58, doi: 10.1016/0020-0190(71)90006-8 ([http://dx.doi.org/10.1016/0020-0190\(71\)90006-8](http://dx.doi.org/10.1016/0020-0190(71)90006-8)).
- Purdom, P., Jr. (1970), "A transitive closure algorithm", *BIT* **10**: 76–94, doi: 10.1007/bf01940892 (<http://dx.doi.org/10.1007/bf01940892>).
- Sedgewick, R. (2004), "19.8 Strong Components in Digraphs", *Algorithms in Java, Part 5 – Graph Algorithms* (3rd ed.), Cambridge MA: Addison-Wesley, pp. 205–216.

Kosaraju's strongly connected components algorithm

In computer science, **Kosaraju's algorithm** (also known as the **Kosaraju–Sharir algorithm**) is a linear time algorithm to find the strongly connected components of a directed graph. Aho, Hopcroft and Ullman credit it to an unpublished paper from 1978 by S. Rao Kosaraju. The same algorithm was independently discovered by Micha Sharir and published by him in 1981. It makes use of the fact that the transpose graph (the same graph with the direction of every edge reversed) has exactly the same strongly connected components as the original graph.

The algorithm

Kosaraju's algorithm works as follows:

- Let G be a directed graph and S be an empty stack.
- While S does not contain all vertices:
 - Choose an arbitrary vertex v not in S . Perform a depth-first search starting at v . Each time that depth-first search finishes expanding a vertex u , push u onto S .
- Reverse the directions of all arcs to obtain the transpose graph.
- While S is nonempty:
 - Pop the top vertex v from S . Perform a depth-first search starting at v in the transpose graph. The set of visited vertices will give the strongly connected component containing v ; record this and remove all these vertices from the graph G and the stack S . Equivalently, breadth-first search (BFS) can be used instead of depth-first search.

Complexity

Provided the graph is described using an adjacency list, Kosaraju's algorithm performs two complete traversals of the graph and so runs in $\Theta(V+E)$ (linear) time, which is asymptotically optimal because there is a matching lower bound (any algorithm must examine all vertices and edges). It is the conceptually simplest efficient algorithm, but is not as efficient in practice as Tarjan's strongly connected components algorithm and the path-based strong component algorithm, which perform only one traversal of the graph.

If the graph is represented as an adjacency matrix, the algorithm requires $O(V^2)$ time.

References

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, 3rd edition. The MIT Press, 2009. ISBN 0-262-03384-4.
- Micha Sharir. A strong connectivity algorithm and its applications to data flow analysis. *Computers and Mathematics with Applications* 7(1):67–72, 1981.

External links

- A description and proof of Kosaraju's Algorithm ^[1]
- Good Math, Bad Math: Computing Strongly Connected Components ^[2]

References

- [1] <http://lcm.csa.iisc.ernet.in/dsa/node171.html>
[2] <http://scienceblogs.com/goodmath/2007/10/30/computing-strongly-connected-c/>

Reachability

In graph theory, **reachability** refers to the ability to get from one vertex to another within a graph. We say that a vertex s can reach a vertex t (or that t is reachable from s) if there exists a sequence of adjacent vertices (i.e. a path) which starts with s and ends with t .

In an undirected graph, it is sufficient to identify the connected components, as any pair of vertices in such a graph can reach each other if and only if they belong to the same connected component. The connected components of a graph can be identified in linear time. The remainder of this article focuses on reachability in a *directed* graph setting.

Definition

For a directed graph $G = (V, E)$, with vertex set V and edge set E , the reachability relation of G is the transitive closure of E , which is to say the set of all ordered pairs (s, t) of vertices in V for which there exists a sequence of vertices $v_0 = s, v_1, v_2, \dots, v_k = t$ such that the edge (v_{i-1}, v_i) is in E for all $1 \leq i \leq k$. If G is acyclic, then its reachability relation is a partial order; any partial order may be defined in this way, for instance as the reachability relation of its transitive reduction. A noteworthy consequence of this is that since partial orders are anti-symmetric, if s can reach t , then we know that t *cannot* reach s . Intuitively, if we could travel from s to t and back to s , then G would contain a cycle, contradicting that it is acyclic. If G is directed but *not* acyclic (i.e. it contains at least one cycle), then its reachability relation will correspond to a preorder instead of a partial order.

Algorithms

Algorithms for determining reachability fall into two classes: those that require preprocessing and those that do not.

If you have only one (or a few) queries to make, it may be more efficient to forgo the use of more complex data structures and compute the reachability of the desired pair directly. This can be accomplished in linear time using algorithms such as breadth first search or iterative deepening depth-first search.

If you will be making many queries, then a more sophisticated method may be used; the exact choice of method depends on the nature of the graph being analysed. In exchange for preprocessing time and some extra storage space, we can create a data structure which can then answer reachability queries on any pair of vertices in as low as $O(1)$ time. Three different algorithms and data structures for three different, increasingly specialized situations are outlined below.

Floyd-Warshall Algorithm

The Floyd–Warshall algorithm can be used to compute the transitive closure of any directed graph, which gives rise to the reachability relation as in the definition, above.

The algorithm requires $O(|V|^3)$ time and $O(|V|^2)$ space in the worst case. This algorithm is not solely interested in reachability as it also computes the shortest path distance between all pairs of vertices. For graphs containing negative cycles, shortest paths may be undefined, but reachability between pairs can still be noted.

Thorup's Algorithm

For planar digraphs, a much faster method is available, as described by Mikkel Thorup in 2004. This method can answer reachability queries on a planar graph in $O(1)$ time after spending $O(n \log n)$ preprocessing time to create a data structure of $O(n \log n)$ size. This algorithm can also supply approximate shortest path distances, as well as route information.

The overall approach is to associate with each vertex a relatively small set of so-called separator paths such that any path from a vertex v to any other vertex w must go through at least one of the separators associated with v or w . An outline of the reachability related sections follows.

Given a graph G , the algorithm begins by organizing the vertices into layers starting from an arbitrary vertex v_0 . The layers are built in alternating steps by first considering all vertices reachable *from* the previous step (starting with just v_0) and then all vertices which reach *to* the previous step until all vertices have been assigned to a layer. By construction of the layers, every vertex appears at most two layers, and every directed path, or dipath, in G is contained within two adjacent layers L_i and L_{i+1} . Let k be the last layer created, that is, the lowest value for k such that $\bigcup_{i=0}^k = V$.

The graph is then re-expressed as a series of digraphs G_0, G_1, \dots, G_{k-1} where each $G_i = r_i \cup L_i \cup L_{i+1}$ and where r_i is the contraction of all previous levels $L_0 \dots L_{i-1}$ into a single vertex. Because every dipath appears in at most two consecutive layers, and because each G_i is formed by two consecutive layers, every dipath in G appears in its entirety in at least one G_i (and no more than 2 consecutive such graphs).

For each G_i , three separators are identified which, when removed, break the graph into three components which each contain at most $1/2$ the vertices of the original. As G_i is built from two layers of opposed dipaths, each separator may consist of up to 2 dipaths, for a total of up to 6 dipaths over all of the separators. Let S be this set of dipaths. The proof that such separators can always be found is related to the Planar Separator Theorem of Lipton and Tarjan, and these separators can be located in linear time.

For each $Q \in S$, the directed nature of Q provides for a natural indexing of its vertices from the start to the end of the path. For each vertex v in G_i , we locate the first vertex in Q reachable by v , and the last vertex in Q that reaches to v . That is, we are looking at how early into Q we can get from v , and how far we can stay in Q and still get back to v . This information is stored with each v . Then for any pair of vertices u and w , u can reach w via Q if u connects to Q earlier than w connects from Q .

Every vertex is labelled as above for each step of the recursion which builds $G_0 \dots, G_k$. As this recursion has logarithmic depth, a total of $O(\log n)$ extra information is stored per vertex. From this point, a logarithmic time query for reachability is as simple as looking over each pair of labels for a common, suitable Q . The original paper then works to tune the query time down to $O(1)$.

In summarizing the analysis of this method, first consider that the layering approach partitions the vertices so that each vertex is considered only $O(1)$ times. The separator phase of the algorithm breaks the graph into components which are at most $1/2$ the size of the original graph, resulting in a logarithmic recursion depth. At each level of the recursion, only linear work is needed to identify the separators as well as the connections possible between vertices. The overall result is $O(n \log n)$ preprocessing time with only $O(\log n)$ additional information stored for each vertex.

Kameda's Algorithm

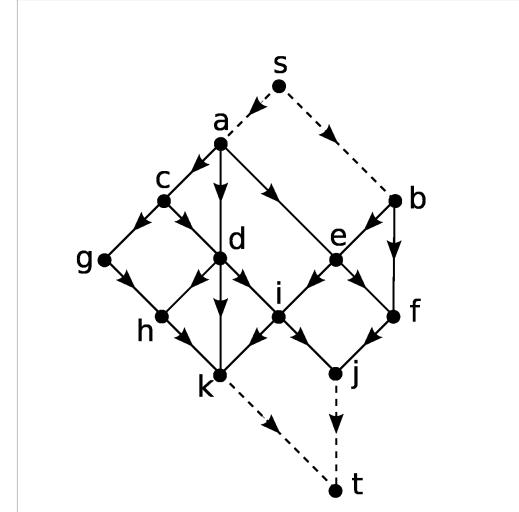
An even faster method for pre-processing, due to T. Kameda in 1975, can be used if the graph is planar, acyclic, and also exhibits the following additional properties: all 0-indegree and all 0-outdegree vertices appear on the same face (we will assume this is the outer face), and it is possible to partition the boundary of that face into two parts such that all 0-indegree vertices appear on one part, and all 0-outdegree vertices appear on the other (i.e. the two types of vertices do not alternate).

If G exhibits these properties, then we can preprocess the graph in only $O(n)$ time, and store only $O(\log n)$ extra bits per vertex, answering reachability queries for any pair of vertices in $O(1)$ time with a simple comparison.

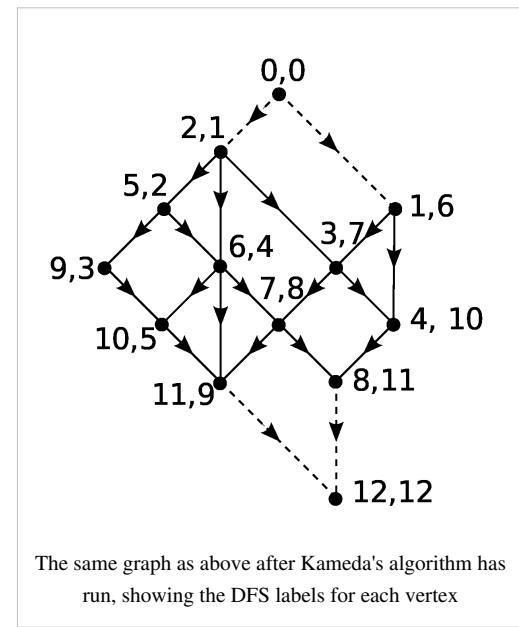
Preprocessing performs the following steps. We add a new vertex s which has an edge to each 0-indegree vertex, and another new vertex t with edges to each 0-outdegree vertex. Note that the properties of G allow us to do so while maintaining planarity, that is, there will still be no edge crossings after these additions. For each vertex we store the list of adjacencies (out-edges) in order of the planarity of the graph (for example, clockwise with respect to the graph's embedding). We then initialize a counter $i = n + 1$ and begin a Depth-First Traversal from s . During this traversal, the adjacency list of each vertex is visited from left-to-right as needed. As vertices are popped from the traversal's stack, they are labelled with the value i , and i is then decremented. Note that t is always labelled with the value $n + 1$ and s is always labelled with 0. The depth-first traversal is then repeated, but this time the adjacency list of each vertex is visited from right-to-left.

When completed, s and t , and their incident edges, are removed. Each remaining vertex stores a 2-dimensional label with values from 1 to n . Given two vertices u and v , and their labels $L(u) = (a_1, a_2)$ and $L(v) = (b_1, b_2)$, we say that $L(u) < L(v)$ if and only if $a_1 \leq b_1$, $a_2 \leq b_2$, and there exists at least one component a_1 or a_2 which is strictly less than b_1 or b_2 , respectively.

The main result of this method then states that v is reachable from u if and only if $L(u) < L(v)$, which is easily calculated in $O(1)$ time.



A suitable digraph for Kameda's method with s and t added.



The same graph as above after Kameda's algorithm has run, showing the DFS labels for each vertex

Related Problems

A related problem is to solve reachability queries with some number k of vertex failures. For example: "Can vertex u still reach vertex v even though vertices s_1, s_2, \dots, s_k have failed and can no longer be used?" A similar problem may consider edge failures rather than vertex failures, or a mix of the two. The breadth-first search technique works just as well on such queries, but constructing an efficient oracle is more challenging.

Another problem related to reachability queries is in quickly recalculating changes to reachability relationships when some portion of the graph is changed. For example, this is a relevant concern to garbage collection which needs to balance the reclamation of memory (so that it may be reallocated) with the performance concerns of the running application.

References

Transitive closure

For other uses, see Closure (disambiguation).

This article is about the transitive closure of a binary relation. For the transitive closure of a set, see transitive set#Transitive closure.

In mathematics, the **transitive closure** of a binary relation R on a set X is the transitive relation R^+ on set X such that R^+ contains R and R^+ is minimal (Lidl and Pilz 1998:337). If the binary relation itself is transitive, then the transitive closure is that same binary relation; otherwise, the transitive closure is a different relation. For example, if X is a set of airports and $x R y$ means "there is a direct flight from airport x to airport y ", then the transitive closure of R on X is the relation R^+ : "it is possible to fly from x to y in one or more flights."

Transitive relations and examples

A relation R on a set X is transitive if, for all x,y,z in X , whenever $x R y$ and $y R z$ then $x R z$. Examples of transitive relations include the equality relation on any set, the "less than or equal" relation on any linearly ordered set, and the relation "x was born before y" on the set of all people. Symbolically, this can be denoted as: if $x < y$ and $y < z$ then $x < z$.

One example of a non-transitive relation is "city x can be reached via a direct flight from city y " on the set of all cities. Simply because there is a direct flight from one city to a second city, and a direct flight from the second city to the third, does not imply there is a direct flight from the first city to the third. The transitive closure of this relation is a different relation, namely "there is a sequence of direct flights that begins at city x and ends at city y ". Every relation can be extended in a similar way to a transitive relation.

Existence and description

For any relation R , the transitive closure of R always exists. To see this, note that the intersection of any family of transitive relations is again transitive. Furthermore, there exists at least one transitive relation containing R , namely the trivial one: $X \times X$. The transitive closure of R is then given by the intersection of all transitive relations containing R .

For finite sets, we can construct the transitive closure step by step, starting from R and adding transitive edges. This gives the intuition for a general construction. For any set X , we can prove that transitive closure is given by the following expression

$$R^+ = \bigcup_{i \in \{1, 2, 3, \dots\}} R^i.$$

where R^i is the i -th power of R , defined inductively by

$$R^1 = R$$

and, for $i > 0$,

$$R^{i+1} = R \circ R^i$$

where \circ denotes composition of relations.

To show that the above definition of R^+ is the least transitive relation containing R , we show that it contains R , that it is transitive, and that it is the smallest set with both of those characteristics.

- $R \subseteq R^+$: R^+ contains all of the R^i , so in particular R^+ contains R .
- R^+ is transitive: every element of R^+ is in one of the R^i , so R^+ must be transitive by the following reasoning: if $(s_1, s_2) \in R^j$ and $(s_2, s_3) \in R^k$, then from composition's associativity, $(s_1, s_3) \in R^{j+k}$ (and thus in R^+) because of the definition of R^i .
- R^+ is minimal: Let G be any transitive relation containing R , we want to show that $R^+ \subseteq G$. It is sufficient to show that for every $i > 0$, $R^i \subseteq G$. Well, since G contains R , $R^1 \subseteq G$. And since G is transitive, whenever $R^i \subseteq G$, $R^{i+1} \subseteq G$ according to the construction of R^i and what it means to be transitive. Therefore, by induction, G contains every R^i , and thus also R^+ .

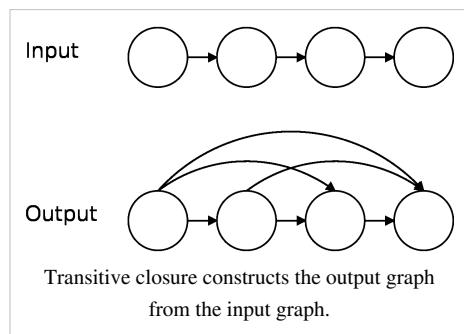
Properties

The intersection of two transitive relations is transitive.

The union of two transitive relations need not be transitive. To preserve transitivity, one must take the transitive closure. This occurs, for example, when taking the union of two equivalence relations or two preorders. To obtain a new equivalence relation or preorder one must take the transitive closure (reflexivity and symmetry—in the case of equivalence relations—are automatic).

In graph theory

In computer science, the concept of transitive closure can be thought of as constructing a data structure that makes it possible to answer reachability questions. That is, can one get from node a to node d in one or more hops? A binary relation tells you only that node a is connected to node b , and that node b is connected to node c , etc. After the transitive closure is constructed, as depicted in the following figure, in an $O(1)$ operation one may determine that node d is reachable from node a . The data structure is typically stored as a matrix, so if



matrix[1][4] = 1, then it is the case that node 1 can reach node 4 through one or more hops.

The transitive closure of a directed acyclic graph (DAG) is the reachability relation of the DAG and a strict partial order.

In logic and computational complexity

The transitive closure of a binary relation cannot, in general, be expressed in first-order logic (FO). This means that one cannot write a formula using predicate symbols R and T that will be satisfied in any model if and only if T is the transitive closure of R . In finite model theory, first-order logic (FO) extended with a transitive closure operator is usually called **transitive closure logic**, and abbreviated FO(TC) or just TC. TC is a sub-type of fixpoint logics. The fact that FO(TC) is strictly more expressive than FO was discovered by Ronald Fagin in 1974; the result was then rediscovered by Alfred Aho and Jeffrey Ullman in 1979, who proposed to use fixpoint logic as a database query language (Libkin 2004:vii). With more recent concepts of finite model theory, proof that FO(TC) is strictly more expressive than FO follows immediately from the fact that FO(TC) is not Gaifman-local (Libkin 2004:49).

In computational complexity theory, the complexity class NL corresponds precisely to the set of logical sentences expressible in TC. This is because the transitive closure property has a close relationship with the NL-complete problem STCON for finding directed paths in a graph. Similarly, the class L is first-order logic with the commutative, transitive closure. When transitive closure is added to second-order logic instead, we obtain PSPACE.

In database query languages

Further information: Hierarchical and recursive queries in SQL

Since the 1980s Oracle Database has implemented a proprietary SQL extension CONNECT BY... START WITH that allows the computation of a transitive closure as part of a declarative query. The SQL 3 (1999) standard added a more general WITH RECURSIVE construct also allowing transitive closures to be computed inside the query processor; as of 2011 the latter is implemented in IBM DB2, Microsoft SQL Server, and PostgreSQL, although not in MySQL (Benedikt and Senellart 2011:189).

Datalog also implements transitive closure computations (Silberschatz et al. 2010:C.3.6).

Algorithms

Efficient algorithms for computing the transitive closure of a graph can be found in Nuutila (1995). The simplest technique is probably the Floyd–Warshall algorithm.Wikipedia:Disputed statement The fastest worst-case methods, which are not practical, reduce the problem to matrix multiplication.

More recent research has explored efficient ways of computing transitive closure on distributed systems based on the MapReduce paradigm (Afrati et al. 2011).

References

- Lidl, R. and Pilz, G., 1998, *Applied abstract algebra*, 2nd edition, Undergraduate Texts in Mathematics, Springer, ISBN 0-387-98290-6
- Keller, U., 2004, *Some Remarks on the Definability of Transitive Closure in First-order Logic and Datalog* ^[1] (unpublished manuscript)
- Erich Grädel; Phokion G. Kolaitis; Leonid Libkin; Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, Scott Weinstein (2007). *Finite Model Theory and Its Applications*. Springer. pp. 151–152. ISBN 978-3-540-68804-4.
- Libkin, Leonid (2004), *Elements of Finite Model Theory*, Springer, ISBN 978-3-540-21202-7

- Heinz-Dieter Ebbinghaus; Jörg Flum (1999). *Finite Model Theory* (2nd ed.). Springer. pp. 123–124, 151–161, 220–235. ISBN 978-3-540-28787-2.
- Aho, A. V.; Ullman, J. D. (1979). "Universality of data retrieval languages". *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '79*. p. 110. doi:10.1145/567752.567763^[2].
- Benedikt, M.; Senellart, P. (2011). "Databases". In Blum, Edward K.; Aho, Alfred V. *Computer Science. The Hardware, Software and Heart of It*. pp. 169–229. doi:10.1007/978-1-4614-1168-0_10^[3]. ISBN 978-1-4614-1167-3.
- Nuutila, E., Efficient Transitive Closure Computation in Large Digraphs.^[4] Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series No. 74, Helsinki 1995, 124 pages. Published by the Finnish Academy of Technology. ISBN 951-666-451-2, ISSN 1237-2404, UDC 681.3.
- Abraham Silberschatz; Henry Korth; S. Sudarshan (2010). *Database System Concepts* (6th ed.). McGraw-Hill. ISBN 978-0-07-352332-3. Appendix C^[5] (online only)
- Foto N. Afrati, Vinayak Borkar, Michael Carey, Neoklis Polyzotis, Jeffrey D. Ullman, Map-Reduce Extensions and Recursive Queries^[6], EDBT 2011, March 22–24, 2011, Uppsala, Sweden, ISBN 978-1-4503-0528-0

External links

- "Transitive closure and reduction^[7]", The Stony Brook Algorithm Repository, Steven Skiena .

References

- [1] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.8266>
- [2] <http://dx.doi.org/10.1145%2F567752.567763>
- [3] http://dx.doi.org/10.1007%2F978-1-4614-1168-0_10
- [4] <http://www.cs.hut.fi/~enu/thesis.html>
- [5] <http://codex.cs.yale.edu/avi/db-book/db6/appendices-dir/c.pdf>
- [6] <http://www.edbt.org/Proceedings/2011-Uppsala/papers/edbt/a1-afrati.pdf>
- [7] <http://www.cs.sunysb.edu/~algorith/files/transitive-closure.shtml>

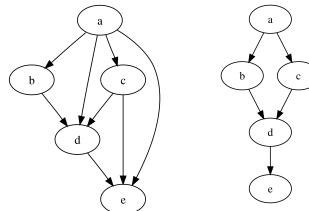
Transitive reduction

In mathematics, a **transitive reduction** of a directed graph is a graph with as few edges as possible that has the same reachability relation as the given graph. Equivalently, the given graph and its transitive reduction should have the same transitive closure as each other, and its transitive reduction should have as few edges as possible among all graphs with this property. Transitive reductions were introduced by Aho, Garey & Ullman (1972), who provided tight bounds on the computational complexity of constructing them.

If a given graph is a finite directed acyclic graph, its transitive reduction is unique, and is a subgraph of the given graph. However, uniqueness is not guaranteed for graphs with cycles, and for infinite graphs not even existence is guaranteed. The closely related concept of a **minimum equivalent graph** is a subgraph of the given graph that has the same reachability relation and as few edges as possible.^[1] For finite directed acyclic graphs, the minimum equivalent graph is the same as the transitive reduction. However, for graphs that may contain cycles, minimum equivalent graphs are NP-hard to construct, while transitive reductions can still be constructed in polynomial time. Transitive reductions can also be defined for more abstract binary relations on sets, by interpreting the pairs of the relation as arcs in a graph.

In directed acyclic graphs

The transitive reduction of a finite directed graph G is a graph with the fewest possible edges that has the same reachability relation as the original graph. That is, if there is a path from a vertex x to a vertex y in graph G , there must also be a path from x to y in the transitive reduction of G , and vice versa. The following image displays drawings of graphs corresponding to a non-transitive binary relation (on the left) and its transitive reduction (on the right).



The transitive reduction of a finite directed acyclic graph G is unique, and consists of the edges of G that form the only path between their endpoints. In particular, it is always a subgraph of the given graph. For this reason, the transitive reduction coincides with the minimum equivalent graph in this case.

In the mathematical theory of binary relations, any relation R on a set X may be thought of as a directed graph that has the set X as its vertex set and that has an arc xy for every ordered pair of elements that are related in R . In particular, this method lets partially ordered sets be reinterpreted as directed acyclic graphs, in which there is an arc xy in the graph whenever there is an order relation $x < y$ between the given pair of elements of the partial order. When the transitive reduction operation is applied to a directed acyclic graph that has been constructed in this way, it generates the covering relation of the partial order, which is frequently given visual expression by means of a Hasse diagram.

Transitive reduction has been used on networks which can be represented as directed acyclic graphs (eg. citation networks) to reveal structural differences between networks.^[2]

In graphs with cycles

In a finite graph that may have cycles, the transitive reduction is not uniquely defined: there may be more than one graph on the same vertex set that has a minimal number of edges and has the same reachability relation as the given graph. Additionally, it may be the case that none of these minimal graphs is a subgraph of the given graph. Nevertheless, it is straightforward to characterize the minimal graphs with the same reachability relation as the given graph G . If G is an arbitrary directed graph, and H is a graph with the minimum possible number of edges having the same reachability relation as G , then H consists of

- A directed cycle for each strongly connected component of G , connecting together the vertices in this component
- An edge xy for each edge XY of the transitive reduction of the condensation of G , where X and Y are two strongly connected components of G that are connected by an edge in the condensation, x is any vertex in component X , and y is any vertex in component Y . The condensation of G is a directed acyclic graph that has a vertex for every strongly connected component of G and an edge for every two components that are connected by an edge in G . In particular, because it is acyclic, its transitive reduction can be defined as in the previous section.

The total number of edges in this type of transitive reduction is then equal to the number of edges in the transitive reduction of the condensation, plus the number of vertices in nontrivial strongly connected components (components with more than one vertex).

The edges of the transitive reduction that correspond to condensation edges can always be chosen to be a subgraph of the given graph G . However, the cycle within each strongly connected component can only be chosen to be a subgraph of G if that component has a Hamiltonian cycle, something that is not always true and is difficult to check. Because of this difficulty, it is NP-hard to find the smallest subgraph of a given graph G with the same reachability (its minimum equivalent graph).

Computational complexity

As Aho et al. show, when the time complexity of graph algorithms is measured only as a function of the number n of vertices in the graph, and not as a function of the number of edges, transitive closure and transitive reduction have the same complexity. It had already been shown that transitive closure and multiplication of Boolean matrices of size $n \times n$ had the same complexity as each other,^[3] so this result put transitive reduction into the same class. The fastest known algorithms for matrix multiplication, as of 2013, take time $O(n^{2.3727})$,^[4] and this same time bound applies to transitive reduction as well.

To prove that transitive reduction is as hard as transitive closure, Aho et al. construct from a given directed acyclic graph G another graph H , in which each vertex of G is replaced by a path of three vertices, and each edge of G corresponds to an edge in H connecting the corresponding middle vertices of these paths. In addition, in the graph H , Aho et al. add an edge from every path start to every path end. In the transitive reduction of H , there is an edge from the path start for u to the path end for v , if and only if edge uv does not belong to the transitive closure of G . Therefore, if the transitive reduction of H can be computed efficiently, the transitive closure of G can be read off directly from it.

To prove that transitive reduction is as easy as transitive closure, Aho et al. rely on the already-known equivalence with Boolean matrix multiplication. They let A be the adjacency matrix of the given graph, and B be the adjacency matrix of its transitive closure (computed using any standard transitive closure algorithm). Then an edge uv belongs to the transitive reduction if and only if there is a nonzero entry in row u and column v of matrix A , and there is not a nonzero entry in the same position of the matrix product AB . In this construction, the nonzero elements of the matrix AB represent pairs of vertices connected by paths of length two or more.

When measured both in terms of the number n of vertices and the number m of edges in a directed acyclic graph, transitive reductions can also be found in time $O(nm)$, a bound that may be faster than the matrix multiplication methods for sparse graphs. To do so, collect edges (u,v) such that the longest-path distance from u to v is one,

calculating those distances by linear-time search from each possible starting vertex, u . This $O(nm)$ time bound matches the complexity of constructing transitive closures by using depth first search or breadth first search to find the vertices reachable from every choice of starting vertex, so again with these assumptions transitive closures and transitive reductions can be found in the same amount of time.

Notes

- [1] Moyles & Thompson (1969).
- [2] <http://arxiv.org/abs/1310.8224>
- [3] Aho et al. credit this result to an unpublished 1971 manuscript of Ian Munro, and to a 1970 Russian-language paper by M. E. Furman.
- [4] Williams (2012).

References

- Aho, A. V.; Garey, M. R.; Ullman, J. D. (1972), "The transitive reduction of a directed graph", *SIAM Journal on Computing* **1** (2): 131–137, doi: 10.1137/0201008 (<http://dx.doi.org/10.1137/0201008>), MR 0306032 (<http://www.ams.org/mathscinet-getitem?mr=0306032>).
- Moyles, Dennis M.; Thompson, Gerald L. (1969), "An Algorithm for Finding a Minimum Equivalent Graph of a Digraph", *Journal of the ACM* **16** (3): 455–460, doi: 10.1145/321526.321534 (<http://dx.doi.org/10.1145/321526.321534>).
- Williams, Virginia Vassilevska (2012), "Multiplying matrices faster than Coppersmith–Winograd", *Proc. 44th ACM Symposium on Theory of Computing (STOC '12)*, pp. 887–898, doi: 10.1145/2213977.2214056 (<http://dx.doi.org/10.1145/2213977.2214056>).
- Clough, James R.; Gollings, Jamie; Loach, Tamar V.; Evans, Tim S. (2014), *Transitive reduction of citation networks*, doi: 10.6084/m9.figshare.987091 (<http://dx.doi.org/10.6084/m9.figshare.987091>).

External links

- Weisstein, Eric W., "Transitive Reduction" (<http://mathworld.wolfram.com/TransitiveReduction.html>), *MathWorld*.

Application: 2-satisfiability

In computer science, **2-satisfiability** (abbreviated as 2-SAT or just 2SAT) is the problem of determining whether a collection of two-valued (Boolean or binary) variables with constraints on pairs of variables can be assigned values satisfying all the constraints. It is a special case of the general Boolean satisfiability problem, which can involve constraints on more than two variables, and of constraint satisfaction problems, which can allow more than two choices for the value of each variable. But in contrast to those problems, which are NP-complete, it has a known polynomial time solution. Instances of the 2-satisfiability problem are typically expressed as 2-CNF or **Krom formulas**.

Problem representations

A 2-SAT problem may be described using a Boolean expression with a special restricted form: a conjunction of disjunctions (and of ors), where each disjunction (or operation) has two arguments that may either be variables or the negations of variables. The variables or their negations appearing in this formula are known as terms and the disjunctions of pairs of terms are known as clauses. For example, the following formula is in conjunctive normal form, with seven variables and eleven clauses:

$$\begin{aligned} & (x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge \\ & (x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge \\ & (x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6). \end{aligned}$$

The 2-satisfiability problem is to find a truth assignment to these variables that makes a formula of this type true: we must choose whether to make each of the variables true or false, so that every clause has at least one term that becomes true. For the expression shown above, one possible satisfying assignment is the one that sets all seven of the variables to true. There are also 15 other ways of setting all the variables so that the formula becomes true. Therefore, the 2-SAT instance represented by this expression is satisfiable.

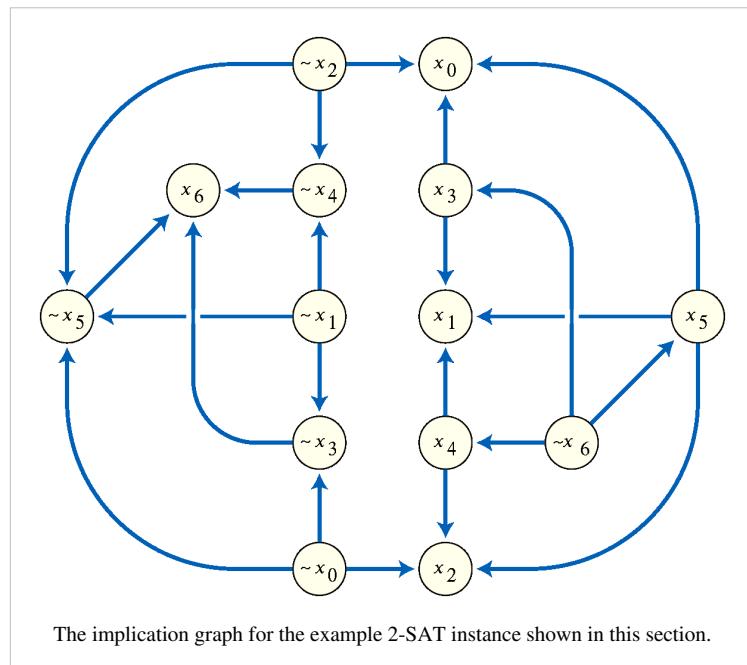
Formulas with the form described above are known as 2-CNF formulas; the "2" in this name stands for the number of terms per clause, and "CNF" stands for conjunctive normal form, a type of Boolean expression in the form of a conjunction of disjunctions. They are also called Krom formulas, after the work of UC Davis mathematician Melvin R. Krom, whose 1967 paper was one of the earliest works on the 2-satisfiability problem.

Each clause in a 2-CNF formula is logically equivalent to an implication from one variable or negated variable to the other. For example,

$$(x_0 \vee \neg x_3) \equiv (\neg x_0 \Rightarrow \neg x_3) \equiv (x_3 \Rightarrow x_0).$$

Because of this equivalence between these different types of operation, a 2-satisfiability instance may also be written in implicative normal form, in which we replace each or operation in the conjunctive normal form by both of the two implications to which it is equivalent.

A third, more graphical way of describing a 2-satisfiability instance is as an implication graph. An implication graph is a directed graph in which there is one vertex per variable or negated variable, and an edge connecting one vertex to



another whenever the corresponding variables are related by an implication in the implicative normal form of the instance. An implication graph must be a skew-symmetric graph, meaning that the undirected graph formed by forgetting the orientations of its edges has a symmetry that takes each variable to its negation and reverses the orientations of all of the edges.

Algorithms

Several algorithms are known for solving the 2-satisfiability problem; the most efficient of them take linear time.

Resolution and transitive closure

Krom (1967) described the following polynomial time decision procedure for solving 2-satisfiability instances.

Suppose that a 2-satisfiability instance contains two clauses that both use the same variable x , but that x is negated in one clause and not in the other. Then we may combine the two clauses to produce a third clause, having the two other terms in the two clauses; this third clause must also be satisfied whenever the first two clauses are both satisfied. For instance, we may combine the clauses $(a \vee b)$ and $(\neg b \vee \neg c)$ in this way to produce the clause $(a \vee \neg c)$. In terms of the implicative form of a 2-CNF formula, this rule amounts to finding two implications $\neg a \Rightarrow b$ and $b \Rightarrow \neg c$, and inferring by transitivity a third implication $\neg a \Rightarrow \neg c$.

Krom writes that a formula is *consistent* if repeated application of this inference rule cannot generate both the clauses $(x \vee x)$ and $(\neg x \vee \neg x)$, for any variable x . As he proves, a 2-CNF formula is satisfiable if and only if it is consistent. For, if a formula is not consistent, it is not possible to satisfy both of the two clauses $(x \vee x)$ and $(\neg x \vee \neg x)$ simultaneously. And, if it is consistent, then the formula can be extended by repeatedly adding one clause of the form $(x \vee x)$ or $(\neg x \vee \neg x)$ at a time, preserving consistency at each step, until it includes such a clause for every variable. At each of these extension steps, one of these two clauses may always be added while preserving consistency, for if not then the other clause could be generated using the inference rule. Once all variables have a clause of this form in the formula, a satisfying assignment of all of the variables may be generated by setting a variable x to true if the formula contains the clause $(x \vee x)$ and setting it to false if the formula contains the clause $(\neg x \vee \neg x)$. If there were a clause not satisfied by this assignment, i.e., one in which both variables appeared with sign opposite to their appearances in the added clauses, it would be possible to resolve this clause with one to reverse the sign of that variable, and then to resolve it with the original clause to produce a clause of the other variable in double with the sign it held in the original clause. Since the formula is known to have remained consistent, this is impossible, so the assignment must satisfy the original formula as well. Krom was concerned primarily with completeness of systems of inference rules, rather than with the efficiency of algorithms. However, his method leads to a polynomial time bound for solving 2-satisfiability problems. By grouping together all of the clauses that use the same variable, and applying the inference rule to each pair of clauses, it is possible to find all inferences that are possible from a given 2-CNF instance, and to test whether it is consistent, in total time $O(n^3)$, where n is the number of variables in the instance: for each variable, there may be $O(n^2)$ pairs of clauses involving that variable, to which the inference rule may be applied. Thus, it is possible to determine whether a given 2-CNF instance is satisfiable in time $O(n^3)$. Because finding a satisfying assignment using Krom's method involves a sequence of $O(n)$ consistency checks, it would take time $O(n^4)$. Even, Itai & Shamir (1976) quote a faster time bound of $O(n^2)$ for this algorithm, based on more careful ordering of its operations. Nevertheless, even this smaller time bound was greatly improved by the later linear time algorithms of Even, Itai & Shamir (1976) and Aspvall, Plass & Tarjan (1979).

In terms of the implication graph of the 2-satisfiability instance, Krom's inference rule can be interpreted as constructing the transitive closure of the graph. As Cook (1971) observes, it can also be seen as an instance of the Davis–Putnam algorithm for solving satisfiability problems using the principle of resolution. Its correctness follows from the more general correctness of the Davis–Putnam algorithm, and its polynomial time bound is clear since each resolution step increases the number of clauses in the instance, which is upper bounded by a quadratic function of the

number of variables.

Limited backtracking

Even, Itai & Shamir (1976) describe a technique involving limited backtracking for solving constraint satisfaction problems with binary variables and pairwise constraints; they apply this technique to a problem of classroom scheduling, but they also observe that it applies to other problems including 2-SAT.

The basic idea of their approach is to build a partial truth assignment, one variable at a time. Certain steps of the algorithms are "choice points", points at which a variable can be given either of two different truth values, and later steps in the algorithm may cause it to backtrack to one of these choice points. However, only the most recent choice can be backtracked over; all choices made earlier than the most recent one are permanent.

Initially, there is no choice point, and all variables are unassigned. At each step, the algorithm chooses the variable whose value to set, as follows:

- If there is a clause in which both of the variables are set, in a way that falsifies the clause, then the algorithm backtracks to its most recent choice point, undoing the assignments it made since that choice, and reverses the decision made at that choice. If no choice point has been reached, or if the algorithm has already backtracked over the most recent choice point, then it aborts the search and reports that the input 2-CNF formula is unsatisfiable.
- If there is a clause in which one of the variables has been set, and the clause may still become either true or false, then the other variable is set in a way that forces the clause to become true.
- If all clauses are either guaranteed to be true for the current assignment or have two unset variables, then the algorithm creates a choice point and sets one of the unassigned variables to an arbitrarily chosen value.

Intuitively, the algorithm follows all chains of inference after making each of its choices; this either leads to a contradiction and a backtracking step, or, if no contradiction is derived, it follows that the choice was a correct one that leads to a satisfying assignment. Therefore, the algorithm either correctly finds a satisfying assignment or it correctly determines that the input is unsatisfiable.

Even et al. did not describe in detail how to implement this algorithm efficiently; they state only that by "using appropriate data structures in order to find the implications of any decision", each step of the algorithm (other than the backtracking) can be performed quickly. However, some inputs may cause the algorithm to backtrack many times, each time performing many steps before backtracking, so its overall complexity may be nonlinear. To avoid this problem, they modify the algorithm so that, after reaching each choice point, it tests in parallel both alternative assignments for the variable set at the choice point, interleaving both parallel tests to produce a sequential algorithm. As soon as one of these two parallel tests reaches another choice point, the other parallel branch is aborted. In this way, the total time spent performing both parallel tests is proportional to the size of the portion of the input formula whose values are permanently assigned. As a result, the algorithm takes linear time in total.

Strongly connected components

Aspvall, Plass & Tarjan (1979) found a simpler linear time procedure for solving 2-satisfiability instances, based on the notion of strongly connected components from graph theory.

Two vertices in a directed graph are said to be strongly connected to each other if there is a directed path from one to the other and vice versa. This is an equivalence relation, and the vertices of the graph may be partitioned into strongly connected components, subsets within which every two vertices are strongly connected. There are several efficient linear time algorithms for finding the strongly connected components of a graph, based on depth first search: Tarjan's strongly connected components algorithm and the path-based strong component algorithm^[1] each perform a single depth first search. Kosaraju's algorithm performs two depth first searches, but is very simple.

In terms of the implication graph, two terms belong to the same strongly connected component whenever there exist chains of implications from one term to the other and vice versa. Therefore, the two terms must have the same value

in any satisfying assignment to the given 2-satisfiability instance. In particular, if a variable and its negation both belong to the same strongly connected component, the instance cannot be satisfied, because it is impossible to assign both of these terms the same value. As Aspvall et al. showed, this is a necessary and sufficient condition: a 2-CNF formula is satisfiable if and only if there is no variable that belongs to the same strongly connected component as its negation.

This immediately leads to a linear time algorithm for testing satisfiability of 2-CNF formulae: simply perform a strong connectivity analysis on the implication graph and check that each variable and its negation belong to different components. However, as Aspvall et al. also showed, it also leads to a linear time algorithm for finding a satisfying assignment, when one exists. Their algorithm performs the following steps:

- Construct the implication graph of the instance, and find its strongly connected components using any of the known linear-time algorithms for strong connectivity analysis.
- Check whether any strongly connected component contains both a variable and its negation. If so, report that the instance is not satisfiable and halt.
- Construct the condensation of the implication graph, a smaller graph that has one vertex for each strongly connected component, and an edge from component i to component j whenever the implication graph contains an edge uv such that u belongs to component i and v belongs to component j . The condensation is automatically a directed acyclic graph and, like the implication graph from which it was formed, it is skew-symmetric.
- Topologically order the vertices of the condensation. In practice this may be efficiently achieved as a side effect of the previous step, as components are generated by Kosaraju's algorithm in topological order and by Tarjan's algorithm in reverse topological order.
- For each component in this order, if its variables do not already have truth assignments, set all the terms in the component to be false. This also causes all of the terms in the complementary component to be set to true.

Due to the topological ordering, when a term x is set to false, all terms that lead to it via a chain of implications will themselves already have been set to false. Symmetrically, when a term is set to true, all terms that can be reached from it via a chain of implications will already have been set to true. Therefore, the truth assignment constructed by this procedure satisfies the given formula, which also completes the proof of correctness of the necessary and sufficient condition identified by Aspvall et al.

As Aspvall et al. show, a similar procedure involving topologically ordering the strongly connected components of the implication graph may also be used to evaluate fully quantified Boolean formulae in which the formula being quantified is a 2-CNF formula.

Applications

Conflict-free placement of geometric objects

A number of exact and approximate algorithms for the automatic label placement problem are based on 2-satisfiability. This problem concerns placing textual labels on the features of a diagram or map. Typically, the set of possible locations for each label is highly constrained, not only by the map itself (each label must be near the feature it labels, and must not obscure other features), but by each other: two labels will be illegible if they overlap each other. In general, label placement is an NP-hard problem. However, if each feature has only two possible locations for its label (say, extending to the left and to the right of the feature) then it may be solved in polynomial time. For, in this case, one may create a 2-satisfiability instance that has a variable for each label and constraints preventing each pair of labels from being assigned overlapping positions. If the labels are all congruent rectangles, the corresponding 2-SAT instance can be shown to have only linearly many constraints, leading to near-linear time algorithms for finding a labeling. Poon, Zhu & Chin (1998) describe a map labeling problem in which each label is a rectangle that may be placed in one of three positions with respect to a line segment that it labels: it may have the segment as one of its sides, or it may be centered on the segment. They represent these three positions using two

binary variables in such a way that, again, testing the existence of a valid labeling becomes a 2-SAT problem.

Formann & Wagner (1991) use this observation as part of an approximation algorithm for the problem of finding square labels of the largest possible size for a given set of points, with the constraint that each label has one of its corners on the point that it labels. To find a labeling with a given size, they eliminate squares that, if doubled, would overlap another points, and they eliminate points that can be labeled in a way that cannot possibly overlap with another point's label, and they show that the remaining points have only two possible label placements, allowing the 2-SAT approach to be used. By searching for the largest size that leads to a solvable 2-SAT instance, they find a solution with approximation ratio at most two. Similarly, if each label is rectangular and must be placed in such a way that the point it labels is somewhere along its bottom edge, then using 2-SAT to find the optimal solution in which the label has the point on a bottom corner leads to an approximation ratio of at most two.

Similar reductions to 2-satisfiability have been applied to other geometric placement problems. In graph drawing, if the vertex locations are fixed and each edge must be drawn as a circular arc with one of two possible locations (for instance as an arc diagram), then the problem of choosing which arc to use for each edge in order to avoid crossings is a 2SAT problem with a variable for each edge and a constraint for each pair of placements that would lead to a crossing. However, in this case it is possible to speed up the solution, compared to an algorithm that builds and then searches an explicit representation of the implication graph, by searching the graph implicitly. In VLSI integrated circuit design, if a collection of modules must be connected by wires that can each bend at most once, then again there are two possible routes for the wires, and the problem of choosing which of these two routes to use, in such a way that all wires can be routed in a single layer of the circuit, can be solved as a 2SAT instance.

Boros et al. (1999) consider another VLSI design problem: the question of whether or not to mirror-reverse each module in a circuit design. This mirror reversal leaves the module's operations unchanged, but it changes the order of the points at which the input and output signals of the module connect to it, possibly changing how well the module fits into the rest of the design. Boros *et al.* consider a simplified version of the problem in which the modules have already been placed along a single linear channel, in which the wires between modules must be routed, and there is a fixed bound on the density of the channel (the maximum number of signals that must pass through any cross-section of the channel). They observe that this version of the problem may be solved as a 2-SAT instance, in which the constraints relate the orientations of pairs of modules that are directly across the channel from each other; as a consequence, the optimal density may also be calculated efficiently, by performing a binary search in which each step involves the solution of a 2-SAT instance.

Data clustering

One way of clustering a set of data points in a metric space into two clusters is to choose the clusters in such a way as to minimize the sum of the diameters of the clusters, where the diameter of any single cluster is the largest distance between any two of its points; this is preferable to minimizing the maximum cluster size, which may lead to very similar points being assigned to different clusters. If the target diameters of the two clusters are known, a clustering that achieves those targets may be found by solving a 2-satisfiability instance. The instance has one variable per point, indicating whether that point belongs to the first cluster or the second cluster. Whenever any two points are too far apart from each other for both to belong to the same cluster, a clause is added to the instance that prevents this assignment.

The same method also can be used as a subroutine when the individual cluster diameters are unknown. To test whether a given sum of diameters can be achieved without knowing the individual cluster diameters, one may try all maximal pairs of target diameters that add up to at most the given sum, representing each pair of diameters as a 2-satisfiability instance and using a 2-satisfiability algorithm to determine whether that pair can be realized by a clustering. To find the optimal sum of diameters one may perform a binary search in which each step is a feasibility test of this type. The same approach also works to find clusterings that optimize other combinations than sums of the cluster diameters, and that use arbitrary dissimilarity numbers (rather than distances in a metric space) to measure the

size of a cluster. The time bound for this algorithm is dominated by the time to solve a sequence of 2-SAT instances that are closely related to each other, and Ramnath (2004) shows how to solve these related instances more quickly than if they were solved independently from each other, leading to a total time bound of $O(n^3)$ for the sum-of-diameters clustering problem.

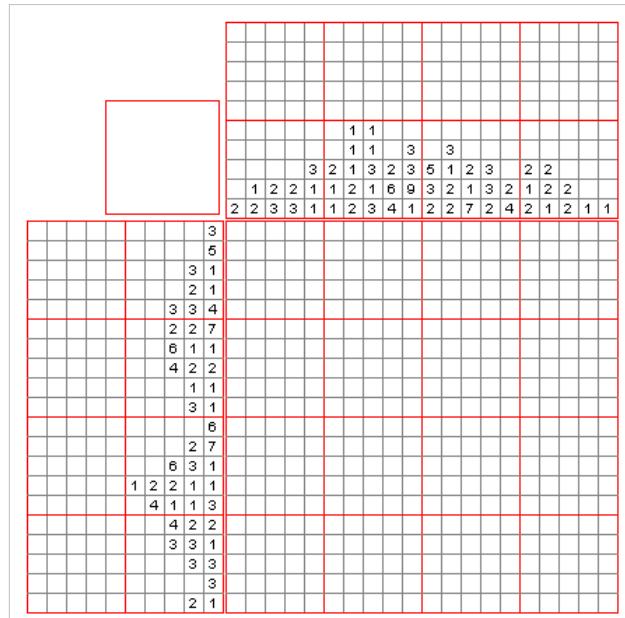
Scheduling

Even, Itai & Shamir (1976) consider a model of classroom scheduling in which a set of n teachers must be scheduled to teach each of m cohorts of students; the number of hours per week that teacher i spends with cohort j is described by entry R_{ij} of a matrix R given as input to the problem, and each teacher also has a set of hours during which he or she is available to be scheduled. As they show, the problem is NP-complete, even when each teacher has at most three available hours, but it can be solved as an instance of 2-satisfiability when each teacher only has two available hours. (Teachers with only a single available hour may easily be eliminated from the problem.) In this problem, each variable v_{ij} corresponds to an hour that teacher i must spend with cohort j , the assignment to the variable specifies whether that hour is the first or the second of the teacher's available hours, and there is a 2-SAT clause preventing any conflict of either of two types: two cohorts assigned to a teacher at the same time as each other, or one cohort assigned to two teachers at the same time.

Miyashiro & Matsui (2005) apply 2-satisfiability to a problem of sports scheduling, in which the pairings of a round-robin tournament have already been chosen and the games must be assigned to the teams' stadiums. In this problem, it is desirable to alternate home and away games to the extent possible, avoiding "breaks" in which a team plays two home games in a row or two away games in a row. At most two teams can avoid breaks entirely, alternating between home and away games; no other team can have the same home-away schedule as these two, because then it would be unable to play the team with which it had the same schedule. Therefore, an optimal schedule has two breakless teams and a single break for every other team. Once one of the breakless teams is chosen, one can set up a 2-satisfiability problem in which each variable represents the home-away assignment for a single team in a single game, and the constraints enforce the properties that any two teams have a consistent assignment for their games, that each team have at most one break before and at most one break after the game with the breakless team, and that no team has two breaks. Therefore, testing whether a schedule admits a solution with the optimal number of breaks can be done by solving a linear number of 2-satisfiability problems, one for each choice of the breakless team. A similar technique also allows finding schedules in which every team has a single break, and maximizing rather than minimizing the number of breaks (to reduce the total mileage traveled by the teams).

Discrete tomography

Tomography is the process of recovering shapes from their cross-sections. In discrete tomography, a simplified version of the problem that has been frequently studied, the shape to be recovered is a polyomino (a subset of the squares in the two-dimensional square lattice), and the cross-sections provide aggregate information about the sets of squares in individual rows and columns of the lattice. For instance, in the popular nonogram puzzles, also known as paint by numbers or griddlers, the set of squares to be determined represents the dark pixels in a binary image, and the input given to the puzzle solver tells him or her how many consecutive blocks of dark pixels to include in each row or column of the image, and how long each of those blocks should be. In other forms of digital tomography, even less information about each row or column is given: only the total number of squares, rather than the number and length of the blocks of squares. An equivalent version of the problem is that we must recover a given 0-1 matrix given only the sums of the values in each row and in each column of the matrix.



Example of a nonogram puzzle being solved.

Although there exist polynomial time algorithms to find a matrix having given row and column sums, the solution may be far from unique: any submatrix in the form of a 2×2 identity matrix can be complemented without affecting the correctness of the solution. Therefore, researchers have searched for constraints on the shape to be reconstructed that can be used to restrict the space of solutions. For instance, one might assume that the shape is connected; however, testing whether there exists a connected solution is NP-complete. An even more constrained version that is easier to solve is that the shape is orthogonally convex: having a single contiguous block of squares in each row and column. Improving several previous solutions, Chrobak & Dürr (1999) showed how to reconstruct connected orthogonally convex shapes efficiently, using 2-SAT. The idea of their solution is to guess the indexes of rows containing the leftmost and rightmost cells of the shape to be reconstructed, and then to set up a 2-SAT problem that tests whether there exists a shape consistent with these guesses and with the given row and column sums. They use four 2-SAT variables for each square that might be part of the given shape, one to indicate whether it belongs to each of four possible "corner regions" of the shape, and they use constraints that force these regions to be disjoint, to have the desired shapes, to form an overall shape with contiguous rows and columns, and to have the desired row and column sums. Their algorithm takes time $O(m^3 n)$ where m is the smaller of the two dimensions of the input shape and n is the larger of the two dimensions. The same method was later extended to orthogonally convex shapes that might be connected only diagonally instead of requiring orthogonal connectivity.^[2]

More recently, as part of a solver for full nonogram puzzles, Batenburg and Kosters (2008, 2009) used 2-SAT to combine information obtained from several other heuristics. Given a partial solution to the puzzle, they use dynamic programming within each row or column to determine whether the constraints of that row or column force any of its squares to be white or black, and whether any two squares in the same row or column can be connected by an implication relation. They also transform the nonogram into a digital tomography problem by replacing the sequence of block lengths in each row and column by its sum, and use a maximum flow formulation to determine whether this digital tomography problem combining all of the rows and columns has any squares whose state can be determined or pairs of squares that can be connected by an implication relation. If either of these two heuristics determines the value of one of the squares, it is included in the partial solution and the same calculations are repeated. However, if

both heuristics fail to set any squares, the implications found by both of them are combined into a 2-satisfiability problem and a 2-satisfiability solver is used to find squares whose value is fixed by the problem, after which the procedure is again repeated. This procedure may or may not succeed in finding a solution, but it is guaranteed to run in polynomial time. Batenburg and Kosters report that, although most newspaper puzzles do not need its full power, both this procedure and a more powerful but slower procedure which combines this 2-SAT approach with the limited backtracking of Even, Itai & Shamir (1976) are significantly more effective than the dynamic programming and flow heuristics without 2-SAT when applied to more difficult randomly generated nonograms.^[3]

Other applications

2-satisfiability has also been applied to problems of recognizing undirected graphs that can be partitioned into an independent set and a small number of complete bipartite subgraphs, inferring business relationships among autonomous subsystems of the internet, and reconstruction of evolutionary trees.

Complexity and extensions

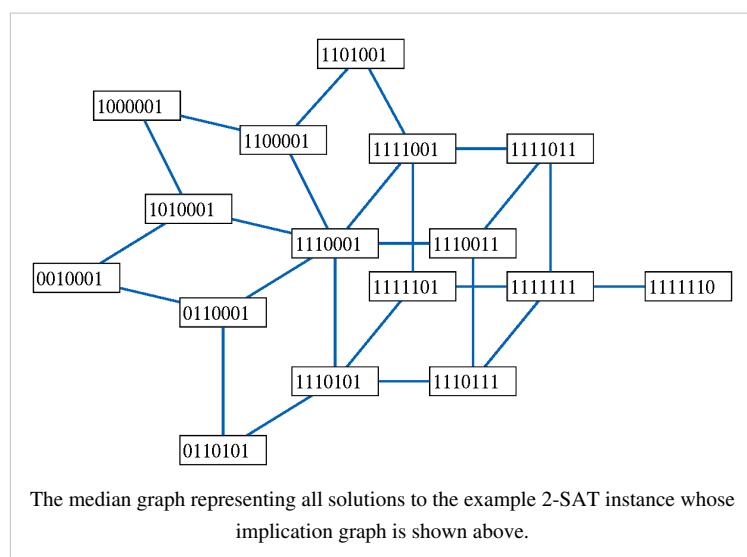
NL-completeness

A nondeterministic algorithm for determining whether a 2-satisfiability instance is *not* satisfiable, using only a logarithmic amount of writable memory, is easy to describe: simply choose (nondeterministically) a variable v and search (nondeterministically) for a chain of implications leading from v to its negation and then back to v . If such a chain is found, the instance cannot be satisfiable. By the Immerman–Szelepcsenyi theorem, it is also possible in nondeterministic logspace to verify that a satisfiable 2-SAT instance is satisfiable.

2-satisfiability is NL-complete,^[4] meaning that it is one of the "hardest" or "most expressive" problems in the complexity class **NL** of problems solvable nondeterministically in logarithmic space. Completeness here means that a deterministic Turing machine using only logarithmic space can transform any other problem in **NL** into an equivalent 2-satisfiability problem. Analogously to similar results for the more well-known complexity class **NP**, this transformation together with the Immerman–Szelepcsenyi theorem allow any problem in **NL** to be represented as a second order logic formula with a single existentially quantified predicate with clauses limited to length 2; such formulae are known as SO-Krom. Similarly, the implicative normal form can be expressed in first order logic with the addition of an operator for transitive closure.

The set of all solutions

The set of all solutions to a 2-satisfiability instance has the structure of a median graph, in which an edge corresponds to the operation of flipping the values of a set of variables that are all constrained to be equal or unequal to each other. In particular, by following edges in this way one can get from any solution to any other solution. Conversely, any median graph can be represented as the set of solutions to a 2-satisfiability instance in this way. The median of any three solutions is formed by setting each variable to the value it holds in



the majority of the three solutions; this median always forms another solution to the instance.^[5]

Feder (1994) describes an algorithm for efficiently listing all solutions to a given 2-satisfiability instance, and for solving several related problems. There also exist algorithms for finding two satisfying assignments that have the maximal Hamming distance from each other.

Counting the number of satisfying assignments

#2SAT is the problem of counting the number of satisfying assignments to a given 2-CNF formula. This counting problem is #P-complete, which implies that it is not solvable in polynomial time unless P = NP. Moreover, there is no fully polynomial randomized approximation scheme for #2SAT unless NP = RP and this even holds when the input is restricted to monotone 2-CNF formulas, i.e., 2-CNF formulas in which each literal is a positive occurrence of a variable.^[6]

The fastest known algorithm for computing the exact number of satisfying assignments to a 2SAT formula runs in time $O(1.246^n)$.

Random 2-satisfiability instances

One can form a 2-satisfiability instance at random, for a given number n of variables and m of clauses, by choosing each clause uniformly at random from the set of all possible two-variable clauses. When m is small relative to n , such an instance will likely be satisfiable, but larger values of m have smaller probabilities of being satisfiable. More precisely, if m/n is fixed as a constant $\alpha \neq 1$, the probability of satisfiability tends to a limit as n goes to infinity: if $\alpha < 1$, the limit is one, while if $\alpha > 1$, the limit is zero. Thus, the problem exhibits a phase transition at $\alpha = 1$.^[7]

Maximum-2-satisfiability

In the maximum-2-satisfiability problem (**MAX-2-SAT**), the input is a formula in conjunctive normal form with two literals per clause, and the task is to determine the maximum number of clauses that can be simultaneously satisfied by an assignment. MAX-2-SAT is NP-hard and it is a particular case of a maximum satisfiability problem.

By formulating MAX-2-SAT as a problem of finding a cut (that is, a partition of the vertices into two subsets) maximizing the number of edges that have one endpoint in the first subset and one endpoint in the second, in a graph related to the implication graph, and applying semidefinite programming methods to this cut problem, it is possible to find in polynomial time an approximate solution that satisfies at least 0.940... times the optimal number of clauses. A *balanced* MAX 2-SAT instance is an instance of MAX 2-SAT where every variable appears positively and negatively with equal weight. For this problem, one can improve the approximation ratio to $\min \left\{ (3 - \cos \theta)^{-1} (2 + (2/\pi)\theta) : \pi/2 \leq \theta \leq \pi \right\} = 0.943\dots$

If the unique games conjecture is true, then it is impossible to approximate MAX 2-SAT, balanced or not, with an approximation constant better than 0.943... in polynomial time. Under the weaker assumption that P \neq NP, the problem is only known to be inapproximable within a constant better than $21/22 = 0.95454\dots$

Various authors have also explored exponential worst-case time bounds for exact solution of MAX-2-SAT instances.^[8]

Weighted-2-satisfiability

In the weighted 2-satisfiability problem (**W2SAT**), the input is an n -variable 2SAT instance and an integer k , and the problem is to decide whether there exists a satisfying assignment in which at most k of the variables are true. One may easily encode the vertex cover problem as a W2SAT problem: given a graph G and a bound k on the size of a vertex cover, create a variable for each vertex of a graph, and for each edge uv of the graph create a 2SAT clause $u \vee v$. Then the satisfying instances of the resulting 2SAT formula encode solutions to the vertex cover problem, and there is a satisfying assignment with k true variables if and only if there is a vertex cover with k vertices. Therefore,

W2SAT is NP-complete.

Moreover, in parameterized complexity W2SAT provides a natural W[1]-complete problem, which implies that W2SAT is not fixed-parameter tractable unless this holds for all problems in W[1]. That is, it is unlikely that there exists an algorithm for W2SAT whose running time takes the form $f(k) \cdot n^{O(1)}$. Even more strongly, W2SAT cannot be solved in time $n^{o(k)}$ unless the exponential time hypothesis fails.

Quantified Boolean formulae

As well as finding the first polynomial-time algorithm for 2-satisfiability, Krom (1967) also formulated the problem of evaluating fully quantified Boolean formulae in which the formula being quantified is a 2-CNF formula. The 2-satisfiability problem is the special case of this quantified 2-CNF problem, in which all quantifiers are existential. Krom also developed an effective decision procedure for these formulae; Aspvall, Plass & Tarjan (1979) showed that it can be solved in linear time, by an extension of their technique of strongly connected components and topological ordering.

Many-valued logics

The 2-SAT problem can also be asked for propositional many-valued logics. The algorithms are not usually linear, and for some logics the problem is even NP-complete; see Hähnle (2001, 2003) for surveys.

References

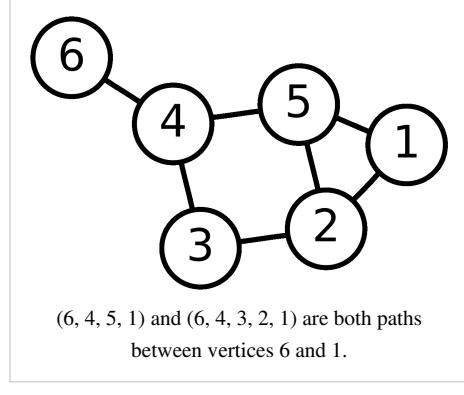
- [1] First published by . Rediscovered in 1999 by Harold N. Gabow, and published in .
- [2] ; .
- [3] ; .
- [4] , Thm. 16.3.
- [5] , to appear . . .
- [6] , Theorem 57.
- [7] ; ; .
- [8] ; ;

Shortest paths

Shortest path problem

In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

This is analogous to the problem of finding the shortest path between two intersections on a road map: the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment.



Definition

The shortest path problem can be defined for graphs whether undirected, directed, or mixed. It is defined here for undirected graphs; for directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

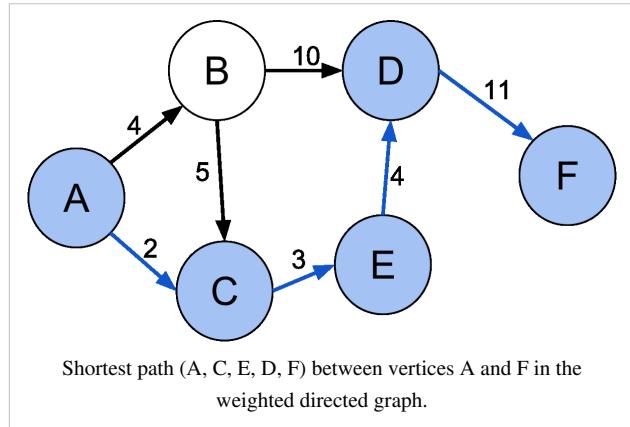
Two vertices are adjacent when they are both incident to a common edge. A path in an undirected graph is a sequence of vertices $P = (v_1, v_2, \dots, v_n) \in V \times V \times \dots \times V$ such that v_i is adjacent to v_{i+1} for $1 \leq i < n$. Such a path P is called a path of length n from v_1 to v_n . (The v_i are variables; their numbering here relates to their position in the sequence and needs not to relate to any canonical labeling of the vertices.)

Let $e_{i,j}$ be the edge incident to both v_i and v_j . Given a real-valued weight function $f : E \rightarrow \mathbb{R}$, and an undirected (simple) graph G , the shortest path from v to v' is the path $P = (v_1, v_2, \dots, v_n)$ (where $v_1 = v$ and $v_n = v'$) that over all possible n minimizes the sum $\sum_{i=1}^{n-1} f(e_{i,i+1})$. When each edge in the graph has unit

weight or $f : E \rightarrow \{1\}$, this is equivalent to finding the path with fewest edges.

The problem is also sometimes called the **single-pair shortest path problem**, to distinguish it from the following variations:

- The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
- The **single-destination shortest path problem**, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v . This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
- The **all-pairs shortest path problem**, in which we have to find shortest paths between every pair of vertices v, v' in the graph.



These generalizations have significantly more efficient algorithms than the simplistic approach of running a single-pair shortest path algorithm on all relevant pairs of vertices.

Algorithms

The most important algorithms for solving this problem are:

- Dijkstra's algorithm solves the single-source shortest path problem.
- Bellman–Ford algorithm solves the single-source problem if edge weights may be negative.
- A* search algorithm solves for single pair shortest path using heuristics to try to speed up the search.
- Floyd–Warshall algorithm solves all pairs shortest paths.
- Johnson's algorithm solves all pairs shortest paths, and may be faster than Floyd–Warshall on sparse graphs.
- Viterbi algorithm solves the shortest stochastic path problem with an additional probabilistic weight on each node.

Additional algorithms and associated evaluations may be found in Cherkassky et al.

Road networks

A road network can be considered as a graph with positive weights. The nodes represent road junctions and each edge of the graph is associated with a road segment between two junctions. The weight of an edge may correspond to the length of the associated road segment, the time needed to traverse the segment or the cost of traversing the segment. Using directed edges it is also possible to model one-way streets. Such graphs are special in the sense that some edges are more important than others for long distance travel (e.g. highways). This property has been formalized using the notion of highway dimension.^[1] There are a great number of algorithms that exploit this property and are therefore able to compute the shortest path a lot quicker than would be possible on general graphs.

All of these algorithms work in two phases. In the first phase, the graph is preprocessed without knowing the source or target node. This phase may take several days for realistic data and some techniques. The second phase is the query phase. In this phase, source and target node are known. The running time of the second phase is generally less than a second. The idea is that the road network is static, so the preprocessing phase can be done once and used for a large number of queries on the same road network.

The algorithm with the fastest known query time is called hub labeling and is able to compute shortest path on the road networks of Europe or the USA in a fraction of a microsecond.^[2] Other techniques that have been used are:

- ALT
- Arc Flags
- Contraction hierarchies
- Transit Node Routing
- Reach based Pruning
- Labeling

Single-source shortest paths

Directed unweighted graphs

Algorithm	Time complexity	Author
Breadth-first search	$O(E)$	

Directed graphs with nonnegative weights

The following table is taken from Schrijver (2004).^[3] A green background indicates an asymptotically best bound in the table.

Algorithm	Time complexity	Author
	$O(V^2 EL)$	Ford 1956
Bellman–Ford algorithm	$O(VE)$	Bellman 1958, Moore 1959
	$O(V^2 \log V)$	Dantzig 1958, Dantzig 1960, Minty (cf. Pollack & Wiegelson 1960), Whiting & Hillier 1960
Dijkstra's algorithm with list	$O(V^2)$	Leyzorek et al. 1957, Dijkstra 1959
Dijkstra's algorithm with modified binary heap	$O((E + V) \log V)$	
...
Dijkstra's algorithm with Fibonacci heap	$O(E + V \log V)$	Fredman & Tarjan 1984, Fredman & Tarjan 1987
	$O(E \log \log L)$	Johnson 1982, Karlsson & Poblete 1983
Gabow's algorithm	$O(E \log_{E/V} L)$	Gabow 1983b, Gabow 1985b
	$O(E + V \sqrt{\log L})$	Ahuja et al. 1990

This list is incomplete; you can help by expanding it ^[4].

Directed graphs with arbitrary weights

Algorithm	Time complexity	Author
Bellman–Ford algorithm	$O(VE)$	Bellman 1958, Moore 1959

This list is incomplete; you can help by expanding it ^[4].

All-pairs shortest paths

The all-pairs shortest paths problem for unweighted directed graphs was introduced by Shimbrel (1953), who observed that it could be solved by a linear number of matrix multiplications, taking a total time of $O(V^4)$.

Subsequent algorithms handle edge weights (which may possibly be negative), and are faster. The Floyd–Warshall algorithm takes $O(V^3)$ time, and Johnson's algorithm (a combination of the Bellman–Ford and Dijkstra algorithms) takes $O(VE + V^2 \log V)$.

Applications

Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like Mapquest or Google Maps. For this application fast specialized algorithms are available.

If one represents a nondeterministic abstract machine as a graph where vertices describe states and edges describe possible transitions, shortest path algorithms can be used to find an optimal sequence of choices to reach a certain goal state, or to establish lower bounds on the time needed to reach a given state. For example, if vertices represents the states of a puzzle like a Rubik's Cube and each directed edge corresponds to a single move or turn, shortest path algorithms can be used to find a solution that uses the minimum possible number of moves.

In a networking or telecommunications mindset, this shortest path problem is sometimes called the min-delay path problem and usually tied with a widest path problem. For example, the algorithm may seek the shortest (min-delay) widest path, or widest shortest (min-delay) path.

A more lighthearted application is the games of "six degrees of separation" that try to find the shortest path in graphs like movie stars appearing in the same film.

Other applications, often studied in operations research, include plant and facility layout, robotics, transportation, and VLSI design".

Related problems

For shortest path problems in computational geometry, see Euclidean shortest path.

The travelling salesman problem is the problem of finding the shortest path that goes through every vertex exactly once, and returns to the start. Unlike the shortest path problem, which can be solved in polynomial time in graphs without negative cycles, the travelling salesman problem is NP-complete and, as such, is believed not to be efficiently solvable for large sets of data (see P = NP problem). The problem of finding the longest path in a graph is also NP-complete.

The Canadian traveller problem and the stochastic shortest path problem are generalizations where either the graph isn't completely known to the mover, changes over time, or where actions (traversals) are probabilistic.

The shortest multiple disconnected path is a representation of the primitive path network within the framework of Reptation theory.

The widest path problem seeks a path so that the minimum label of any edge is as large as possible.

Linear programming formulation

There is a natural linear programming formulation for the shortest path problem, given below. It is very simple compared to most other uses of linear programs in discrete optimization, however it illustrates connections to other concepts.

Given a directed graph (V, A) with source node s , target node t , and cost w_{ij} for each edge (i, j) in A , consider the program with variables x_{ij}

$$\text{minimize } \sum_{ij \in A} w_{ij} x_{ij} \text{ subject to } x \geq 0 \text{ and for all } i, \sum_j x_{ij} - \sum_j x_{ji} = \begin{cases} 1, & \text{if } i = s; \\ -1, & \text{if } i = t; \\ 0, & \text{otherwise.} \end{cases}$$

The intuition behind this is that x_{ij} is an indicator variable for whether edge (i, j) is part of the shortest path: 1 when it is, and 0 if it is not. We wish to select the set of edges with minimal weight, subject to the constraint that this set forms a path from s to t (represented by the equality constraint: for all vertices except s and t the number of incoming and outgoing edges that are part of the path must be the same (i.e., that it should be a path from s to t).

This LP has the special property that it is integral; more specifically, every basic optimal solution (when one exists) has all variables equal to 0 or 1, and the set of edges whose variables equal 1 form an s - t dipath. See Ahuja et al. for one proof, although the origin of this approach dates back to mid-20th century.

The dual for this linear program is

$$\text{maximize } y_t - y_s \text{ subject to for all } ij, y_j - y_i \leq w_{ij}$$

and feasible duals correspond to the concept of a consistent heuristic for the A* algorithm for shortest paths. For any feasible dual y the reduced costs $w'_{ij} = w_{ij} - y_j + y_i$ are nonnegative and A* essentially runs Dijkstra's algorithm on these reduced costs.

General algebraic framework on semirings: the algebraic path problem

Many problems can be framed as a form of the shortest path for some suitably substituted notions of addition along a path and taking the minimum. The general approach to these is to consider the two operations be those of a semiring. Semiring multiplication is done along the path, and the addition is between paths. This general framework is known as the algebraic path problem.^{[5][6]}

Most of the classic shortest-path algorithms (and new ones) can be formulated as solving linear systems over such algebraic structures.

More recently, an even more general framework for solving these (and much less obviously related problems) has been developed under the banner of valuation algebras.

References

- [1] Abraham, Ittai; Fiat, Amos; Goldberg, Andrew V.; Werneck, Renato F. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms" (<http://research.microsoft.com/pubs/115272/soda10.pdf>). research.microsoft.com/pubs/115272/soda10.pdf). ACM-SIAM Symposium on Discrete Algorithms, pages 782-793, 2010.
- [2] Abraham, Ittai; Delling, Daniel; Goldberg, Andrew V.; Werneck, Renato F. research.microsoft.com/pubs/142356/HL-TR.pdf "A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks" (<http://research.microsoft.com/pubs/142356/HL-TR.pdf>). Symposium on Experimental Algorithms, pages 230-241, 2011.
- [3] Here: vol.A, sect.7.5b, p.103
- [4] http://en.wikipedia.org/w/index.php?title=Shortest_path_problem&action=edit
- [5] Mehryar Mohri, " Semiring frameworks and algorithms for shortest-distance problems (<http://www.cs.nyu.edu/~mohri/pub/jalc.pdf>)", Journal of Automata, Languages and Combinatorics, Volume 7 Issue 3, January 2002, Pages 321 - 350
- [6] <http://www.iam.unibe.ch/~run/talks/2008-06-05-Bern-Joneczy.pdf>
- Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* **16**: 87–90. MR 0102435 (<http://www.ams.org/mathscinet-getitem?mr=0102435>).
- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. "Single-Source Shortest Paths and All-Pairs Shortest Paths". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 580–642. ISBN 0-262-03293-7.
- Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" (<http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf>). *Numerische Mathematik* **1**: 269–271. doi: 10.1007/BF01386390 (<http://dx.doi.org/10.1007/BF01386390>).
- Fredman, Michael Lawrence; Tarjan, Robert E. (1984). "Fibonacci heaps and their uses in improved network optimization algorithms" (<http://www.computer.org/portal/web/csdl/doi/10.1109/SFCS.1984.715934>). 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346. doi: 10.1109/SFCS.1984.715934 (<http://dx.doi.org/10.1109/SFCS.1984.715934>). ISBN 0-8186-0591-X.
- Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (<http://portal.acm.org/citation.cfm?id=28874>). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi: 10.1145/28869.28874 (<http://dx.doi.org/10.1145/28869.28874>).

- Leyzorek, M.; Gray, R. S.; Johnson, A. A.; Ladew, W. C.; Meaker, S. R., Jr.; Petry, R. M.; Seitz, R. N. (1957). *Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — A Study of Model Techniques for Communication Systems*. Cleveland, Ohio: Case Institute of Technology.
- Moore, E. F. (1959). "The shortest path through a maze". *Proceedings of an International Symposium on the Theory of Switching (Cambridge, Massachusetts, 2–5 April 1957)*. Cambridge: Harvard University Press. pp. 285–292.
- Shimbel, Alfonso (1953). "Structural parameters of communication networks". *Bulletin of Mathematical Biophysics* **15** (4): 501–507. doi: 10.1007/BF02476438 (<http://dx.doi.org/10.1007/BF02476438>).

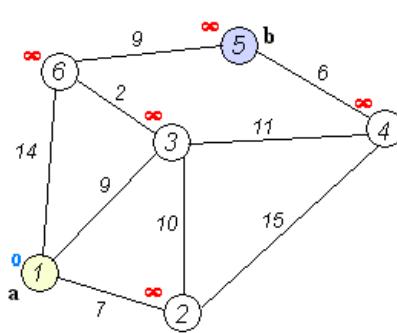
Further reading

- D. Frigioni; A. Marchetti-Spaccamela and U. Nanni (1998). "Fully dynamic output bounded single source shortest path problem" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.9856>). *Proc. 7th Annu. ACM-SIAM Symp. Discrete Algorithms*. Atlanta, GA. pp. 212–221.

Dijkstra's algorithm for single-source shortest paths with positive edge lengths

Not to be confused with Dykstra's projection algorithm.

Dijkstra's algorithm



Dijkstra's algorithm. It picks the unvisited vertex with the lowest-distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(E + V \log V)$

Graph and tree search algorithms

- $\alpha-\beta$
- A*
- B*
- Backtracking
- Beam
- Bellman–Ford
- Best-first
- Bidirectional
- Borůvka
- Branch & bound
- BFS
- British Museum
- D*
- DFS
- Depth-limited
- Dijkstra
- Edmonds
- Floyd–Warshall
- Fringe search
- Hill climbing
- IDA*
- Iterative deepening
- Johnson

• Jump point
• Kruskal
• Lexicographic BFS
• Prim
• SMA*
• Uniform-cost
Listings
• <i>Graph algorithms</i>
• <i>Search algorithms</i>
• <i>List of graph algorithms</i>
Related topics
• Dynamic programming
• Graph traversal
• Tree traversal
• Search games
• v
• t
• e [1]

Dijkstra's algorithm, conceived by computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$ (where $|V|$ is the number of vertices). The idea of this algorithm is also given in (Leyzorek et al. 1957). The implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V| \log |V|)$ (where $|E|$ is the number of edges) is due to (Fredman & Tarjan 1984). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. However, special cases can indeed be improved. For example, when arc weights are integers and bounded by a constant C , a usage of a special priority queue structure by Van Emde Boas et al. (1977) (Ahuja et al. 1990) will take a complexity of $O(|E| \log \log |C|)$. Another interesting implementation based on a combination of a new Radix-heap and the well-known Fibonacci-heap runs in $O(|E| + |V| \sqrt{\log |C|})$ complexity (Ahuja et al. 1990).

Algorithm

Let the node at which we are starting be called the **initial node**. Let the **distance of node Y** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be $6 + 2 = 8$. If *B* was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

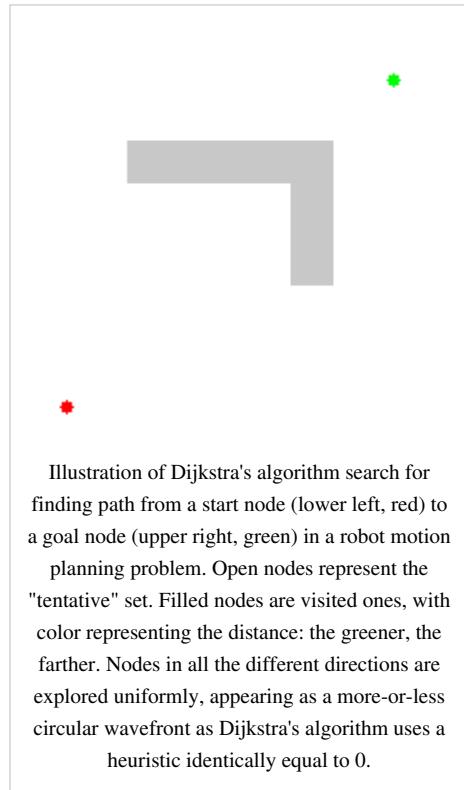


Illustration of Dijkstra's algorithm search for finding path from a start node (lower left, red) to a goal node (upper right, green) in a robot motion planning problem. Open nodes represent the "tentative" set. Filled nodes are visited ones, with color representing the distance: the greener, the farther. Nodes in all the different directions are explored uniformly, appearing as a more-or-less circular wavefront as Dijkstra's algorithm uses a heuristic identically equal to 0.

Description

*Note: For ease of understanding, this discussion uses the terms **intersection**, **road** and **map** — however, in formal notation these terms are **vertex**, **edge** and **graph**, respectively.*

Suppose you would like to find the shortest path between two intersections on a city map, a starting point and a destination. The order is conceptually simple: to start, mark the distance to every intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that intersection has not yet been *visited*; some variants of this method simply leave the intersection unlabeled. Now, at each iteration, select a *current* intersection. For the first iteration, the current intersection will be the starting point and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first), the current intersection will be the closest unvisited intersection to the starting point—this will be easy to find.

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring intersection, mark the current intersection as *visited* and select the unvisited intersection with lowest distance (from

the starting point) – or the lowest label—as the current intersection. Nodes marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

Continue this process of updating the neighboring intersections with the shortest distances, then marking the current intersection as visited and moving onto the closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection) you have determined the shortest path to it, from the starting point, and can trace your way back, following the arrows in reverse.

Of note is the fact that this algorithm makes no attempt to direct "exploration" towards the destination as one might expect. Rather, the sole consideration in determining the next "current" intersection is its distance from the starting point. This algorithm, therefore "expands outward" from the starting point, interactively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path, however, it may also reveal one of the algorithm's weaknesses: its relative slowness in some topologies.

Pseudocode

In the following algorithm, the code `u := vertex in Q with min dist[u]`, searches for the vertex `u` in the vertex set `Q` that has the least `dist[u]` value. `length(u, v)` returns the length of the edge joining (i.e. the distance between) the two neighbor-nodes `u` and `v`. The variable `alt` on line 17 is the length of the path from the root node to the neighbor node `v` if it were to go through `u`. If this path is shorter than the current shortest path recorded for `v`, that current path is replaced with this `alt` path. The `previous` array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```

1  function Dijkstra(Graph, source):
2      dist[source] := 0                                // Distance from source to source
3      for each vertex v in Graph:                      // Initializations
4          if v ≠ source
5              dist[v] := infinity                     // Unknown distance function from source to v
6              previous[v] := undefined                // Previous node in optimal path from source
7          end if
8          add v to Q                                 // All nodes initially in Q (unvisited nodes)
9      end for
10
11     while Q is not empty:                          // The main loop
12         u := vertex in Q with min dist[u]    // Source node in first case
13         remove u from Q
14
15         for each neighbor v of u:                  // where v has not yet been removed from Q.
16             alt := dist[u] + length(u, v)
17             if alt < dist[v]:                      // A shorter path to v has been found
18                 dist[v] := alt
19                 previous[v] := u
20             end if
21         end for
22     end while
23     return dist[], previous[]
24 end function
```

If we are only interested in a shortest path between vertices `source` and `target`, we can terminate the search at line 13 if `u = target`. Now we can read the shortest path from `source` to `target` by reverse iteration:

```

1  S := empty sequence
2  u := target
3  while previous[u] is defined:                                // Construct the shortest path with a stack S
4      insert u at the beginning of S                            // Push the vertex into the stack
5      u := previous[u]                                         // Traverse from target to source
6  end while

```

Now sequence `S` is the list of vertices constituting one of the shortest paths from `source` to `target`, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between `source` and `target` (there might be several different ones of the same length). Then instead of storing only a single node in each entry of `previous[]` we would store all nodes satisfying the relaxation condition. For example, if both `r` and `source` connect to `target` and both of them lie on different shortest paths through `target` (because the edge cost is the same in both cases), then we would add both `r` and `source` to `previous[target]`. When the algorithm completes, `previous[]` data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these shortest paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search.

Using a priority queue

A min-priority queue is an abstract data structure that provides 3 basic operations : `add_with_priority()`, `decrease_priority()` and `extract_min()`. As mentioned earlier, using such a data structure can lead to faster computing times than using a basic queue. Notably, Fibonacci heap (Fredman & Tarjan 1984) or Brodal queue offer optimal implementations for those 3 operations. As the algorithm is slightly different, we mention it here, in pseudo-code as well :

```

1  function Dijkstra(Graph, source):
2      dist[source] := 0                                     // Initializations
3      for each vertex v in Graph:
4          if v ≠ source
5              dist[v] := infinity                         // Unknown distance from source to v
6              previous[v] := undefined                    // Predecessor of v
7      end if
8      PQ.add_with_priority(v, dist[v])
9  end for
10
11
12  while PQ is not empty:                                 // The main loop
13      u := PQ.extract_min()                           // Remove and return best vertex
14      for each neighbor v of u:                      // where v has not yet been removed from PQ.
15          alt = dist[u] + length(u, v)
16          if alt < dist[v]
17              dist[v] := alt
18              previous[v] := u

```

```

19             PQ.decrease_priority(v, alt)
20         end if
21     end for
22 end while
23 return previous[]

```

It should be noted that other data structures can be used to achieve even faster computing times in practice.

Running time

An upper bound of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of $|E|$ and $|V|$ using big-O notation.

For any implementation of vertex set Q the running time is in $O(|E| \cdot dk_Q + |V| \cdot em_Q)$, where dk_Q and em_Q are times needed to perform decrease key and extract minimum operations in set Q , respectively.

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q . In this case, the running time is $O(|E| + |V|^2) = O(|V|^2)$.

For sparse graphs, that is, graphs with far fewer than $O(|V|^2)$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. With a self-balancing binary search tree or binary heap, the algorithm requires $\Theta((|E| + |V|) \log |V|)$ time (which is dominated by $\Theta(|E| \log |V|)$, assuming the graph is connected). To avoid $O(|V|)$ look-up in decrease-key step on a vanilla binary heap, it is necessary to maintain a supplementary index mapping each vertex to the heap's index (and keep it up to date as priority queue Q changes), making it take only $O(\log |V|)$ time instead. The Fibonacci heap improves this to $O(|E| + |V| \log |V|)$. Note that for directed acyclic graphs, it is possible to find shortest paths from a given starting vertex in linear time, by processing the vertices in a topological order, and calculating the path length for each vertex to be the minimum length obtained via any of its incoming edges.^[1]

Related problems and algorithms

The functionality of Dijkstra's original algorithm can be extended with a variety of modifications. For example, sometimes it is desirable to present solutions which are less than mathematically optimal. To obtain a ranked list of less-than-optimal solutions, the optimal solution is first calculated. A single edge appearing in the optimal solution is removed from the graph, and the optimum solution to this new graph is calculated. Each edge of the original solution is suppressed in turn and a new shortest-path calculated. The secondary solutions are then ranked and presented after the first optimal solution.

Dijkstra's algorithm is usually the working principle behind link-state routing protocols, OSPF and IS-IS being the most common ones.

Unlike Dijkstra's algorithm, the Bellman–Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex s . The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed. It is possible to adapt Dijkstra's algorithm to handle negative weight edges by combining it with the Bellman–Ford algorithm (to remove negative edges and detect negative cycles), such an algorithm is called Johnson's algorithm.

The A* algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored, if additional information is available that provides a lower bound on the "distance" to the target. This approach can be viewed from the perspective of linear programming: there is a natural linear program for computing

shortest paths, and solutions to its dual linear program are feasible if and only if they form a consistent heuristic (speaking roughly, since the sign conventions differ from place to place in the literature). This feasible dual / consistent heuristic defines a non-negative reduced cost and A* is essentially running Dijkstra's algorithm with these reduced costs. If the dual satisfies the weaker condition of admissibility, then A* is instead more akin to the Bellman–Ford algorithm.

The process that underlies Dijkstra's algorithm is similar to the greedy process used in Prim's algorithm. Prim's purpose is to find a minimum spanning tree that connects all nodes in the graph; Dijkstra is concerned with only two nodes. Prim's does not evaluate the total weight of the path from the starting node, only the individual path.

Breadth-first search can be viewed as a special-case of Dijkstra's algorithm on unweighted graphs, where the priority queue degenerates into a FIFO queue.

Fast marching method can be viewed as a continuous version of Dijkstra's algorithm which computes the geodesic distance on a triangle mesh.

Dynamic programming perspective

From a dynamic programming point of view, Dijkstra's algorithm is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the **Reaching** method.^[2]

In fact, Dijkstra's explanation of the logic behind the algorithm, namely

Problem 2. Find the path of minimum total length between two given nodes P and Q .

We use the fact that, if R is a node on the minimal path from P to Q , knowledge of the latter implies the knowledge of the minimal path from P to R .

is a paraphrasing of Bellman's famous Principle of Optimality in the context of the shortest path problem.

Notes

[1] http://www.boost.org/doc/libs/1_44_0/libs/graph/doc/dag_shortest_paths.html

[2] Online version of the paper with interactive computational modules. (http://www.ifors.ms.unimelb.edu.au/tutorial/dijkstra_new/index.html)

References

- Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" (<http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf>). *Numerische Mathematik* **1**: 269–271. doi: 10.1007/BF01386390 (<http://dx.doi.org/10.1007/BF01386390>).
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7.
- Fredman, Michael Lawrence; Tarjan, Robert E. (1984). "Fibonacci heaps and their uses in improved network optimization algorithms". 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346. doi: 10.1109/SFCS.1984.715934 (<http://dx.doi.org/10.1109/SFCS.1984.715934>).
- Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (<http://portal.acm.org/citation.cfm?id=28874>). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi: 10.1145/28869.28874 (<http://dx.doi.org/10.1145/28869.28874>).
- Zhan, F. Benjamin; Noon, Charles E. (February 1998). "Shortest Path Algorithms: An Evaluation Using Real Road Networks". *Transportation Science* **32** (1): 65–73. doi: 10.1287/trsc.32.1.65 (<http://dx.doi.org/10.1287/trsc.32.1.65>).
- Leyzorek, M.; Gray, R. S.; Johnson, A. A.; Ladew, W. C.; Meaker, Jr., S. R.; Petry, R. M.; Seitz, R. N. (1957). *Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — A Study of Model*

Techniques for Communication Systems. Cleveland, Ohio: Case Institute of Technology.

- Knuth, D.E. (1977). "A Generalization of Dijkstra's Algorithm". *Information Processing Letters* **6** (1): 1–5. doi: 10.1016/0020-0190(77)90002-3 ([http://dx.doi.org/10.1016/0020-0190\(77\)90002-3](http://dx.doi.org/10.1016/0020-0190(77)90002-3)).
- Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James B.; Tarjan, Robert E. (April 1990). "Faster Algorithms for the Shortest Path Problem". *Journal of Association for Computing Machinery (ACM)* **37** (2): 213—223. doi: 10.1145/77600.77615 (<http://dx.doi.org/10.1145/77600.77615>).

External links



Wikimedia Commons has media related to *Dijkstra's algorithm*.

- C/C++
 - Dijkstra's Algorithm in C Programming Language (<http://www.rawbytes.com/dijkstras-algorithm-in-c/>)
 - Dijkstra's Algorithm in C++ (<https://github.com/xtaci/algorithms/blob/master/include/dijkstra.h>)
 - Implementation in Boost C++ library (http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/dijkstra_shortest_paths.html)
- Java
 - Applet by Carla Laffra of Pace University (<http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html>)
 - A Java library for path finding with Dijkstra's Algorithm and example Applet (<http://www.stackframe.com/software/PathFinder>)
 - Dijkstra's Algorithm Applet (<http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm>)
 - Dijkstra's algorithm as bidirectional version in Java (<https://github.com/graphhopper/graphhopper/tree/90879ad05c4dfedf0390d44525065f727b043357/core/src/main/java/com/graphhopper/routing>)
 - Hipster: An Open-Source Java Library for Heuristic Search with Dijkstra's algorithm and examples (<http://www.hipster4j.org>)
 - Open Source Java Graph package with implementation of Dijkstra's Algorithm (<http://code.google.com/p/annas/>)
 - Shortest Path Problem: Dijkstra's Algorithm (<http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml>)
 - Visualization of Dijkstra's Algorithm (http://students.ceid.upatras.gr/~papagel/english/java_docs/minDijk.htm)
- C#/.Net
 - Dijkstra's Algorithm in C# (<http://www.codeproject.com/KB/recipes/ShortestPathCalculation.aspx>)
 - Fast Priority Queue Implementation of Dijkstra's Algorithm in C# (<http://www.codeproject.com/KB/recipes/FastHeapDijkstra.aspx>)
 - QuickGraph, Graph Data Structures and Algorithms for .NET (<http://quickgraph.codeplex.com/>)
- Dijkstra's Algorithm Simulation (<http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html>)
- Oral history interview with Edsger W. Dijkstra (<http://purl.umn.edu/107247>), Charles Babbage Institute University of Minnesota, Minneapolis.
- Animation of Dijkstra's algorithm (<http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html>)
- Haskell implementation of Dijkstra's Algorithm (<http://bonsaicode.wordpress.com/2011/01/04/programming-praxis-dijkstras-algorithm/>) on Bonsai code
- Implementation in T-SQL (<http://hansolav.net/sql/graphs.html>)

- A MATLAB program for Dijkstra's algorithm (<http://www.mathworks.com/matlabcentral/fileexchange/20025-advanced-dijkstras-minimum-path-algorithm>)
- Step through Dijkstra's Algorithm in an online JavaScript Debugger (http://www.turb0js.com/a/Dijkstras_Algorithm)

Bellman–Ford algorithm for single-source shortest paths allowing negative edge lengths

Bellman–Ford algorithm

Class	Single-source shortest path problem (for weighted directed graphs)
Data structure	Graph
Worst case performance	$O(V E)$
Worst case space complexity	$O(V)$

Graph and tree search algorithms

- $\alpha-\beta$
- A*
- B*
- Backtracking
- Beam
- Bellman–Ford
- Best-first
- Bidirectional
- Borůvka
- Branch & bound
- BFS
- British Museum
- D*
- DFS
- Depth-limited
- Dijkstra
- Edmonds
- Floyd–Warshall
- Fringe search
- Hill climbing
- IDA*
- Iterative deepening
- Johnson
- Jump point
- Kruskal
- Lexicographic BFS
- Prim
- SMA*

• Uniform-cost
Listings
• <i>Graph algorithms</i>
• <i>Search algorithms</i>
• <i>List of graph algorithms</i>
Related topics
• Dynamic programming
• Graph traversal
• Tree traversal
• Search games
• v
• t
• $e^{[1]}$

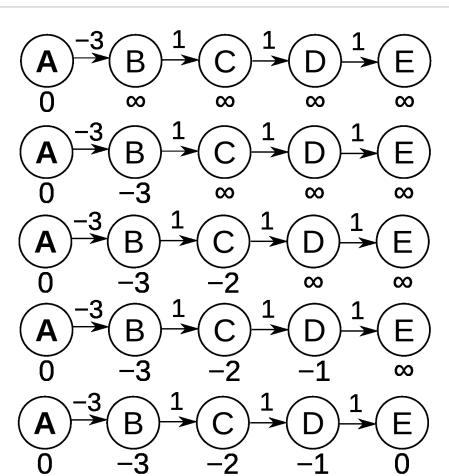
The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The algorithm is usually named after two of its developers, Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956, respectively; however, Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the **Bellman–Ford–Moore algorithm**.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.^[1] If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no *cheapest* path: any path can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence.^[2]

Algorithm

Like Dijkstra's Algorithm, Bellman–Ford is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path. However, Dijkstra's algorithm greedily selects the minimum-weight node that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes *all* the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.

Bellman–Ford runs in $O(|V| \cdot |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively.



In this example graph, assuming that A is the source and edges are processed in the worst order, from right to left, it requires the full $|V|-1$ or 4 iterations for the distance estimates to converge. Conversely, if the edges are processed in the best order, from left to right, the algorithm converges in a single iteration.

```

function BellmanFord(list vertices, list edges, vertex source)
    ::weight[], predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (weight and predecessor) with shortest-path
    // (less cost/weight/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then weight[v] := 0
        else weight[v] := infinity
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if weight[u] + w < weight[v]:
                weight[v] := weight[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if weight[u] + w < weight[v]:
            error "Graph contains a negative-weight cycle"
    return weight[], predecessor[]

```

Simply put, the algorithm initializes the distance to the source to 0 and all other nodes to infinity. Then for all edges, if the distance to the destination can be shorten by taking the edge, the distance is updated to the new lower value. At each iteration i that the edges are scanned, the algorithm finds all shortest paths of at most length i edges. Since the longest possible path without a cycle can be $|V| - 1$ edges, the edges must be scanned $|V| - 1$ times to ensure the shortest path has been found for all nodes. A final scan of all the edges is performed and if any distance is updated, then a path of length $|V|$ edges has been found which can only occur if at least one negative cycle exists in the graph.

Proof of correctness

The correctness of the algorithm can be shown by induction. The precise statement shown by induction is:

Lemma. After i repetitions of *for* cycle:

- If $\text{Distance}(u)$ is not infinity, it is equal to the length of some path from s to u ;
- If there is a path from s to u with at most i edges, then $\text{Distance}(u)$ is at most the length of the shortest path from s to u with at most i edges.

Proof. For the base case of induction, consider $i=0$ and the moment before *for* cycle is executed for the first time. Then, for the source vertex, $\text{source.distance} = 0$, which is correct. For other vertices u , $u.\text{distance} = \text{infinity}$, which is also correct because there is no path from *source* to u with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex's distance is updated by $v.\text{distance} := u.\text{distance} + uv.\text{weight}$. By inductive assumption, $u.\text{distance}$ is the length of

some path from *source* to *u*. Then *u.distance* + *uv.weight* is the length of the path from *source* to *v* that follows the path from *source* to *u* and then goes to *v*.

For the second part, consider the shortest path from *source* to *u* with at most *i* edges. Let *v* be the last vertex before *u* on this path. Then, the part of the path from *source* to *v* is the shortest path from *source* to *v* with at most *i*-1 edges. By inductive assumption, *v.distance* after *i*-1 cycles is at most the length of this path. Therefore, *uv.weight* + *v.distance* is at most the length of the path from *s* to *u*. In the *i*th cycle, *u.distance* gets compared with *uv.weight* + *v.distance*, and is set equal to it if *uv.weight* + *v.distance* was smaller. Therefore, after *i* cycles, *u.distance* is at most the length of the shortest path from *source* to *u* that uses at most *i* edges.

If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so at step 3 no further improvements can be made. Conversely, suppose no improvement can be made. Then for any cycle with vertices *v*[0], ..., *v*[*k*-1],

```
v[i].distance <= v[(i-1) mod k].distance + v[(i-1) mod k]v[i].weight
```

Summing around the cycle, the *v*[*i*].*distance* terms and the *v*[*i*-1 (*mod* *k*)].*distance* terms cancel, leaving

```
0 <= sum from 1 to k of v[i-1 (mod k)]v[i].weight
```

I.e., every cycle has nonnegative weight.

Finding negative cycles

When the algorithm is used to find shortest paths, the existence of negative cycles is a problem, preventing the algorithm from finding a correct answer. However, since it terminates upon finding a negative cycle, the Bellman–Ford algorithm can be used for applications in which this is the target to be sought - for example in cycle-cancelling techniques in network flow analysis.

Applications in routing

A distributed variant of the Bellman–Ford algorithm is used in distance-vector routing protocols, for example the Routing Information Protocol (RIP). The algorithm is distributed because it involves a number of nodes (routers) within an Autonomous system, a collection of IP networks typically owned by an ISP. It consists of the following steps:

1. Each node calculates the distances between itself and all other nodes within the AS and stores this information as a table.
2. Each node sends its table to all neighboring nodes.
3. When a node receives distance tables from its neighbors, it calculates the shortest routes to all other nodes and updates its own table to reflect any changes.

The main disadvantages of the Bellman–Ford algorithm in this setting are as follows:

- It does not scale well.
- Changes in network topology are not reflected quickly since updates are spread node-by-node.
- Count to infinity (if link or node failures render a node unreachable from some set of other nodes, those nodes may spend forever gradually increasing their estimates of the distance to it, and in the meantime there may be routing loops).

Improvements

The Bellman–Ford algorithm may be improved in practice (although not in the worst case) by the observation that, if an iteration of the main loop of the algorithm terminates without making any changes, the algorithm can be immediately terminated, as subsequent iterations will not make any more changes. With this early termination condition, the main loop may in some cases use many fewer than $|V| - 1$ iterations, even though the worst case of the algorithm remains unchanged.

Yen (1970) described two more improvements to the Bellman–Ford algorithm for a graph without negative-weight cycles; again, while making the algorithm faster in practice, they do not change its $O(|V|^2|E|)$ worst case time bound. His first improvement reduces the number of relaxation steps that need to be performed within each iteration of the algorithm. If a vertex v has a distance value that has not changed since the last time the edges out of v were relaxed, then there is no need to relax the edges out of v a second time. In this way, as the number of vertices with correct distance values grows, the number whose outgoing edges need to be relaxed in each iteration shrinks, leading to a constant-factor savings in time for dense graphs.

Yen's second improvement first assigns some arbitrary linear order on all vertices and then partitions the set of all edges into two subsets. The first subset, E_f , contains all edges (v_i, v_j) such that $i < j$; the second, E_b , contains edges (v_i, v_j) such that $i > j$. Each vertex is visited in the order $v_1, v_2, \dots, v_{|V|}$, relaxing each outgoing edge from that vertex in E_f . Each vertex is then visited in the order $v_{|V|}, v_{|V|-1}, \dots, v_1$, relaxing each outgoing edge from that vertex in E_b . Each iteration of the main loop of the algorithm, after the first one, adds at least two edges to the set of edges whose relaxed distances match the correct shortest path distances: one from E_f and one from E_b . This modification reduces the worst-case number of iterations of the main loop of the algorithm from $|V| - 1$ to $|V|/2$.^[3]

Another improvement, by Bannister & Eppstein (2012), replaces the arbitrary linear order of the vertices used in Yen's second improvement by a random permutation. This change makes the worst case for Yen's improvement (in which the edges of a shortest path strictly alternate between the two subsets E_f and E_b) very unlikely to happen. With a randomly permuted vertex ordering, the expected number of iterations needed in the main loop is at most $|V|/3$.^[4]

Notes

[1] Sedgewick (2002).

[2] Kleinberg & Tardos (2006).

[3] Cormen et al., 2nd ed., Problem 24-1, pp. 614–615.

References

Original sources

- Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* **16**: 87–90. MR 0102435 (<http://www.ams.org/mathscinet-getitem?mr=0102435>).
- Ford Jr., Lester R. (August 14, 1956). *Network Flow Theory* (<http://www.rand.org/pubs/papers/P923.html>). Paper P-923. Santa Monica, California: RAND Corporation.
- Moore, Edward F. (1959). "The shortest path through a maze". *Proc. Internat. Sympos. Switching Theory 1957, Part II*. Cambridge, Mass.: Harvard Univ. Press. pp. 285–292. MR 0114710 (<http://www.ams.org/mathscinet-getitem?mr=0114710>).
- Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks". *Quarterly of Applied Mathematics* **27**: 526–530. MR 0253822 (<http://www.ams.org/mathscinet-getitem?mr=0253822>).
- Bannister, M. J.; Eppstein, D. (2012). "Randomized speedup of the Bellman–Ford algorithm" (<http://siamomnibooksonline.com/2012ANALCO/data/papers/005.pdf>). *Analytic Algorithmics and Combinatorics (ANALCO12), Kyoto, Japan*. pp. 41–47. arXiv: 1111.5414 (<http://arxiv.org/abs/1111.5414>).

Secondary sources

- Bang-Jensen, Jørgen; Gutin, Gregory (2000). "Section 2.3.4: The Bellman-Ford-Moore algorithm" (<http://www.cs.rhul.ac.uk/books/dbook/>). *Digraphs: Theory, Algorithms and Applications* (First ed.). ISBN 978-1-84800-997-4.
- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. *Introduction to Algorithms*. MIT Press and McGraw-Hill., Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 24.1: The Bellman–Ford algorithm, pp. 588–592. Problem 24-1, pp. 614–615. Third Edition. MIT Press, 2009. ISBN 978-0-262-53305-8. Section 24.1: The Bellman–Ford algorithm, pp. 651–655.
- Heineman, George T.; Pollice, Gary; Selkow, Stanley (2008). "Chapter 6: Graph Algorithms". *Algorithms in a Nutshell*. O'Reilly Media. pp. 160–164. ISBN 978-0-596-51624-6.
- Kleinberg, Jon; Tardos, Éva (2006). *Algorithm Design*. New York: Pearson Education, Inc.
- Sedgewick, Robert (2002). "Section 21.7: Negative Edge Weights" (<http://safari.oreilly.com/0201361213/ch21lev1sec7>). *Algorithms in Java* (3rd ed.). ISBN 0-201-36121-3.

External links

- C++ code example (https://github.com/xtaci/algorithms/blob/master/include/bellman_ford.h)
- Open Source Java Graph package with Bellman-Ford Algorithms (<http://code.google.com/p/annas/>)

Johnson's algorithm for all-pairs shortest paths in sparse graphs

Johnson's algorithm

Class	All-pairs shortest path problem (for weighted graphs)
Data structure	Graph
Worst case performance	$O(V ^2 \log V + V E)$

For the scheduling algorithm of the same name, see Job Shop Scheduling.

Graph and tree search algorithms	
•	$\alpha-\beta$
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall
•	Fringe search
•	Hill climbing
•	IDA*
•	Iterative deepening
•	Johnson
•	Jump point
•	Kruskal
•	Lexicographic BFS
•	Prim
•	SMA*
•	Uniform-cost
Listings	
•	<i>Graph algorithms</i>
•	<i>Search algorithms</i>
•	<i>List of graph algorithms</i>

Related topics
<ul style="list-style-type: none"> • Dynamic programming • Graph traversal • Tree traversal • Search games
<ul style="list-style-type: none"> • v • t • $e^{[1]}$

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse, edge weighted, directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the Bellman–Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph.^[1] It is named after Donald B. Johnson, who first published the technique in 1977.

A similar reweighting technique is also used in Suurballe's algorithm for finding two disjoint paths of minimum total length between the same two vertices in a graph with non-negative edge weights.

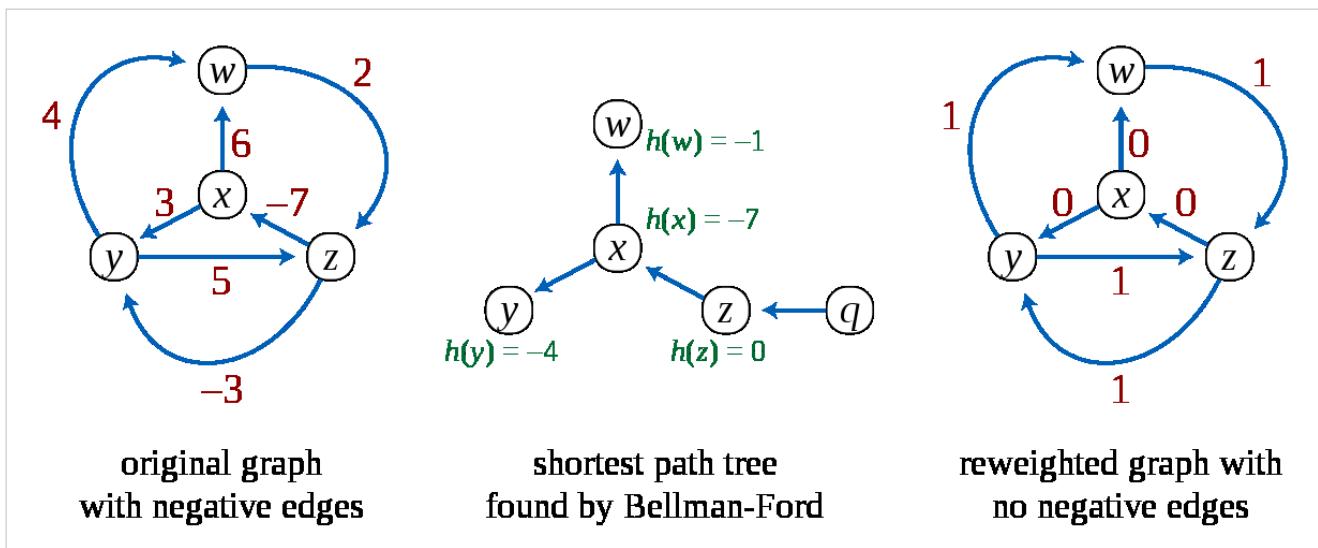
Algorithm description

Johnson's algorithm consists of the following steps:

1. First, a new node q is added to the graph, connected by zero-weight edges to each of the other nodes.
2. Second, the Bellman–Ford algorithm is used, starting from the new vertex q , to find for each vertex v the minimum weight $h(v)$ of a path from q to v . If this step detects a negative cycle, the algorithm is terminated.
3. Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from u to v , having length $w(u,v)$, is given the new length $w(u,v) + h(u) - h(v)$.
4. Finally, q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.

Example

The first three stages of Johnson's algorithm are depicted in the illustration below.



The graph on the left of the illustration has two negative edges, but no negative cycles. At the center is shown the new vertex q , a shortest path tree as computed by the Bellman–Ford algorithm with q as starting vertex, and the values $h(v)$ computed at each other node as the length of the shortest path from q to that node. Note that these values

are all non-positive, because q has a length-zero edge to each vertex and the shortest path can be no longer than that edge. On the right is shown the reweighted graph, formed by replacing each edge weight $w(u,v)$ by $w(u,v) + h(u) - h(v)$. In this reweighted graph, all edge weights are non-negative, but the shortest path between any two nodes uses the same sequence of edges as the shortest path between the same two nodes in the original graph. The algorithm concludes by applying Dijkstra's algorithm to each of the four starting nodes in the reweighted graph.

Correctness

In the reweighted graph, all paths between a pair s and t of nodes have the same quantity $h(s) - h(t)$ added to them. The previous statement can be proven as follows: Let p be an s - t path. Its weight W in the reweighted graph is given by the following expression:

$$(w(s, p_1) + h(s) - h(p_1)) + (w(p_1, p_2) + h(p_1) - h(p_2)) + \dots + (w(p_n, t) + h(p_n) - h(t)).$$

Every $+h(p_i)$ is cancelled by $-h(p_i)$ in the previous bracketed expression; therefore, we are left with the following expression for W :

$$(w(s, p_1) + w(p_1, p_2) + \dots + w(p_n, t)) + h(s) - h(t)$$

The bracketed expression is the weight of p in the original weighting.

Since the reweighting adds the same amount to the weight of every s - t path, a path is a shortest path in the original weighting if and only if it is a shortest path after reweighting. The weight of edges that belong to a shortest path from q to any node is zero, and therefore the lengths of the shortest paths from q to every node become zero in the reweighted graph; however, they still remain shortest paths. Therefore, there can be no negative edges: if edge uv had a negative weight after the reweighting, then the zero-length path from q to u together with this edge would form a negative-length path from q to v , contradicting the fact that all vertices have zero distance from q . The non-existence of negative edges ensures the optimality of the paths found by Dijkstra's algorithm. The distances in the original graph may be calculated from the distances calculated by Dijkstra's algorithm in the reweighted graph by reversing the reweighting transformation.

Analysis

The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is $O(V^2 \log V + VE)$: the algorithm uses $O(VE)$ time for the Bellman–Ford stage of the algorithm, and $O(V \log V + E)$ for each of V instantiations of Dijkstra's algorithm. Thus, when the graph is sparse, the total time can be faster than the Floyd–Warshall algorithm, which solves the same problem in time $O(V^3)$.

References

[1] . Section 25.3, "Johnson's algorithm for sparse graphs", pp. 636–640.

External links

- Boost: All Pairs Shortest Paths (http://www.boost.org/doc/libs/1_40_0/libs/graph/doc/johnson_all_pairs_shortest.html)

Floyd–Warshall algorithm for all-pairs shortest paths in dense graphs

"Floyd's algorithm" redirects here. For cycle detection, see Floyd's cycle-finding algorithm. For computer graphics, see Floyd–Steinberg dithering.

Floyd–Warshall algorithm

Class	All-pairs shortest path problem (for weighted graphs)
Data structure	Graph
Worst case performance	$O(V ^3)$
Best case performance	$\Omega(V ^3)$
Worst case space complexity	$\Theta(V ^2)$

Graph and tree search algorithms

- $\alpha-\beta$
- A*
- B*
- Backtracking
- Beam
- Bellman–Ford
- Best-first
- Bidirectional
- Borůvka
- Branch & bound
- BFS
- British Museum
- D*
- DFS
- Depth-limited
- Dijkstra
- Edmonds
- Floyd–Warshall
- Fringe search
- Hill climbing
- IDA*
- Iterative deepening
- Johnson
- Jump point
- Kruskal
- Lexicographic BFS
- Prim
- SMA*
- Uniform-cost

[Listings](#)

<ul style="list-style-type: none"> • Graph algorithms • Search algorithms • List of graph algorithms
Related topics
<ul style="list-style-type: none"> • Dynamic programming • Graph traversal • Tree traversal • Search games
<ul style="list-style-type: none"> • v • t • $e^{[1]}$

In computer science, the **Floyd–Warshall algorithm** (also known as **Floyd's algorithm**, **Roy–Warshall algorithm**, **Roy–Floyd algorithm**, or the **WFI algorithm**) is a graph analysis algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles, see below) and also for finding transitive closure of a relation R . A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices, though it does not return details of the paths themselves.

The Floyd–Warshall algorithm was published in its currently recognized form by Robert Floyd in 1962. However, it is essentially the same as algorithms previously published by Bernard Roy in 1959 and also by Stephen Warshall in 1962 for finding the transitive closure of a graph. The modern formulation of Warshall's algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962.

The algorithm is an example of dynamic programming.

Algorithm

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta(|V|^3)$ comparisons in a graph. This is remarkable considering that there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Consider a graph G with vertices V numbered 1 through N . Further consider a function $\text{shortestPath}(i, j, k)$ that returns the shortest possible path from i to j using vertices only from the set $\{1, 2, \dots, k\}$ as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to $k + 1$.

For each of these pairs of vertices, the true shortest path could be either (1) a path that only uses vertices in the set $\{1, \dots, k\}$ or (2) a path that goes from i to $k + 1$ and then from $k + 1$ to j . We know that the best path from i to j that only uses vertices 1 through k is defined by $\text{shortestPath}(i, j, k)$, and it is clear that if there were a better path from i to $k + 1$ to j , then the length of this path would be the concatenation of the shortest path from i to $k + 1$ (using vertices in $\{1, \dots, k\}$) and the shortest path from $k + 1$ to j (also using vertices in $\{1, \dots, k\}$).

If $w(i, j)$ is the weight of the edge between vertices i and j , we can define $\text{shortestPath}(i, j, k + 1)$ in terms of the following recursive formula: the base case is

$$\text{shortestPath}(i, j, 0) = w(i, j)$$

and the recursive case is

$$\text{shortestPath}(i, j, k+1) = \min(\text{shortestPath}(i, j, k), \text{shortestPath}(i, k+1, k) + \text{shortestPath}(k+1, j, k))$$

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing $\text{shortestPath}(i, j, k)$ for all (i, j) pairs for $k = 1$, then $k = 2$, etc. This process continues until $k = n$, and we have found the shortest path for all (i, j) pairs using any intermediate vertices. Pseudocode for this basic version follows:

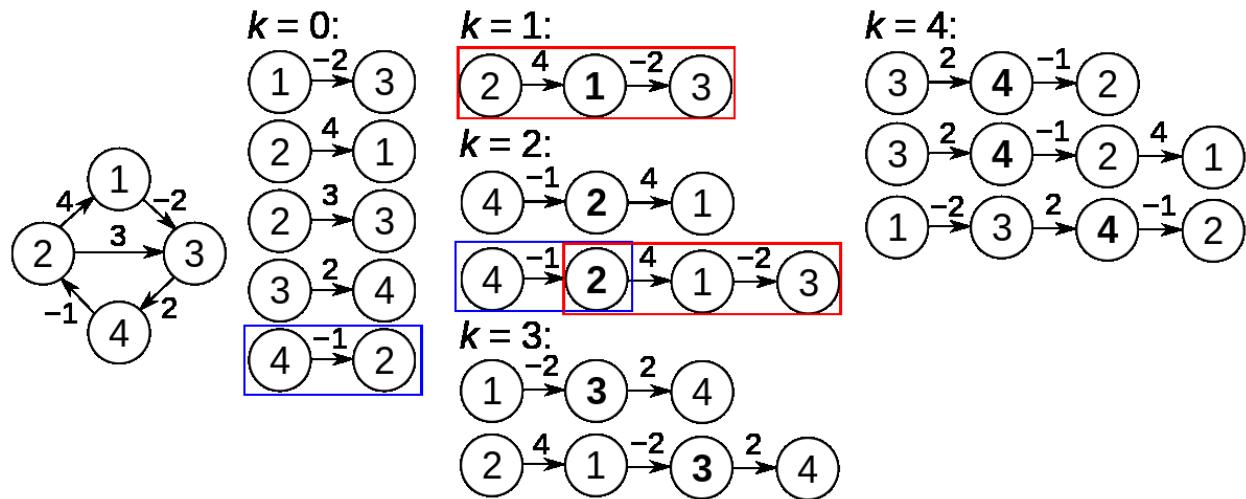
```

1 let dist be a  $|V| \times |V|$  array of minimum distances initialized to  $\infty$  (infinity)
2 for each vertex v
3   dist[v][v]  $\leftarrow 0$ 
4 for each edge  $(u, v)$ 
5   dist[u][v]  $\leftarrow w(u, v)$  // the weight of the edge  $(u, v)$ 
6 for k from 1 to  $|V|$ 
7   for i from 1 to  $|V|$ 
8     for j from 1 to  $|V|$ 
9       if dist[i][j] > dist[i][k] + dist[k][j]
10      dist[i][j]  $\leftarrow$  dist[i][k] + dist[k][j]
11    end if

```

Example

The algorithm above is executed on the graph on the left below:



Prior to the first iteration of the outer loop, labeled $k=0$ above, the only known paths correspond to the single edges in the graph. At $k=1$, paths that go through the vertex 1 are found: in particular, the path $2 \rightarrow 1 \rightarrow 3$ is found, replacing the path $2 \rightarrow 3$ which has fewer edges but is longer. At $k=2$, paths going through the vertices $\{1,2\}$ are found. The red and blue boxes show how the path $4 \rightarrow 2 \rightarrow 1 \rightarrow 3$ is assembled from the two known paths $4 \rightarrow 2$ and $2 \rightarrow 1 \rightarrow 3$ encountered in previous iterations, with 2 in the intersection. The path $4 \rightarrow 2 \rightarrow 3$ is not considered, because $2 \rightarrow 1 \rightarrow 3$ is the shortest path encountered so far from 2 to 3. At $k=3$, paths going through the vertices $\{1,2,3\}$ are found. Finally, at $k=4$, all shortest paths are found.

Behavior with negative cycles

A negative cycle is a cycle whose edges sum to a negative value. There is no shortest path between any pair of vertices i, j which form part of a negative cycle, because path-lengths from i to j can be arbitrarily small (negative). For numerically meaningful output, the Floyd–Warshall algorithm assumes that there are no negative cycles. Nevertheless, if there are negative cycles, the Floyd–Warshall algorithm can be used to detect them. The intuition is as follows:

- The Floyd–Warshall algorithm iteratively revises path lengths between all pairs of vertices (i, j) , including where $i = j$;
- Initially, the length of the path (i, i) is zero;
- A path $\{(i, k), (k, i)\}$ can only improve upon this if it has length less than zero, i.e. denotes a negative cycle;
- Thus, after the algorithm, (i, i) will be negative if there exists a negative-length path from i back to i .

Hence, to detect negative cycles using the Floyd–Warshall algorithm, one can inspect the diagonal of the path matrix, and the presence of a negative number indicates that the graph contains at least one negative cycle. To avoid numerical problems one should check for negative numbers on the diagonal of the path matrix within the inner for loop of the algorithm. Obviously, in an undirected graph a negative edge creates a negative cycle (i.e., a closed walk) involving its incident vertices. Considering all edges of the above example graph as undirected, e.g. the vertex sequence 4 - 2 - 4 is a cycle with weight sum -2.

Path reconstruction

The Floyd–Warshall algorithm typically only provides the lengths of the paths between all pairs of vertices. With simple modifications, it is possible to create a method to reconstruct the actual path between any two endpoint vertices. While one may be inclined to store the actual path from each vertex to each other vertex, this is not necessary, and in fact, is very costly in terms of memory. Instead, the Shortest-path tree can be calculated for each node in $\Theta(|E|)$ time using $\Theta(|V|)$ memory to store each tree which allows us to efficiently reconstruct a path from any two connected vertices.

```

let dist be a  $|V| \times |V|$  array of minimum distances initialized to  $\infty$  (infinity)
let next be a  $|V| \times |V|$  array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction ()
    for each edge  $(u, v)$ 
        dist[u][v]  $\leftarrow w(u, v)$  // the weight of the edge  $(u, v)$ 
        next[u][v]  $\leftarrow v$ 
    for k from 1 to  $|V|$  // standard Floyd–Warshall implementation
        for i from 1 to  $|V|$ 
            for j from 1 to  $|V|$ 
                if dist[i][k] + dist[k][j] < dist[i][j] then
                    dist[i][j]  $\leftarrow$  dist[i][k] + dist[k][j]
                    next[i][j]  $\leftarrow$  next[i][k]

procedure Path( $u, v$ )
    if next[u][v] = null then
        return []
    path = [u]
    while u  $\neq v$ 
        u  $\leftarrow$  next[u][v]
        path.append(u)
    return path

```

Analysis

Let n be $|V|$, the number of vertices. To find all n^2 of $\text{shortestPath}(i,j,k)$ (for all i and j) from those of $\text{shortestPath}(i,j,k-1)$ requires $2n^2$ operations. Since we begin with $\text{shortestPath}(i,j,0) = \text{edgeCost}(i,j)$ and compute the sequence of n matrices $\text{shortestPath}(i,j,1)$, $\text{shortestPath}(i,j,2)$, ..., $\text{shortestPath}(i,j,n)$, the total number of operations used is $n \cdot 2n^2 = 2n^3$. Therefore, the complexity of the algorithm is $\Theta(n^3)$.

Applications and generalizations

The Floyd–Warshall algorithm can be used to solve the following problems, among others:

- Shortest paths in directed graphs (Floyd's algorithm).
- Transitive closure of directed graphs (Warshall's algorithm). In Warshall's original formulation of the algorithm, the graph is unweighted and represented by a Boolean adjacency matrix. Then the addition operation is replaced by logical conjunction (AND) and the minimum operation by logical disjunction (OR).
- Finding a regular expression denoting the regular language accepted by a finite automaton (Kleene's algorithm, a closely related generalization of the Floyd–Warshall algorithm)
- Inversion of real matrices (Gauss–Jordan algorithm)
- Optimal routing. In this application one is interested in finding the path with the maximum flow between two vertices. This means that, rather than taking minima as in the pseudocode above, one instead takes maxima. The edge weights represent fixed constraints on flow. Path weights represent bottlenecks; so the addition operation above is replaced by the minimum operation.
- Fast computation of Pathfinder networks.
- Widest paths/Maximum bandwidth paths
- Computing canonical form of difference bound matrices (DBMs)

Implementations

Implementations are available for many programming languages.

- For C++, in the boost::graph^[1] library
- For C#, at QuickGraph^[2]
- For Java, in the Apache Commons Graph^[3] library
- For MATLAB, in the Matlab_bgl^[4] package
- For Perl, in the Graph^[5] module
- For Python, in the NetworkX library
- For R, in package e1071^[6]

References

- [1] <http://www.boost.org/libs/graph/doc/>
- [2] <http://www.codeplex.com/quickgraph>
- [3] <http://commons.apache.org/sandbox/commons-graph/>
- [4] <http://www.mathworks.com/matlabcentral/fileexchange/10922>
- [5] <https://metacpan.org/module/Graph>
- [6] <http://cran.r-project.org/web/packages/e1071/index.html>
- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L. (1990). *Introduction to Algorithms* (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8.
 - Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565;
 - Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.
- Floyd, Robert W. (June 1962). "Algorithm 97: Shortest Path". *Communications of the ACM* 5 (6): 345. doi:10.1145/367766.368168 (<http://dx.doi.org/10.1145/367766.368168>).

- Ingerman, Peter Z. (November 1962). "Algorithm 141: Path Matrix". *Communications of the ACM* **5** (11): 556. doi: 10.1145/368996.369016 (<http://dx.doi.org/10.1145/368996.369016>).
- Kleene, S. C. (1956). "Representation of events in nerve nets and finite automata". In C. E. Shannon and J. McCarthy. *Automata Studies*. Princeton University Press. pp. 3–42.
- Warshall, Stephen (January 1962). "A theorem on Boolean matrices". *Journal of the ACM* **9** (1): 11–12. doi: 10.1145/321105.321107 (<http://dx.doi.org/10.1145/321105.321107>).
- Kenneth H. Rosen (2003). *Discrete Mathematics and Its Applications, 5th Edition*. Addison Wesley. ISBN 0-07-119881-4.
- Roy, Bernard (1959). "Transitivité et connexité.". *C. R. Acad. Sci. Paris* **249**: 216–218.

External links

- Interactive animation of the Floyd–Warshall algorithm (http://www.pms.informatik.uni-muenchen.de/lehre/compgeometry/Gosper/shortest_path/shortest_path.html#visualization)
- The Floyd–Warshall algorithm in C#, as part of QuickGraph (<http://quickgraph.codeplex.com/>)
- Visualization of Floyd's algorithm (http://students.ceid.upatras.gr/~papagei/english/java_docs/allmin.htm)

Suurballe's algorithm for two shortest disjoint paths

In theoretical computer science and network routing, **Suurballe's algorithm** is an algorithm for finding two disjoint paths in a nonnegatively-weighted directed graph, so that both paths connect the same pair of vertices and have minimum total length. The algorithm was conceived by J. W. Suurballe and published in 1974. The main idea of Suurballe's algorithm is to use Dijkstra's algorithm to find one path, to modify the weights of the graph edges, and then to run Dijkstra's algorithm a second time. The modification to the weights is similar to the weight modification in Johnson's algorithm, and preserves the non-negativity of the weights while allowing the second instance of Dijkstra's algorithm to find the correct second path.

The objective is strongly related to that of minimum cost flow algorithms, where in this case there are two units of "flow" and nodes have unit "capacity".

Definitions

Let G be a weighted directed graph containing a set V of n vertices and a set E of m directed edges; let s be a designated source vertex in G , and let t be a designated destination vertex. Let each edge (u,v) in E , from vertex u to vertex v , have a non-negative cost $w(u,v)$.

Define $d(s,u)$ to be the cost of the shortest path to node u from node s in the shortest path tree rooted at s .

Algorithm

Suurballe's algorithm performs the following steps:

1. Find the shortest path tree T rooted at node s by running Dijkstra's algorithm. This tree contains for every vertex u , a shortest path from s to u . Let P_1 be the shortest cost path from s to t . The edges in T are called *tree edges* and the remaining edges are called *non tree edges*.
2. Modify the cost of each edge in the graph by replacing the cost $w(u,v)$ of every edge (u,v) by $w'(u,v) = w(u,v) - d(s,v) + d(s,u)$. According to the resulting modified cost function, all tree edges have a cost of 0, and non tree edges have a non negative cost.

3. Create a residual graph G_t formed from G by removing the edges of G that are directed into s and by reversing the direction of the zero length edges along path P_1 .
4. Find the shortest path P_2 in the residual graph G_t by running Dijkstra's algorithm.
5. Discard the reversed edges of P_2 from both paths. The remaining edges of P_1 and P_2 form a subgraph with two outgoing edges at s , two incoming edges at t , and one incoming and one outgoing edge at each remaining vertex. Therefore, this subgraph consists of two edge-disjoint paths from s to t and possibly some additional (zero-length) cycles. Return the two disjoint paths from the subgraph.

Example

The following example shows how Suurballe's algorithm finds the shortest pair of disjoint paths from A to F .

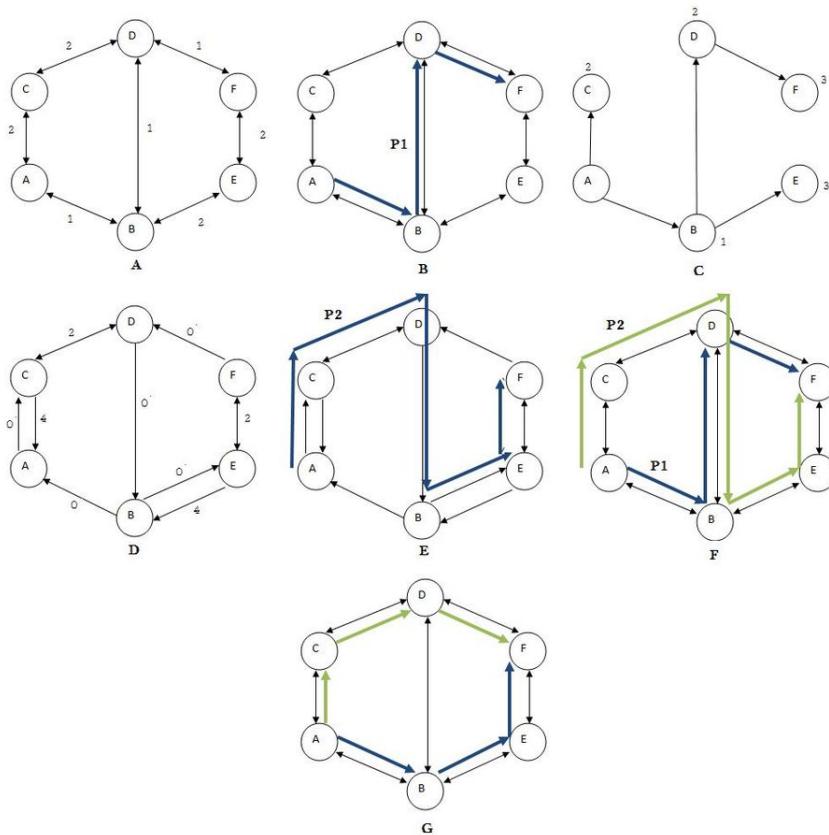


Figure A illustrates a weighted graph G .

Figure B calculates the shortest path P_1 from A to F ($A-B-D-F$).

Figure C illustrates the shortest path tree T rooted at A , and the computed distances from A to every vertex.

Figure D shows the updated cost of each edge and the edges of path P_1 reversed.

Figure E calculates path P_2 in the residual graph G_t ($A-C-D-B-E-F$).

Figure F illustrates both path P_1 and path P_2 .

Figure G finds the shortest pair of disjoint paths by combining the edges of paths P_1 and P_2 and then discarding the common reversed edges between both paths ($B-D$). As a result we get the two shortest pair of disjoint paths ($A-B-E-F$) and ($A-C-D-F$).

Correctness

The weight of any path from s to t in the modified system of weights equals the weight in the original graph, minus $d(s,t)$. Therefore, the shortest two disjoint paths under the modified weights are the same paths as the shortest two paths in the original graph, although they have different weights.

Suurballe's algorithm may be seen as a special case of the successive shortest paths method for finding a minimum cost flow with total flow amount two from s to t . The modification to the weights does not affect the sequence of paths found by this method, only their weights. Therefore the correctness of the algorithm follows from the correctness of the successive shortest paths method.

Analysis and running time

This algorithm requires two iterations of Dijkstra's algorithm. Using Fibonacci heaps, both iterations can be performed in time $O(m + n \log n)$ on a graph with n vertices and m edges. Therefore, the same time bound applies to Suurballe's algorithm.

Variations

The version of Suurballe's algorithm as described above finds paths that have disjoint edges, but that may share vertices. It is possible to use the same algorithm to find vertex-disjoint paths, by replacing each vertex by a pair of adjacent vertices, one with all of the incoming adjacencies of the original vertex, and one with all of the outgoing adjacencies. Two edge-disjoint paths in this modified graph necessarily correspond to two vertex-disjoint paths in the original graph, and vice versa, so applying Suurballe's algorithm to the modified graph results in the construction of two vertex-disjoint paths in the original graph. Suurballe's original 1974 algorithm was for the vertex-disjoint version of the problem, and was extended in 1984 by Suurballe and Tarjan to the edge-disjoint version.

By using a modified version of Dijkstra's algorithm that simultaneously computes the distances to each vertex t in the graphs G_t , it is also possible to find the total lengths of the shortest pairs of paths from a given source vertex s to every other vertex in the graph, in an amount of time that is proportional to a single instance of Dijkstra's algorithm.

References

Bidirectional search

Graph and tree search algorithms	
•	α - β
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall
•	Fringe search
•	Hill climbing
•	IDA*
•	Iterative deepening
•	Johnson
•	Jump point
•	Kruskal
•	Lexicographic BFS
•	Prim
•	SMA*
•	Uniform-cost
Listings	
•	<i>Graph algorithms</i>
•	<i>Search algorithms</i>
•	<i>List of graph algorithms</i>
Related topics	
•	Dynamic programming
•	Graph traversal
•	Tree traversal
•	Search games
•	v
•	t
•	e ^[1]

Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, stopping when the two meet in the middle. The reason for this approach is that in many cases it is faster: for instance, in a simplified model of search problem complexity in which both searches expand a tree with branching factor b , and the distance from start to goal is d , each of the two searches has complexity $O(b^{d/2})$ (in Big O notation), and the sum of these two search times is much less than the $O(b^d)$ complexity that would result from a single search from the beginning to the goal.

As in A* search, bi-directional search can be guided by a heuristic estimate of the remaining distance to the goal (in the forward tree) or from the start (in the backward tree).

Ira Pohl (1971) was the first one to design and implement a bi-directional heuristic search algorithm. Andrew Goldberg and others explained the correct termination conditions for the bidirectional version of Dijkstra's Algorithm.^[1]

Description

A Bidirectional Heuristic Search is a state space search from some state s to another state t , searching from s to t and from t to s simultaneously (or quasi-simultaneously if done on a sequential machine). It returns a valid list of operators that if applied to s will give us t .

While it may seem as though the operators have to be invertible for the reverse search, it is only necessary to be able to find, given any node n , the set of parent nodes of n such that there exists some valid operator from each of the parent nodes to n . This has often been likened to a one-way street in the route-finding domain: it is not necessary to be able to travel down both directions, but it is necessary when standing at the end of the street to determine the beginning of the street as a possible route.

Similarly, for those edges that have inverse arcs (i.e. arcs going in both directions) it is not necessary that each direction be of equal cost. The reverse search will always use the inverse cost (i.e. the cost of the arc in the forward direction). More formally, if n is a node with parent p , then $k_1(p, n) = k_2(n, p)$, defined as being the cost from p to n . (Auer Kaindl 2004)

Terminology and notation

b

the branching factor of a search tree

$k(n, m)$

the cost associated with moving from node n to node m

$g(n)$

the cost from the root to the node n

$h(n)$

the heuristic estimate of the distance between the node n and the goal

s

the start state

t

the goal state (sometimes g , not to be confused with the function)

d

the current search direction. By convention, d is equal to 1 for the forward direction and 2 for the backward direction (Kwa 1989)

d'

the opposite search direction (i.e. $d' = 3 - d$)

 $TREE_d$

the search tree in direction d . If $d = 1$, the root is s , if $d = 2$, the root is t

 $OPEN_d$

the leaves of $TREE_d$ (sometimes referred to as $FRINGE_d$). It is from this set that a node is chosen for expansion. In bidirectional search, these are sometimes called the search 'frontiers' or 'wavefronts', referring to how they appear when a search is represented graphically. In this metaphor, a 'collision' occurs when, during the expansion phase, a node from one wavefront is found to have successors in the opposing wavefront.

 $CLOSED_d$

the non-leaf nodes of $TREE_d$. This set contains the nodes already visited by the search

Approaches for Bidirectional Heuristic Search

Bidirectional algorithms can be broadly split into three categories: Front-to-Front, Front-to-Back (or Front-to-End), and Perimeter Search (Kaindl Kainz 1997). These differ by the function used to calculate the heuristic.

Front-to-Back

Front-to-Back algorithms calculate the h value of a node n by using the heuristic estimate between n and the root of the opposite search tree, s or t .

Front-to-Back is the most actively researched of the three categories. The current best algorithm (at least in the Fifteen puzzle domain) is the BiMAX-BS*F algorithm, created by Auer and Kaindl (Auer, Kaindl 2004).

Front-to-Front

Front-to-Front algorithms calculate the h value of a node n by using the heuristic estimate between n and some subset of $OPEN'_d$. The canonical example is that of the BHFFA (Bidirectional Heuristic Front-to-Front Algorithm) (de Champeaux 1977/1983), where the h function is defined as the minimum of all heuristic estimates between the current node and the nodes on the opposing front. Or, formally:

$$h_d(n) = \min_i \{H(n, o_i) | o_i \in OPEN'_{d'}\}$$

where $H(n, o)$ returns an admissible (i.e. not overestimating) heuristic estimate of the distance between nodes n and o .

Front-to-Front suffers from being excessively computationally demanding. Every time a node n is put into the open list, its $f = g + h$ value must be calculated. This involves calculating a heuristic estimate from n to every node in the opposing $OPEN$ set, as described above. The $OPEN$ sets increase in size exponentially for all domains with $b > 1$.

References

- [1] Efficient Point-to-Point Shortest Path Algorithms ([http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP shortest path algorithms.pdf](http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf))
- de Champeaux, Dennis; Sint, Lenie (1977), "An improved bidirectional heuristic search algorithm", *Journal of the ACM* **24** (2): 177–191, doi: 10.1145/322003.322004 (<http://dx.doi.org/10.1145/322003.322004>).
 - de Champeaux, Dennis (1983), "Bidirectional heuristic search again", *Journal of the ACM* **30** (1): 22–32, doi: 10.1145/322358.322360 (<http://dx.doi.org/10.1145/322358.322360>).
 - Pohl, Ira (1971), "Bi-directional Search", in Meltzer, Bernard; Michie, Donald, *Machine Intelligence* **6**, Edinburgh University Press, pp. 127–140.
 - Russell, Stuart J.; Norvig, Peter (2002), "3.4 Uninformed search strategies", *Artificial Intelligence: A Modern Approach* (2nd ed.), Prentice Hall.

A* search algorithm

"A*" and "A star" redirect here. For other uses, see A* (disambiguation).

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(E) = O(b^d)$
Worst case space complexity	$O(V) = O(b^d)$

Graph and tree search algorithms	
•	$\alpha\text{-}\beta$
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall
•	Fringe search
•	Hill climbing
•	IDA*
•	Iterative deepening
•	Johnson
•	Jump point
•	Kruskal
•	Lexicographic BFS
•	Prim
•	SMA*
•	Uniform-cost
Listings	
•	<i>Graph algorithms</i>
•	<i>Search algorithms</i>
•	<i>List of graph algorithms</i>
Related topics	

• Dynamic programming
• Graph traversal
• Tree traversal
• Search games
• v t e [1]

In computer science, A* (pronounced "A star" ( listen)) is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. However, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, although other work has found A* to be superior to other approaches.

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first described the algorithm in 1968. It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better time performance by using heuristics.

Description

A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As A* traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

It uses a knowledge-plus-heuristic cost function of node x (usually denoted $f(x)$) to determine the order in which the search visits nodes in the tree. The cost function is a sum of two functions:

- the past path-cost function, which is the known distance from the starting node to the current node x (usually denoted $g(x)$)
- a future path-cost function, which is an admissible "heuristic estimate" of the distance from x to the goal (usually denoted $h(x)$).

The $h(x)$ part of the $f(x)$ function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

If the heuristic h satisfies the additional condition $h(x) \leq d(x, y) + h(y)$ for every edge (x, y) of the graph (where d denotes the length of that edge), then h is called monotone, or consistent. In such a case, A* can be implemented more efficiently—roughly speaking, no node needs to be processed more than once (see *closed set* below)—and A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) := d(x, y) + h(y) - h(x)$.

History

In 1968 Nils Nilsson suggested a heuristic approach for Shakey the Robot to navigate through a room containing obstacles. This path-finding algorithm, called A1, was a faster version of the then best known formal approach, Dijkstra's algorithm, for finding shortest paths in graphs. Bertram Raphael suggested some significant improvements upon this algorithm, calling the revised version A2. Then Peter E. Hart introduced an argument that established A2, with only minor changes, to be the best possible algorithm for finding shortest paths. Hart, Nilsson and Raphael then jointly developed a proof that the revised A2 algorithm was *optimal* for finding shortest paths under certain well-defined conditions.

Process

Like all informed search algorithms, it first searches the routes that *appear* to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account; the $g(x)$ part of the heuristic is the cost from the starting point, not simply the local cost from the previously expanded node.

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the *open set* or *fringe*. The lower $f(x)$ for a given node x , the higher its priority. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

Additionally, if the heuristic is *monotonic* (or consistent, see below), a *closed set* of nodes already traversed may be used to make the search more efficient.

Pseudocode

The following pseudocode describes the algorithm Wikipedia:Disputed statement:

```
function A*(start,goal)
    closedset := the empty set      // The set of nodes already
evaluated.

    openset := {start}    // The set of tentative nodes to be
evaluated, initially containing the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start,
goal)

    while openset is not empty
        current := the node in openset having the lowest f_score[]
value

        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] +
dist_between(current,neighbor)
```

```

    if neighbor not in openset or tentative_g_score < g_score[neighbor]
        came_from[neighbor] := current
        g_score[neighbor] := tentative_g_score
        f_score[neighbor] := g_score[neighbor] +
heuristic_cost_estimate(neighbor, goal)
    if neighbor not in openset
        add neighbor to openset

return failure

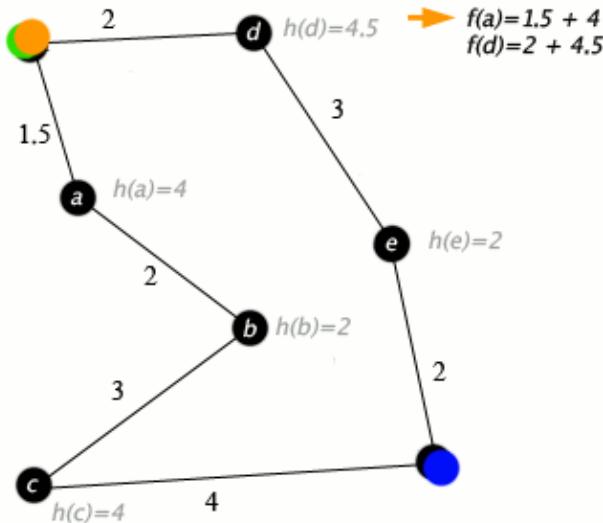
function reconstruct_path(came_from, current_node)
    if current_node in came_from
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node

```

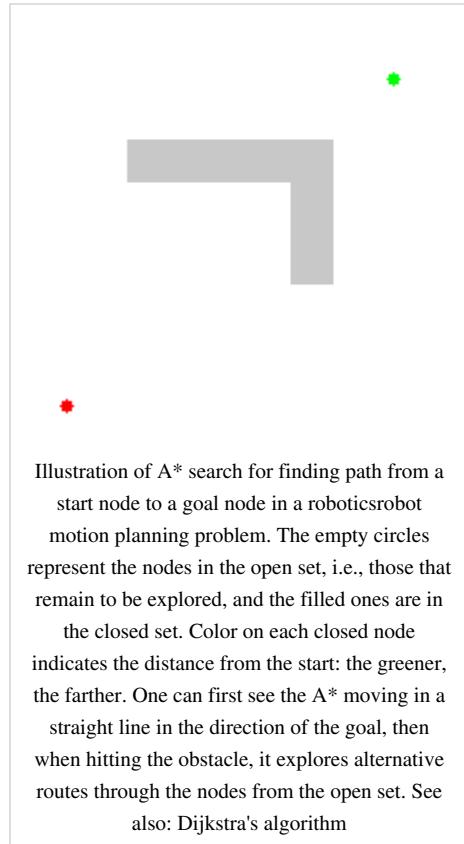
Remark: the above pseudocode assumes that the heuristic function is *monotonic* (or consistent, see below), which is a frequent case in many practical problems, such as the Shortest Distance Path in road networks. However, if the assumption is not true, nodes in the **closed** set may be rediscovered and their cost improved. In other words, the closed set can be omitted (yielding a tree search algorithm) if a solution is guaranteed to exist, or if the algorithm is adapted so that new nodes are added to the open set only if they have a lower f value than at any previous iteration.

Example

An example of an A star (A*) algorithm in action where nodes are cities connected with roads and $h(x)$ is the straight-line distance to target point:



Key: green: start; blue: goal; orange: visited



Properties

Like breadth-first search, A* is *complete* and will always find a solution if one exists.

If the heuristic function h is admissible, meaning that it never overestimates the actual minimal cost of reaching the goal, then A* is itself admissible (or *optimal*) if we do not use a closed set. If a closed set is used, then h must also be *monotonic* (or consistent) for A* to be optimal. This means that for any pair of adjacent nodes x and y , where $d(x,y)$ denotes the length of the edge between them, we must have:

$$h(x) \leq d(x, y) + h(y)$$

This ensures that for any path X from the initial node to x :

$$L(X) + h(x) \leq L(X) + d(x, y) + h(y) = L(Y) + h(y)$$

where L is a function that denotes the length of a path, and Y is the path X extended to include y . In other words, it is impossible to decrease (total distance so far + estimated remaining distance) by extending a path to include a neighboring node. (This is analogous to the restriction to nonnegative edge weights in Dijkstra's algorithm.) Monotonicity implies admissibility when the heuristic estimate at any goal node itself is zero, since (letting $P = (f, v_1, v_2, \dots, v_n, g)$ be a shortest path from any node f to the nearest goal g):

$$h(f) \leq d(f, v_1) + h(v_1) \leq d(f, v_1) + d(v_1, v_2) + h(v_2) \leq \dots \leq L(P) + h(g) = L(P)$$

A* is also optimally efficient for any heuristic h , meaning that no optimal algorithm employing the same heuristic will expand fewer nodes than A*, except when there are multiple partial solutions where h exactly predicts the cost of the optimal path. Even in this case, for each graph there exists some order of breaking ties in the priority queue such that A* examines the fewest possible nodes.

Special cases

Dijkstra's algorithm, as another example of a uniform-cost search algorithm, can be viewed as a special case of A* where $h(x) = 0$ for all x . General depth-first search can be implemented using the A* by considering that there is a global counter C initialized with a very large value. Every time we process a node we assign C to all of its newly discovered neighbors. After each single assignment, we decrease the counter C by one. Thus the earlier a node is discovered, the higher its $h(x)$ value. It should be noted, however, that both Dijkstra's algorithm and depth-first search can be implemented more efficiently without including a $h(x)$ value at each node.

Implementation details

There are a number of simple optimizations or implementation details that can significantly affect the performance of an A* implementation. The first detail to note is that the way the priority queue handles ties can have a significant effect on performance in some situations. If ties are broken so the queue behaves in a LIFO manner, A* will behave like depth-first search among equal cost paths (avoiding exploring more than one equally optimal solution).

When a path is required at the end of the search, it is common to keep with each node a reference to that node's parent. At the end of the search these references can be used to recover the optimal path. If these references are being kept then it can be important that the same node doesn't appear in the priority queue more than once (each entry corresponding to a different path to the node, and each with a different cost). A standard approach here is to check if a node about to be added already appears in the priority queue. If it does, then the priority and parent pointers are changed to correspond to the lower cost path. When finding a node in a queue to perform this check, many standard implementations of a min-heap require $O(n)$ time. Augmenting the heap with a hash table can reduce this to constant time.

Wikipedia:Please clarify.

Admissibility and optimality

A* is admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A* uses an "optimistic" estimate of the cost of a path through every node that it considers—optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

Here is the main idea of the proof:

When A* terminates its search, it has found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.

Suppose now that some other search algorithm B terminates its search with a path whose actual cost is *not* less than the estimated cost of a path through some open node. Based on the heuristic information it has, Algorithm B cannot rule out the possibility that a path through that node has a lower cost. So while B might consider fewer nodes than A*, it cannot be admissible. Accordingly, A* considers the fewest nodes of any admissible search algorithm.

This is only true if both:

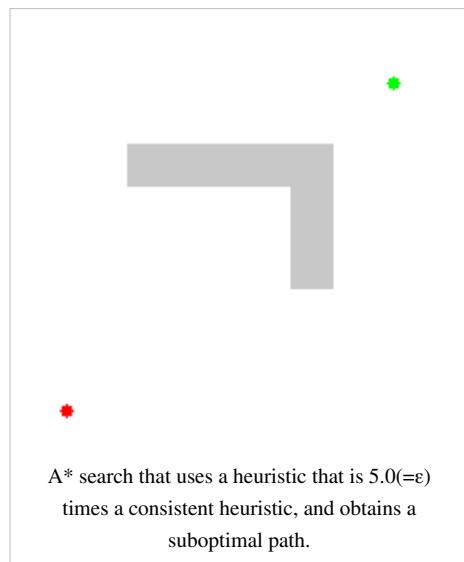
- A* uses an admissible heuristic. Otherwise, A* is not guaranteed to expand fewer nodes than another search algorithm with the same heuristic.
- A* solves only one search problem rather than a series of similar search problems. Otherwise, A* is not guaranteed to expand fewer nodes than incremental heuristic search algorithms.

Bounded relaxation

While the admissibility criterion guarantees an optimal solution path, it also means that A* must examine all equally meritorious paths to find the optimal path. It is possible to speed up the search at the expense of optimality by relaxing the admissibility criterion. Oftentimes we want to bound this relaxation, so that we can guarantee that the solution path is no worse than $(1 + \varepsilon)$ times the optimal solution path. This new guarantee is referred to as ε -admissible.

There are a number of ε -admissible algorithms:

- Weighted A*. If $h_a(n)$ is an admissible heuristic function, in the weighted version of the A* search one uses $h_w(n) = \varepsilon h_a(n)$, $\varepsilon > 1$ as the heuristic function, and perform the A* search as usual (which eventually happens faster than using h_a since fewer nodes are expanded). The path hence found by the search algorithm can have a cost of at most ε times that of the least cost path in the graph.
- Static Weighting uses the cost function $f(n) = g(n) + (1 + \varepsilon)h(n)$.
- Dynamic Weighting uses the cost function $f(n) = g(n) + (1 + \varepsilon w(n))h(n)$, where $w(n) = \begin{cases} 1 - \frac{d(n)}{N} & d(n) \leq N \\ 0 & \text{otherwise} \end{cases}$, and where $d(n)$ is the depth of the search and N is the anticipated length of the solution path.
- Sampled Dynamic Weighting uses sampling of nodes to better estimate and debias the heuristic error.
- A_{ε}^* . uses two heuristic functions. The first is the FOCAL list, which is used to select candidate nodes, and the second h_F is used to select the most promising node from the FOCAL list.



- A_ϵ selects nodes with the function $A f(n) + B h_F(n)$, where A and B are constants. If no nodes can be selected, the algorithm will backtrack with the function $C f(n) + D h_F(n)$, where C and D are constants.
- AlphA* attempts to promote depth-first exploitation by preferring recently expanded nodes. AlphA* uses the cost function $f_\alpha(n) = (1 + w_\alpha(n)) f(n)$, where $w_\alpha(n) = \begin{cases} \lambda & g(\pi(n)) \leq g(\tilde{n}) \\ \Lambda & \text{otherwise} \end{cases}$, where λ and Λ are constants with $\lambda \leq \Lambda$, $\pi(n)$ is the parent of n , and \tilde{n} is the most recently expanded node.

Complexity

The time complexity of A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

where h^* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h^* that returns the true distance from x to the goal.

Applications

A* is commonly used for the common pathfinding problem in applications such as games, but was originally designed as a general graph traversal algorithm. It finds applications to diverse problems, including the problem of parsing using stochastic grammars in NLP.

Variants of A*

- D*
- Field D*
- IDA*
- Fringe
- Fringe Saving A* (FSA*)
- Generalized Adaptive A* (GAA*)
- Lifelong Planning A* (LPA*)
- Simplified Memory bounded A* (SMA*)
- Jump point search
- Theta*
- A* can be adapted to a bidirectional search algorithm. Special care needs to be taken for the stopping criterion.^[1]

References

[1] from Princeton University

Further reading

- Hart, P. E.; Nilsson, N. J.; Raphael, B. (1972). "Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"". *SIGART Newsletter* 37: 28–29. doi: 10.1145/1056777.1056779 (<http://dx.doi.org/10.1145/1056777.1056779>).
- Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto, California: Tioga Publishing Company. ISBN 0-935382-01-1.

External links

- A* Pathfinding for Beginners (<http://www.policyalmanac.org/games/aStarTutorial.htm>)
- A* with Jump point search (<http://harablog.wordpress.com/2011/09/07/jump-point-search/>)
- Clear visual A* explanation, with advice and thoughts on path-finding (<http://theory.stanford.edu/~amitp/GameProgramming/>)
- Variation on A* called Hierarchical Path-Finding A* (HPA*) (<http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf>)
- A* Algorithm tutorial (<http://www.heyes-jones.com/astar.html>)
- A* Pathfinding in Objective-C (Xcode) (<http://www.humblebeesoft.com/blog/?p=18>)
- Dyna-h, an A*-similar heuristic approach in Reinforcement Learning (<http://dx.doi.org/10.1016/j.knosys.2011.09.008>)

Longest path problem

In graph theory and theoretical computer science, the **longest path problem** is the problem of finding a simple path of maximum length in a given graph. A path is called simple if it does not have any repeated vertices; the length of a path may either be measured by its number of edges, or (in weighted graphs) by the sum of the weights of its edges. In contrast to the shortest path problem, which can be solved in polynomial time in graphs without negative-weight cycles, the longest path problem is NP-hard, meaning that it cannot be solved in polynomial time for arbitrary graphs unless P = NP. Stronger hardness results are also known showing that it is difficult to approximate. However, it has a linear time solution for directed acyclic graphs, which has important applications in finding the critical path in scheduling problems.

NP-hardness

The NP-hardness of the unweighted longest problem can be shown using a reduction from the Hamiltonian path problem: a graph G has a Hamiltonian path if and only if its longest path has length $n - 1$, where n is the number of vertices in G . Because the Hamiltonian path problem is NP-complete, this reduction shows that the decision version of the longest path problem is also NP-complete. In this decision problem, the input is a graph G and a number k ; the desired output is "yes" if G contains a path of k or more edges, and *no* otherwise.

If the longest path problem could be solved in polynomial time, it could be used to solve this decision problem, by finding a longest path and then comparing its length to the number k . Therefore, the longest path problem is NP-hard. It is not NP-complete, because it is not a decision problem.

In weighted complete graphs with non-negative edge weights, the weighted longest path problem is the same as the Travelling salesman path problem, because the longest path always includes all vertices.

Acyclic graphs and critical paths

A longest path between two given vertices s and t in a weighted graph G is the same thing as a shortest path in a graph $-G$ derived from G by changing every weight to its negation. Therefore, if shortest paths can be found in $-G$, then longest paths can also be found in G .

For most graphs, this transformation is not useful because it creates cycles of negative length in $-G$. But if G is a directed acyclic graph, then no negative cycles can be created, and longest paths in G can be found in linear time by applying a linear time algorithm for shortest paths in $-G$, which is also a directed acyclic graph. For instance, for each vertex v in a given DAG, the length of the longest path ending at v may be obtained by the following steps:

1. Find a topological ordering of the given DAG.

2. For each vertex v of the DAG, in the topological ordering, compute the length of the longest path ending at v by looking at its incoming neighbors and adding one to the maximum length recorded for those neighbors. If v has no incoming neighbors, set the length of the longest path ending at v to zero. In either case, record this number so that later steps of the algorithm can access it.

Once this has been done, the longest path in the whole DAG may be obtained by starting at the vertex v with the largest recorded value, then repeatedly stepping backwards to its incoming neighbor with the largest recorded value, and reversing the sequence of vertices found in this way.

The critical path method for scheduling a set of activities involves the construction of a directed acyclic graph in which the vertices represent project milestones and the edges represent activities that must be performed after one milestone and before another; each edge is weighted by an estimate of the amount of time the corresponding activity will take to complete. In such a graph, the longest path from the first milestone to the last one is the critical path, which describes the total time for completing the project.

Longest paths of directed acyclic graphs may also be applied in layered graph drawing: assigning each vertex v of a directed acyclic graph G to the layer whose number is the length of the longest path ending at v results in a layer assignment for G with the minimum possible number of layers.

Approximation

Björklund, Husfeldt & Khanna (2004) write that the longest path problem in unweighted undirected graphs "is notorious for the difficulty of understanding its approximation hardness". The best polynomial time approximation algorithm known for this case achieves only a very weak approximation ratio, $n / \exp(\Omega(\sqrt{\log n}))$.^[1] For all $\epsilon > 0$, it is not possible to approximate the longest path to within a factor of $2^{(\log n)^{1-\epsilon}}$ unless NP is contained within quasi-polynomial deterministic time; however, there is a big gap between this inapproximability result and the known approximation algorithms for this problem.

In the case of unweighted but directed graphs, strong inapproximability results are known. For every $\epsilon > 0$ the problem cannot be approximated to within a factor of $n^{1-\epsilon}$ unless P=NP, and with stronger complexity-theoretic assumptions it cannot be approximated to within a factor of $n / \log^{2+\epsilon} n$. The color-coding technique can be used to find paths of logarithmic length, if they exist, but this gives an approximation ratio of only $O(n / \log n)$.

Parameterized complexity

The longest path problem is fixed-parameter tractable when parameterized by the length of the path. For instance, it can be solved in time linear in the size of the input graph (but exponential in the length of the path), by an algorithm that performs the following steps:

1. Perform a depth-first search of the graph. Let d be the depth of the resulting depth-first search tree.
2. Use the sequence of root-to-leaf paths of the depth-first search tree, in the order in which they were traversed by the search, to construct a path decomposition of the graph, with pathwidth d .
3. Apply dynamic programming to this path decomposition to find a longest path in time $O(d!2^d n)$, where n is the number of vertices in the graph.

Since the output path has length at least as large as d , the running time is also bounded by $O(\ell!2^\ell n)$, where ℓ is the length of the longest path.^[2] Using color-coding, the dependence on path length can be reduced to singly exponential. A similar dynamic programming technique shows that the longest path problem is also fixed-parameter tractable when parameterized by the treewidth of the graph.

For graphs of bounded clique-width, the longest path can also be solved by a polynomial time dynamic programming algorithm. However, the exponent of the polynomial depends on the clique-width of the graph, so this algorithm is not fixed-parameter tractable. The longest path problem, parameterized by clique-width, is hard for the

parameterized complexity class $W[1]$, showing that a fixed-parameter tractable algorithm is unlikely to exist.

Special classes of graphs

The longest path problem may be solved in polynomial time on the complements of comparability graphs.^[3] It may also be solved in polynomial time on any class of graphs with bounded treewidth or bounded clique-width, such as the distance-hereditary graphs. However, it is NP-hard even when restricted to split graphs, circle graphs, or planar graphs.

References

- [1] . For earlier work with even weaker approximation bounds, see and .
- [2] . For an earlier FPT algorithm with slightly better dependence on the path length, but worse dependence on the size of the graph, see .
- [3] . For earlier work on more restrictive subclasses of co-comparability graphs, see and .

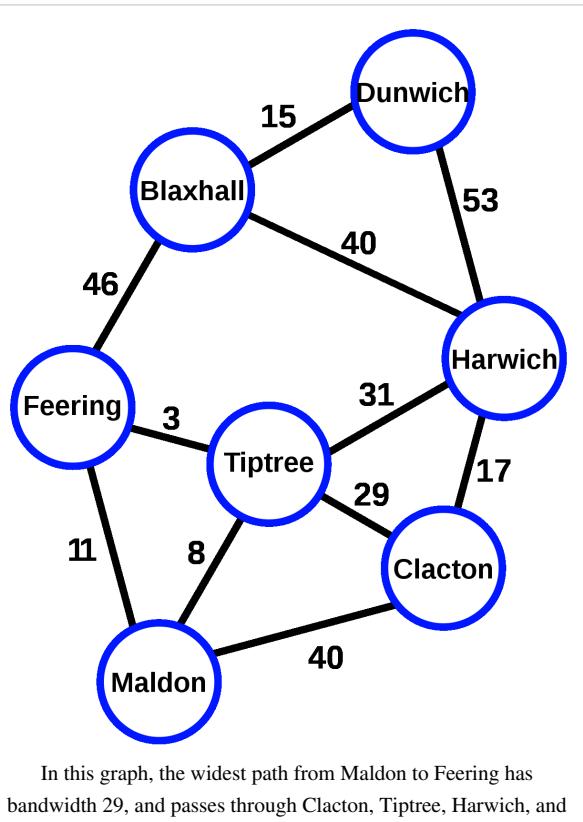
External links

- "Find the Longest Path (<http://valis.cs.uiuc.edu/~sariel/misc/funny/longestpath.mp3>)", song by Dan Barrett

Widest path problem

In graph algorithms, the **widest path problem**, also known as the **bottleneck shortest path problem** or the **maximum capacity path problem**, is the problem of finding a path between two designated vertices in a weighted graph, maximizing the weight of the minimum-weight edge in the path.

For instance, if the graph represents connections between routers in the Internet, and the weight of an edge represents the bandwidth of a connection between two routers, the widest path problem is the problem of finding an end-to-end path between two Internet nodes that has the maximum possible bandwidth. The weight of the minimum-weight edge is known as the capacity or bandwidth of the path. As well as its applications in network routing, the widest path problem is also an important component of the Schulze method for deciding the winner of a multiway election, and has been applied to digital compositing, metabolic analysis, and the computation of maximum flows. It is possible to adapt most shortest path algorithms to compute widest paths, by modifying them to use the bottleneck distance instead of path length. However, in many cases even faster algorithms are possible.



In this graph, the widest path from Maldon to Feering has bandwidth 29, and passes through Clacton, Tiptree, Harwich, and Blaxhall.

A closely related problem, the **minimax path problem**, asks for the path that minimizes the maximum weight of any of its edges. It has applications that include transportation planning. Any algorithm for the widest path problem can be transformed into an algorithm for the minimax path problem, or vice versa, by reversing the sense of all the

weight comparisons performed by the algorithm, or equivalently by replacing every edge weight by its negation.

Undirected graphs

In an undirected graph, a widest path may be found as the path between the two vertices in the maximum spanning tree of the graph, and a minimax path may be found as the path between the two vertices in the minimum spanning tree.

In any graph, directed or undirected, there is a straightforward algorithm for finding a widest path once the weight of its minimum-weight edge is known: simply delete all smaller edges and search for any path among the remaining edges using breadth first search or depth first search. Based on this test, there also exists a linear time algorithm for finding a widest $s-t$ path in an undirected graph, that does not use the maximum spanning tree. The main idea of the algorithm is to apply the linear-time path-finding algorithm to the median edge weight in the graph, and then either to delete all smaller edges or contract all larger edges according to whether a path does or does not exist, and recurse in the resulting smaller graph.

Fernandez, Garfinkel & Arbiol (1998) use undirected bottleneck shortest paths in order to form composite aerial photographs that combine multiple images of overlapping areas. In the subproblem to which the widest path problem applies, two images have already been transformed into a common coordinate system; the remaining task is to select a *seam*, a curve that passes through the region of overlap and divides one of the two images from the other. Pixels on one side of the seam will be copied from one of the images, and pixels on the other side of the seam will be copied from the other image; unlike other compositing methods that average pixels from both images, this produces a valid photographic image of every part of the region being photographed. They weight the edges of a grid graph by a numeric estimate of how visually apparent a seam through that point would be; the choice of a bottleneck shortest path as the seam, rather than a more conventional shortest path, forces their system to find a seam that is difficult to discern at all of its points, rather than allowing it to trade off greater visibility in one part of the image for lesser visibility elsewhere.

If all edge weights of an undirected graph are positive, then the minimax distances between pairs of points (the maximum edge weights of minimax paths) form an ultrametric; conversely every finite ultrametric space comes from minimax distances in this way. A data structure constructed from the minimum spanning tree allows the minimax distance between any pair of vertices to be computed in constant time per distance, using lowest common ancestor queries in a Cartesian tree. The root of the Cartesian tree represents the heaviest minimum spanning tree edge, and the children of the root are Cartesian trees recursively constructed from the subtrees of the minimum spanning tree formed by removing the heaviest edge. The leaves of the Cartesian tree represent the vertices of the input graph, and the minimax distance between two vertices equals the weight of the Cartesian tree node that is their lowest common ancestor. Once the minimum spanning tree edges have been sorted, this Cartesian tree can be constructed in linear time.

Directed graphs

In directed graphs, the maximum spanning tree solution cannot be used. Instead, several different algorithms are known; the choice of which algorithm to use depends on whether a start or destination vertex for the path is fixed, or whether paths for many start or destination vertices must be found simultaneously.

All pairs

The all-pairs widest path problem has applications in the Schulze method for choosing a winner in multiway elections in which voters rank the candidates in preference order. The Schulze method constructs a complete directed graph in which the vertices represent the candidates and every two vertices are connected by an edge. Each edge is directed from the winner to the loser of a pairwise contest between the two candidates it connects, and is labeled with

the margin of victory of that contest. Then the method computes widest paths between all pairs of vertices, and the winner is the candidate whose vertex has wider paths to each opponent than vice versa. The results of an election using this method are consistent with the Condorcet method – a candidate who wins all pairwise contests automatically wins the whole election – but it generally allows a winner to be selected, even in situations where the Condorcet method itself fails.^[1] The Schulze method has been used by several organizations including the Wikimedia Foundation.^[2]

To compute the widest path widths for all pairs of nodes in a dense directed graph, such as the ones that arise in the voting application, the asymptotically fastest known approach takes time $O(n^{(3+\omega)/2})$ where ω is the exponent for fast matrix multiplication. Using the best known algorithms for matrix multiplication, this time bound becomes $O(n^{2.688})$.^[3] Instead, the reference implementation for the Schulze method uses a modified version of the simpler Floyd–Warshall algorithm, which takes $O(n^3)$ time. For sparse graphs, it may be more efficient to repeatedly apply a single-source widest path algorithm.

Single source

If the edges are sorted by their weights, then a modified version of Dijkstra's algorithm can compute the bottlenecks between a designated start vertex and every other vertex in the graph, in linear time. The key idea behind the speedup over a conventional version of Dijkstra's algorithm is that the sequence of bottleneck distances to each vertex, in the order that the vertices are considered by this algorithm, is a monotonic subsequence of the sorted sequence of edge weights; therefore, the priority queue of Dijkstra's algorithm can be replaced by an array indexed by the numbers from 1 to m (the number of edges in the graph), where array cell i contains the vertices whose bottleneck distance is the weight of the edge with position i in the sorted order. This method allows the widest path problem to be solved as quickly as sorting; for instance, if the edge weights are represented as integers, then the time bounds for integer sorting a list of m integers would apply also to this problem.

Single source and single destination

Berman & Handler (1987) suggest that service vehicles and emergency vehicles should use minimax paths when returning from a service call to their base. In this application, the time to return is less important than the response time if another service call occurs while the vehicle is in the process of returning. By using a minimax path, where the weight of an edge is the maximum travel time from a point on the edge to the farthest possible service call, one can plan a route that minimizes the maximum possible delay between receipt of a service call and arrival of a responding vehicle. Ullah, Lee & Hassoun (2009) use maximin paths to model the dominant reaction chains in metabolic networks; in their model, the weight of an edge is the free energy of the metabolic reaction represented by the edge.

Another application of widest paths arises in the Ford–Fulkerson algorithm for the maximum flow problem. Repeatedly augmenting a flow along a maximum capacity path in the residual network of the flow leads to a small bound, $O(m \log U)$, on the number of augmentations needed to find a maximum flow; here, the edge capacities are assumed to be integers that are at most U . However, this analysis does not depend on finding a path that has the exact maximum of capacity; any path whose capacity is within a constant factor of the maximum suffices. Combining this approximation idea with the shortest path augmentation method of the Edmonds–Karp algorithm leads to a maximum flow algorithm with running time $O(mn \log U)$.

It is possible to find maximum-capacity paths and minimax paths with a single source and single destination very efficiently even in models of computation that allow only comparisons of the input graph's edge weights and not arithmetic on them. The algorithm maintains a set S of edges that are known to contain the bottleneck edge of the optimal path; initially, S is just the set of all m edges of the graph. At each iteration of the algorithm, it splits S into an ordered sequence of subsets S_1, S_2, \dots of approximately equal size; the number of subsets in this partition is chosen in such a way that all of the split points between subsets can be found by repeated median-finding in time

$O(m)$. The algorithm then reweights each edge of the graph by the index of the subset containing the edge, and uses the modified Dijkstra algorithm on the reweighted graph; based on the results of this computation, it can determine in linear time which of the subsets contains the bottleneck edge weight. It then replaces S by the subset S_i that it has determined to contain the bottleneck weight, and starts the next iteration with this new set S . The number of subsets into which S can be split increases exponentially with each step, so the number of iterations is proportional to the iterated logarithm function, $O(\log^* n)$, and the total time is $O(m \log^* n)$. In a model of computation where each edge weight is a machine integer, the use of repeated bisection in this algorithm can be replaced by a list-splitting technique of Han & Thorup (2002), allowing S to be split into $O(\sqrt{m})$ smaller sets S_i in a single step and leading to a linear overall time bound.

Euclidean point sets

A variant of the minimax path problem has also been considered for sets of points in the Euclidean plane. As in the undirected graph problem, this can be solved efficiently by finding a Euclidean minimum spanning tree. However, it becomes more complicated when a path is desired that not only minimizes the hop length but also, among paths with the same hop length, minimizes or approximately minimizes the total length of the path. The solution can be approximated using geometric spanners.

In number theory, the unsolved Gaussian moat problem asks whether or not minimax paths in the Gaussian prime numbers have bounded or unbounded minimax length. That is, does there exist a constant B such that, for every pair of points p and q in the infinite Euclidean point set defined by the Gaussian primes, the minimax path in the Gaussian primes between p and q has minimax edge length at most B ?

External links

- Java implementations^[4] of the shortest minimax path problem. Algorithms use incremental link removal method to find the shortest minimax paths. Further, it also provides a modified version of Floyd-Warshall.

References

- [1] More specifically, the only kind of tie that the Schulze method fails to break is between two candidates who have equally wide paths to each other.
- [2] See Jesse Plamondon-Willard, Board election to use preference voting, May 2008; Mark Ryan, 2008 Wikimedia Board Election results, June 2008; 2008 Board Elections, June 2008; and 2009 Board Elections, August 2009.
- [3] . For an earlier algorithm that also used fast matrix multiplication to speed up all pairs widest paths, see and Chapter 5 of
- [4] <https://gist.github.com/vy/6580214>

Canadian traveller problem

In computer science and graph theory, the **Canadian Traveller Problem** (CTP) is a generalization of the shortest path problem to graphs that are *partially observable*. In other words, the graph is revealed while it is being explored, and explorative edges are charged even if they do not contribute to the final path.

This optimization problem was introduced by Christos Papadimitriou and Mihalis Yannakakis in 1989 and a number of variants of the problem have been studied since. The name supposedly originates from conversations of the authors who learned of the difficulty Canadian drivers had with snowfall randomly blocking roads. The stochastic version, where each edge is associated with a probability of independently being in the graph, has been given considerable attention in operations research under the name "the Stochastic Shortest Path Problem with Recourse" (SSPPR).

Problem description

For a given instance, there are a number of possibilities, or *realizations*, of how the hidden graph may look. Given an instance, a description of how to follow the instance in the best way is called a *policy*. The CTP task is to compute the expected cost of the optimal policies. To compute an actual description of an optimal policy may be a harder problem.

Given an instance and policy for the instance, every realization produces its own (deterministic) walk in the graph. Note that the walk is not necessarily a path since the best strategy may be to, e.g., visit every vertex of a cycle and return to the start. This differs from the shortest path problem (with strictly positive weights), where repetitions in a walk implies that a better solution exists.

Variants

There are primarily five parameters distinguishing the number of variants of the Canadian Traveller Problem. The first parameter is how to value the walk produced by a policy for a given instance and realization. In the Stochastic Shortest Path Problem with Recourse, the goal is simply to minimize the cost of the walk (defined as the sum over all edges of the cost of the edge times the number of times that edge was taken). For the Canadian Traveller Problem, the task is to minimize the competitive ratio of the walk; i.e., to minimize the number of times longer the produced walk is to the shortest path in the realization.

The second parameter is how to evaluate a policy with respect to different realizations consistent with the instance under consideration. In the Canadian Traveller Problem, one wishes to study the worst case and in SSPPR, the average case. For average case analysis, one must furthermore specify an a priori distribution over the realizations.

The third parameter is restricted to the stochastic versions and is about what assumptions we can make about the distribution of the realizations and how the distribution is represented in the input. In the Stochastic Canadian Traveller Problem and in the Edge-independent Stochastic Shortest Path Problem (i-SSPPR), each uncertain edge (or cost) has an associated probability of being in the realization and the event that an edge is in the graph is independent of which other edges are in the realization. Even though this is a considerable simplification, the problem is still #P-hard. Another variant is to make no assumption on the distribution but require that each realization with non-zero probability be explicitly stated (such as "Probability 0.1 of edge set { {3,4},{1,2} }, probability 0.2 of..."). This is called the Distribution Stochastic Shortest Path Problem (d-SSPPR or R-SSPPR) and is NP-complete. The first variant is harder than the second because the former can represent in logarithmic space some distributions that the latter represents in linear space.

The fourth and final parameter is how the graph changes over time. In CTP and SSPPR, the realization is fixed but not known. In the Stochastic Shortest Path Problem with Recourse and Resets or the Expected Shortest Path problem, a new realization is chosen from the distribution after each step taken by the policy. This problem can be

solved in polynomial time by reducing it to a Markov decision process with polynomial horizon. The Markov generalization, where the realization of the graph may influence the next realization, is known to be much harder.

An additional parameter is how new knowledge is being discovered on the realization. In traditional variants of CTP, the agent uncovers the exact weight (or status) of an edge upon reaching an adjacent vertex. A new variant was recently suggested where an agent also has the ability to perform remote sensing from any location on the realization. In this variant, the task is to minimize the travel cost plus the cost of sensing operations.

Formal definition

We define the variant studied in the paper from 1989. That is, the goal is to minimize the competitive ratio in the worst case. It is necessary that we begin by introducing certain terms.

Consider a given graph and the family of undirected graphs that can be constructed by adding one or more edges from a given set. Formally, let $\mathcal{G}(V, E, F) = \{(V, E + F') | F' \subseteq F\}$, $E \cap F = \emptyset$ where we think of E as the edges that must be in the graph and of F as the edges that may be in the graph. We say that $G \in \mathcal{G}(V, E, F)$ is a *realization* of the graph family. Furthermore, let W be an associated cost matrix where w_{ij} is the cost of going from vertex i to vertex j , assuming that this edge is in the realization.

For any vertex v in V , we call $E_B(v, V)$ its incident edges with respect to the edge set B on V . Furthermore, for a realization $G \in \mathcal{G}(V, E, F)$, let $d_B(s, t)$ be the cost of the shortest path in the graph from s to t . This is called the off-line problem because an algorithm for such a problem would have complete information of the graph. We say that a strategy π to navigate such a graph is a mapping from $(\mathcal{P}(E), \mathcal{P}(F), V)$ to V , where $\mathcal{P}(X)$ denotes the powerset of X . We define the cost $c(\pi, B)$ of a strategy π with respect to a particular realization $G = (V, B)$ as follows.

- Let $v_0 = s$, $E_0 = E$ and $F_0 = F$.
- For $i = 0, 1, 2, \dots$, define
 - $E_{i+1} = E_i \cup E_B(v_i, V)$,
 - $F_{i+1} = F_i - E_F(v_i, V)$, and
 - $v_{i+1} = \pi(E_{i+1}, F_{i+1}, v_i)$.
- If there exists a T such that $v_T = t$, then $c(\pi, B) = \sum_{i=0}^{T-1} w_{v_i, v_{i+1}}$; otherwise let $c(\pi, B) = \infty$.

In other words, we evaluate the policy based on the edges we currently know are in the graph (E_i) and the edges we know might be in the graph (F_i). When we take a step in the graph, the edges incident to our new location become known to us. Those edges that are in the graph are added to E_i , and regardless of whether the edges are in the graph or not, they are removed from the set of unknown edges, F_i . If the goal is never reached, we say that we have an infinite cost. If the goal is reached, we define the cost of the walk as the sum of the costs of all of the edges traversed, with cardinality.

Finally, we define the Canadian traveller problem.

Given a CTP instance (V, E, F, s, t, r) , decide whether there exists a policy π such that for every realization $(V, B) \in \mathcal{G}(V, E, F)$, the cost $c(\pi, B)$ of the policy is no more than r times the off-line optimal, $d_B(s, t)$.

Papadimitriou and Yannakakis noted that this defines a two-player game, where the players compete over the cost of their respective paths and the edge set is chosen by the second player (nature).

Complexity

The original paper analysed the complexity of the problem and reported it to be PSPACE-complete. It was also shown that finding an optimal path in the case where each edge has an associated probability of being in the graph (i-SSPPR) is a PSPACE-easy but #P-hard problem.^[1] It is an open problem to bridge this gap.

The directed version of the stochastic problem is known in operations research as the Stochastic Shortest Path Problem with Recourse.

Applications

The problem is said to have applications in operations research, transportation planning, artificial intelligence, machine learning, communication networks, and routing. A variant of the problem has been studied for robot navigation with probabilistic landmark recognition.

Open problems

Despite the age of the problem and its many potential applications, many natural questions still remain open. Is there a constant-factor approximation or is the problem APX-hard? Is i-SSPPR #P-complete? An even more fundamental question has been left unanswered: is there a polynomial-size *description* of an optimal policy, setting aside for a moment the time necessary to compute the description?^[2]

Notes

[1] Papadimitriou and Yannakakis, 1989, p. 148

[2] Karger and Nikolova, 2008, p. 1

References

- C.H. Papadimitriou; M. Yannakakis (1989). "Shortest paths without a map". *Lecture notes in computer science*. Proc. 16th ICALP **372**. Springer-Verlag. pp. 610–620.
- David Karger; Evdokia Nikolova (2008). *Exact Algorithms for the Canadian Traveller Problem on Paths and Trees*.
- Zahy Bnaya; Ariel Felner, Solomon Eyal Shimony (2009). *Canadian Traveller Problem with remote sensing*. International Joint Conference On Artificial Intelligence (IJCAI).

K shortest path routing

The **K shortest path routing** algorithm is an extension algorithm of the shortest path routing algorithm in a given network.

It is sometimes crucial to have more than one path between two nodes in a given network. In the event there are additional constraints, other paths different from the shortest path can be computed. To find the shortest path one can use shortest path algorithms such as Dijkstra's algorithm or Bellman Ford algorithm and extend them to find more than one path. The K Shortest path routing algorithm is a generalization of the shortest path problem. The algorithm not only finds the shortest path, but also K other paths in order of increasing cost. K is the number of shortest paths to find. The problem can be restricted to have the K shortest path without loops (loopless K shortest path) or with loop.

History

Since 1957 there have been many papers published on the K Shortest path routing algorithm problem. Most of the fundamental works on not just finding the single shortest path between a pair of nodes, but instead listing a sequence of the K shortest paths, were done between the 1960s and up to 2001. Since then, most of the recent research has been about the applications of the algorithm and its variants. In 2010, Michael Gunter et al. published a book on *Symbolic calculation of K-shortest paths and related measures with the stochastic process algebra tool CASPA*.^[1]

Important works on the K Shortest path problem:

Year of Publication	Author	Title
1971	Jin Y. Yen	Finding the K Shortest Loopless Paths in a Network ^[2]
1972	M. T. Ardon et al.	A recursive algorithm for generating circuits and related subgraphs ^[3]
1973	P. M. Camerini et al.	The k shortest spanning trees of a graph
1975	K. Aihara	An approach to enumerating elementary paths and cutsets by Gaussian elimination method
1976	T. D. Am et al.	An algorithm for generating all the paths between two vertices in a digraph and its application ^[4]
1988	Ravindra K. Ahuja et al.	Faster algorithms for the shortest path problem ^[5]
1989	S. Anily et al.	Ranking the best binary trees ^[6]
1990	Ravindra K. Ahuja et al.	Faster algorithms for the shortest path problem ^[5]
1993	El-Amin et al.	An expert system for transmission line route selection ^[7]
1997	David Eppstein	Finding the k Shortest Paths ^[8]
1977	Ingo Althöfer	On the K-best mode in computer chess: measuring the similarity of move proposals ^[9]
1999	Ingo Althöfer	Decision Support Systems With Multiple Choice Structure ^[10]
2011	Husain Aljazzar, Stefan Leue	K*: A heuristic search algorithm for finding the k shortest paths ^[11]

The BibTeX database^[12] contains more articles.

Algorithm

The Dijkstra algorithm can be generalized to find the K Shortest path.

Definitions:

- $G(V, E)$: weighted directed graph, with set of vertices V and set of directed edges E ,
 - $w(u, v)$: cost of directed edge from node u to node v (costs are non-negative).
- Links that do not satisfy constraints on the shortest path are removed from the graph
- s : the source node
 - t : the destination node
 - K : the number of shortest paths to find
 - P_u : a path from s to u
 - B is a heap data structure containing paths
 - P : set of shortest paths from s to t
 - count_u : number of shortest paths found to node u

Algorithm:

```

*P = empty,
*countu = 0, for all u in V
insert path Ps = {s} into B with cost 0
while B is not empty and countt < K:
    – let Pu be the shortest cost path in B with cost C
    – B = B - {Pu}, countu = countu + 1
    – if u = t then P = P ∪ Pu
    – if countu ≤ K then
        • for each vertex v adjacent to u:
            – let Pv be a new path with cost C + w(u, v) formed by concatenating edge (u, v) to
              path Pu
            – insert Pv into B

```

There are mainly two variants of the K shortest path routing algorithm:

First variant

In the first variant, the paths are not required to be loopless (this is the simple one): David Eppstein's algorithm achieves the best running time complexity.

Second variant

In the second variant, attributed to Jin Y. Yen, the paths are required to be loopless.^[13] (This restriction introduced another level of complexity.) Yen's algorithm is used where simple paths only are considered, whereas Eppstein's algorithm is used when non-simple paths are allowed (e.g., paths are allowed to revisit the same node multiple times).

Paths are not required to be loopless

In all that follows, **m** is the number of edges and **n** is the number of vertices.

Eppstein's algorithm provides the best results. Eppstein's model finds the K shortest paths (allowing cycles) connecting a given pair of vertices in a digraph, in time $O(m + n \log n + K)$.

Eppstein's algorithm uses a graph transformation technique.

This model can also find the K shortest paths from a given source **s** to each vertex in the graph, in total time $O(m + n \log n + kn)$.

Loopless K shortest path routing algorithm

The best running time for this model is attributed to Jin. Y. Yen.^[13] Yen's algorithm finds the lengths of all shortest paths from a fixed node to all other nodes in an n -node non negative-distance network. This technique only requires $2n^2$ additions and n^2 comparisons - which is less than other available algorithms require.

The running time complexity is $O(Kn(m + n \log n))$. m represents the number of edges and n is the number of vertices.

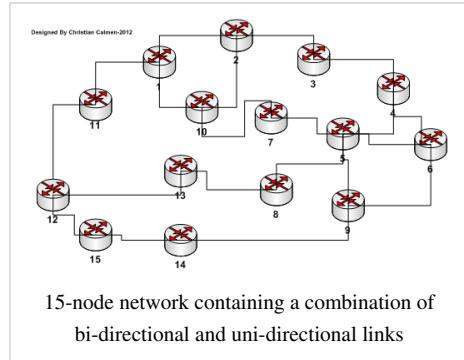
Some examples and description

Example #1

The following example makes use of Yen's model to find the first K shortest paths between communicating end nodes. That is, it finds the first, second, third, etc. up to the K^{th} shortest path. More details can be found here^[14]. The code provided in this example attempts to solve the K Shortest path routing problem for a 15-nodes network containing a combination of unidirectional and bidirectional links:

Example #2

Another example is the use of K Shortest algorithm to track multiple objects. The technique implements a multiple object tracker based on the K Shortest paths routing algorithm. A set of probabilistic occupancy maps is used as input. An object detector provides the input. The complete details can be found at "Computer Vision Laboratory^[15] – CVLAB".



Applications

The K Shortest path routing is a good alternative for:

- Geographic path planning^[16]
- Network routing, especially in optical mesh network where there are additional constraints that cannot be solved by using ordinary shortest path algorithms.
- Hypothesis generation in computational linguistics
- Sequence alignment and metabolic pathway finding in bioinformatics^[17]
- Multiple object tracking^[18] as described above
- Road Networks: road junctions are the nodes (vertices) and each edge (link) of the graph is associated with a road segment between two junctions.

Variations

There are two main variations of the K Shortest path routing algorithm as mentioned above. Other variations only fall in between these.

- In the first variation, loops are allowed, that is paths are allowed to revisit the same node multiple times. The following papers deal with this variation.
- The second variation deals with simple paths. It is also called loopless K Shortest path routing problem and is attributed to J. Y. Yen

Related problems

- Dijkstra's algorithm solves the single-source shortest path problems.
- The Bellman–Ford algorithm solves the single-source problem if edge weights may be negative.
- The breadth-first search algorithm is used when the search is only limited to two operations.
- The Floyd–Warshall algorithm solves all pairs shortest paths.
- Johnson's algorithm solves all pairs' shortest paths, and may be faster than Floyd–Warshall on sparse graphs.
- Perturbation theory finds (at worst) the locally shortest path.

Cherkassky et al.^[19] provide more algorithms and associated evaluations.

Notes

- [1] Michael Günther et al.: "Symbolic calculation of K-shortest paths and related measures with the stochastic process algebra tool CASPA". In: Int'l Workshop on Dynamic Aspects in Dependability Models for Fault-Tolerant Systems (DYADEM-FTS), ACM Press (2010) 13–18.
- [2] <http://adrian.idv.hk/lib/exe/fetch.php/y71-shortestpath.pdf>
- [3] <http://130.203.133.150/showciting;jsessionid=113691273F14638C24AC0F1657DA966F?cid=14257806>
- [4] http://www.researchgate.net/publication/216545964_An_algorithm_for_generating_all_the_paths_between_two_vertices_in_a_digraph_and_its_application
- [5] [https://domino.mpi-inf.mpg.de/intranet/ag1/ag1publ.nsf/AuthorEditorIndividualView/c9dd2bf40fd35d33c125713a0036b99d/\\$FILE/p213-ahuja.pdf?OpenElement](https://domino.mpi-inf.mpg.de/intranet/ag1/ag1publ.nsf/AuthorEditorIndividualView/c9dd2bf40fd35d33c125713a0036b99d/$FILE/p213-ahuja.pdf?OpenElement)
- [6] <http://dl2.acm.org/citation.cfm?id=75797>
- [7] <http://www.sciencedirect.com/science/article/pii/0142061590900109>
- [8] http://pdf.aminer.org/001/059/121/finding_the_k_shortest_paths.pdf
- [9] <http://fano.ics.uci.edu/cites/Document/On-the-K-best-mode-in-computer-chess-measuring-the-similarity-of-move-proposals.html>
- [10] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.9728>
- [11] <http://dx.doi.org/10.1016/j.artint.2011.07.003>
- [12] <http://www.ics.uci.edu/~eppstein/bibs/kpath.bib>
- [13] Yen J. Y: "Finding the K-Shortest Loopless Paths in a Network". Management Science 1971; 17:712–716
- [14] <http://www.technical-recipes.com/2012/the-k-shortest-paths-algorithm-in-c/#more-2432>
- [15] <http://cylab.epfl.ch/software/ksp/>
- [16] <http://raweb.inria.fr/rapportsactivite/RA2009/aspi/uid62.html>
- [17] <http://bioinformatics.oxfordjournals.org/content/21/16/3401.full.pdf?keytype=ref&ijkey=L BKAnjRh0mW0xP4>
- [18] <http://cylab.epfl.ch/publications/publications/2011/BerclazFTF11.pdf>
- [19] Cherkassky, Boris V.; Goldberg, Andrew V.; Radzik, Tomasz (1996). "Shortest paths algorithms: theory and experimental evaluation". Mathematical Programming. Ser. A 73 (2): 129–174.

References

- Michael Günther et al. (<http://www.unibw.de/inf3/personen/profs/siegle/Papers/DYADEM2010.pdf>): *Symbolic calculation of k-shortest paths and related measures with the stochastic process algebra tool CASPA*. In: Int'l Workshop on Dynamic Aspects in Dependability Models for Fault-Tolerant Systems (DYADEM-FTS), ACM Press (2010) 13–18
- Yen J. Y (<http://adrian.idv.hk/lib/exe/fetch.php/paper/y71-shortestpath.pdf>): *Finding the K-Shortest Loopless Paths in a Network*. Management Science 1971; 17:712–716
- David Eppstein (http://pdf.aminer.org/001/059/121/finding_the_k_shortest_paths.pdf): *Finding the k shortest paths*. 35th IEEE Symp. Foundations of Comp. Sci., Santa Fe, 1994, pp. 154–165. Tech. Rep. 94–26, ICS, UCI, 1994. SIAM J. Computing 28(2):652–673, 1998.
- <http://www.technical-recipes.com/2012/the-k-shortest-paths-algorithm-in-c/#more-2432>
- Multiple objects tracking technique using K-shortest path algorithm: <http://cvlab.epfl.ch/software/ksp/>
- BibTeX database: <http://www.ics.uci.edu/~eppstein/bibs/kpath.bib>
- Computer Vision Laboratory: <http://cvlab.epfl.ch/software/ksp/>

External links

- Implementation of Yen's algorithm (<http://code.google.com/p/k-shortest-paths/>)

Application: Centrality analysis of social networks

For the statistical concept, see Central tendency.

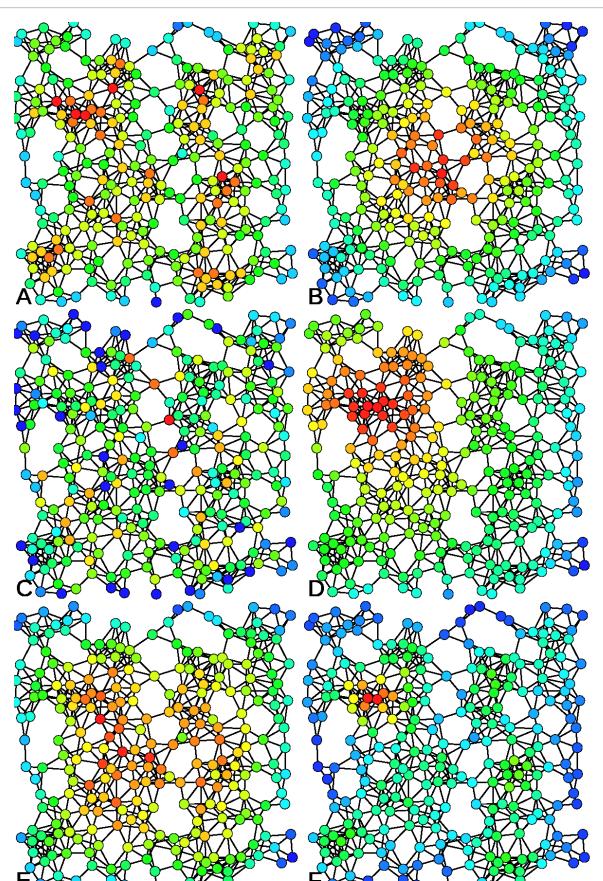
In graph theory and network analysis, **centrality** refers to indicators which identify the most important vertices within a graph. Applications include identifying the most influential person(s) in a social network, key infrastructure nodes in the Internet or urban networks, and super spreaders of disease. Centrality concepts were first developed in social network analysis, and many of the terms used to measure centrality reflect their sociological origin.^[1]

Definition and characterization of centrality indices

Centrality indices are answers to the question "What characterizes an important vertex?" The answer is given in terms of a real-valued function on the vertices of a graph, where the values produced are expected to provide a ranking which identifies the most important nodes.

The word "importance" has a wide number of meanings, leading to many different definitions of centrality. Two categorization schemes have been proposed. "Importance" can be conceived in relation to a type of flow or transfer across the network. This allows centralities to be classified by the type of flow they consider important. "Importance" can alternately be conceived as involvement in the cohesiveness of the network. This allows centralities to be classified based on how they measure cohesiveness. Both of these approaches divide centralities in distinct categories. A further conclusion is that a centrality which is appropriate for one category will often "get it wrong" when applied to a different category.

When centralities are categorized by their approach to cohesiveness, it becomes apparent that the majority of centralities inhabit one category. They are counts of the number of walks starting from a given vertex, and differ only in how walks are defined and counted. Restricting consideration to this group allows for a soft characterization which places centralities on a spectrum from walks of length one (degree centrality) to infinite walks (eigenvalue centrality). The observation that many centralities share this familial relationships perhaps explains the high rank correlations between these indices.



Examples of A) Degree centrality, B) Closeness centrality, C) Betweenness centrality, D) Eigenvector centrality, E) Katz centrality and F) Alpha centrality of the same graph.

Characterization by network flows

A network can be considered a description of the paths along which something flows. This allows a two-dimensional characterization based on the type of flow and the type of path encoded by the centrality. Flows represent either *parallel* or *serial* duplication. Parallel duplication implies that the item does not replicate during transfer. An example is the passing of a paperback novel from one person to another, or a specific dollar bill moving through the economy. Transfer is a special subclass of parallel duplication, since it implies a fixed final destination of the flow. Serial duplication implies that the item can duplicate during transfer. Examples include infectious disease and gossip/rumor spreading.

Likewise, the type of path can be constrained to: Geodiscs (shortest paths), paths (no vertex is visited more than once), trails (vertices can be visited multiple times, no edge is traversed more than once), or walks (vertices and edges can be visited/traversed multiple times).

Characterization by walk structure

An alternate classification can be derived from how the centrality is constructed. This again splits into two dimensions. Centralities are either *Radial* or *Medial*. Radial centralities count walks which start/end from the given vertex. The degree and eigenvalue centralities are examples of radial centralities, counting the number of walks of length one or length infinity. Medial centralities count walks which pass through the given vertex. The canonical example is Freedman's betweenness centrality, the number of shortest paths which pass through the given vertex.

Likewise, the counting can capture either the *volume* or the *length* of walks. Volume is the total number of walks of the given type. The three examples from the previous paragraph fall into this category. Length captures the distance from the given vertex to the remaining vertices in the graph. Freedman's closeness centrality, the total geodesic distance from a given vertex to all other vertices, is the best known example. Note that this classification is independent of the type of walk counted (i.e. walk, trail, path, geodisc).

Borgatti and Everett propose that this typology provides insight into how best to compare centrality measures. Centralities placed in the same box in this 2x2 classification are similar enough to make plausible alternatives; one can reasonably compare which is better for a given application. Measures from different boxes, however, are categorically distinct. Any evaluation of relative fitness can only occur within the context of predetermining which category is more applicable, rendering the comparison moot.

Radial-Volume centralities exist on a spectrum

The characterization by walk structure shows that almost all centralities in wide use are radial-volume measures. These encode the belief that a vertex's centrality is a function of the centrality of the vertices it is associated with. Centralities distinguish themselves on how association is defined.

Bonacich showed that if association is defined in terms of walks, then a family of centralities can be defined based on the length of walk considered. The degree counts walks of length one, the eigenvalue centrality counts walks of length infinity. Alternate definitions of association are also reasonable. The alpha centrality allows vertices to have an external source of influence. Estrada's subgraph centrality proposes only counting closed paths (triangles, squares, ...).

The heart of such measures is the observation that powers of the graph's adjacency matrix gives the number of walks corresponding to that power. Similarly, the matrix exponential is also closely related to the number of walks of a given length. An initial transformation of the adjacency matrix allows differing definition of the type of walk counted. Under either approach, the centrality of a vertex can be expressed as an infinite sum, either

$$\sum_{k=0}^{\infty} \beta^k A_R^k$$

for matrix powers or

$$\sum_{k=0}^{\infty} \frac{(\beta A_R)^k}{k!}$$

for matrix exponentials, where k is walk length, A_R is the transformed adjacency matrix, and β is a discount parameter which ensures convergence of the sum. Bonacich's family of measures does not transform the adjacency matrix. The alpha centrality replaces the adjacency matrix with its resolvent. The subgraph centrality replaces the adjacency matrix with its trace. A startling conclusion is that regardless of the initial transformation of the adjacency matrix, all such approaches have common limiting behavior. As β approaches zero, the indices converge to the degree centrality. As β approaches its maximal value, the indices converge to the eigenvalue centrality.

Important limitations

Centrality indices have two important limitations, one obvious and the other subtle. The obvious limitation is that a centrality which is optimal for one application is often sub-optimal for a different application. Indeed, if this were not so, we would not need so many different centralities.

The more subtle limitation is the commonly held fallacy that vertex centrality indicates the relative importance of vertices. Centrality indices are explicitly designed to produce a ranking which allows indication of the most important vertices. This they do well, under the limitation just noted. The error is two-fold. Firstly, a ranking only orders vertices by importance, it does not quantify the difference in importance between different levels of the ranking. Secondly, and more importantly, the features which (correctly) identify the most important vertices in a given network/application do not generalize to the remaining vertices. The rankings are meaningless for the vast majority of network nodes. This explains why, for example, only the first few results of a Google image search appear in a reasonable order.

While the failure of centrality indices to generalize to the rest of the network may at first seem counter-intuitive, it follows directly from the above definitions. Complex networks have heterogeneous topology. To the extent that the optimal measure depends on the network structure of the most important vertices, a measure which is optimal for such vertices is sub-optimal for the remainder of the network.

Degree centrality

Main article: Degree (graph theory)

Historically first and conceptually simplest is **degree centrality**, which is defined as the number of links incident upon a node (i.e., the number of ties that a node has). The degree can be interpreted in terms of the immediate risk of a node for catching whatever is flowing through the network (such as a virus, or some information). In the case of a directed network (where ties have direction), we usually define two separate measures of degree centrality, namely indegree and outdegree. Accordingly, indegree is a count of the number of ties directed to the node and outdegree is the number of ties that the node directs to others. When ties are associated to some positive aspects such as friendship or collaboration, indegree is often interpreted as a form of popularity, and outdegree as gregariousness.

The degree centrality of a vertex v , for a given graph $G := (V, E)$ with $|V|$ vertices and $|E|$ edges, is defined as

$$C_D(v) = \deg(v)$$

Calculating degree centrality for all the nodes in a graph takes $\Theta(V^2)$ in a dense adjacency matrix representation of the graph, and for edges takes $\Theta(E)$ in a sparse matrix representation.

Sometimes the interest is in finding the centrality of a graph within a graph.

The definition of centrality on the node level can be extended to the whole graph. Let v^* be the node with highest degree centrality in G . Let $X := (Y, Z)$ be the Y -node connected graph that maximizes the following quantity (with y^* being the node with highest degree centrality in X):

$$H = \sum_{j=1}^{|Y|} C_D(y*) - C_D(y_j)$$

Correspondingly, the degree centrality of the graph G is as follows:

$$C_D(G) = \frac{\sum_{i=1}^{|V|} [C_D(v*) - C_D(v_i)]}{H}$$

The value of H is maximized when the graph X contains one central node to which all other nodes are connected (a star graph), and in this case $H = (n - 1)(n - 2)$.

Closeness centrality

In connected graphs there is a natural distance metric between all pairs of nodes, defined by the length of their shortest paths. The **farness** of a node s is defined as the sum of its distances to all other nodes, and its closeness is defined as the inverse of the farness.^{[2][3]} Thus, the more central a node is the lower its total distance to all other nodes. Closeness can be regarded as a measure of how long it will take to spread information from s to all other nodes sequentially.

In the classic definition of the closeness centrality, the spread of information is modeled by the use of shortest paths. This model might not be the most realistic for all types of communication scenarios. Thus, related definitions have been discussed to measure closeness, like the random walk closeness centrality introduced by Noh and Rieger (2004). It measures the speed with which randomly walking messages reach a vertex from elsewhere in the network—a sort of random-walk version of closeness centrality.^[4]

The *information centrality* of Stephenson and Zelen (1989) is another closeness measure, which bears some similarity to that of Noh and Rieger. In essence it measures the harmonic mean of the resistance distances towards a vertex \mathbf{i} , which is smaller if \mathbf{i} has many paths of small resistance connecting it to other vertices.^[5]

Note that by definition of graph theoretic distances, the classic closeness centrality of all nodes in an unconnected graph would be 0. In a work by Dangalchev (2006) relating network vulnerability, the definition for closeness is modified such that it can be applied to graphs which lack connectivity:^[6]

$$C_C(v) = \sum_{t \in V \setminus v} 2^{-d_G(v,t)}.$$

This definition allows to create formulae for the closeness of two or more joined graphs. For example if vertex p of graph G_1 is connected to vertex q of graph G_2 then the closeness of the resulting graph is equal to:

$$C(G_1 + G_2) = C(G_1) + C(G_2) + (1 + C(p))(1 + C(q)).$$

Another extension to networks with disconnected components has been proposed by Opsahl (2010), and later studied by Boldi and Vigna (2013) in general directed graphs:

$$C_H(x) = \sum_{y \neq x} \frac{1}{d(y,x)}$$

The formula above, with the convention $1/\infty = 0$, defines *harmonic centrality*. It is a natural modification of Bavelas's definition of closeness following the general principle proposed by Marchiori and Latora (2000) that in networks with infinite distances the harmonic mean behaves better than the arithmetic mean. Indeed, Bavelas's closeness can be described as the denormalized reciprocal of the **arithmetic** mean of distances, whereas harmonic centrality is the denormalized reciprocal of the **harmonic** mean of distances.

Betweenness centrality

Main article: Betweenness centrality

Betweenness is a centrality measure of a vertex within a graph (there is also edge betweenness, which is not discussed here). Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes. It was introduced as a measure for quantifying the control of a human on the communication between other humans in a social network by Linton Freeman. In his conception, vertices that have a high probability to occur on a randomly chosen shortest path between two randomly chosen vertices have a high betweenness.

The betweenness of a vertex v in a graph $G := (V, E)$ with V vertices is computed as follows:

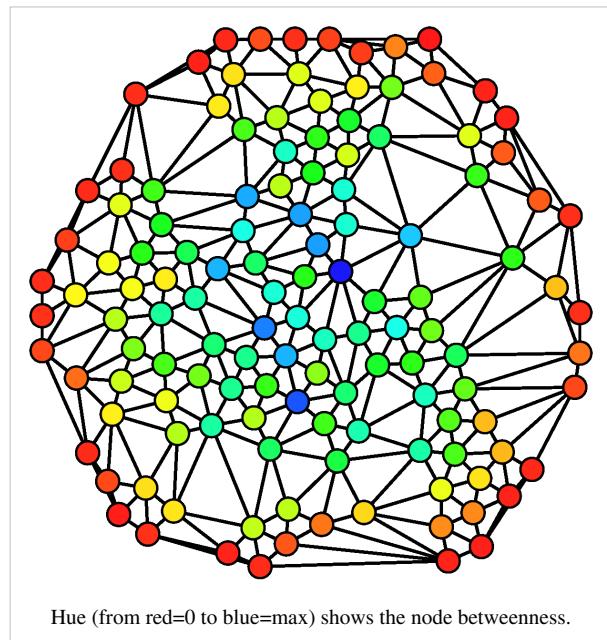
1. For each pair of vertices (s,t) , compute the shortest paths between them.
2. For each pair of vertices (s,t) , determine the fraction of shortest paths that pass through the vertex in question (here, vertex v).
3. Sum this fraction over all pairs of vertices (s,t) .

More compactly the betweenness can be represented as:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v . The betweenness may be normalised by dividing through the number of pairs of vertices not including v , which for directed graphs is $(n - 1)(n - 2)$ and for undirected graphs is $(n - 1)(n - 2)/2$. For example, in an undirected star graph, the center vertex (which is contained in every possible shortest path) would have a betweenness of $(n - 1)(n - 2)/2$ (if normalised) while the leaves (which are contained in no shortest paths) would have a betweenness of 0.

From a calculation aspect, both betweenness and closeness centralities of all vertices in a graph involve calculating the shortest paths between all pairs of vertices on a graph, which requires $\Theta(V^3)$ time with the Floyd–Warshall algorithm. However, on sparse graphs, Johnson's algorithm may be more efficient, taking $O(V^2 \log V + VE)$ time. In the case of unweighted graphs the calculations can be done with Brandes' algorithm which takes $O(VE)$ time. Normally, these algorithms assume that graphs are undirected and connected with the allowance of loops and multiple edges. When specifically dealing with network graphs, often graphs are without loops or multiple edges to maintain simple relationships (where edges represent connections between two people or vertices). In this case, using Brandes' algorithm will divide final centrality scores by 2 to account for each shortest path being counted twice.



Hue (from red=0 to blue=max) shows the node betweenness.

Eigenvector centrality

Eigenvector centrality is a measure of the influence of a node in a network. It assigns relative scores to all nodes in the network based on the concept that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. Google's PageRank is a variant of the Eigenvector centrality measure.^[7] Another closely related centrality measure is Katz centrality.

Using the adjacency matrix to find eigenvector centrality

For a given graph $G := (V, E)$ with $|V|$ number of vertices let $A = (a_{v,t})$ be the adjacency matrix, i.e. $a_{v,t} = 1$ if vertex v is linked to vertex t , and $a_{v,t} = 0$ otherwise. The centrality score of vertex v can be defined as:

$$x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t = \frac{1}{\lambda} \sum_{t \in G} a_{v,t} x_t$$

where $M(v)$ is a set of the neighbors of v and λ is a constant. With a small rearrangement this can be rewritten in vector notation as the eigenvector equation

$$\mathbf{Ax} = \lambda \mathbf{x}$$

In general, there will be many different eigenvalues λ for which an eigenvector solution exists. However, the additional requirement that all the entries in the eigenvector be positive implies (by the Perron–Frobenius theorem) that only the greatest eigenvalue results in the desired centrality measure. The v^{th} component of the related eigenvector then gives the centrality score of the vertex v in the network. Power iteration is one of many eigenvalue algorithms that may be used to find this dominant eigenvector. Furthermore, this can be generalized so that the entries in A can be real numbers representing connection strengths, as in a stochastic matrix.

Katz centrality and PageRank

Main article: Katz centrality

Katz centrality^[8] is a generalization of degree centrality. Degree centrality measures the number of direct neighbors, and Katz centrality measures the number of all nodes that can be connected through a path, while the contributions of distant nodes are penalized. Mathematically, it is defined as $x_i = \sum_{k=1}^{\infty} \sum_{j=1}^N \alpha^k (A^k)_{ji}$ where α is an attenuation factor in $(0, 1)$.

Katz centrality can be viewed as a variant of eigenvector centrality. Another form of Katz centrality is $x_i = \alpha \sum_{j=1}^N a_{ij} (x_j + 1)$. Compared to the expression of eigenvector centrality, x_j is replaced by $x_j + 1$.

It is shown that^[9] the principal eigenvector (associated with the largest eigenvalue of A , the adjacency matrix) is the limit of Katz centrality as α approaches $1/\lambda$ from below.

PageRank satisfies the following equation $x_i = \alpha \sum_j a_{ji} \frac{x_j}{L(j)} + \frac{1-\alpha}{N}$, where $L(j) = \sum_j a_{ij}$ is the number of neighbors of node j (or number of outbound links in a directed graph). Compared to eigenvector centrality and Katz centrality, one major difference is the scaling factor $L(j)$. Another difference between PageRank and eigenvector centrality is that the PageRank vector is a left hand eigenvector (note the factor a_{ji} has indices reversed).^[10]

Percolation Centrality

A slew of centrality measures exist to determine the ‘importance’ of a single node in a complex network. However, these measures quantify the importance of a node in purely topological terms, and the value of the node does not depend on the ‘state’ of the node in any way. It remains constant regardless of network dynamics. This is true even for the weighted betweenness measures. However, a node may very well be centrally located in terms of betweenness centrality or another centrality measure, but may not be ‘centrally’ located in the context of a network in which there is percolation. Percolation of a ‘contagion’ occurs in complex networks in a number of scenarios. For example, viral or bacterial infection can spread over social networks of people, known as contact networks. The spread of disease can also be considered at a higher level of abstraction, by contemplating a network of towns or population centres, connected by road, rail or air links. Computer viruses can spread over computer networks. Rumours or news about business offers and deals can also spread via social networks of people. In all of these scenarios, a ‘contagion’ spreads over the links of a complex network, altering the ‘states’ of the nodes as it spreads, either recoverably or otherwise. For example, in an epidemiological scenario, individuals go from ‘susceptible’ to ‘infected’ state as the infection spreads. The states the individual nodes can take in the above examples could be binary (such as received/not received a piece of news), discrete (susceptible/infected/recovered), or even continuous (such as the proportion of infected people in a town), as the contagion spreads. The common feature in all these scenarios is that the spread of contagion results in the change of node states in networks. Percolation centrality (PC) was proposed with this in mind, which specifically measures the importance of nodes in terms of aiding the percolation through the network. This measure was proposed by Piraveenan et al.

The Percolation Centrality is defined for a given node, at a given time, as the proportion of ‘percolated paths’ that go through that node. A ‘percolated path’ is a shortest path between a pair of nodes, where the source node is percolated (e.g., infected). The target node can be percolated or non-percolated, or in a partially percolated state.

$$PC^t(v) = \frac{1}{N-2} \sum_{s \neq v \neq r} \frac{\sigma_{sr}(v)}{\sigma_{sr}} \frac{x_s^t}{\sum [x_i^t] - x_v^t}$$

where $\sigma_{sr}(v)$ is total number of shortest paths from node s to node r and σ_{sr} is the number of those paths that pass through v . The percolation state of the node i at time t is denoted by x_i^t and two special cases are when $x_i^t = 0$ which indicates a non-percolated state at time t whereas when $x_i^t = 1$ which indicates a fully percolated state at time t . The values in between indicate partially percolated states (e.g., in a network of townships, this would be the percentage of people infected in that town).

The attached weights to the percolation paths depend on the percolation levels assigned to the source nodes, based on the premise that the higher the percolation level of a source node is, the more important are the paths that originate from that node. Nodes which lie on shortest paths originating from highly percolated nodes are therefore potentially more important to the percolation. The definition of PC may also be extended to include target node weights as well. Percolation centrality calculations run in $O(NM)$ time with an efficient implementation adopted from Brandes' fast algorithm and if the calculation needs to consider target nodes weights, the worst case time is $O(N^3)$.

Cross-Clique Centrality

Cross-Clique centrality of a single node, in a complex graph determines the connectivity of a node to different Cliques. A node with high cross-clique connectivity facilitates the propagation of information or disease in a graph. Cliques are subgraphs in which every node is connected to every other node in the clique. The cross-clique connectivity of a node v for a given graph $G := (V, E)$ with $|V|$ vertices and $|E|$ edges, is defined as $X(v)$ where $X(v)$ is the number of cliques to which vertex v belongs. This measure was proposed in.

Centralization

The *centralization* of any network is a measure of how central its most central node is in relation to how central all the other nodes are. Centralization measures then (a) calculate the sum in differences in centrality between the most central node in a network and all other nodes; and (b) divide this quantity by the theoretically largest such sum of differences in any network of the same size. Thus, every centrality measure can have its own centralization measure.

Defined formally, if $C_x(p_i)$ is any centrality measure of point i , if $C_x(p_*)$ is the largest such measure in the network, and if $\max \sum_{i=1}^N C_x(p_*) - C_x(p_i)$ is the largest sum of differences in point centrality C_x for any graph

of with the same number of nodes, then the centralization of the network is:

$$C_x = \frac{\sum_{i=1}^N C_x(p_*) - C_x(p_i)}{\max \sum_{i=1}^N C_x(p_*) - C_x(p_i)}$$

Extensions

Empirical and theoretical research have extended the concept of centrality in the context of static networks to dynamic centrality^[11] in the context of time-dependent and temporal networks.^{[12][13][14]}

For generalizations to weighted networks, see Opsahl et al. (2010).

The concept of centrality was extended to a group level as well. For example, **Group Betweenness** centrality shows the proportion of geodesics connecting pairs of non-group members that pass through the group.^{[15][16]}

Notes and references

- [1] Newman, M.E.J. 2010. *Networks: An Introduction*. Oxford, UK: Oxford University Press.
- [2] Alex Bavelas. Communication patterns in task-oriented groups. *J. Acoust. Soc. Am.*, **22**(6):725–730, 1950.
- [3] Sabidussi, G. (1966) The centrality index of a graph. *Psychometrika* **31**, 581–603.
- [4] J. D. Noh and H. Rieger, Phys. Rev. Lett. **92**, 118701 (2004).
- [5] Stephenson, K. A. and Zelen, M., 1989. Rethinking centrality: Methods and examples. *Social Networks* **11**, 1–37.
- [6] Dangalchev Ch., Residual Closeness in Networks, *Phisica A* **365**, 556 (2006).
- [7] <http://www.ams.org/samplings/feature-column/fcarc-pagerank>
- [8] Katz, L. 1953. A New Status Index Derived from Sociometric Index. *Psychometrika*, 39–43.
- [9] Bonacich, P., 1991. Simultaneous group and individual centralities. *Social Networks* **13**, 155–168.
- [10] How does Google rank webpages? (http://scenic.princeton.edu/network20q/lectures/Q3_notes.pdf) 20Q: About Networked Life
- [11] Braha, D. and Bar-Yam, Y. 2006. "From Centrality to Temporary Fame: Dynamic Centrality in Complex Networks." *Complexity* **12**: 59-63.
- [12] Hill,S.A. and Braha, D. 2010. "Dynamic Model of Time-Dependent Complex Networks." *Physical Review E* **82**, 046105.
- [13] Gross, T. and Sayama, H. (Eds.). 2009. *Adaptive Networks: Theory, Models and Applications*. Springer.
- [14] Holme, P. and Saramäki, J. 2013. *Temporal Networks*. Springer.
- [15] Everett, M. G. and Borgatti, S. P. (2005). Extending centrality. In P. J. Carrington, J. Scott and S. Wasserman (Eds.), *Models and methods in social network analysis* (pp. 57-76). New York: Cambridge University Press.
- [16] Puzis, R., Yagil, D., Elovici, Y., Braha, D. (2009). Collaborative attack on Internet users' anonymity (http://necsi.edu/affiliates/braha/Internet_Research_Anonymity.pdf), *Internet Research* **19**(1)

Further reading

- Koschützki, D.; Lehmann, K. A.; Peeters, L.; Richter, S.; Tenfelde-Podehl, D. and Zlotowski, O. (2005) Centrality Indices. In Brandes, U. and Erlebach, T. (Eds.) *Network Analysis: Methodological Foundations*, pp. 16–61, LNCS 3418, Springer-Verlag.

External links

- https://networkx.lanl.gov/trac/attachment/ticket/119/page_rank.py
- http://www.faculty.ucr.edu/~hanneman/nettext/C10_Centrality.html
- <http://socnetv.sourceforge.net/docs/analysis.html#CC>

Application: Schulze voting system

Part of the Politics series
Voting systems
Politics portal
• v
• t
• e [1]

The **Schulze method** is a voting system developed in 1997 by Markus Schulze that selects a single winner using votes that express preferences. The method can also be used to create a sorted list of winners. The Schulze method is also known as **Schwartz Sequential Dropping (SSD)**, **Cloneproof Schwartz Sequential Dropping (CSSD)**, the **Beatpath Method**, **Beatpath Winner**, **Path Voting**, and **Path Winner**.

The Schulze method is a Condorcet method, which means the following: if there is a candidate who is preferred over every other candidate in pairwise comparisons, then this candidate will be the winner when the Schulze method is applied.

The output of the Schulze method (defined below) gives an ordering of candidates. Therefore, if several positions are available, the method can be used for this purpose without modification, by letting the k top-ranked candidates win the k available seats. Furthermore, for proportional representation elections, a single transferable vote variant has been proposed.

The Schulze method is used by several organizations including Debian, Ubuntu, Gentoo, Software in the Public Interest, Free Software Foundation Europe, Pirate Party associations and many others.

Description of the method

Ballot

The input to the Schulze method is the same as for other ranked single-winner election methods: each voter must furnish an ordered preference list on candidates where ties are allowed (a strict weak order).

One typical way for voters to specify their preferences on a ballot (see right) is as follows. Each ballot lists all the candidates, and each voter ranks this list in order of preference using numbers: the voter places a '1' beside the most preferred candidate(s), a '2' beside the second-most preferred, and so forth. Each voter may optionally:

- give the same preference to more than one candidate. This indicates that this voter is indifferent between these candidates.
- use non-consecutive numbers to express preferences. This has no impact on the result of the elections, since only the order in which the candidates are ranked by the voter matters, and not the absolute numbers of the preferences.
- keep candidates unranked. When a voter doesn't rank all candidates, then this is interpreted as if this voter (i) strictly prefers all ranked to all unranked candidates, and (ii) is indifferent among all unranked candidates.

Rank any number of options in your order of preference.

<input type="checkbox"/>	Joe Smith
1	John Citizen
3	Jane Doe
<input type="checkbox"/>	Fred Rubble
2	Mary Hill

Computation

$d[V, W]$ is assumed to be the number of voters who prefer candidate V to candidate W .

A *path* from candidate X to candidate Y of *strength* p is a sequence of candidates $C(1), \dots, C(n)$ with the following properties:

1. $C(1) = X$ and $C(n) = Y$.
2. For all $i = 1, \dots, (n - 1)$: $d[C(i), C(i + 1)] > d[C(i + 1), C(i)]$.
3. For all $i = 1, \dots, (n - 1)$: $d[C(i), C(i + 1)] = p$.

$p[A, B]$, the *strength of the strongest path* from candidate A to candidate B , is the maximum value such that there is a path from candidate A to candidate B of that strength. If there is no path from candidate A to candidate B at all, then $p[A, B] = 0$.

Candidate D is *better* than candidate E if and only if $p[D, E] > p[E, D]$.

Candidate D is a *potential winner* if and only if $p[D, E] > p[E, D]$ for every other candidate E .

It can be proven that $p[X, Y] > p[Y, X]$ and $p[Y, Z] > p[Z, Y]$ together imply $p[X, Z] > p[Z, X]$.^{§4.1}

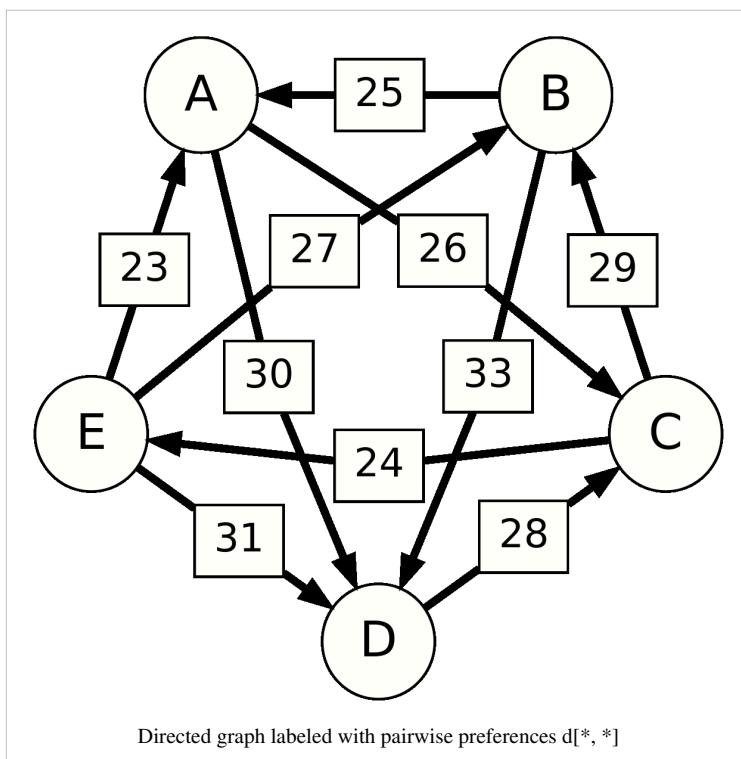
Therefore, it is guaranteed (1) that the above definition of "better" really defines a transitive relation and (2) that there is always at least one candidate D with $p[D, E] > p[E, D]$ for every other candidate E .

Example

In the following example 45 voters rank 5 candidates.

- 5 $ACBED$ (meaning, 5 voters have order of preference: $A > C > B > E > D$)
- 5 $ADECB$
- 8 $BEDAC$
- 3 $CABED$
- 7 $CAEBD$
- 2 $CBADE$
- 7 $DCEBA$
- 8 $EBADC$

The pairwise preferences have to be computed first. For example, when comparing A and B pairwise, there are $5 + 5 + 3 + 7 = 20$ voters who prefer A to B , and $8 + 2 + 7 + 8 = 25$ voters who prefer B to A . So $d[A, B] = 20$ and $d[B, A] = 25$. The full set of pairwise preferences is:



Matrix of pairwise preferences

	$d[*, A]$	$d[*, B]$	$d[*, C]$	$d[*, D]$	$d[*, E]$
$d[A, *]$		20	26	30	22
$d[B, *]$	25		16	33	18
$d[C, *]$	19	29		17	24
$d[D, *]$	15	12	28		14
$d[E, *]$	23	27	21	31	

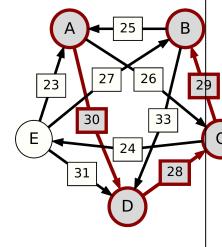
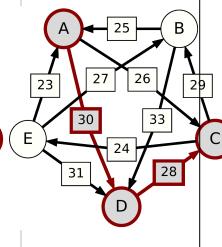
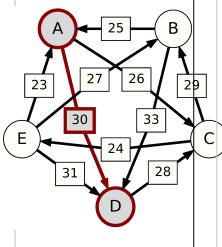
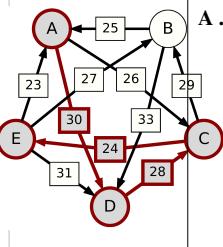
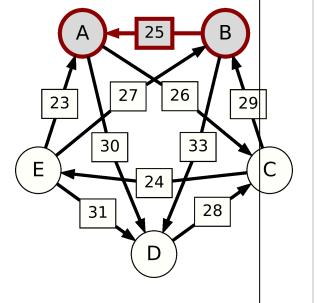
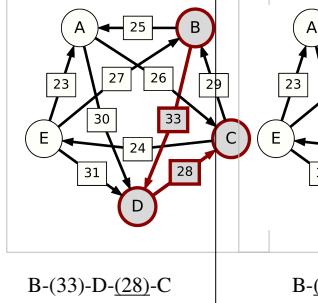
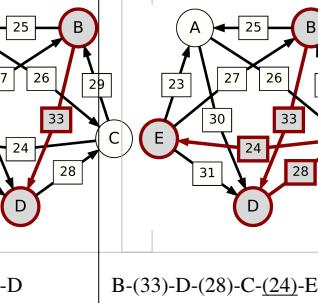
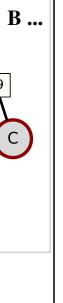
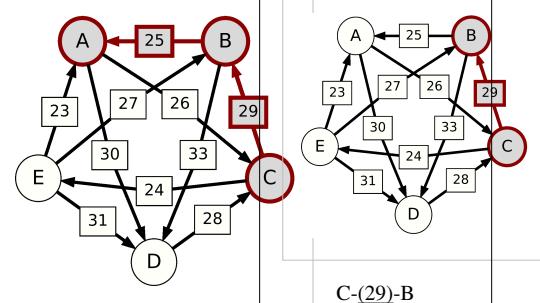
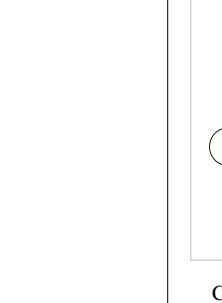
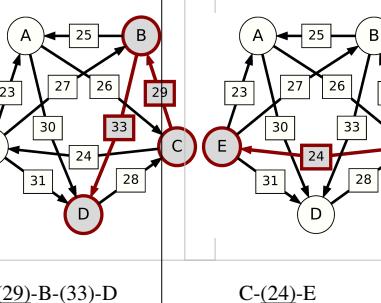
The cells for $d[X, Y]$ have a light green background if $d[X, Y] > d[Y, X]$, otherwise the background is light red. There is no undisputed winner by only looking at the pairwise differences here.

Now the strongest paths have to be identified. To help visualize the strongest paths, the set of pairwise preferences is depicted in the diagram on the right in the form of a directed graph. An arrow from the node representing a candidate X to the one representing a candidate Y is labelled with $d[X, Y]$. To avoid cluttering the diagram, an arrow has only been drawn from X to Y when $d[X, Y] > d[Y, X]$ (i.e. the table cells with light green background), omitting the one in the opposite direction (the table cells with light red background).

One example of computing the strongest path strength is $p[B, D] = 33$: the strongest path from B to D is the direct path (B, D) which has strength 33. But when computing $p[A, C]$, the strongest path from A to C is not the direct path (A, C) of strength 26, rather the strongest path is the indirect path (A, D, C) which has strength $\min(30, 28) = 28$. The *strength* of a path is the strength of its weakest link.

For each pair of candidates X and Y, the following table shows the strongest path from candidate X to candidate Y in red, with the weakest link underlined.

Strongest paths

	... to A	... to B	... to C	... to D	... to E	
from A ...						from A ...
from B ...						from B ...
from C ...						from C ...

C-(29)-B-(25)-A

C-(29)-B-(33)-D

C-(24)-E

C-(29)-B

C-(29)-B-(33)-D

C-(24)-E

B-(25)-A

B-(33)-D

B-(33)-D-(28)-C-(24)-E

B-(33)-D-(28)-C

B-(33)-D

B-(33)-D-(28)-C-(24)-E

C-(29)-B-(25)-A

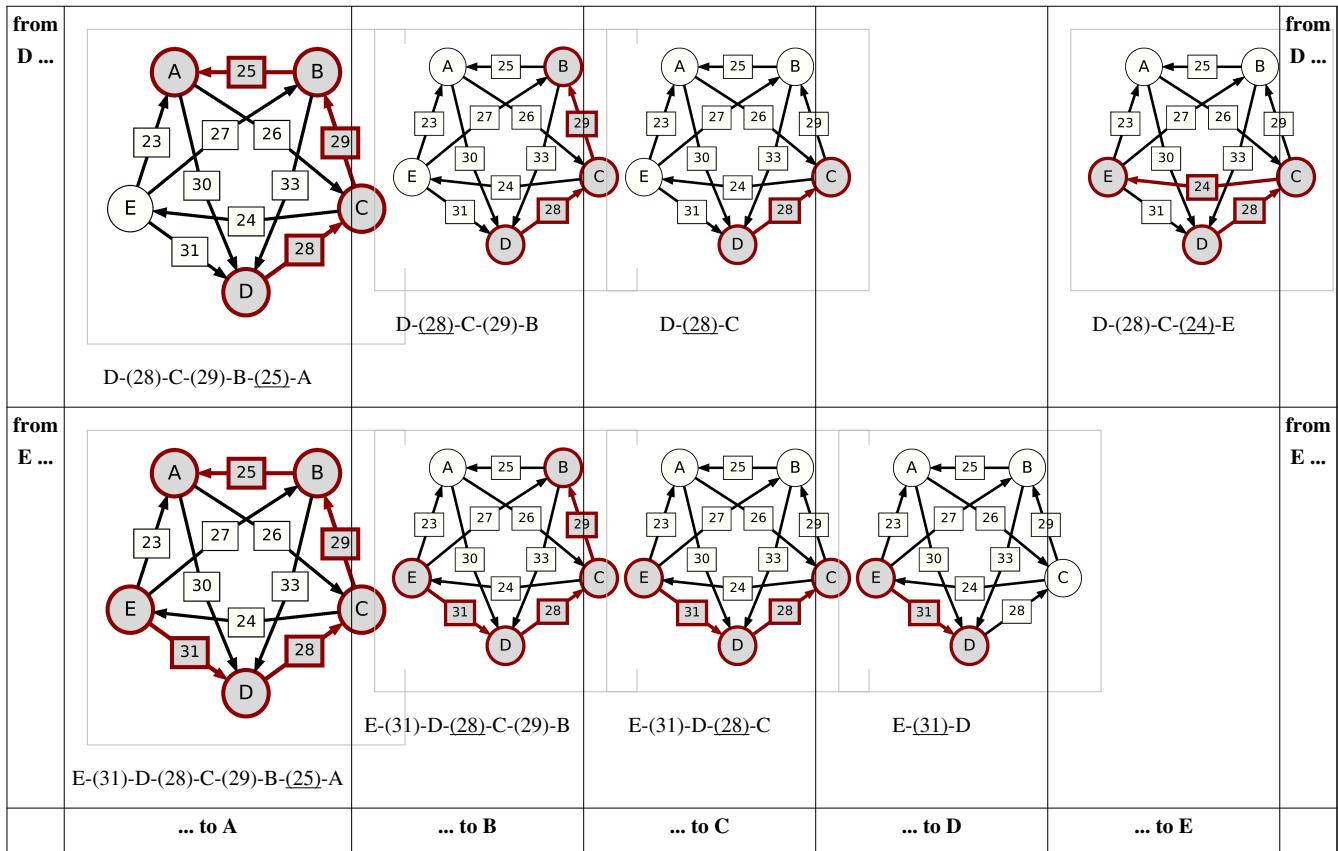
C-(29)-B-(33)-D

C-(24)-E

C-(29)-B

C-(29)-B-(33)-D

C-(24)-E



Strengths of the strongest paths

	$p^{*,A}$	$p^{*,B}$	$p^{*,C}$	$p^{*,D}$	$p^{*,E}$
$p[A,*]$		28	28	30	24
$p[B,*]$	25		28	33	24
$p[C,*]$	25	29		29	24
$p[D,*]$	25	28	28		24
$p[E,*]$	25	28	28	31	

Now the output of the Schulze method can be determined. For example, when comparing A and B, since $28 = p[A,B] > p[B,A] = 25$, for the Schulze method candidate A is *better* than candidate B. Another example is that $31 = p[E,D] > p[D,E] = 24$, so candidate E is *better* than candidate D. Continuing in this way, the result is that the Schulze ranking is $E > A > C > B > D$, and E wins. In other words, E wins since $p[E,X] \geq p[X,E]$ for every other candidate X.

Implementation

The only difficult step in implementing the Schulze method is computing the strongest path strengths. However, this is a well-known problem in graph theory sometimes called the widest path problem. One simple way to compute the strengths therefore is a variant of the Floyd–Warshall algorithm. The following pseudocode illustrates the algorithm.

```
# Input: d[i,j], the number of voters who prefer candidate i to
candidate j.
# Output: p[i,j], the strength of the strongest path from candidate i
to candidate j.
```

```

for i from 1 to C
    for j from 1 to C
        if (i ≠ j) then
            if (d[i,j] > d[j,i]) then
                p[i,j] := d[i,j]
            else
                p[i,j] := 0

for i from 1 to C
    for j from 1 to C
        if (i ≠ j) then
            for k from 1 to C
                if (i ≠ k and j ≠ k) then
                    p[j,k] := max ( p[j,k], min ( p[j,i], p[i,k] ) )

```

This algorithm is efficient, and has running time proportional to C^3 where C is the number of candidates. (This does not account for the running time of computing the $d^{*,*}$ values, which if implemented in the most straightforward way, takes time proportional to C^2 times the number of voters.)

Ties and alternative implementations

When allowing users to have ties in their preferences, the outcome of the Schulze method naturally depends on how these ties are interpreted in defining $d^{*,*}$. Two natural choices are that $d[A, B]$ represents either the number of voters who strictly prefer A to B ($A > B$), or the *margin* of (voters with $A > B$) minus (voters with $B > A$). But no matter how the ds are defined, the Schulze ranking has no cycles, and assuming the ds are unique it has no ties.

Although ties in the Schulze ranking are unlikely,^[2] they are possible. Schulze's original paper proposed breaking ties in accordance with a voter selected at random, and iterating as needed.

An alternative, slower, way to describe the winner of the Schulze method is the following procedure:

1. draw a complete directed graph with all candidates, and all possible edges between candidates
2. iteratively [a] delete all candidates not in the Schwartz set (i.e. any candidate which cannot reach all others) and
[b] delete the weakest link
3. the winner is the last non-deleted candidate.

Satisfied and failed criteria

Satisfied criteria

The Schulze method satisfies the following criteria:

- Unrestricted domain
- Non-imposition (a.k.a. citizen sovereignty)
- Non-dictatorship
- Pareto criterion^{:§4.3}
- Monotonicity criterion^{:§4.5}
- Majority criterion
- Majority loser criterion
- Condorcet criterion
- Condorcet loser criterion

- Schwartz criterion
- Smith criterion^{:§4.7}
- Independence of Smith-dominated alternatives^{:§4.7}
- Mutual majority criterion
- Independence of clones^{:§4.6}
- Reversal symmetry^{:§4.4}
- Mono-append^[3]
- Mono-add-plump
- Resolvability criterion^{:§4.2}
- Polynomial runtime^{:§2.3"}
- prudence^{:§4.9"}
- MinMax sets^{:§4.8"}
- Woodall's plurality criterion if winning votes are used for $d[X, Y]$
- Symmetric-completion if margins are used for $d[X, Y]$

Failed criteria

Since the Schulze method satisfies the Condorcet criterion, it automatically fails the following criteria:

- Participation^{:§3.4}
- Consistency
- Invulnerability to compromising
- Invulnerability to burying
- Later-no-harm

Likewise, since the Schulze method is not a dictatorship and agrees with unanimous votes, Arrow's Theorem implies it fails the criterion

- Independence of irrelevant alternatives

The Schulze method also fails

- Peyton Young's criterion Local Independence of Irrelevant Alternatives.

Comparison table

The following table compares the Schulze method with other preferential single-winner election methods:

	Monotonic	Condorcet	Majority	Condorcet loser	Majority loser	Mutual majority	Smith	ISDA	LIIA	Clone independence	Reversal symmetry	Polynomial time	Participation, Consistency	Resolvability
Schulze	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes
Ranked Pairs	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Copeland	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No
Kemeny-Young	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	Yes
Nanson	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	No	Yes
Baldwin	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	Yes	No	Yes
Instant-runoff voting	No	No	Yes	Yes	Yes	Yes	No	No	No	Yes	No	Yes	No	Yes
Borda	Yes	No	No	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	Yes
Bucklin	Yes	No	Yes	No	Yes	Yes	No	No	No	No	No	Yes	No	Yes
Coombs	No	No	Yes	Yes	Yes	Yes	No	No	No	No	No	Yes	No	Yes

Minimax	Yes	Yes	Yes	No	No	No	No	No	No	No	No	Yes	No	Yes
Plurality	Yes	No	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes
Anti-plurality	Yes	No	No	No	Yes	No	No	No	No	No	No	Yes	Yes	Yes
Contingent voting	No	No	Yes	Yes	Yes	No	No	No	No	No	No	Yes	No	Yes
Sri Lankan contingent voting	No	No	Yes	No	No	No	No	No	No	No	No	Yes	No	Yes
Supplementary voting	No	No	Yes	No	No	No	No	No	No	No	No	Yes	No	Yes
Dodgson	No	Yes	Yes	No	No	No	No	No	No	No	No	No	No	Yes

The main difference between the Schulze method and the ranked pairs method can be seen in this example:

Suppose the MinMax score of a set X of candidates is the strength of the strongest pairwise win of a candidate $A \notin X$ against a candidate $B \in X$. Then the Schulze method, but not Ranked Pairs, guarantees that the winner is always a candidate of the set with minimum MinMax score.^[4,8] So, in some sense, the Schulze method minimizes the largest majority that has to be reversed when determining the winner.

On the other hand, Ranked Pairs minimizes the largest majority that has to be reversed to determine the order of finish, in the minlexmax sense.^[4] In other words, when Ranked Pairs and the Schulze method produce different orders of finish, for the majorities on which the two orders of finish disagree, the Schulze order reverses a larger majority than the Ranked Pairs order.

History

The Schulze method was developed by Markus Schulze in 1997. It was first discussed in public mailing lists in 1997–1998^[5] and in 2000.^[6] Subsequently, Schulze method users included Software in the Public Interest (2003),^[7] Debian (2003),^[8] Gentoo (2005),^[9] TopCoder (2005),^[10] Wikimedia (2008),^[11] KDE (2008),^[12] the Free Software Foundation Europe (2008),^[13] the Pirate Party of Sweden (2009),^[14] and the Pirate Party of Germany (2010).^[15] In the French Wikipedia, the Schulze method was one of two multi-candidate methods approved by a majority in 2005,^[16] and it has been used several times.^[17]

In 2011, Schulze published the method in the academic journal *Social Choice and Welfare*.^[1]

Users

The Schulze method is not currently used in parliamentary elections. However, it has been used for parliamentary primaries in the Swedish Pirate Party. It is also starting to receive support in other public organizations. Organizations which currently use the Schulze method are:

- Alternative for Germany^[18]
- Annodex Association^[19]
- Associated Student Government at Northwestern University^[20]
- Associated Student Government at University of Freiburg^[21]
- Berufsverband der Kinder- und Jugendärzte (BVKJ)^[22]
- Blitzed^[23]
- BoardGameGeek^[24]
- Cassandra^[25]
- Codex Alpe Adria
- Collective Agency^[26]
- College of Marine Science
- Computer Science Departmental Society for York University (HackSoc)^[27]
- County Highpointers^[28]
- Debian
- Demokratische Bildung Berlin^[29]
- EuroBillTracker^[30]
- European Democratic Education Conference (EUDEC)
- Fair Trade Northwest^[31]
- FFmpeg^[32]
- Five Star Movement of Campobasso,^[33] Monte Compatri,^[34] Montemurlo,^[35] and Pescara^[36]
- Flemish Student Society of Leuven^[37]
- Free Geek^[38]
- Free Hardware Foundation of Italy^[39]
- Free Software Foundation Europe (FSFE)
- Gentoo Foundation
- GNU Privacy Guard (GnuPG)^[40]
- Gothenburg Hacker Space (GHS)^[41]
- Graduate Student Organization at the State University of New York: Computer Science (GSOCS)
- Haskell^[42]
- Hillegass Parker House
- Ithaca Generator^{[43][44]}
- Kanawha Valley Scrabble Club^[45]
- KDE e.V.
- Kingman Hall^[46]
- Knight Foundation^[47]
- Kubuntu^[48]
- Kumoricon^[49]
- League of Professional System Administrators (LOPSA)^[50]
- Libre-Entreprise^{[51][52]}

For more information, see:

- [Board elections 2008](#) ↗
- [Candidates](#) ↗
- [Schulze method](#) ↗

2	John Doe
3	Joe Bloggs
1	Jane Doe
	John Smith
2	A. N. Other

OK

sample ballot for Wikimedia's Board of Trustees elections

- LiquidFeedback
- Lumiera/Cinelerra^[53]
- Madisonium^{[54][55]}
- Mathematical Knowledge Management Interest Group (MKM-IG)^[56]
- Metalab
- Music Television (MTV)^[57]
- Neo^[58]
- Noisebridge^[59]
- North Shore Cyclists (NSC)^[60]
- OpenEmbedded^[61]
- OpenStack^[62]
- Park Alumni Society (PAS)^[63]
- Pirate Party Australia^[64]
- Pirate Party of Austria^[65]
- Pirate Party of Belgium^[66]
- Pirate Party of Brazil
- Pirate Party of France^[67]
- Pirate Party of Germany
- Pirate Party of Iceland^[68]
- Pirate Party of Italy^[69]
- Pirate Party of Mexico^[70]
- Pirate Party of the Netherlands^[71]
- Pirate Party of New Zealand^[72]
- Pirate Party of Sweden
- Pirate Party of Switzerland^[73]
- Pirate Party of the United States^[74]
- Pittsburgh Ultimate^[75]
- RLLMUK^[76]
- RPMrepo^[77]
- Sender Policy Framework (SPF)^[78]
- Software in the Public Interest (SPI)
- Squeak^[79]
- Students for Free Culture^[80]
- Sugar Labs^[81]
- SustainableUnion^[82]
- Sverok^[83]
- Technology House^{[84][85]}
- TestPAC^[86]
- TopCoder
- Ubuntu^[87]
- University of British Columbia Math Club^[88]
- Vidya Gaem Awards^[89]
- Wikipedia in French, Hebrew,^[90] Hungarian,^[91] and Russian.^[92]

Notes

- [1] http://en.wikipedia.org/w/index.php?title=Template:Electoral_systems&action=edit
- [2] Under reasonable probabilistic assumptions when the number of voters is much larger than the number of candidates
- [3] Douglas R. Woodall, Properties of Preferential Election Rules (<http://www.votingmatters.org.uk/ISSUE3/P5.HTM>), *Voting Matters*, issue 3, pages 8-15, December 1994
- [4] Tideman, T. Nicolaus, "Independence of clones as a criterion for voting rules," *Social Choice and Welfare* vol 4 #3 (1987), pp 185-206.
- [5] See:
 - Markus Schulze, Condorect sub-cycle rule (<http://lists.electorama.com/pipermail/election-methods-electorama.com/1997-October/001570.html>), October 1997 (In this message, the Schulze method is mistakenly believed to be identical to the ranked pairs method.)
 - Mike Ossipoff, Party List P.S. (<http://groups.yahoo.com/group/election-methods-list/message/467>), July 1998
 - Markus Schulze, Tiebreakers, Subcycle Rules (<http://groups.yahoo.com/group/election-methods-list/message/673>), August 1998
 - Markus Schulze, Maybe Schulze is decisive (<http://groups.yahoo.com/group/election-methods-list/message/845>), August 1998
 - Norman Petry, Schulze Method - Simpler Definition (<http://groups.yahoo.com/group/election-methods-list/message/867>), September 1998
 - Markus Schulze, Schulze Method (<http://groups.yahoo.com/group/election-methods-list/message/2291>), November 1998
- [6] See:
 - Anthony Towns, Disambiguation of 4.1.5 (<http://lists.debian.org/debian-vote/2000/11/msg00121.html>), November 2000
 - Norman Petry, Constitutional voting, definition of cumulative preference (<http://lists.debian.org/debian-vote/2000/12/msg00045.html>), December 2000
- [7] Process for adding new board members (<http://www.spi-inc.org/corporate/resolutions/2003/2003-01-06.wta.1/>), January 2003
- [8] See:
 - Constitutional Amendment: Condorcet/Clone Proof SSD Voting Method (http://www.debian.org/vote/2003/vote_0002), June 2003
 - Constitution for the Debian Project (<http://www.debian.org-devel/constitution>), appendix A6
 - Debian Voting Information (<http://www.debian.org/vote/>)
- [9] See:
 - Gentoo Foundation Charter (<http://www.gentoo.org/foundation/en/>)
 - Aron Griffis, 2005 Gentoo Trustees Election Results (<http://article.gmane.org/gmane.linux.gentoo.nfp/252/match=Condorcet+Schwartz+drop+dropped>), May 2005
 - Lars Weiler, Gentoo Weekly Newsletter 23 May 2005 (<http://article.gmane.org/gmane.linux.gentoo.weekly-news/121/match=Condorcet>)
 - Daniel Drake, Gentoo metastructure reform poll is open (<http://article.gmane.org/gmane.linux.gentoo.devel/28603/match=Condorcet+Cloneproof+Schwartz+Sequential+Dropping>), June 2005
 - Grant Goodyear, Results now more official (<http://article.gmane.org/gmane.linux.gentoo.devel/42260/match=Schulze+method>), September 2006
 - 2007 Gentoo Council Election Results (<http://dev.gentoo.org/~fox2mike/elections/council/2007/council2007-results>), September 2007
 - 2008 Gentoo Council Election Results (<http://www.gentoo.org/proj/en/elections/council/2008/council-2008-results.txt>), June 2008
 - 2008 Gentoo Council Election Results (<http://www.gentoo.org/proj/en/elections/council/2008/council-200811-results.txt>), November 2008
 - 2009 Gentoo Council Election Results (<http://www.gentoo.org/proj/en/elections/council/2009/council-200906-results.txt>), June 2009
 - 2009 Gentoo Council Election Results (<http://www.gentoo.org/proj/en/elections/council/2009/council-200912-results.txt>), December 2009
 - 2010 Gentoo Council Election Results (<http://www.gentoo.org/proj/en/elections/council/2010/council-201006-results.txt>), June 2010
- [10] See:
 - 2006 TopCoder Open Logo Design Contest (http://www.topcoder.com/tc?module=Static&d1=tournaments&d2=tco06&d3=logo_rules), November 2005
 - 2006 TopCoder Collegiate Challenge Logo Design Contest (http://www.topcoder.com/tc?module=Static&d1=tournaments&d2=tccc06&d3=logo_rules), June 2006
 - 2007 TopCoder High School Tournament Logo (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2030>), September 2006
 - 2007 TopCoder Arena Skin Contest (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2046>), November 2006
 - 2007 TopCoder Open Logo Contest (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2047>), January 2007
 - 2007 TopCoder Open Web Design Contest (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2050>), January 2007
 - 2007 TopCoder Collegiate Challenge T-Shirt Design Contest (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2122>), September 2007
 - 2008 TopCoder Open Logo Design Contest (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2127>), September 2007

- 2008 TopCoder Open Web Site Design Contest (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2133>), October 2007
- 2008 TopCoder Open T-Shirt Design Contest (<http://studio.topcoder.com/?module=ViewContestDetails&ct=2183>), March 2008

[11] See:

- Jesse Plamondon-Willard, Board election to use preference voting, May 2008
- Mark Ryan, 2008 Wikimedia Board Election results, June 2008
- 2008 Board Elections, June 2008
- 2009 Board Elections, August 2009

[12] section 3.4.1 of the Rules of Procedures for Online Voting (http://ev.kde.org/rules/online_voting.php)

[13] See:

- article 6 section 3 of the constitution (<http://www.fsfeurope.org/about/legal/Constitution.en.pdf>)
- Fellowship vote for General Assembly seats (<http://www.fsfeurope.org/news/2009/news-20090301-01.en.html>), March 2009
- And the winner of the election for FSFE's Fellowship GA seat is ... (<http://fsfe.org/news/2009/news-20090601-01.en.html>), June 2009

[14] See:

- Inför primärvalen (<http://forum.piratpartiet.se/FindPost174988.aspx>), October 2009
- Dags att kandidera till riksdagen (<http://forum.piratpartiet.se/FindPost176567.aspx>), October 2009
- Råresultat primärvalet (<http://forum.piratpartiet.se/FindPost193877.aspx>), January 2010

[15] 11 of the 16 regional sections and the federal section of the Pirate Party of Germany are using LiquidFeedback (<http://liquidfeedback.org/>) for unbinding internal opinion polls. In 2010/2011, the Pirate Parties of Neukölln (link (<http://wiki.piratenpartei.de/BE:Neuk%F6lln/Gebietsversammlungen/2010.3/Protokoll>)), Mitte (link (<http://berlin.piratenpartei.de/2011/01/18/kandidaten-der-piraten-in-mitte-aufgestellt/>)), Steglitz-Zehlendorf (link (http://wiki.piratenpartei.de/wiki/images/d/da/BE_Gebietsversammlung_Steglitz_Zehlendorf_2011_01_20_Protokoll.pdf)), Lichtenberg (link (<http://piraten-lichtenberg.de/?p=205>)), and Tempelhof-Schöneberg (link (http://wiki.piratenpartei.de/BE:Gebietsversammlungen/Tempelhof-Schoeneberg/Protokoll_2011.1)) adopted the Schulze method for its primaries. Furthermore, the Pirate Party of Berlin (in 2011) (link (<http://wiki.piratenpartei.de/BE:Parteitag/2011.1/Protokoll>)) and the Pirate Party of Regensburg (in 2012) (link (http://wiki.piratenpartei.de/BY:Regensburg/Gr%C3%A4ndung/Gesch%C3%A4ftsordnung#Anlage_A)) adopted this method for their primaries.

[16] Choix dans les votes

[17] fr:Spécial:Pages liées/Méthode Schulze

[18] §12(4), §12(15), and §14(3) of the bylaws (<https://www.alternativefuer.de/pdf/Bundessatzung-Parteibeschluss.pdf>), April 2013

[19] Election of the Annodex Association committee for 2007 (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_50fcf592ae8f13d9), February 2007

[20] Ajith, Van Atta win ASG election (<http://www.northbynorthwestern.com/story/ajith-van-atta-win-asg-election/>), April 2013

[21] §6 and §7 of its bylaws (<http://www.u-asta.uni-freiburg.de/vs/stura/stura-go-entgeltig-beschlossene-form-13.05.2014.pdf>), May 2014

[22] §9a of the bylaws (http://www.kinderaerzte-im-netz.de/bvkj/contentkin/psfile/pdf/56/BVKJ_Satzu4e2d51acd5583.pdf), October 2013

[23] Condorcet method for admin voting (http://wiki.blitzed.org/Condorcet_method_for_admin_voting), January 2005

[24] See:

- Important notice for Golden Geek voters (<http://www.boardgamegeek.com/article/1751580>), September 2007
- Golden Geek Awards 2008 - Nominations Open (<http://www.boardgamegeek.com/article/2582330>), August 2008
- Golden Geek Awards 2009 - Nominations Open (<http://www.boardgamegeek.com/article/3840078>), August 2009
- Golden Geek Awards 2010 - Nominations Open (<http://www.boardgamegeek.com/article/5492260>), September 2010
- Golden Geek Awards 2011 - Nominations Open (<http://boardgamegeek.com/thread/694044>), September 2011

[25] Project Logo (<http://article.gmane.org/gmane.comp.db.cassandra.devel/424/match=condorcet+schwartz+sequential+dropping+beatpath>), October 2009

[26] Civics Meeting Minutes (<http://collectiveagency.co/2012/03/21/civics-meeting-minutes-32012/>), March 2012

[27] Report on HackSoc Elections (<http://www.hacksoc.org/HackSocElections.pdf>), December 2008

[28] Adam Helman, Family Affair Voting Scheme - Schulze Method (http://www.cohp.org/records/votes/family_affair_voting_scheme.html)

[29] appendix 1 of the constitution (http://www.demokratische-schule-x.de/media/DBB_Satzung_Stand_August_2010.pdf)

[30] See:

- Candidate cities for EBTM05 (<http://forum.eurobilltracker.eu/viewtopic.php?t=4920&highlight=condorcet+beatpath+ssd>), December 2004
- Meeting location preferences (<http://forum.eurobilltracker.eu/viewtopic.php?t=4921&highlight=condorcet>), December 2004
- Date for EBTM07 Berlin (<http://forum.eurobilltracker.eu/viewtopic.php?t=9353&highlight=condorcet+beatpath>), January 2007
- Vote the date of the Summer EBTM08 in Ljubljana (<http://forum.eurobilltracker.eu/viewtopic.php?t=10564&highlight=condorcet+beatpath>), January 2008
- New Logo for EBT (<http://forum.eurobilltracker.com/viewtopic.php?f=26&t=17919&start=15#p714947>), August 2009

[31] article XI section 2 of the bylaws (http://fairtradenorthwest.org/FTNW_Bylaws.pdf)

[32] Democratic election of the server admins (<http://article.gmane.org/gmane.comp.video.ffmpeg.devel/113026/match=%22schulze+method%22+%22Cloneproof+schwartz+sequential+dropping%22+Condorcet>), July 2010

- [33] Campobasso. Comunali, scattano le primarie a 5 Stelle (<http://www.quotidianomolise.com/campobasso-comunali-scattano-le-primarie-a-5-stelle/>), February 2014
- [34] article 25(5) of the bylaws (<http://www.5stellemontecompatri.it/wp-content/uploads/2013/11/REGOLAMENTO-M5S-Montecompatri1.doc>), October 2013
- [35] 2° Step Comunarie di Montemurlo (<http://www.montemurlo5stelle.net/2013/11/2-step-comunarie-di-montemurlo.html>), November 2013
- [36] article 12 of the bylaws (http://movimento5stellepescara.it/wp-content/uploads/2014/02/Regolamento-m5S-pescara_rev.01_11_02_-2014.pdf), February 2014
- [37] article 51 of the statutory rules (<http://www.vtk.be/page/file/ef87370c1d5798758d1730a14f7410d96783301e/>)
- [38] Voters Guide (http://wiki.freegeek.org/images/7/7a/Voters_guide.pdf), September 2011
- [39] See:
- Eletto il nuovo Consiglio nella Free Hardware Foundation (<http://fhf.it/notizie/nuovo-consiglio-nella-fhf>), June 2008
 - Poll Results (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_5b6e434828ec547b), June 2008
- [40] GnuPG Logo Vote (<http://logo-contest.gnupg.org/results.html>), November 2006
- [41] §14 of the bylaws (<http://gbg.hackerspace.se/site/om-ghs/stadgar/>)
- [42] Haskell Logo Competition (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?num_winners=1&id=E_d21b0256a4fd5ed7&algorithm=beatpath), March 2009
- [43] <http://ithacagenerator.ohhttp://hip.wikispot.org/Bylawsrg/>
- [44] article VI section 10 of the bylaws (<http://ithacagenerator.org/alternate-membership-terms/bylaws/>), November 2012
- [45] A club by any other name ... (<http://wvscrabble.blogspot.com/2009/04/club-by-any-other-name.html>), April 2009
- [46] See:
- Ka-Ping Yee, Condorcet elections (<http://www.livejournal.com/users/zestyping/102718.html>), March 2005
 - Ka-Ping Yee, Kingman adopts Condorcet voting (<http://www.livejournal.com/users/zestyping/111588.html>), April 2005
- [47] Knight Foundation awards \$5000 to best created-on-the-spot projects (<http://civic.mit.edu/blog/andrew/knight-foundation-awards-5000-to-best-created-on-the-spot-projects>), June 2009
- [48] Kubuntu Council 2013 (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_31619806caaf95b5), May 2013
- [49] See:
- Mascot 2007 contest (<http://www.kumoricon.org/forums/index.php?topic=2599.45>), July 2006
 - Mascot 2008 and cover 2007 contests (<http://www.kumoricon.org/forums/index.php?topic=4497.0>), May 2007
 - Mascot 2009 and program cover 2008 contests (<http://www.kumoricon.org/forums/index.php?topic=6653.0>), April 2008
 - Mascot 2010 and program cover 2009 contests (<http://www.kumoricon.org/forums/index.php?topic=10048.0>), May 2009
 - Mascot 2011 and book cover 2010 contests (<http://www.kumoricon.org/forums/index.php?topic=12955.0>), May 2010
 - Mascot 2012 and book cover 2011 contests (<http://www.kumoricon.org/forums/index.php?topic=15340.0>), May 2011
- [50] article 8.3 of the bylaws (http://governance.lopsa.org/LOPSA_Bylaws)
- [51] <http://www.libre-entreprise.org/>
- [52] See:
- Choix de date pour la réunion Libre-entreprise durant le Salon Solution Linux 2006 (<http://www.libre-entreprise.org/index.php/Election:DateReunionSolutionLinux2006>), January 2006
 - Entrée de Libricks dans le réseau Libre-entreprise (<http://www.libre-entreprise.org/index.php/Election:EntreeLibricks>), February 2008
- [53] Lumiera Logo Contest (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_7df51370797b45d6), January 2009
- [54] <http://madisonium.org/>
- [55] bylaws (https://docs.google.com/document/d/1UhAGouwOEbzM6Uv3fQd78e5P1VfuYldtjEK7XO_B3Y/edit)
- [56] The MKM-IG uses Condorcet with dual dropping (<http://condorcet-dd.sourceforge.net/>). That means: The Schulze ranking and the ranked pairs ranking are calculated and the winner is the top-ranked candidate of that of these two rankings that has the better Kemeny score.
- See:
- MKM-IG Charter
 - Michael Kohlhase, MKM-IG Trustees Election Details & Ballot (<http://lists.jacobs-university.de/pipermail/projects-mkm-ig/2004-November/000041.html>), November 2004
 - Andrew A. Adams, MKM-IG Trustees Election 2005 (<http://lists.jacobs-university.de/pipermail/projects-mkm-ig/2005-December/000072.html>), December 2005
 - Lionel Elie Mamane, Elections 2007: Ballot (<http://lists.jacobs-university.de/pipermail/projects-mkm-ig/2007-August/000406.html>), August 2007
- [57] Benjamin Mako Hill, Voting Machinery for the Masses (<http://www.oscon.com/oscon2008/public/schedule/detail/3230>), July 2008
- [58] See:
- Wahlen zum Neo-2-Freeze: Formalitäten (<http://wiki.neo-layout.org/wiki/Neo-2-Freeze/Wahl?version=10#a7.Wahlverfahren>), February 2010
 - Hinweise zur Stimmabgabe (<http://wiki.neo-layout.org/wiki/Neo-2-Freeze/Wahl/Stimmabgabe?version=11>), March 2010
 - Ergebnisse (<http://wiki.neo-layout.org/wiki/Neo-2-Freeze/Wahl/Ergebnisse?version=9>), March 2010

- [59] 2009 Director Elections (https://www.noisebridge.net/index.php?title=2009_Director_Elections&oldid=8951)

[60] NSC Jersey election (<http://www.nscyc.org/JerseyWinner>), NSC Jersey vote (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_6c53f2bddb068673), September 2007

[61] Online Voting Policy (http://www.openembedded.org/index.php/Online_Voting_Policy)

[62] See:

 - 2010 OpenStack Community Election (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_f35052f9f6d58f36&rkey=4603fbf32e182e6c), November 2010
 - OpenStack Governance Elections Spring 2012 (<http://www.openstack.org/blog/2012/02/openstack-governance-elections-spring-2012/>), February 2012

[63] <http://www.parkscholars.org/index.php>

[64] National Congress 2011 Results (<http://pirateparty.org.au/2011/11/18/national-congress-2011-results/>), November 2011

[65] §6(10) of the bylaws (<http://wiki.piratenpartei.at/wiki/Satzung>)

[66] The Belgian Pirate Party Announces Top Candidates for the European Elections (<http://piratetimes.net/the-belgian-pirate-party-announces-top-candidates-for-the-european-elections>), January 2014

[67] §11.2.E of the statutory rules (<http://partipirate.org/ri.pdf>)

[68] article 7.5 of the bylaws (<http://www.piratar.is/um-pirata/log-felagsins/>)

[69] Rules adopted on 18 December 2011 (<http://www.partito-pirata.it/statute/>)

[70] Vote Result for Name Definition (<http://wikipartido.mx/Timon-war/faces/votacionydebate/votacion/votResultados.xhtml?vid=1>)

[71] Help mee met het nieuwe Piratenpartij-logo! (<https://www.piratenpartij.nl/blog/argure/help-mee-met-het-nieuwe-piratenpartij-logo>), August 2013

[72] 23 January 2011 meeting minutes (http://pirateparty.org.nz/wiki/23_January_2011_meeting_minutes)

[73] Piratenversammlung der Piratenpartei Schweiz (<http://blog.florian-pankerl.de/?p=444>), September 2010

[74] Article IV Section 4 of the constitution ([http://www.pirate-party.us/wiki/Pirate_National_Committee_\(PNC\)/Constitution](http://www.pirate-party.us/wiki/Pirate_National_Committee_(PNC)/Constitution))

[75] 2006 Community for Pittsburgh Ultimate Board Election (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_89773564141f0859), September 2006

[76] Committee Elections (<http://www.rllmukforum.com/index.php?topic/260622-committee-elections-2012/>), April 2012

[77] LogoVoting (<http://rpmrepo.org/driesverachtert/LogoVoting>), December 2007

[78] See:

 - SPF Council Election Procedures (http://www.openspf.org/Council_Election)
 - 2006 SPF Council Election (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_1fd503d126aaa609), January 2006
 - 2007 SPF Council Election (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_8e5a1ca7f86a5d5d), January 2007

[79] Squeak Oversight Board Election 2010 (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?num_winners=7&id=E_716d8c257e6cf36b&algorithm=beatpath), March 2010

[80] See:

 - Bylaws of the Students for Free Culture (<http://wiki.freeculture.org/Bylaws>), article V, section 1.1.1
 - Free Culture Student Board Elected Using Selectricty (<http://blog.selectricty.org/?p=4>), February 2008

[81] Election status update (<http://lists.sugarlabs.org/archive/iaep/2009-September/008620.html>), September 2009

[82] §10 III of its bylaws (http://sustainableunion.yolasite.com/resources/130614_Satzung_sud_fBayern.pdf), June 2013

[83] Minutes of the 2010 Annual Sverok Meeting (<http://www.sverok.se/wp-content/uploads/2010/11/Protokoll-Riksmötte-20102.pdf>), November 2010

[84] <http://techhouse.brown.edu/>

[85] constitution (<https://mygroups.brown.edu/organization/technologyhouse/DocumentLibrary/View/29406>), December 2010

[86] article VI section 6 of the bylaws (<http://testpacpleaseignore.org/official-test-pac-bylaws/>)

[87] Ubuntu IRC Council Position (<https://lists.ubuntu.com/archives/ubuntu-irc/2012-May/001538.html>), May 2012

[88] See this mail (<http://twitter.com/alex/status/11353642881>).

[89] Pairwise Voting Results (<http://2012.vidyagaemawards.com/voting/results/pairwise>)

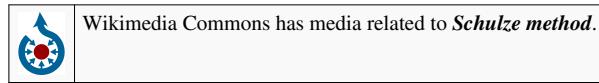
[90] See e.g. here ([http://he.wikipedia.org/w/index.php?title=ברלמן-ויקיפדיה:סדרה-לנRpfiyka&oldid=7014412](http://he.wikipedia.org/w/index.php?title=ברלמן-ויקיפדיה:סדרה-לנՐפִיָּקָה&oldid=7014412)) (May 2009), here (<http://he.wikipedia.org/w/index.php?title=ברלמן-ויקיפדיה:סדרה-לנRpfiyka&oldid=7388447>) (August 2009), and here (<http://he.wikipedia.org/w/index.php?title=ברלמן-ויקיפדיה:סדרה-לנRpfiyka&oldid=8057743>) (December 2009).

[91] See here and here.

[92] See:

 - Result of 2007 Arbitration Committee Elections
 - Result of 2008 Arbitration Committee Elections
 - Result of 2009 Arbitration Committee Elections
 - Result of 2010 Arbitration Committee Elections

External links

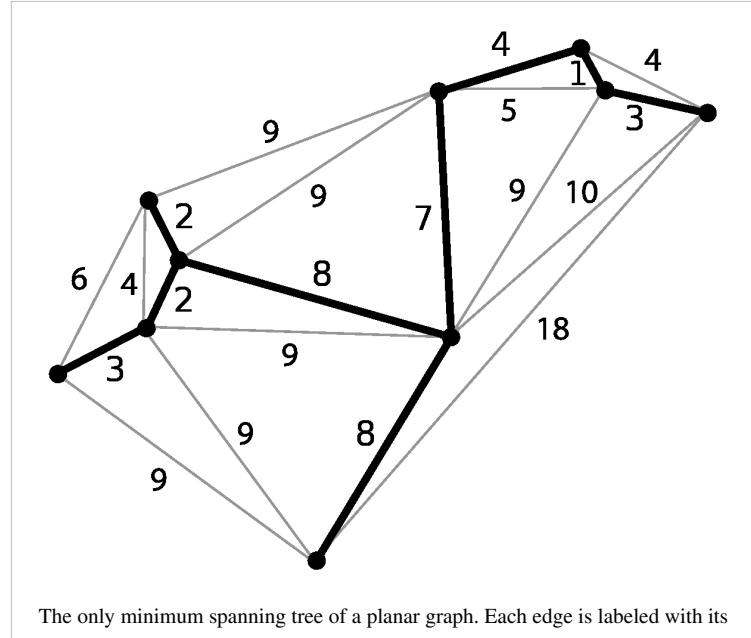


- Official website (<http://m-schulze.9mail.de/>)
- Condorcet Computations (http://www.dsi.unifi.it/~PP2009/talks/Talks_giovedi/Talks_giovedi/grabmeier.pdf) by Johannes Grabmeier
- Spieltheorie (<http://www.informatik.uni-freiburg.de/~ki/teaching/ss09/gametheory/spieltheorie.pdf>) (German) by Bernhard Nebel
- Accurate Democracy (http://accuratedemocracy.com/voting_rules.htm) by Rob Loring
- Christoph Börgers (2009), *Mathematics of Social Choice: Voting, Compensation, and Division* (<http://books.google.com/books?id=dccBaphP1G4C&pg=PA37>), SIAM, ISBN 0-89871-695-0
- Nicolaus Tideman (2006), *Collective Decisions and Voting: The Potential for Public Choice* (http://books.google.com/books?id=RN5q_LuByUoC&pg=PA228), Burlington: Ashgate, ISBN 0-7546-4717-X
- preftools (<http://www.public-software-group.org/preftools>) by the Public Software Group
- Condorcet Class (https://github.com/julien-boudry/Condorcet_Schulze-PHP_Class) PHP library supporting multiple Condorcet methods, including that of Schulze.
- Arizonans for Condorcet Ranked Voting (<http://www.azsos.gov/election/2008/general/ballotmeasuretext/I-21-2008.pdf>)

Minimum spanning trees

Minimum spanning tree

Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree (MST)** or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.



The only minimum spanning tree of a planar graph. Each edge is labeled with its weight, which here is roughly proportional to its length.

One example would be a telecommunications company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths (eg. along roads), then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight — there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects to every house; there might be several spanning trees possible. A *minimum spanning tree* would be one with the lowest total cost, thus would represent the least expensive path for laying the cable.

Properties

Possible multiplicity

There may be several minimum spanning trees of the same weight having a minimum number of edges; in particular, if all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum. If there are n vertices in the graph, then each tree has $n-1$ edges.

Uniqueness

If each edge has a distinct weight then there will be only one, unique minimum spanning tree. This is true in many realistic situations, such as the telecommunications company example above, where it's unlikely any two paths have *exactly* the same cost. This generalizes to spanning forests as well. If the edge weights are not unique, only the (multi-)set of weights in minimum spanning trees is unique, that is the same for all minimum spanning trees.^[1]

A proof of uniqueness by contradiction is as follows.

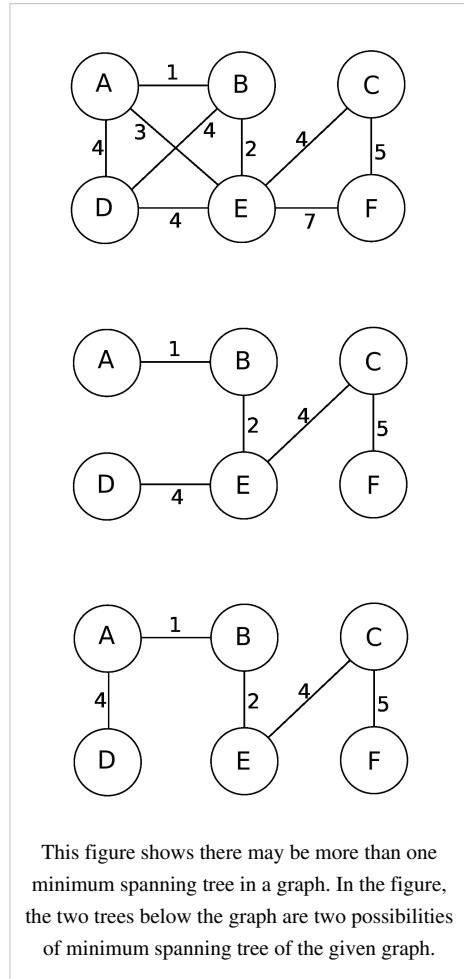
1. Suppose there are two different MSTs A and B .
2. Let e_1 be the edge of least weight that is in one of the MSTs and not the other. Without loss of generality, assume e_1 is in A but not in B .
3. As B is a MST, $\{e_1\} \cup B$ must contain a cycle C .
4. Then C has an edge e_2 whose weight is greater than the weight of e_1 , since all edges in B with less weight are in A by the choice of e_1 , and C must have at least one edge that is not in A because otherwise A would contain a cycle in contradiction with its being an MST.
5. Replacing e_2 with e_1 in B yields a spanning tree with a smaller weight.
6. This contradicts the assumption that B is a MST.

Minimum-cost subgraph

If the weights are *positive*, then a minimum spanning tree is in fact a minimum-cost subgraph connecting all vertices, since subgraphs containing cycles necessarily have more total weight.

Cycle property

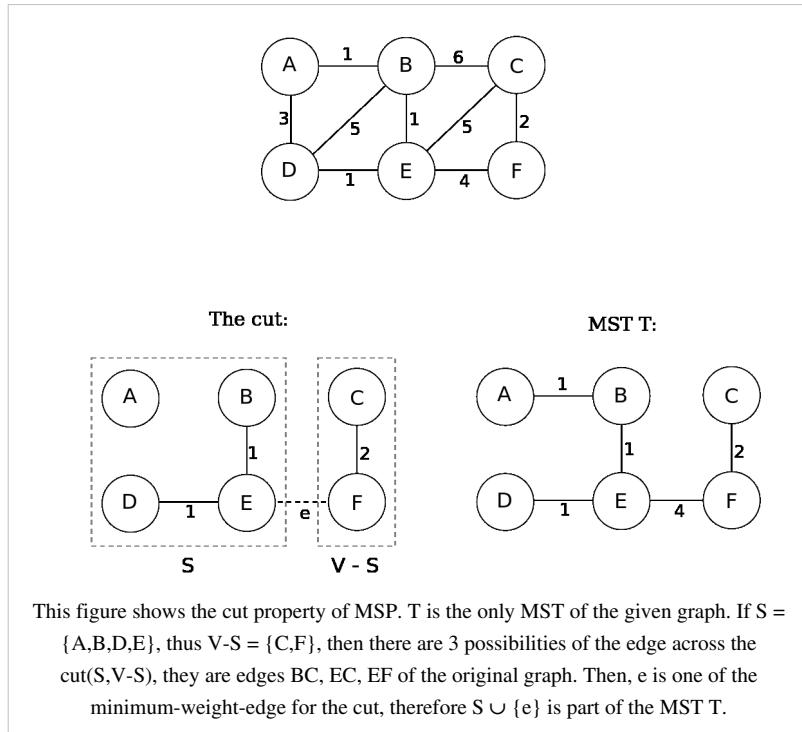
For any cycle C in the graph, if the weight of an edge e of C is larger than the weights of all other edges of C , then this edge cannot belong to an MST. Assuming the contrary, i.e. that e belongs to an MST T_1 , then deleting e will break T_1 into two subtrees with the two ends of e in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge f of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree T_2 with weight less than that of T_1 , because the weight of f is less than the weight of e .



This figure shows there may be more than one minimum spanning tree in a graph. In the figure, the two trees below the graph are two possibilities of minimum spanning tree of the given graph.

Cut property

For any cut C in the graph, if the weight of an edge e of C is strictly smaller than the weights of all other edges of C, then this edge belongs to all MSTs of the graph. To prove this, assume the contrary: in the figure at right, make edge BC (weight 6) part of the MST T instead of edge e (weight 4). Adding e to T will produce a cycle, while replacing BC with e would produce MST of smaller weight. Thus, a tree containing BC is not a MST, a contradiction that violates our assumption. By a similar argument, if more than one edge is of minimum weight across a cut, then each such edge is contained in a minimum spanning tree.



Minimum-cost edge

If the edge of a graph with the minimum cost e is unique, then this edge is included in any MST. Indeed, if e was not included in the MST, removing any of the (larger cost) edges in the cycle formed after adding e to the MST, would yield a spanning tree of smaller weight.

Algorithms

The first algorithm for finding a minimum spanning tree was developed by Czech scientist Otakar Borůvka in 1926 (see Borůvka's algorithm). Its purpose was an efficient electrical coverage of Moravia. There are now two algorithms commonly used, Prim's algorithm and Kruskal's algorithm. All three are greedy algorithms that run in polynomial time, so the problem of finding such trees is in **FP**, and related decision problems such as determining whether a particular edge is in the MST or determining if the minimum total weight exceeds a certain value are in **P**. Another greedy algorithm not as commonly used is the reverse-delete algorithm, which is the reverse of Kruskal's algorithm.

If the edge weights are integers, then deterministic algorithms are known that solve the problem in $O(m + n)$ integer operations, where m is the number of edges, n is the number of vertices. In a comparison model, in which the only allowed operations on edge weights are pairwise comparisons, Karger, Klein & Tarjan (1995) found a linear time randomized algorithm based on a combination of Borůvka's algorithm and the reverse-delete algorithm. Whether the problem can be solved deterministically in linear time by a comparison-based algorithm remains an open question, however. The fastest non-randomized comparison-based algorithm with known complexity, by Bernard Chazelle, is based on the soft heap, an approximate priority queue. Its running time is $O(m \alpha(m,n))$, where α is the classical functional inverse of the Ackermann function. The function α grows extremely slowly, so that for all practical purposes it may be considered a constant no greater than 4; thus Chazelle's algorithm takes very close to linear time. Seth Pettie and Vijaya Ramachandran have found a provably optimal deterministic comparison-based minimum spanning tree algorithm, the computational complexity of which is unknown.

Research has also considered parallel algorithms for the minimum spanning tree problem. With a linear number of processors it is possible to solve the problem in $O(\log n)$ time. Bader & Cong (2003) demonstrate an algorithm

that can compute MSTs 5 times faster on 8 processors than an optimized sequential algorithm.

Other specialized algorithms have been designed for computing minimum spanning trees of a graph so large that most of it must be stored on disk at all times. These *external storage* algorithms, for example as described in "Engineering an External Memory Minimum Spanning Tree Algorithm" by Roman, Dementiev et al., can operate, by authors' claims, as little as 2 to 5 times slower than a traditional in-memory algorithm. They rely on efficient external storage sorting algorithms and on graph contraction techniques for reducing the graph's size efficiently.

The problem can also be approached in a distributed manner. If each node is considered a computer and no node knows anything except its own connected links, one can still calculate the distributed minimum spanning tree.

MST on complete graphs

Alan M. Frieze showed that given a complete graph on n vertices, with edge weights that are independent identically distributed random variables with distribution function F satisfying $F'(0) > 0$, then as n approaches $+\infty$ the expected weight of the MST approaches $\zeta(3)/F'(0)$, where ζ is the Riemann zeta function. Frieze and Steele also proved convergence in probability. Svante Janson proved a central limit theorem for weight of the MST.

For uniform random weights in $[0, 1]$, the exact expected size of the minimum spanning tree has been computed for small complete graphs.

Vertices	Expected size	Approximative expected size
2	1 / 2	0.5
3	3 / 4	0.75
4	31 / 35	0.8857143
5	893 / 924	0.9664502
6	278 / 273	1.0183151
7	30739 / 29172	1.053716
8	199462271 / 184848378	1.0790588
9	126510063932 / 115228853025	1.0979027

Applications

Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids (which they were first invented for, as mentioned above). They are invoked as subroutines in algorithms for other problems, including the Christofides algorithm for approximating the traveling salesman problem,^[2] approximating the multi-terminal minimum cut problem (which is equivalent in the single-terminal case to the maximum flow problem), and approximating the minimum-cost weighted perfect matching.

Other practical applications based on minimal spanning trees include:

- Taxonomy.
- Cluster analysis: clustering points in the plane, single-linkage clustering (a method of hierarchical clustering), graph-theoretic clustering, and clustering gene expression data.
- Constructing trees for broadcasting in computer networks. On Ethernet networks this is accomplished by means of the Spanning tree protocol.
- Image registration and segmentation^[3] — see minimum spanning tree-based segmentation.
- Curvilinear feature extraction in computer vision.
- Handwriting recognition of mathematical expressions.

- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Regionalisation of socio-geographic areas, the grouping of areas into homogeneous, contiguous regions.
- Comparing ecotoxicology data.
- Topological observability in power systems.
- Measuring homogeneity of two-dimensional materials.
- Minimax process control.

In pedagogical contexts, minimum spanning tree algorithms serve as a common introductory example of both graph algorithms and greedy algorithms due to their simplicity.

Related problems

The problem of finding the Steiner tree of a subset of the vertices, that is, minimum tree that spans the given subset, is known to be NP-Complete.^[4]

A related problem is the k -minimum spanning tree (k -MST), which is the tree that spans some subset of k vertices in the graph with minimum weight.

A set of k -smallest spanning trees is a subset of k spanning trees (out of all possible spanning trees) such that no spanning tree outside the subset has smaller weight. (Note that this problem is unrelated to the k -minimum spanning tree.)

The Euclidean minimum spanning tree is a spanning tree of a graph with edge weights corresponding to the Euclidean distance between vertices which are points in the plane (or space).

The rectilinear minimum spanning tree is a spanning tree of a graph with edge weights corresponding to the rectilinear distance between vertices which are points in the plane (or space).

In the distributed model, where each node is considered a computer and no node knows anything except its own connected links, one can consider distributed minimum spanning tree. The mathematical definition of the problem is the same but there are different approaches for a solution.

The capacitated minimum spanning tree is a tree that has a marked node (origin, or root) and each of the subtrees attached to the node contains no more than a c nodes. c is called a tree capacity. Solving CMST optimally is NP-hard, but good heuristics such as Esau-Williams and Sharma produce solutions close to optimal in polynomial time.

The degree constrained minimum spanning tree is a minimum spanning tree in which each vertex is connected to no more than d other vertices, for some given number d . The case $d = 2$ is a special case of the traveling salesman problem, so the degree constrained minimum spanning tree is NP-hard in general.

For directed graphs, the minimum spanning tree problem is called the Arborescence problem and can be solved in quadratic time using the Chu–Liu/Edmonds algorithm.

A **maximum spanning tree** is a spanning tree with weight greater than or equal to the weight of every other spanning tree. Such a tree can be found with algorithms such as Prim's or Kruskal's after multiplying the edge weights by -1 and solving the MST problem on the new graph. A path in the maximum spanning tree is the widest path in the graph between its two endpoints: among all possible paths, it maximizes the weight of the minimum-weight edge. Maximum spanning trees find applications in parsing algorithms for natural languages and in training algorithms for conditional random fields.

The **dynamic MST** problem concerns the update of a previously computed MST after an edge weight change in the original graph or the insertion/deletion of a vertex.

The minimum labeling spanning tree problem is to find a spanning tree with least types of labels if each edge in a graph is associated with a label from a finite label set instead of a weight.

Minimum bottleneck spanning tree

A bottleneck edge is the highest weighted edge in a spanning tree. A spanning tree is a **minimum bottleneck spanning tree** (or **MBST**) if the graph does not contain a spanning tree with a smaller bottleneck edge weight. A MST is necessarily a MBST (provable by the cut property), but a MBST is not necessarily a MST.^{[5][6]}

References

- [1] Do the minimum spanning trees of a weighted graph have the same number of edges with a given weight? (<http://cs.stackexchange.com/questions/2204/do-the-minimum-spanning-trees-of-a-weighted-graph-have-the-same-number-of-edges>)
- [2] Nicos Christofides, Worst-case analysis of a new heuristic for the travelling salesman problem, Report 388, Graduate School of Industrial Administration, CMU, 1976.
- [3] P. Felzenszwalb, D. Huttenlocher: Efficient Graph-Based Image Segmentation. IJCV 59(2) (September 2004)
- [4] . ND12
- [5] <http://flashing-thoughts.blogspot.ru/2010/06/everything-about-bottleneck-spanning.html>
- [6] <http://pages.cpsc.ucalgary.ca/~dcatalin/413/t4.pdf>

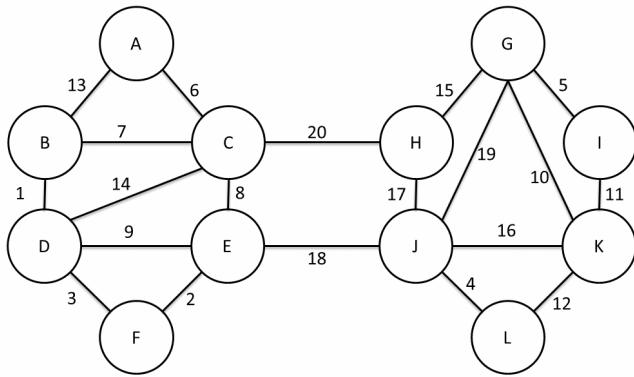
Additional reading

- Otakar Boruvka on Minimum Spanning Tree Problem (translation of the both 1926 papers, comments, history) (2000) (<http://citeseer.ist.psu.edu/nesetril00otakar.html>) Jaroslav Nesetril, Eva Milková, Helena Nesetrilová. (Section 7 gives his algorithm, which looks like a cross between Prim's and Kruskal's.)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 23: Minimum Spanning Trees, pp. 561–579.
- Eisner, Jason (1997). State-of-the-art algorithms for minimum spanning trees: A tutorial discussion (<http://www.cs.jhu.edu/~jason/papers/eisner.mst-tutorial.pdf>). Manuscript, University of Pennsylvania, April. 78 pp.
- Kromkowski, John David. "Still Unmelted after All These Years", in Annual Editions, Race and Ethnic Relations, 17/e (2009 McGraw Hill) (Using minimum spanning tree as method of demographic analysis of ethnic diversity across the United States).

External links

- Implemented in BGL, the Boost Graph Library (http://www.boost.org/libs/graph/doc/table_of_contents.html)
- The Stony Brook Algorithm Repository - Minimum Spanning Tree codes (<http://www.cs.sunysb.edu/~algorith/files/minimum-spanning-tree.shtml>)
- Implemented in QuickGraph for .Net (<http://www.codeplex.com/quickgraph>)

Borůvka's algorithm



An animation, describing Boruvka's (Sollin's) algorithm, for finding a minimum spanning tree in a graph - An example on the runtime of the algorithm

Graph and tree search algorithms

- $\alpha-\beta$
- A*
- B*
- Backtracking
- Beam
- Bellman–Ford
- Best-first
- Bidirectional
- Borůvka
- Branch & bound
- BFS
- British Museum
- D*
- DFS
- Depth-limited
- Dijkstra
- Edmonds
- Floyd–Warshall
- Fringe search
- Hill climbing
- IDA*
- Iterative deepening
- Johnson
- Jump point
- Kruskal
- Lexicographic BFS
- Prim
- SMA*
- Uniform-cost

Listings
• <i>Graph algorithms</i>
• <i>Search algorithms</i>
• <i>List of graph algorithms</i>
Related topics
• Dynamic programming
• Graph traversal
• Tree traversal
• Search games
• v
• t
• $e^{[1]}$

Borůvka's algorithm is an algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct.

It was first published in 1926 by Otakar Borůvka as a method of constructing an efficient electricity network for Moravia. The algorithm was rediscovered by Choquet in 1938; again by Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki in 1951; and again by Sollin in 1965. Because Sollin was the only computer scientist in this list living in an English speaking country, this algorithm is frequently called **Sollin's algorithm**, especially in the parallel computing literature.

The algorithm begins by first examining each vertex and adding the cheapest edge from that vertex to another in the graph, without regard to already added edges, and continues joining these groupings in a like manner until a tree spanning all vertices is completed.

Pseudocode

Designating each vertex or set of connected vertices a "component", pseudocode for Borůvka's algorithm is:

```

Input: A connected graph G whose edges have distinct weights
Initialize a forest T to be a set of one-vertex trees, one for each
vertex of the graph.
While T has more than one component:
  For each component C of T:
    Begin with an empty set of edges S
    For each vertex v in C:
      Find the cheapest edge from v to a vertex outside of C, and add
      it to S
    Add the cheapest edge in S to T
Output: T is the minimum spanning tree of G.

```

As in Kruskal's algorithm, tracking components of T can be done efficiently using a disjoint-set data structure. In graphs where edges have identical weights, edges with equal weights can be ordered based on the lexicographic order of their endpoints.

Complexity

Borůvka's algorithm can be shown to take $O(\log V)$ iterations of the outer loop until it terminates, and therefore to run in time $O(E \log V)$, where E is the number of edges, and V is the number of vertices in G . In planar graphs, and more generally in families of graphs closed under graph minor operations, it can be made to run in linear time, by removing all but the cheapest edge between each pair of components after each stage of the algorithm.^[1]

Example

Image	components	Description
	{A}, {B}, {C}, {D}, {E}, {F}, {G}	This is our original weighted graph. The numbers near the edges indicate their weight. Initially, every vertex by itself is a component (blue circles).
	{A,B,D,F}, {C,E,G}	In the first iteration of the outer loop, the minimum weight edge out of every component is added. Some edges are selected twice (AD, CE). Two components remain.
	{A,B,C,D,E,F,G}	In the second and final iteration, the minimum weight edge out of each of the two remaining components is added. These happen to be the same edge. One component remains and we are done. The edge BD is not considered because both endpoints are in the same component.

Other algorithms

Other algorithms for this problem include Prim's algorithm and Kruskal's algorithm. Fast parallel algorithms can be obtained by combining Prim's algorithm with Borůvka's.

A faster randomized minimum spanning tree algorithm based in part on Borůvka's algorithm due to Karger, Klein, and Tarjan runs in expected $O(E)$ time. The best known (deterministic) minimum spanning tree algorithm by Bernard Chazelle is also based in part on Borůvka's and runs in $O(E \alpha(E,V))$ time, where α is the inverse of the Ackermann function. These randomized and deterministic algorithms combine steps of Borůvka's algorithm, reducing the number of components that remain to be connected, with steps of a different type that reduce the number of edges between pairs of components.

Notes

[1] ; .

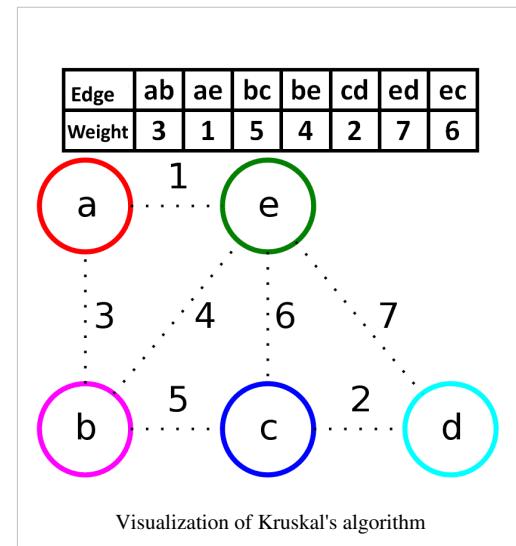
Kruskal's algorithm

Graph and tree search algorithms	
•	α - β
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall
•	Fringe search
•	Hill climbing
•	IDA*
•	Iterative deepening
•	Johnson
•	Jump point
•	Kruskal
•	Lexicographic BFS
•	Prim
•	SMA*
•	Uniform-cost
Listings	
•	<i>Graph algorithms</i>
•	<i>Search algorithms</i>
•	<i>List of graph algorithms</i>
Related topics	
•	Dynamic programming
•	Graph traversal
•	Tree traversal
•	Search games
•	v
•	t
•	e ^[1]

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

This algorithm first appeared in *Proceedings of the American Mathematical Society*, pp. 48–50 in 1956, and was written by Joseph Kruskal.

Other algorithms for this problem include Prim's algorithm, Reverse-delete algorithm, and Borůvka's algorithm.



Description

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty and F is not yet spanning
 - remove an edge with minimum weight from S
 - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

Pseudocode

The following code is implemented with disjoint-set data structure:

```

KRUSKAL (G) :
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET (v)
4 foreach (u, v) ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ { (u, v) }
7     UNION(u, v)
8 return A

```

Complexity

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

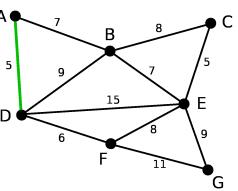
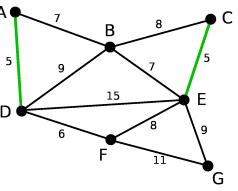
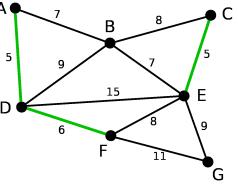
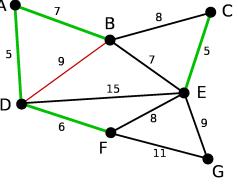
- E is at most V^2 and $\log V^2 = 2 \log V$ is $O(\log V)$.
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain $V \leq E+1$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from S " to operate in constant time. Next, we use a disjoint-set data structure (Union&Find) to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

Provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the algorithm can use more sophisticated disjoint-set data structure to run in $O(E \alpha(V))$ time, where α is the extremely slowly growing inverse of the single-valued Ackermann function.

Example

Download the example data. ^[1]

Image	Description
	AD and CE are the shortest edges, with length 5, and AD has been arbitrarily chosen, so it is highlighted.
	CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.
	The next edge, DF with length 6, is highlighted using much the same method.
	The next-shortest edges are AB and BE , both with length 7. AB is chosen arbitrarily, and is highlighted. The edge BD has been highlighted in red, because there already exists a path (in green) between B and D , so it would form a cycle (ABD) if it were chosen.

	The process continues to highlight the next-smallest edge, BE with length 7. Many more edges are highlighted in red at this stage: BC because it would form the loop BCE , DE because it would form the loop DEBA , and FE because it would form FEBAD .
	Finally, the process finishes with the edge EG of length 9, and the minimum spanning tree is found.

Proof of correctness

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed spanning tree is of minimal weight.

Spanning tree

Let P be a connected, weighted graph and let Y be the subgraph of P produced by the algorithm. Y cannot have a cycle, been within one subtree and not between two different trees. Y cannot be disconnected, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of P .

Minimality

We show that the following proposition P is true by induction: If F is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains F .

- Clearly P is true at the beginning, when F is empty: any minimum spanning tree will do, and there exists one because a weighted connected graph always has a minimum spanning tree.
- Now assume P is true for some non-final edge set F and let T be a minimum spanning tree that contains F . If the next chosen edge e is also in T , then P is true for $F + e$. Otherwise, $T + e$ has a cycle C and there is another edge f that is in C but not F . (If there were no such edge f , then e could not have been added to F , since doing so would have created the cycle C .) Then $T - f + e$ is a tree, and it has the same weight as T , since T has minimum weight and the weight of f cannot be less than the weight of e , otherwise the algorithm would have chosen f instead of e . So $T - f + e$ is a minimum spanning tree containing $F + e$ and again P holds.
- Therefore, by the principle of induction, P holds when F has become a spanning tree, which is only possible if F is a minimum spanning tree itself.

References

- [1] <http://www.carlschroedl.com/blog/comp/kruskals-minimum-spanning-tree-algorithm/2012/05/14/>
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 23.2: The algorithms of Kruskal and Prim, pp. 567–574.
 - Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*, Fourth Edition. John Wiley & Sons, Inc., 2006. ISBN 0-471-73884-0. Section 13.7.1: Kruskal's Algorithm, pp. 632..

External links

- Kruskal's algorithm explanation and example with c implementation (http://scantree.com/Data_Structure/kruskal's-algorithm)
- Download the example minimum spanning tree data. (<http://www.carlschroedl.com/blog/comp/kruskals-minimum-spanning-tree-algorithm/2012/05/14/>)
- Animation of Kruskal's algorithm (Requires Java plugin) (<http://students.ceid.upatras.gr/~papagel/project/kruskal.htm>)
- download kruskal algorithm Implement with C++ and java(graphical)(Requires java 7+) (http://www.programyar.com/wp-content/uploads/2012/08/wood_cutter_kruskal_algorithm_with_javac++.zip)
- C# Implementation (http://www.codeproject.com/KB/recipes/Kruskal_Algorithm.aspx)
- Open source java graph library with implementation of Kruskal's algorithm (<https://github.com/monmohan/mgraphlib>)
- Auto-generated PowerPoint Slides for Teaching and Learning (https://docs.google.com/file/d/0B2_b0Jz3VKm8RG8zOTI1dFRrRnM/edit?usp=sharing)

Prim's algorithm

Graph and tree search algorithms	
•	$\alpha-\beta$
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall

<ul style="list-style-type: none"> • Fringe search • Hill climbing • IDA* • Iterative deepening • Johnson • Jump point • Kruskal • Lexicographic BFS • Prim • SMA* • Uniform-cost
Listings
<ul style="list-style-type: none"> • <i>Graph algorithms</i> • <i>Search algorithms</i> • <i>List of graph algorithms</i>
Related topics
<ul style="list-style-type: none"> • Dynamic programming • Graph traversal • Tree traversal • Search games
<ul style="list-style-type: none"> • v • t • $e^{[1]}$

In computer science, **Prim's algorithm** is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is also sometimes called the **DJP algorithm**, the **Jarník algorithm**, or the **Prim–Jarník algorithm**.

Other algorithms for this problem include Kruskal's algorithm and Borůvka's algorithm. These algorithms find the minimum spanning forest in a possibly disconnected graph. By running Prim's algorithm for each connected component of the graph, it can also be used to find the minimum spanning forest.

Description

Informal

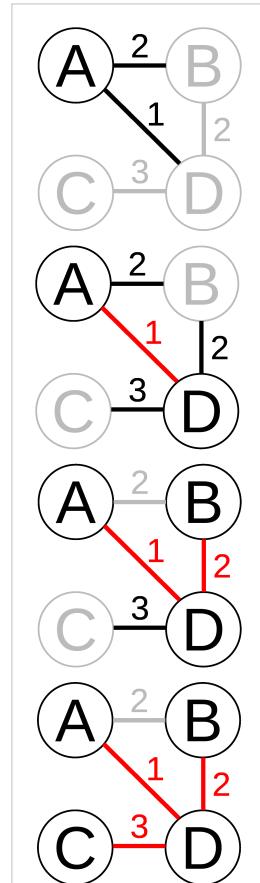
1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Technical

If a graph is empty then we are done immediately. Thus, we assume otherwise.

The algorithm starts with a tree consisting of a single vertex, and continuously increases its size one edge at a time, until it spans all vertices.

- Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
- Initialize: $V_{\text{new}} = \{x\}$, where x is an arbitrary node (starting point) from V , $E_{\text{new}} = \{\}$
- Repeat until $V_{\text{new}} = V$:
 - Choose an edge $\{u, v\}$ with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
 - Add v to V_{new} , and $\{u, v\}$ to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree



Prim's algorithm starting at vertex A. In the second step, BD is chosen to add to the tree instead of AB arbitrarily, as both have weight 2. Afterwards, AB is excluded because it is between two nodes that are already in the tree.

Time complexity



Prim's algorithm has many applications, such as in the generation of this maze, which applies Prim's algorithm to a randomly weighted grid graph.

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(V ^2)$
binary heap and adjacency list	$O((V + E) \log V) = O(E \log V)$
Fibonacci heap and adjacency list	$O(E + V \log V)$

A simple implementation of Prim's, using an adjacency matrix graph representation and linearly searching an array of weights to find the minimum weight edge, to add requires $O(|V|^2)$ running time. Switching to an adjacency list representation brings this down to $O(|V||E|)$, which is strictly better for sparse graphs. However, this running time can be greatly improved further by using heaps to implement finding minimum weight edges in the algorithm's inner loop.

A first improved version uses a heap to store all edges of the input graph, ordered by their weight. This leads to an $O(|E| \log |E|)$ worst-case running time. But storing vertices instead of edges can improve it still further. The heap should order the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed minimum spanning tree (MST) (or infinity if no such edge exists). Every time a vertex v is chosen and added to the MST, a decrease-key operation is performed on all vertices w outside the partial MST such that v is connected to w , setting the key to the minimum of its previous value and the edge cost of (v,w) .

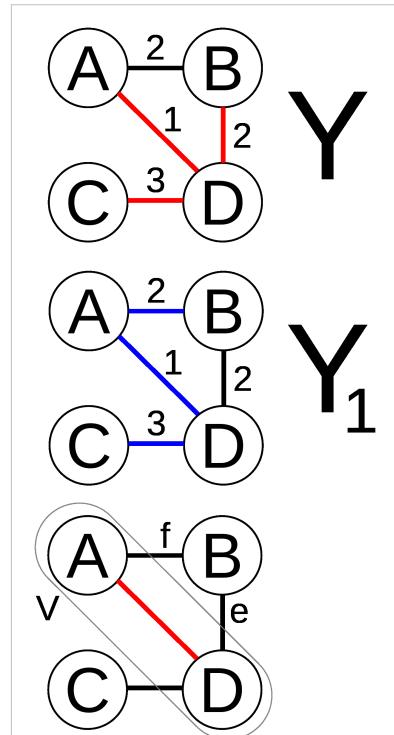
Using a simple binary heap data structure, Prim's algorithm can now be shown to run in time $O(|E| \log |V|)$ where $|E|$ is the number of edges and $|V|$ is the number of vertices. Using a more sophisticated Fibonacci heap, this can be brought down to $O(|E| + |V| \log |V|)$, which is asymptotically faster when the graph is dense enough that $|E|$ is $\omega(|V|)$.

Proof of correctness

Let P be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in a subgraph to a vertex outside the subgraph. Since P is connected, there will always be a path to every vertex. The output Y of Prim's algorithm is a tree, because the edge and vertex added to tree Y are connected. Let Y_1 be a minimum spanning tree of graph P . If $Y_1 = Y$ then Y is a minimum spanning tree. Otherwise, let e be the first edge added during the construction of tree Y that is not in tree Y_1 , and V be the set of vertices connected by the edges added before edge e . Then one endpoint of edge e is in set V and the other is not. Since tree Y_1 is a spanning tree of graph P , there is a path in tree Y_1 joining the two endpoints. As one travels along the path, one must encounter an edge f joining a vertex in set V to one that is not in set V . Now, at the iteration when edge e was added to tree Y , edge f could also have been added and it would be added instead of edge e if its weight was less than e , and since edge f was not added, we conclude that

$$w(f) \geq w(e).$$

Let tree Y_2 be the graph obtained by removing edge f from and adding edge e to tree Y_1 . It is easy to show that tree Y_2 is connected, has the same number of edges as tree Y_1 , and the total weights of its edges is not larger than that of tree Y_1 , therefore it is also a minimum spanning tree of graph P and it contains edge e and all the edges added before it during the construction of set V . Repeat the steps above and we will eventually obtain a minimum spanning tree of graph P that is identical to tree Y . This shows Y is a minimum spanning tree.



Demonstration of proof. In this case, the graph $Y_1 = Y - f + e$ is already equal to Y .

In general, the process may need to be repeated.

References

- V. Jarník: *O jistém problému minimálním* [About a certain minimal problem], Práce Moravské Přírodovědecké Společnosti, 6, 1930, pp. 57–63. (in Czech)
- R. C. Prim: *Shortest connection networks and some generalizations*. In: *Bell System Technical Journal*, 36 (1957), pp. 1389–1401
- D. Cheriton and R. E. Tarjan: *Finding minimum spanning trees*. In: *SIAM Journal on Computing*, 5 (Dec. 1976), pp. 724–741
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Third Edition. MIT Press, 2009. ISBN 0-262-03384-4. Section 23.2: The algorithms of Kruskal and Prim, pp. 631–638.

External links



Wikimedia Commons has media related to *Prim's Algorithm*.

- Prim's algorithm with 'C' implementation (http://scantree.com/Data_Structure/prim's-algorithm)
- Animated example of Prim's algorithm (<http://students.ceid.upatras.gr/~papagel/project/prim.htm>)
- Minimum spanning tree demonstration Python program by Ronald L. Rivest (<http://people.csail.mit.edu/rivest/programs.html>)
- Open Source Java Graph package with implementation of Prim's Algorithm (<http://code.google.com/p/annas/>)
- Open Source C# class library with implementation of Prim's Algorithm (<http://code.google.com/p/ngenerics/>)
- Open Source Java graph library with implementation of Prim's Algorithm (<https://github.com/monmohan/mgraphlib/>)

Edmonds's algorithm for directed minimum spanning trees

Graph and tree search algorithms	
•	α - β
•	A*
•	B*
•	Backtracking
•	Beam
•	Bellman–Ford
•	Best-first
•	Bidirectional
•	Borůvka
•	Branch & bound
•	BFS
•	British Museum
•	D*
•	DFS
•	Depth-limited
•	Dijkstra
•	Edmonds
•	Floyd–Warshall
•	Fringe search
•	Hill climbing
•	IDA*
•	Iterative deepening
•	Johnson
•	Jump point
•	Kruskal
•	Lexicographic BFS
•	Prim
•	SMA*
•	Uniform-cost
Listings	
•	<i>Graph algorithms</i>
•	<i>Search algorithms</i>
•	<i>List of graph algorithms</i>
Related topics	
•	Dynamic programming
•	Graph traversal
•	Tree traversal
•	Search games
•	v
•	t
•	e ^[1]

This article is about the optimum branching algorithm. For the maximum matching algorithm, see Blossom algorithm.

In graph theory, a branch of mathematics, **Edmonds' algorithm** or **Chu–Liu/Edmonds' algorithm** is an algorithm for finding a maximum or minimum *optimum branchings*. This is similar to the minimum spanning tree problem which concerns undirected graphs. However, when nodes are connected by weighted edges that are directed, a minimum spanning tree algorithm cannot be used.

The optimum branching algorithm was proposed independently first by Yoeng-jin Chu and Tseng-hong Liu (1965) and then by Edmonds (1967). To find a maximum path length, the largest edge value is found and connected between the two nodes, then the next largest value, and so on. If an edge creates a loop, it is erased. A minimum path length is found by starting from the smallest value.

Running time

The running time of this algorithm is $O(EV)$. A faster implementation of the algorithm due to Robert Tarjan runs in time $O(E \log V)$ for sparse graphs and $O(V^2)$ for dense graphs. This is as fast as Prim's algorithm for an undirected minimum spanning tree. In 1986, Gabow, Galil, Spencer, and Tarjan produced a faster implementation, with running time $O(E + V \log V)$.

Algorithm

Description

The algorithm has a conceptual recursive description. We will denote by f the function which, given a weighted directed graph D with a distinguished vertex r called the *root*, returns a spanning tree rooted at r of minimal cost. The precise description is as follows. Given a weighted directed graph D with root r we first replace any set of parallel edges (edges between the same pair of vertices in the same direction) by a single edge with weight equal to the minimum of the weights of these parallel edges.

Now, for each node v other than the root, mark an (arbitrarily chosen) incoming edge of lowest cost. Denote the other endpoint of this edge by $\pi(v)$. The edge is now denoted as $(\pi(v), v)$ with associated cost $w(\pi(v), v)$. If the marked edges form an SRT (Shortest Route Tree), $f(D)$ is defined to be this SRT. Otherwise, the set of marked edges form at least one cycle. Call (an arbitrarily chosen) one of these cycles C . We now define a weighted directed graph D' having a root r' as follows. The nodes of D' are the nodes of D not in C plus a new node denoted v_C .

If (u, v) is an edge in D with $u \notin C$ and $v \in C$, then include in D' the edge $e = (u, v_C)$, and define $w(e) = w(u, v) - w(\pi(v), v)$.

If (u, v) is an edge in D with $u \in C$ and $v \notin C$, then include in D' the edge $e = (v_C, v)$, and define $w(e) = w(u, v)$.

If (u, v) is an edge in D with $u \notin C$ and $v \notin C$, then include in D' the edge $e = (u, v)$, and define $w(e) = w(u, v)$.

We include no other edges in D' .

The root r' of D' is simply the root r in D .

Using a call to $f(D')$, find an SRT of D' . First, mark in D all shared edges with D' that are marked in the SRT of D' . Also, mark in D all the edges in C . Now, suppose that in the SRT of D' , the (unique) incoming edge at v_C is (u, v_C) . This edge comes from some pair (u, v) with $u \notin C$ and $v \in C$. Unmark $(\pi(v), v)$ and mark (u, v) . Also, for each marked (v_C, v) in the SRT of D' and coming from an edge (u, v) in D with $u \in C$ and $v \notin C$, mark the edge (u, v) . Now the set of marked edges do form an SRT, which we define to be

the value of $f(D)$.

Observe that $f(D)$ is defined in terms of $f(D')$ for weighted directed rooted graphs D' having strictly fewer vertices than D , and finding $f(D)$ for a single-vertex graph is trivial.

Implementation

Let BV be a vertex bucket and BE be an edge bucket. Let v be a vertex and e be an edge of maximum positive weight that is incident to v . C_i is a circuit. $G_0 = (V_0, E_0)$ is the original digraph. u_i is a replacement vertex for C_i .

```

 $BV \leftarrow BE \leftarrow \emptyset$ 
i=0

A:
if  $BV = V_i$  then goto B
for some vertex  $v \notin BV$  and  $v \in V_i$  {
     $BV \leftarrow BV \cup \{v\}$ 
    find an edge  $e = (x, v)$  such that  $w(e) = \max\{w(y, v) | (y, v) \in E_i\}$ 
    if  $w(e) \leq 0$  then goto A
}
if  $BE \cup \{e\}$  contains a circuit {
    i=i+1
    construct  $G_i$  by shrinking  $C_i$  to  $u_i$ 
    modify BE, BV and some edge weights
}
 $BE \leftarrow BE \cup e$ 
goto A

B:
while i ≠ 0 {
    reconstruct  $G_{i-1}$  and rename some edges in BE
    if  $u_i$  was a root of an out-tree in BE {
         $BE \leftarrow BE \cup \{e | e \in C_i \text{ and } e \neq e_0^i\}$ 
    } else{
         $BE \leftarrow BE \cup \{e | e \in C_i \text{ and } e \neq \tilde{e}_i\}$ 
    }
    i=i-1
}
Maximum branching weight =  $\sum_{e \in BE} w(e)$ 
```

References

- Chu, Y. J.; Liu, T. H. (1965), "On the Shortest Arborescence of a Directed Graph", *Science Sinica* **14**: 1396–1400
- Edmonds, J. (1967), "Optimum Branchings", *J. Res. Nat. Bur. Standards* **71B**: 233–240
- Tarjan, R. E. (1977), "Finding Optimum Branchings", *Networks* **7**: 25–35
- Camerini, P.M.; Fratta, L.; Maffioli, F. (1979), "A note on finding optimum branchings", *Networks* **9**: 309–312
- Gibbons, Alan (1985), *Algorithmic Graph Theory*, Cambridge University press, ISBN 0-521-28881-9
- Gabow, H. N.; Galil, Z.; Spencer, T.; Tarjan, R. E. (1986), "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs", *Combinatorica* **6**: 109–122

External links

- The Directed Minimum Spanning Tree Problem ^[1] Description of the algorithm summarized by Shanchieh Jay Yang, May 2000.
- Edmonds's algorithm (edmonds-alg) ^[2] – An open source implementation of Edmonds's algorithm written in C++ and licensed under the MIT License. This source is using Tarjan's implementation for the dense graph.

References

- [1] <http://www.ce.rit.edu/~sjyec/dmst.html>
[2] <http://edmonds-alg.sourceforge.net/>

Degree-constrained spanning tree

In graph theory, a **degree-constrained spanning tree** is a spanning tree where the maximum vertex degree is limited to a certain constant k . The **degree-constrained spanning tree problem** is to determine whether a particular graph has such a spanning tree for a particular k .

Formal definition

Input: n -node undirected graph $G(V,E)$; positive integer $k \leq n$.

Question: Does G have a spanning tree in which no node has degree greater than k ?

NP-completeness

This problem is NP-complete (Garey & Johnson 1979). This can be shown by a reduction from the Hamiltonian path problem. It remains NP-complete even if k is fixed to a value ≥ 2 . If the problem is defined as the degree must be $\leq k$, the $k = 2$ case of degree-confined spanning tree is the Hamiltonian path problem.

Degree-constrained minimum spanning tree

On a weighted graph, a Degree-constrained minimum spanning tree (DCMST) is a degree-constrained spanning tree in which the sum of its edges has the minimum possible sum. Finding a DCMST is an NP-Hard problem.^[1]

Heuristic algorithms that can solve the problem in polynomial time have been proposed, including Genetic and Ant-Based Algorithms.

Approximation Algorithm

Fürer & Raghavachari (1994) gave an approximation algorithm for the problem which, on any given instance, either shows that the instance has no tree of maximum degree k or it finds and returns a tree of maximum degree $k+1$.

References

- [1] Bui, T. N. and Zrncic, C. M. 2006. An ant-based algorithm for finding degree-constrained minimum spanning tree. (http://www.cs.york.ac.uk/rts/docs/GECCO_2006/docs/p11.pdf) In GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, pages 11–18, New York, NY, USA. ACM.
- Garey, Michael R.; Johnson, David S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, ISBN 0-7167-1045-5. A2.1: ND1, p. 206.
- Fürer, Martin; Raghavachari, Balaji (1994), "Approximating the minimum-degree Steiner tree to within one of optimal", *Journal of Algorithms* 17 (3): 409–423, doi: 10.1006/jagm.1994.1042 (<http://dx.doi.org/10.1006/jagm.1994.1042>).

Maximum-leaf spanning tree

In graph theory, a **connected dominated set** and a **maximum leaf spanning tree** are two closely related structures defined on an undirected graph.

Definitions

A connected dominating set of a graph G is a set D of vertices with two properties:

1. Any node in D can reach any other node in D by a path that stays entirely within D . That is, D induces a connected subgraph of G .
2. Every vertex in G either belongs to D or is adjacent to a vertex in D . That is, D is a dominating set of G .

A **minimum connected dominating set** of a graph G is a connecting dominating set with the smallest possible cardinality among all connected dominating sets of G . The **connected domination number** of G is the number of vertices in the minimum connected dominating set.

Any spanning tree T of a graph G has at least two leaves, vertices that have only one edge of T incident to them. A maximum leaf spanning tree is a spanning tree that has the largest possible number of leaves among all spanning trees of G . The **max leaf number** of G is the number of leaves in the maximum leaf spanning tree.

Complementarity

If d is the connected domination number of an n -vertex graph G , where $n > 2$, and l is its max leaf number, then the three quantities d , l , and n obey the simple equation

$$n = d + l.$$

If D is a connected dominating set, then there exists a spanning tree in G whose leaves include all vertices that are not in D : form a spanning tree of the subgraph induced by D , together with edges connecting each remaining vertex v that is not in D to a neighbor of v in D . This shows that $l \geq n - d$.

In the other direction, if T is any spanning tree in G , then the vertices of T that are not leaves form a connected dominating set of G . This shows that $n - l \geq d$. Putting these two inequalities together proves the equality $n = d + l$.

Therefore, in any graph, the sum of the connected domination number and the max leaf number equals the total number of vertices. Computationally, this implies that finding the minimum dominating set is equally difficult to finding a maximum leaf spanning tree.

Algorithms

It is NP-complete to test whether there exists a connected dominating set with size less than a given threshold, or equivalently to test whether there exists a spanning tree with at least a given number of leaves. Therefore, it is believed that the minimum connected dominating set problem and the maximum leaf spanning tree problem cannot be solved in polynomial time.

When viewed in terms of approximation algorithms, connected domination and maximum leaf spanning trees are not the same: approximating one to within a given approximation ratio is not the same as approximating the other to the same ratio. There exists an approximation for the minimum connected dominating set that achieves a factor of $2 \ln \Delta + O(1)$, where Δ is the maximum degree of a vertex in G . The maximum leaf spanning tree problem is MAX-SNP hard, implying that no polynomial time approximation scheme is likely. However, it can be approximated to within a factor of 2 in polynomial time.

Applications

Connected dominating sets are useful in the computation of routing for mobile ad hoc networks. In this application, a small connected dominating set is used as a backbone for communications, and nodes that are not in this set communicate by passing messages through neighbors that are in the set.

The max leaf number has been employed in the development of fixed-parameter tractable algorithms: several NP-hard optimization problems may be solved in polynomial time for graphs of bounded max leaf number.

References

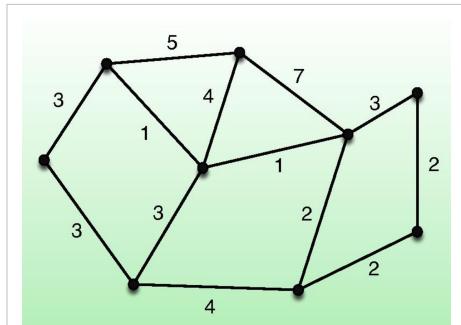
K-minimum spanning tree

The **k -minimum spanning tree problem**, studied in theoretical computer science, asks for a tree of minimum cost that has exactly k vertices and forms a subgraph of a larger graph. It is also called the **k -MST** or **edge-weighted k -cardinality tree**. Finding this tree is NP-hard, but it can be approximated to within a constant approximation ratio in polynomial time.

The input to the problem consists of an undirected graph with weights on its edges, and a number k . The output is a tree with k vertices and $k - 1$ edges, with all of the edges of the output tree belonging to the input graph. The cost of the output is the sum of the weights of its edges, and the goal is to find the tree that has minimum cost.

The k -MST problem has been shown to be NP-hard by a reduction from the Steiner tree problem.

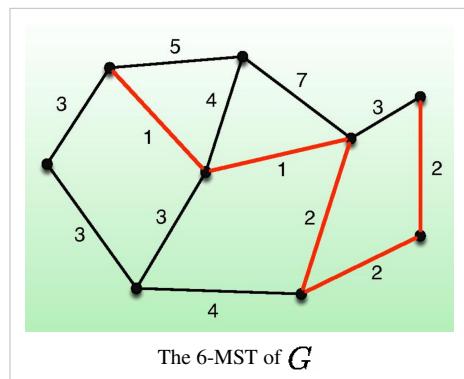
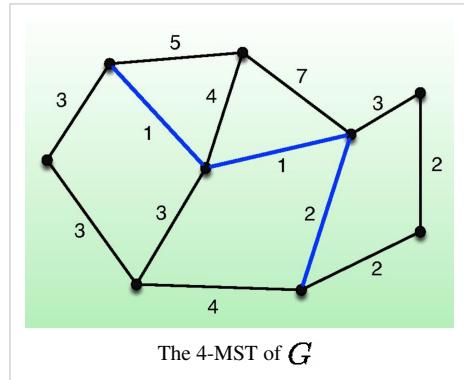
The best approximation known for the problem achieves an approximation ratio of 2, and is by Garg (2005). This approximation relies heavily on the primal-dual schema of Goemans & Williamson (1992). When the input consists of points in the Euclidean plane (any two of which can be connected in the tree with cost equal to their distance) there exists a polynomial time approximation scheme devised by Arora (1998).



An example of an undirected graph G with edge costs

References

- Arora, Sanjeev (1998), "Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems", *Journal of the ACM* **45** (5): 753–782, doi:10.1145/290179.290180 [1].
- Garg, Naveen (2005), "Saving an epsilon: a 2-approximation for the k-MST problem in graphs", *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pp. 396–402, doi:10.1145/1060590.1060650 [2].
- Ravi, R.; Sundaram, R.; Marathe, M.; Rosenkrantz, D.; Ravi, S. (1996), "Spanning trees short or small", *SIAM Journal on Discrete Mathematics* **9** (2): 178–200, doi:10.1137/S0895480194266331 [3].
- Goemans, M.; Williamson, P. (1992), "A general approximation technique for constrained forest problems", *SIAM Journal on Computing* **24** (2): 296–317, doi:10.1137/S0097539793242618 [4].



External links

- Minimum k-spanning tree in "A compendium of NP optimization problems" [5]
- KCTLIB [6], KCTLIB -- A Library for the Edge-Weighted K-Cardinality Tree Problem

References

- [1] <http://dx.doi.org/10.1145%2F290179.290180>
- [2] <http://dx.doi.org/10.1145%2F1060590.1060650>
- [3] <http://dx.doi.org/10.1137%2FS0895480194266331>
- [4] <http://dx.doi.org/10.1137%2FS0097539793242618>
- [5] <http://www.nada.kth.se/~viggo/wwwcompendium/node71.html>
- [6] <http://iridia.ulb.ac.be/~cblum/kctlib/>

Capacitated minimum spanning tree

Capacitated minimum spanning tree is a minimal cost spanning tree of a graph that has a designated root node r and satisfies the capacity constraint c . The capacity constraint ensures that all subtrees (maximal subgraphs connected to the root by a single edge) incident on the root node r have no more than c nodes. If the tree nodes have weights, then the capacity constraint may be interpreted as follows: the sum of weights in any subtree should be no greater than c . The edges connecting the subgraphs to the root node are called *gates*. Finding the optimal solution is NP-hard.

Algorithms

Suppose we have a graph $G = (V, E)$, $n = |G|$ with a root $r \in G$. Let a_i be all other nodes in G . Let c_{ij} be the edge cost between vertices a_i and a_j which form a cost matrix $C = c_{ij}$.

Esau-Williams heuristic

Esau-Williams heuristic finds suboptimal CMST that are very close to the exact solutions, but on average EW produces better results than many other heuristics.

Initially, all nodes are connected to the root r (star graph) and the network's cost is $\sum_{i=0}^n c_{ri}$; each of these edges is a gate. At each iteration, we seek the closest neighbor a_j for every node in $G - r$ and evaluate the tradeoff function: $t(a_i) = g_i - c_{ij}$. We look for the greatest $t(a_i)$ among the positive tradeoffs and, if the resulting subtree does not violate the capacity constraints, remove the gate g_i connecting the i -th subtree to a_j by an edge c_{ij} . We repeat the iterations until we can not make any further improvements to the tree.
Esau-Williams heuristics for computing a suboptimal CMST:

```

function CMST( $c, C, r$ ) :
     $T = \{c_{1r}, c_{2r}, \dots, c_{nr}\}$ 
    while have changes:
        for each node  $a_i$ 
             $a_i$  = closest node in a different subtree
             $t(a_i) = g_i - c_{ij}$ 
             $t_{\max} = \max(t(a_i))$ 
             $k = i$  such that  $t(a_k) = t_{\max}$ 
            if ( $\text{cost}(i) + \text{cost}(j) \leq c$ )
                 $T = T - g_k$ 
                 $T = T \cup c_{kj}$ 
    return  $T$ 

```

It is easy to see that EW finds a solution in polynomial time.

Sharma's heuristic

Sharma's heuristic.

Applications

CMST problem is important in network design: when many terminal computers have to be connected to the central hub, the star configuration is usually not the minimum cost design. Finding a CMST that organizes the terminals into subnetworks can lower the cost of implementing a network.

Limitations

But CMST is still not provide the minimum cost for long situated nodes. overcome this drawback ESAU Williams has solved this problem.

References

Application: Single-linkage clustering

Single-linkage clustering is one of several methods of agglomerative hierarchical clustering. In the beginning of the process, each element is in a cluster of its own. The clusters are then sequentially combined into larger clusters, until all elements end up being in the same cluster. At each step, the two clusters separated by the shortest distance are combined. The definition of 'shortest distance' is what differentiates between the different agglomerative clustering methods. In single-linkage clustering, the link between two clusters is made by a single element pair, namely those two elements (one in each cluster) that are closest to each other. The shortest of these links that remains at any step causes the fusion of the two clusters whose elements are involved. The method is also known as **nearest neighbour clustering**. The result of the clustering can be visualized as a dendrogram, which shows the sequence of cluster fusion and the distance at which each fusion took place.^[1]

Mathematically, the linkage function – the distance $D(X,Y)$ between clusters X and Y – is described by the expression

$$D(X, Y) = \min_{x \in X, y \in Y} d(x, y),$$

where X and Y are any two sets of elements considered as clusters, and $d(x,y)$ denotes the distance between the two elements x and y .

A drawback of this method is the so-called *chaining phenomenon*, which refers to the gradual growth of a cluster as one element at a time gets added to it. This may lead to impractically heterogeneous clusters and difficulties in defining classes that could usefully subdivide the data.

Naive Algorithm

The following algorithm is an agglomerative scheme that erases rows and columns in a proximity matrix as old clusters are merged into new ones. The $N \times N$ proximity matrix D contains all distances $d(i,j)$. The clusterings are assigned sequence numbers $0, 1, \dots, (n - 1)$ and $L(k)$ is the level of the k th clustering. A cluster with sequence number m is denoted (m) and the proximity between clusters (r) and (s) is denoted $d[(r),(s)]$.

The algorithm is composed of the following steps:

1. Begin with the disjoint clustering having level $L(0) = 0$ and sequence number $m = 0$.
2. Find the most similar pair of clusters in the current clustering, say pair $(r), (s)$, according to $d[(r),(s)] = \min d[(i),(j)]$ where the minimum is over all pairs of clusters in the current clustering.
3. Increment the sequence number: $m = m + 1$. Merge clusters (r) and (s) into a single cluster to form the next clustering m . Set the level of this clustering to $L(m) = d[(r),(s)]$
4. Update the proximity matrix, D , by deleting the rows and columns corresponding to clusters (r) and (s) and adding a row and column corresponding to the newly formed cluster. The proximity between the new cluster, denoted (r,s) and old cluster (k) is defined as $d[(k), (r,s)] = \min d[(k),(r)], d[(k),(s)]$.
5. If all objects are in one cluster, stop. Else, go to step 2.

Optimally efficient algorithm

The algorithm explained above is easy to understand but of complexity $\mathcal{O}(n^3)$. In 1973, R. Sibson proposed an optimally efficient algorithm of only complexity $\mathcal{O}(n^2)$ known as SLINK.

Other linkages

This is essentially the same as Kruskal's algorithm for minimum spanning trees. However, in single linkage clustering, the order in which clusters are formed is important, while for minimum spanning trees what matters is the set of pairs of points that form distances chosen by the algorithm.

Alternative linkage schemes include complete linkage and Average linkage clustering - implementing a different linkage in the naive algorithm is simply a matter of using a different formula to calculate inter-cluster distances in the initial computation of the proximity matrix and in step 4 of the above algorithm. An optimally efficient algorithm is however not available for arbitrary linkages. The formula that should be adjusted has been highlighted using bold text.

References

[1] Legendre, P. & Legendre, L. 1998. Numerical Ecology. Second English Edition. 853 pages.

External links

- Single linkage clustering algorithm implementation in Ruby (AI4R) (<http://ai4r.org>)
- Linkages used in Matlab (<http://www.mathworks.com/help/toolbox/stats/linkage.html>)

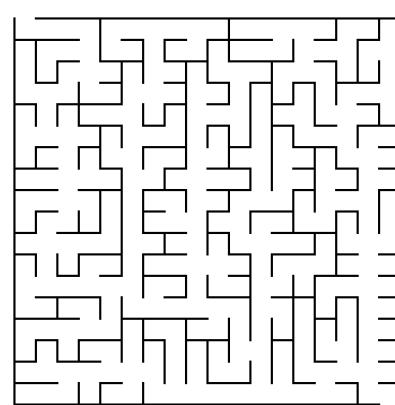
Application: Maze generation

Maze generation algorithms are automated methods for the creation of mazes.

Graph theory based methods

A maze can be generated by starting with a predetermined arrangement of cells (most commonly a rectangular grid but other arrangements are possible) with wall sites between them. This predetermined arrangement can be considered as a connected graph with the edges representing possible wall sites and the nodes representing cells. The purpose of the maze generation algorithm can then be considered to be making a subgraph where it is challenging to find a route between two particular nodes.

If the subgraph is not connected, then there are regions of the graph that are wasted because they do not contribute to the search space. If the graph contains loops, then there may be multiple paths between the chosen nodes. Because of this, maze generation is often approached as generating a random spanning tree. Loops which can confound naive maze solvers may be introduced by adding random edges to the result during the course of the algorithm.



This maze generated by modified version of Prim's algorithm, below.

Depth-first search

This algorithm is a randomized version of the depth-first search algorithm. Frequently implemented with a stack, this approach is one of the simplest ways to generate a maze using a computer. Consider the space for a maze being a large grid of cells (like a large chess board), each cell starting with four walls. Starting from a random cell, the computer then selects a random neighbouring cell that has not yet been visited. The computer removes the 'wall' between the two cells and adds the new cell to a stack (this is analogous to drawing the line on the floor). The computer continues this process, with a cell that has no unvisited neighbours being considered a dead-end. When at a dead-end it backtracks through the path until it reaches a cell with an unvisited neighbour, continuing the path generation by visiting this new, unvisited cell (creating a new junction). This process continues until every cell has been visited, causing the computer to backtrack all the way back to the beginning cell. This approach guarantees that the maze space is completely visited.



Animation of generator's thinking process using Depth-First Search

As stated, the algorithm is very simple and does not produce over-complex mazes. More specific refinements to the algorithm can help to generate mazes that are harder to solve.

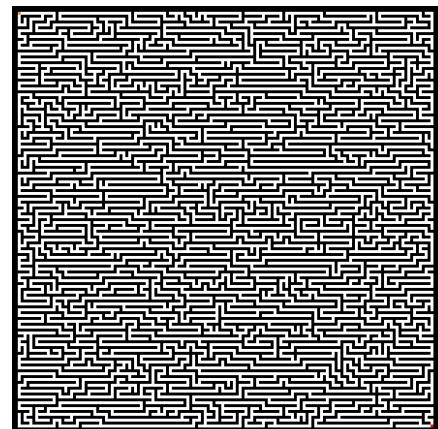
1. Start at a particular cell and call it the "exit."
2. Mark the current cell as visited, and get a list of its neighbors. For each neighbor, starting with a randomly selected neighbor:
 1. If that neighbor hasn't been visited, remove the wall between this cell and that neighbor, and then recurse with that neighbor as the current cell.

As given above this algorithm involves deep recursion which may cause stack overflow issues on some computer architectures. The algorithm can be rearranged into a loop by storing backtracking information in the maze itself.

This also provides a quick way to display a solution, by starting at any given point and backtracking to the exit.

Mazes generated with a depth-first search have a low branching factor and contain many long corridors. Also mazes will typically be relatively easy to find the way to the square that was first picked at the beginning of the algorithm, since most paths lead to or from there, but it is hard to find the way out.

To add difficulty and a fun factor to depth-first search generated mazes, you can influence the likelihood of which neighbor you should visit, instead of it being completely random. By making it more likely to visit neighbors to your sides, you can have a more horizontal maze generation. Experimenting with directional "influence" in certain places could lead to creating fun designs, such as a checkerboard pattern, an X, and more.



Horizontal Influence

Recursive backtracker

The depth-first search algorithm of maze generation is frequently implemented using backtracking:

1. Make the initial cell the current cell and mark it as visited
2. While there are unvisited cells
 1. If the current cell has any neighbours which have not been visited
 1. Choose randomly one of the unvisited neighbours
 2. Push the current cell to the stack
 3. Remove the wall between the current cell and the chosen cell
 4. Make the chosen cell the current cell and mark it as visited
 2. Else if stack is not empty
 1. Pop a cell from the stack
 2. Make it the current cell
 3. Else
 1. Pick a random unvisited cell, make it the current cell and mark it as visited

Randomized Kruskal's algorithm

This algorithm is a randomized version of Kruskal's algorithm.

1. Create a list of all walls, and create a set for each cell, each containing just that one cell.
2. For each wall, in some random order:
 1. If the cells divided by this wall belong to distinct sets:
 1. Remove the current wall.
 2. Join the sets of the formerly divided cells.

There are several data structures that can be used to model the sets of cells. An efficient implementation using a disjoint-set data structure can perform each union and find operation on two sets in nearly constant amortized time (specifically, $O(\alpha(V))$ time; $\alpha(x) < 5$ for any plausible value of x), so the running time of this algorithm is essentially proportional to the number of walls available to the maze.



An animation of generating a 30 by 20 maze using Kruskal's algorithm.

It matters little whether the list of walls is initially randomized or if a wall is randomly chosen from a nonrandom list, either way is just as easy to code.

Because the effect of this algorithm is to produce a minimal spanning tree from a graph with equally weighted edges, it tends to produce regular patterns which are fairly easy to solve.

Randomized Prim's algorithm

This algorithm is a randomized version of Prim's algorithm.

1. Start with a grid full of walls.
2. Pick a cell, mark it as part of the maze. Add the walls of the cell to the wall list.
3. While there are walls in the list:
 1. Pick a random wall from the list. If the cell on the opposite side isn't in the maze yet:
 1. Make the wall a passage and mark the cell on the opposite side as part of the maze.
 2. Add the neighboring walls of the cell to the wall list.
 2. If the cell on the opposite side already was in the maze, remove the wall from the list.

Like the depth-first algorithm, it will usually be relatively easy to find the way to the starting cell, but hard to find the way anywhere else.

Note that simply running classical Prim's on a graph with random weights would create mazes stylistically identical to Kruskal's, because they are both minimal spanning tree algorithms. Instead, this algorithm introduces stylistic variation because the edges closer to the starting point have a lower effective weight.



An animation of generating a 30 by 20 maze using Prim's algorithm.

Modified version

Although the classical Prim's algorithm keeps a list of edges, for maze generation we could instead maintain a list of adjacent cells. If the randomly chosen cell has multiple edges that connect it to the existing maze, select one of these edges at random. This will tend to branch slightly more than the edge-based version above.

Recursive division method

Illustration of Recursive Division

<i>original chamber</i>	<i>division by two walls</i>	<i>holes in walls</i>	<i>continue subdividing...</i>	<i>completed</i>

Mazes can be created with *recursive division*, an algorithm which works as follows: Begin with the maze's space with no walls. Call this a chamber. Divide the chamber with a randomly positioned wall (or multiple walls) where

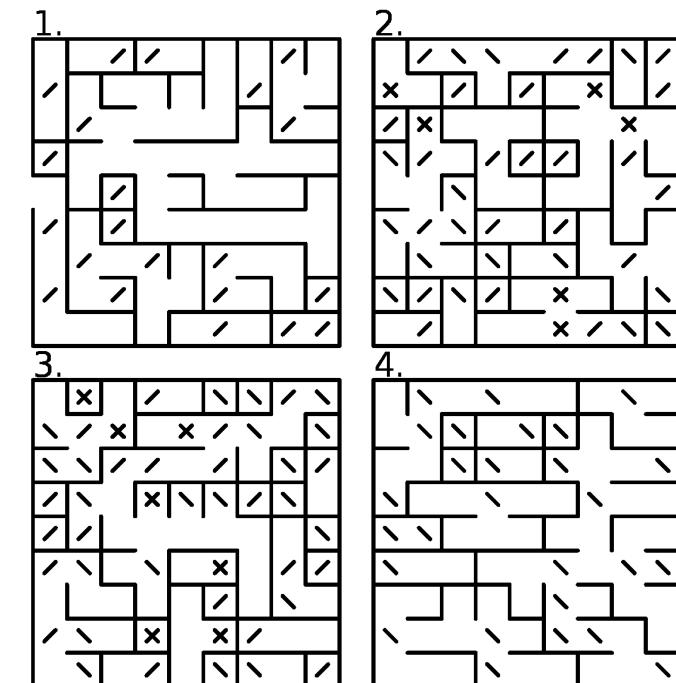
each wall contains a randomly positioned passage opening within it. Then recursively repeat the process on the subchambers until all chambers are minimum sized. This method results in mazes with long straight walls crossing their space, making it easier to see which areas to avoid.

For example, in a rectangular maze, build at random points two walls that are perpendicular to each other. These two walls divide the large chamber into four smaller chambers separated by four walls. Choose three of the four walls at random, and open a one cell-wide hole at a random point in each of the three. Continue in this manner recursively, until every chamber has a width of one cell in either of the two directions.

Simple algorithms

Other algorithms exist that require only enough memory to store one line of a 2D maze or one plane of a 3D maze. They prevent loops by storing which cells in the current line are connected through cells in the previous lines, and never remove walls between any two cells already connected.

Most maze generation algorithms require maintaining relationships between cells within it, to ensure the end result will be solvable. Valid simply connected mazes can however be generated by focusing on each cell independently. A binary tree maze is a standard orthogonal maze where each cell always has a passage leading up or leading left, but never both. To create a binary tree maze, for each cell flip a coin to decide whether to add a passage leading up or left. Always pick the same direction for cells on the boundary, and the end result will be a valid simply connected maze that looks like a binary tree, with the upper left corner its root.



3D version of Prim's algorithm. Vertical layers are labeled 1 through 4 from bottom to top. Stairs up are indicated with "/"; stairs down with "\", and stairs up-and-down with "x". Source code is included with the image description.

A related form of flipping a coin for each cell is to create an image using a random mix of forward slash and backslash characters. This doesn't generate a valid simply connected maze, but rather a selection of closed loops and unicursal passages. (The manual for the Commodore 64 presents a BASIC program using this algorithm, but using PETSCII diagonal line graphic characters instead for a smoother graphic appearance.)

Cellular automaton algorithms

Certain types of cellular automata can be used to generate mazes. Two well-known such cellular automata, Maze and Mazectric, have rulestrings B3/S12345 and B3/S1234. In the former, this means that cells survive from one generation to the next if they have at least one and at most five neighbours. In the latter, this means that cells survive if they have one to four neighbours. If a cell has exactly three neighbours, it is born. It is similar to Conway's Game of Life in that patterns that do not have a living cell adjacent to 1, 4, or 5 other living cells in any generation will behave identically to it. However, for large patterns, it behaves very differently.

For a random starting pattern, these maze-generating cellular automata will evolve into complex mazes with well-defined walls outlining corridors. Mazecetric, which has the rule B3/S1234 has a tendency to generate longer and straighter corridors compared with Maze, with the rule B3/S12345. Since these cellular automaton rules are deterministic, each maze generated is uniquely determined by its random starting pattern. This is a significant drawback since the mazes tend to be relatively predictable.

Like some of the graph-theory based methods described above, these cellular automata typically generate mazes from a single starting pattern; hence it will usually be relatively easy to find the way to the starting cell, but harder to find the way anywhere else.

Python code example

```

import numpy
from numpy.random import random_integers as rand
import matplotlib.pyplot as pyplot

def maze(width=81, height=51, complexity=.75, density=.75):
    # Only odd shapes
    shape = ((height // 2) * 2 + 1, (width // 2) * 2 + 1)
    # Adjust complexity and density relative to maze size
    complexity = int(complexity * (5 * (shape[0] + shape[1])))
    density     = int(density * (shape[0] // 2 * shape[1] // 2))
    # Build actual maze
    Z = numpy.zeros(shape, dtype=bool)
    # Fill borders
    Z[0, :] = Z[-1, :] = 1
    Z[:, 0] = Z[:, -1] = 1
    # Make aisles
    for i in range(density):
        x, y = rand(0, shape[1] // 2) * 2, rand(0, shape[0] // 2) * 2
        Z[y, x] = 1
        for j in range(complexity):
            neighbours = []
            if x > 1:                neighbours.append((y, x - 2))
            if x < shape[1] - 2:    neighbours.append((y, x + 2))
            if y > 1:                neighbours.append((y - 2, x))
            if y < shape[0] - 2:    neighbours.append((y + 2, x))
            if len(neighbours):
                y_,x_ = neighbours[rand(0, len(neighbours) - 1)]
                if Z[y_, x_] == 0:
                    Z[y_, x_] = 1
                    Z[y_ + (y - y_) // 2, x_ + (x - x_) // 2] = 1
                    x, y = x_, y_
    return Z

pyplot.figure(figsize=(10, 5))
pyplot.imshow(maze(80, 40), cmap=pyplot.cm.binary,
interpolation='nearest')
pyplot.xticks([]), pyplot.yticks([])

```

```
pyplot.show()
```

References

External links

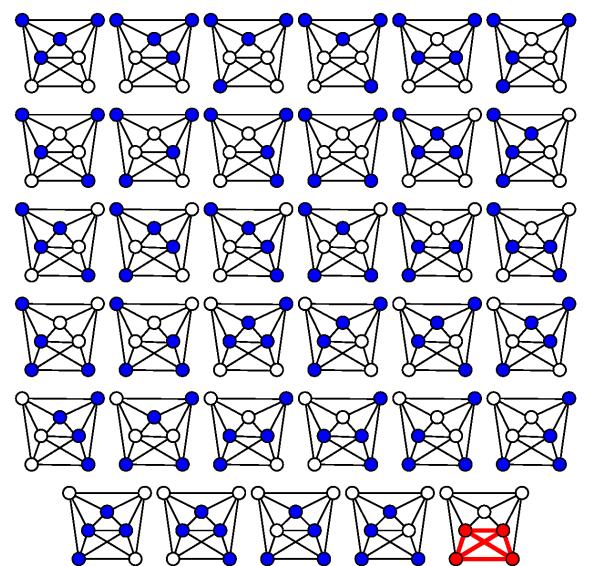
- Jamis Buck: HTML 5 Presentation with Demos of Maze generation Algorithms (<http://www.jamisbuck.org/presentations/rubyconf2011/index.html>)
- Think Labyrinth: Maze algorithms (<http://www.astrolog.org/labyrnth/algrithm.htm#perfect>) (details on these and other maze generation algorithms)
- Maze Generation (<http://www.martinfoltin.sk/mazes>) - Master's Thesis (Java Applet enabling users to have a maze created using various algorithms and human solving of mazes)
- Collection of maze generation code (<http://rosettacode.org/wiki/Maze>) in different languages in Rosetta Code
- Maze generation and navigation in 3D (<http://totologic.blogspot.com/2013/04/maze-generation-in-3d.html>)

Cliques, independent sets, and coloring

Clique problem

In computer science, the **clique problem** refers to any of the problems related to finding particular complete subgraphs ("cliques") in a graph, i.e., sets of elements where each pair of elements is connected.

For example, the **maximum clique problem** arises in the following real-world setting. Consider a social network, where the graph's vertices represent people, and the graph's edges represent mutual acquaintance. To find a largest subset of people who all know each other, one can systematically inspect all subsets, a process that is too time-consuming to be practical for social networks comprising more than a few dozen people. Although this brute-force search can be improved by more efficient algorithms, all of these algorithms take exponential time to solve the problem. Therefore, much of the theory about the clique problem is devoted to identifying special types of graph that admit more efficient algorithms, or to establishing the computational difficulty of the general problem in various models of computation. Along with its applications in social networks, the clique problem also has many applications in bioinformatics and computational chemistry.^[1]



The brute force algorithm finds a 4-clique in this 7-vertex graph (the complement of the 7-vertex path graph) by systematically checking all $C(7,4)=35$ 4-vertex subgraphs for completeness.

Clique problems include:

- finding the maximum clique (a clique with the largest number of vertices),
- finding a maximum weight clique in a weighted graph,
- listing all maximal cliques (cliques that cannot be enlarged)
- solving the decision problem of testing whether a graph contains a clique larger than a given size.

These problems are all hard: the clique decision problem is NP-complete (one of Karp's 21 NP-complete problems), the problem of finding the maximum clique is both fixed-parameter intractable and hard to approximate, and listing all maximal cliques may require exponential time as there exist graphs with exponentially many maximal cliques. Nevertheless, there are algorithms for these problems that run in exponential time or that handle certain more specialized input graphs in polynomial time.^[1]

History

Although complete subgraphs have been studied for longer in mathematics,^[2] the term "clique" and the problem of algorithmically listing cliques both come from the social sciences, where complete subgraphs are used to model social cliques, groups of people who all know each other. The "clique" terminology comes from Luce & Perry (1949), and the first algorithm for solving the clique problem is that of Harary & Ross (1957), who were motivated by the sociological application.

Since the work of Harary and Ross, many others have devised algorithms for various versions of the clique problem. In the 1970s, researchers began studying these algorithms from the point of view of worst-case analysis; see, for instance, Tarjan & Trojanowski (1977), an early work on the worst-case complexity of the maximum clique problem. Also in the 1970s, beginning with the work of Cook (1971) and Karp (1972), researchers began finding mathematical justification for the perceived difficulty of the clique problem in the theory of NP-completeness and related intractability results. In the 1990s, a breakthrough series of papers beginning with Feige et al. (1991) and reported at the time in major newspapers, showed that it is not even possible to approximate the problem accurately and efficiently.

Definitions

Main article: Clique (graph theory)

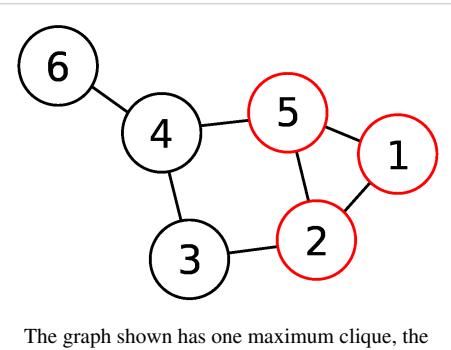
An undirected graph is formed by a finite set of vertices and a set of unordered pairs of vertices, which are called edges. By convention, in algorithm analysis, the number of vertices in the graph is denoted by n and the number of edges is denoted by m . A clique in a graph G is a complete subgraph of G ; that is, it is a subset S of the vertices such that every two vertices in S are connected by an edge in G . A maximal clique is a clique to which no more vertices can be added; a maximum clique is a clique that includes the largest possible number of vertices, and the clique number $\omega(G)$ is the number of vertices in a maximum clique of G .

Several closely related clique-finding problems have been studied.

- In the maximum clique problem, the input is an undirected graph, and the output is a maximum clique in the graph. If there are multiple maximum cliques, only one need be output.
- In the weighted maximum clique problem, the input is an undirected graph with weights on its vertices (or, less frequently, edges) and the output is a clique with maximum total weight. The maximum clique problem is the special case in which all weights are equal.
- In the maximal clique listing problem, the input is an undirected graph, and the output is a list of all its maximal cliques. The maximum clique problem may be solved using as a subroutine an algorithm for the maximal clique listing problem, because the maximum clique must be included among all the maximal cliques.
- In the k -clique problem, the input is an undirected graph and a number k , and the output is a clique of size k if one exists (or, sometimes, all cliques of size k).
- In the clique decision problem, the input is an undirected graph and a number k , and the output is a Boolean value: true if the graph contains a k -clique, and false otherwise.

The first four of these problems are all important in practical applications; the clique decision problem is not, but is necessary in order to apply the theory of NP-completeness to clique-finding problems.

The clique problem and the independent set problem are complementary: a clique in G is an independent set in the complement graph of G and vice versa. Therefore, many computational results may be applied equally well to either problem, and some research papers do not clearly distinguish between the two problems. However, the two problems have different properties when applied to restricted families of graphs; for instance, the clique problem may be solved in polynomial time for planar graphs while the independent set problem remains NP-hard on planar graphs.



The graph shown has one maximum clique, the triangle {1,2,5}, and four more maximal cliques, the pairs {2,3}, {3,4}, {4,5}, and {4,6}.

Algorithms

Maximal versus maximum

A maximal clique, sometimes called inclusion-maximal, is a clique that is not included in a larger clique. Note, therefore, that every clique is contained in a maximal clique.

Maximal cliques can be very small. A graph may contain a non-maximal clique with many vertices and a separate clique of size 2 which is maximal. While a maximum (i.e., largest) clique is necessarily maximal, the converse does not hold. There are some types of graphs in which every maximal clique is maximum (the complements of well-covered graphs, notably including complete graphs, triangle-free graphs without isolated vertices, complete multipartite graphs, and k-trees) but other graphs have maximal cliques that are not maximum.

Finding a maximal clique is straightforward: Starting with an arbitrary clique (for instance, a single vertex), grow the current clique one vertex at a time by iterating over the graph's remaining vertices, adding a vertex if it is connected to each vertex in the current clique, and discarding it otherwise. This algorithm runs in linear time. Because of the ease of finding maximal cliques, and their potential small size, more attention has been given to the much harder algorithmic problem of finding a maximum or otherwise large clique than has been given to the problem of finding a single maximal clique.

Cliques of fixed size

A brute force algorithm to test whether a graph G contains a k -vertex clique, and to find any such clique that it contains, is to examine each subgraph with at least k vertices and check to see whether it forms a clique. This algorithm takes time $O(n^k k^2)$: there are $O(n^k)$ subgraphs to check, each of which has $O(k^2)$ edges whose presence in G needs to be checked. Thus, the problem may be solved in polynomial time whenever k is a fixed constant. When k is part of the input to the problem, however, the time is exponential.^[3]

The simplest nontrivial case of the clique-finding problem is finding a triangle in a graph, or equivalently determining whether the graph is triangle-free. In a graph with m edges, there may be at most $\Theta(m^{3/2})$ triangles; the worst case occurs when G is itself a clique. Therefore, algorithms for listing all triangles must take at least $\Omega(m^{3/2})$ time in the worst case, and algorithms are known that match this time bound.^[4] For instance, Chiba & Nishizeki (1985) describe an algorithm that sorts the vertices in order from highest degree to lowest and then iterates through each vertex v in the sorted list, looking for triangles that include v and do not include any previous vertex in the list. To do so the algorithm marks all neighbors of v , searches through all edges incident to a neighbor of v outputting a triangle for every edge that has two marked endpoints, and then removes the marks and deletes v from the graph. As the authors show, the time for this algorithm is proportional to the arboricity of the graph ($a(G)$) times the number of edges, which is $O(m a(G))$. Since the arboricity is at most $O(m^{1/2})$, this algorithm runs in time $O(m^{3/2})$. More generally, all k -vertex cliques can be listed by a similar algorithm that takes time proportional to the number of edges times the $(k - 2)$ nd power of the arboricity. For graphs of constant arboricity, such as planar graphs (or in general graphs from any non-trivial minor-closed graph family), this algorithm takes $O(m)$ time, which is optimal since it is linear in the size of the input.

If one desires only a single triangle, or an assurance that the graph is triangle-free, faster algorithms are possible. As Itai & Rodeh (1978) observe, the graph contains a triangle if and only if its adjacency matrix and the square of the adjacency matrix contain nonzero entries in the same cell; therefore, fast matrix multiplication techniques such as the Coppersmith–Winograd algorithm can be applied to find triangles in time $O(n^{2.376})$, which may be faster than $O(m^{3/2})$ for sufficiently dense graphs. Alon, Yuster & Zwick (1994) have improved the $O(m^{3/2})$ algorithm for finding triangles to $O(m^{1.41})$ by using fast matrix multiplication. This idea of using fast matrix multiplication to find triangles has also been extended to problems of finding k -cliques for larger values of k .^[5]

Listing all maximal cliques

By a result of Moon & Moser (1965), any n -vertex graph has at most $3^{n/3}$ maximal cliques. The Bron–Kerbosch algorithm is a recursive backtracking procedure of Bron & Kerbosch (1973) that augments a candidate clique by considering one vertex at a time, either adding it to the candidate clique or to a set of excluded vertices that cannot be in the clique but must have some non-neighbor in the eventual clique; variants of this algorithm can be shown to have worst-case running time $O(3^{n/3})$. Therefore, this provides a worst-case-optimal solution to the problem of listing all maximal independent sets; further, the Bron–Kerbosch algorithm has been widely reported as being faster in practice than its alternatives.^[6]

As Tsukiyama et al. (1977) showed, it is also possible to list all maximal cliques in a graph in an amount of time that is polynomial per generated clique. An algorithm such as theirs in which the running time depends on the output size is known as an output-sensitive algorithm. Their algorithm is based on the following two observations, relating the maximal cliques of the given graph G to the maximal cliques of a graph $G \setminus v$ formed by removing an arbitrary vertex v from G :

- For every maximal clique C of $G \setminus v$, either C continues to form a maximal clique in G , or $C \cup \{v\}$ forms a maximal clique in G . Therefore, G has at least as many maximal cliques as $G \setminus v$ does.
- Each maximal clique in G that does not contain v is a maximal clique in $G \setminus v$, and each maximal clique in G that does contain v can be formed from a maximal clique C in $G \setminus v$ by adding v and removing the non-neighbors of v from C .

Using these observations they can generate all maximal cliques in G by a recursive algorithm that, for each maximal clique C in $G \setminus v$, outputs C and the clique formed by adding v to C and removing the non-neighbors of v . However, some cliques of G may be generated in this way from more than one parent clique of $G \setminus v$, so they eliminate duplicates by outputting a clique in G only when its parent in $G \setminus v$ is lexicographically maximum among all possible parent cliques. On the basis of this principle, they show that all maximal cliques in G may be generated in time $O(mn)$ per clique, where m is the number of edges in G and n is the number of vertices; Chiba & Nishizeki (1985) improve this to $O(ma)$ per clique, where a is the arboricity of the given graph. Makino & Uno (2004) provide an alternative output-sensitive algorithm based on fast matrix multiplication, and Johnson & Yannakakis (1988) show that it is even possible to list all maximal cliques in lexicographic order with polynomial delay per clique, although the reverse of this order is NP-hard to generate.

On the basis of this result, it is possible to list all maximal cliques in polynomial time, for families of graphs in which the number of cliques is polynomially bounded. These families include chordal graphs, complete graphs, triangle-free graphs, interval graphs, graphs of bounded boxicity, and planar graphs. In particular, the planar graphs, and more generally, any family of graphs that is both sparse (having a number of edges at most a constant times the number of vertices) and closed under the operation of taking subgraphs, have $O(n)$ cliques, of at most constant size, that can be listed in linear time.

Finding maximum cliques in arbitrary graphs

It is possible to find the maximum clique, or the clique number, of an arbitrary n -vertex graph in time $O(3^{n/3}) = O(1.4422^n)$ by using one of the algorithms described above to list all maximal cliques in the graph and returning the largest one. However, for this variant of the clique problem better worst-case time bounds are possible. The algorithm of Tarjan & Trojanowski (1977) solves this problem in time $O(2^{n/3}) = O(1.2599^n)$; it is a recursive backtracking scheme similar to that of the Bron–Kerbosch algorithm, but is able to eliminate some recursive calls when it can be shown that some other combination of vertices not used in the call is guaranteed to lead to a solution at least as good. Jian (1986) improved this to $O(2^{0.304n}) = O(1.2346^n)$. Robson (1986) improved this to $O(2^{0.276n}) = O(1.2108^n)$ time, at the expense of greater space usage, by a similar backtracking scheme with a more complicated case analysis, together with a dynamic programming technique in which the optimal solution is precomputed for all small connected subgraphs of the complement graph and these partial solutions are used to shortcut the

backtracking recursion. The fastest algorithm known today is due to Robson (2001) which runs in time $O(2^{0.249n}) = O(1.1888^n)$.

There has also been extensive research on heuristic algorithms for solving maximum clique problems without worst-case runtime guarantees, based on methods including branch and bound,^[7] local search,^[8] greedy algorithms,^[9] and constraint programming. Non-standard computing methodologies for finding cliques include DNA computing^[10] and adiabatic quantum computation. The maximum clique problem was the subject of an implementation challenge sponsored by DIMACS in 1992–1993, and a collection of graphs used as benchmarks for the challenge is publicly available.^[11]

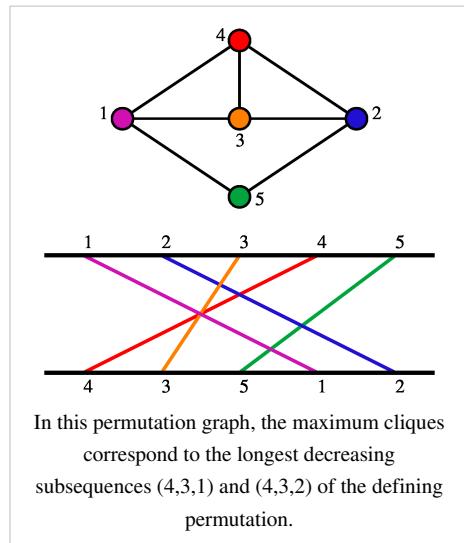
Special classes of graphs

Planar graphs, and other families of sparse graphs, have been discussed above: they have linearly many maximal cliques, of bounded size, that can be listed in linear time. In particular, for planar graphs, any clique can have at most four vertices, by Kuratowski's theorem.

Perfect graphs are defined by the properties that their clique number equals their chromatic number, and that this equality holds also in each of their induced subgraphs. For perfect graphs, it is possible to find a maximum clique in polynomial time, using an algorithm based on semidefinite programming. However, this method is complex and non-combinatorial, and specialized clique-finding algorithms have been developed for many subclasses of perfect graphs. In the complement graphs of bipartite graphs, König's theorem allows the maximum clique problem to be solved using techniques for matching. In another class of perfect graphs, the permutation graphs, a maximum clique is a longest decreasing subsequence of the permutation defining the graph and can be found using known algorithms for the longest decreasing subsequence problem.^[12] In chordal graphs, the maximal cliques are a subset of the n cliques formed as part of an elimination ordering.

In some cases, these algorithms can be extended to other, non-perfect, classes of graphs as well: for instance, in a circle graph, the neighborhood of each vertex is a permutation graph, so a maximum clique in a circle graph can be found by applying the permutation graph algorithm to each neighborhood.^[13] Similarly, in a unit disk graph (with a known geometric representation), there is a polynomial time algorithm for maximum cliques based on applying the algorithm for complements of bipartite graphs to shared neighborhoods of pairs of vertices.

The algorithmic problem of finding a maximum clique in a random graph drawn from the Erdős–Rényi model (in which each edge appears with probability 1/2, independently from the other edges) was suggested by Karp (1976). Although the clique number of such graphs is very close to $2 \log_2 n$, simple greedy algorithms as well as more sophisticated randomized approximation techniques only find cliques with size $\log_2 n$, and the number of maximal cliques in such graphs is with high probability exponential in $\log^2 n$ preventing a polynomial time solution that lists all of them. Because of the difficulty of this problem, several authors have investigated variants of the problem in which the random graph is augmented by adding a large clique, of size proportional to \sqrt{n} . It is possible to find this hidden clique with high probability in polynomial time, using either spectral methods or semidefinite programming.



Approximation algorithms

Several authors have considered approximation algorithms that attempt to find a clique or independent set that, although not maximum, has size as close to the maximum as can be found in polynomial time. Although much of this work has focused on independent sets in sparse graphs, a case that does not make sense for the complementary clique problem, there has also been work on approximation algorithms that do not use such sparsity assumptions.^[14]

Feige (2004) describes a polynomial time algorithm that finds a clique of size $\Omega((\log n/\log \log n)^2)$ in any graph that has clique number $\Omega(n/\log^k n)$ for any constant k . By combining this algorithm to find cliques in graphs with clique numbers between $n/\log n$ and $n/\log^3 n$ with a different algorithm of Boppana & Halldórsson (1992) to find cliques in graphs with higher clique numbers, and choosing a two-vertex clique if both algorithms fail to find anything, Feige provides an approximation algorithm that finds a clique with a number of vertices within a factor of $O(n(\log \log n)^2/\log^3 n)$ of the maximum. Although the approximation ratio of this algorithm is weak, it is the best known to date, and the results on hardness of approximation described below suggest that there can be no approximation algorithm with an approximation ratio significantly less than linear.

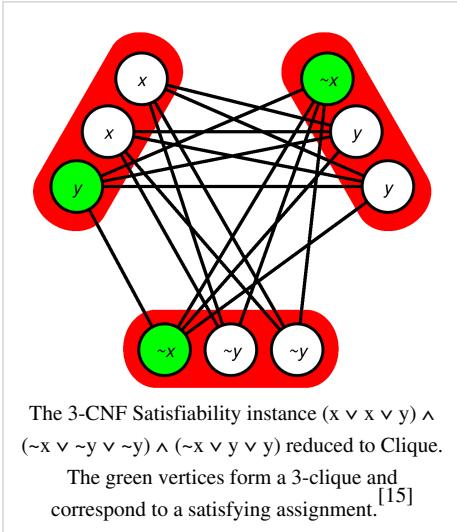
Lower bounds

NP-completeness

The clique decision problem is NP-complete. It was one of Richard Karp's original 21 problems shown NP-complete in his 1972 paper "Reducibility Among Combinatorial Problems". This problem was also mentioned in Stephen Cook's paper introducing the theory of NP-complete problems. Thus, the problem of finding a maximum clique is NP-hard: if one could solve it, one could also solve the decision problem, by comparing the size of the maximum clique to the size parameter given as input in the decision problem.

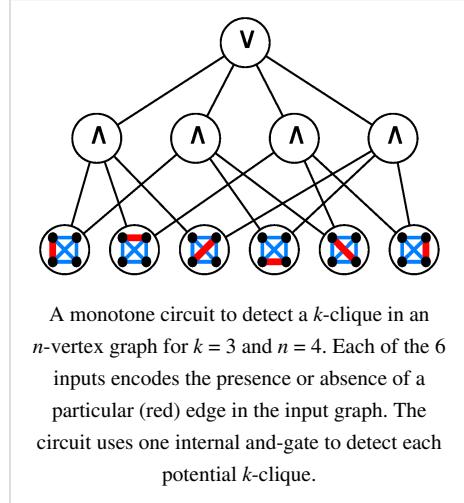
Karp's NP-completeness proof is a many-one reduction from the Boolean satisfiability problem for formulas in conjunctive normal form, which was proved NP-complete in the Cook–Levin theorem.^[16] From a given CNF formula, Karp forms a graph that has a vertex for every pair (v,c) , where v is a variable or its negation and c is a clause in the formula that contains v . Vertices are connected by an edge if they represent compatible variable assignments for different clauses: that is, there is an edge from (v,c) to (u,d) whenever $c \neq d$ and u and v are not each other's negations. If k denotes the number of clauses in the CNF formula, then the k -vertex cliques in this graph represent ways of assigning truth values to some of its variables in order to satisfy the formula; therefore, the formula is satisfiable if and only if a k -vertex clique exists.

Some NP-complete problems (such as the travelling salesman problem in planar graphs) may be solved in time that is exponential in a sublinear function of the input size parameter n . However, as Impagliazzo, Paturi & Zane (2001) describe, it is unlikely that such bounds exist for the clique problem in arbitrary graphs, as they would imply similarly subexponential bounds for many other standard NP-complete problems.



Circuit complexity

The computational difficulty of the clique problem has led it to be used to prove several lower bounds in circuit complexity. Because the existence of a clique of a given size is a monotone graph property (if a clique exists in a given graph, it will exist in any supergraph) there must exist a monotone circuit, using only and gates and or gates, to solve the clique decision problem for a given fixed clique size. However, the size of these circuits can be proven to be a super-polynomial function of the number of vertices and the clique size, exponential in the cube root of the number of vertices.^[17] Even if a small number of NOT gates are allowed, the complexity remains superpolynomial. Additionally, the depth of a monotone circuit for the clique problem using gates of bounded fan-in must be at least a polynomial in the clique size.^[18]

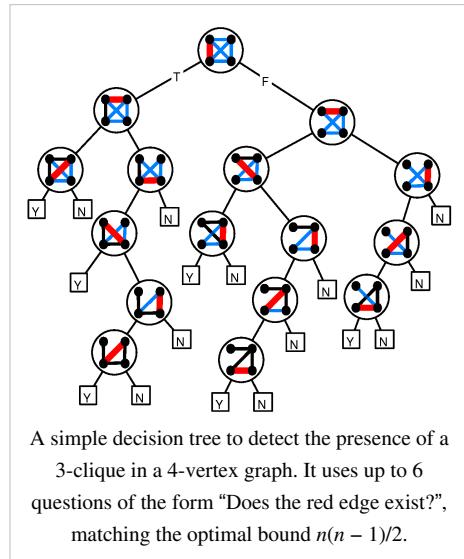


Decision tree complexity

The (deterministic) decision tree complexity of determining a graph property is the number of questions of the form "Is there an edge between vertex u and vertex v ?" that have to be answered in the worst case to determine whether a graph has a particular property. That is, it is the minimum height of a boolean decision tree for the problem. Since there are at most $n(n - 1)/2$ possible questions to be asked, any graph property can be determined with $n(n - 1)/2$ questions. It is also possible to define random and quantum decision tree complexity of a property, the expected number of questions (for a worst case input) that a randomized or quantum algorithm needs to have answered in order to correctly determine whether the given graph has the property.

Because the property of containing a clique is a monotone property (adding an edge can only cause more cliques to exist within the graph, not fewer), it is covered by the Aanderaa–Karp–Rosenberg conjecture, which states that the deterministic decision tree complexity of determining any non-trivial monotone graph property is exactly $n(n - 1)/2$. For deterministic decision trees, the property of containing a k -clique ($2 \leq k \leq n$) was shown to have decision tree complexity exactly $n(n - 1)/2$ by Bollobás (1976). Deterministic decision trees also require exponential size to detect cliques, or large polynomial size to detect cliques of bounded size.

The Aanderaa–Karp–Rosenberg conjecture also states that the randomized decision tree complexity of non-trivial monotone functions is $\Theta(n^2)$. The conjecture is resolved for the property of containing a k -clique ($2 \leq k \leq n$), since it is known to have randomized decision tree complexity $\Theta(n^2)$.^[19] For quantum decision trees, the best known lower bound is $\Omega(n)$, but no matching algorithm is known for the case of $k \geq 3$.^[20]



Fixed-parameter intractability

Parameterized complexity is the complexity-theoretic study of problems that are naturally equipped with a small integer parameter k , and for which the problem becomes more difficult as k increases, such as finding k -cliques in graphs. A problem is said to be fixed-parameter tractable if there is an algorithm for solving it on inputs of size n in time $f(k) n^{O(1)}$; that is, if it can be solved in polynomial time for any fixed value of k and moreover if the exponent of the polynomial does not depend on k .

For the clique problem, the brute force search algorithm has running time $O(n^k k^2)$, and although it can be improved by fast matrix multiplication the running time still has an exponent that is linear in k . Thus, although the running time of known algorithms for the clique problem is polynomial for any fixed k , these algorithms do not suffice for fixed-parameter tractability. Downey & Fellows (1995) defined a hierarchy of parametrized problems, the W hierarchy, that they conjectured did not have fixed-parameter tractable algorithms; they proved that independent set (or, equivalently, clique) is hard for the first level of this hierarchy, W[1]. Thus, according to their conjecture, clique is not fixed-parameter tractable. Moreover, this result provides the basis for proofs of W[1]-hardness of many other problems, and thus serves as an analogue of the Cook–Levin theorem for parameterized complexity.

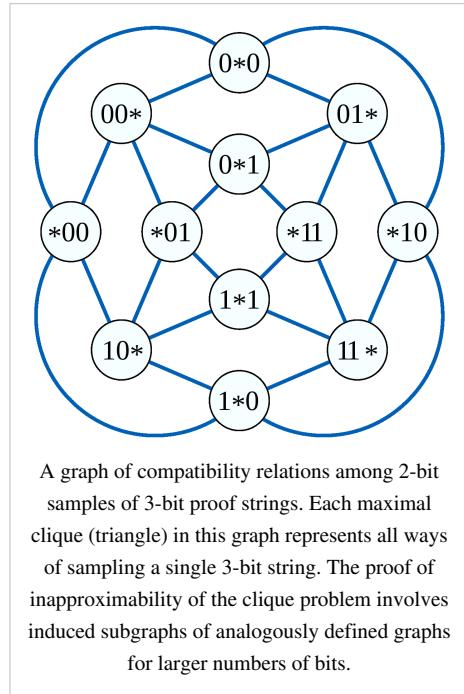
Chen et al. (2006) showed that the clique problem cannot be solved in time $n^{o(k)}$ unless the exponential time hypothesis fails.

Although the problems of listing maximal cliques or finding maximum cliques are unlikely to be fixed-parameter tractable with the parameter k , they may be fixed-parameter tractable for other parameters of instance complexity. For instance, both problems are known to be fixed-parameter tractable when parametrized by the degeneracy of the input graph.

Hardness of approximation

The computational complexity of approximating the clique problem has been studied for a long time; for instance, Garey & Johnson (1978) observed that, because of the fact that the clique number takes on small integer values and is NP-hard to compute, it cannot have a fully polynomial-time approximation scheme. However, little more was known until the early 1990s, when several authors began to make connections between the approximation of maximum cliques and probabilistically checkable proofs, and used these connections to prove hardness of approximation results for the maximum clique problem.^[21] After many improvements to these results it is now known that, unless P = NP, there can be no polynomial time algorithm that approximates the maximum clique to within a factor better than $O(n^{1-\varepsilon})$, for any $\varepsilon > 0$.^[22]

The rough idea of these inapproximability results^[23] is to form a graph that represents a probabilistically checkable proof system for an NP-complete problem such as Satisfiability. A proof system of this type is defined by a family of proof strings (sequences of bits) and proof checkers: algorithms that, after a polynomial amount of computation over a given Satisfiability instance, examine a small number of randomly chosen bits of the proof string and on the basis of that examination either declare it to be a valid proof or declare it to be invalid. False negatives are not allowed: a valid proof must always be declared to be valid, but an invalid proof may be declared to be valid as long as the probability that a checker makes a mistake of this type is low. To transform a probabilistically checkable proof system into a clique problem, one forms a graph in which the vertices represent all the possible ways that a



proof checker could read a sequence of proof string bits and end up accepting the proof. Two vertices are connected by an edge whenever the two proof checker runs that they describe agree on the values of the proof string bits that they both examine. The maximal cliques in this graph consist of the accepting proof checker runs for a single proof string, and one of these cliques is large if and only if there exists a proof string that many proof checkers accept. If the original Satisfiability instance is satisfiable, there will be a large clique defined by a valid proof string for that instance, but if the original instance is not satisfiable, then all proof strings are invalid, any proof string has only a small number of checkers that mistakenly accept it, and all cliques are small. Therefore, if one could distinguish in polynomial time between graphs that have large cliques and graphs in which all cliques are small, one could use this ability to distinguish the graphs generated from satisfiable and unsatisfiable instances of the Satisfiability problem, not possible unless $P = NP$. An accurate polynomial-time approximation to the clique problem would allow these two sets of graphs to be distinguished from each other, and is therefore also impossible.

Free software for searching maximum clique

Name	License	API language	Brief info
NetworkX	BSD	Python	approximate solution, see the routine <code>max_clique</code> [24]
maxClique	CRAPL	Java	exact algorithms and DIMACS instances [25]
OpenOpt	BSD	Python	exact and approximate solutions, possibility to specify nodes that have to be included / excluded; see MCP [26] class for more details and examples

Notes

- [1] For more details and references, see clique (graph theory).
- [2] Complete subgraphs make an early appearance in the mathematical literature in the graph-theoretic reformulation of Ramsey theory by .
- [3] E.g., see .
- [4] provide an algorithm with $O(m^{3/2})$ running time that finds a triangle if one exists but does not list all triangles; list all triangles in time $O(m^{3/2})$.
- [5] ;;;.
- [6] ;.
- [7] ;;;;;;.
- [8] ;.
- [9] ;.
- [10] . Although the title refers to maximal cliques, the problem this paper solves is actually the maximum clique problem.
- [11] DIMACS challenge graphs for the clique problem (<ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/>), accessed 2009-12-17.
- [12] , p. 159. provide an alternative quadratic-time algorithm for maximum cliques in comparability graphs, a broader class of perfect graphs that includes the permutation graphs as a special case.
- [13] ;, p. 247.
- [14] ;;.
- [15] Adapted from
- [16] gives essentially the same reduction, from 3-SAT instead of Satisfiability, to show that subgraph isomorphism is NP-complete.
- [17] . For earlier and weaker bounds on monotone circuits for the clique problem, see and .
- [18] used communication complexity to prove this result.
- [19] For instance, this follows from .
- [20] ;.
- [21] ;;.
- [22] showed inapproximability for this ratio using a stronger complexity theoretic assumption, the inequality of NP and ZPP; described more precisely the inapproximation ratio, and derandomized the construction weakening its assumption to $P \neq NP$.
- [23] This reduction is originally due to and used in all subsequent inapproximability proofs; the proofs differ in the strengths and details of the probabilistically checkable proof systems that they rely on.
- [24] http://networkx.lanl.gov/preview/reference/generated/networkx.algorithms.approximation.cliques.max_clique.html

- [25] <http://www.dcs.gla.ac.uk/~pat/maxClique/distribution/>
 [26] <http://openopt.org/MCP>

References

- Abello, J.; Pardalos, P. M.; Resende, M. G. C. (1999), "On maximum clique problems in very large graphs" (<http://www2.research.att.com/~mgcr/abstracts/vlclq.html>), in Abello, J.; Vitter, J., *External Memory Algorithms*, DIMACS Series on Discrete Mathematics and Theoretical Computer Science **50**, American Mathematical Society, pp. 119–130, ISBN 0-8218-1184-3.
- Alon, N.; Boppana, R. (1987), "The monotone circuit complexity of boolean functions", *Combinatorica* **7** (1): 1–22, doi: 10.1007/BF02579196 (<http://dx.doi.org/10.1007/BF02579196>).
- Alon, N.; Krivelevich, M.; Sudakov, B. (1998), "Finding a large hidden clique in a random graph", *Random Structures & Algorithms* **13** (3–4): 457–466, doi: 10.1002/(SICI)1098-2418(199810/12)13:3/4<457::AID-RSA14>3.0.CO;2-W ([http://dx.doi.org/10.1002/\(SICI\)1098-2418\(199810/12\)13:3/4<457::AID-RSA14>3.0.CO;2-W](http://dx.doi.org/10.1002/(SICI)1098-2418(199810/12)13:3/4<457::AID-RSA14>3.0.CO;2-W)).
- Alon, N.; Yuster, R.; Zwick, U. (1994), "Finding and counting given length cycles", *Proceedings of the 2nd European Symposium on Algorithms, Utrecht, The Netherlands*, pp. 354–364.
- Amano, K.; Maruoka, A. (1998), "A superpolynomial lower bound for a circuit computing the clique function with at most $(1/6)\log \log n$ negation gates" (<http://www.springerlink.com/content/m64ju7clmqhqmv9g/>), *Proc. Symp. Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **1450**, Springer-Verlag, pp. 399–408.
- Arora, Sanjeev; Lund, Carsten; Motwani, Rajeev; Sudan, Madhu; Szegedy, Mario (1998), "Proof verification and the hardness of approximation problems", *Journal of the ACM* **45** (3): 501–555, doi: 10.1145/278298.278306 (<http://dx.doi.org/10.1145/278298.278306>), ECCC TR98-008 (<http://eccc.uni-trier.de/report/1998/008/>). Originally presented at the 1992 Symposium on Foundations of Computer Science, doi: 10.1109/SFCS.1992.267823 (<http://dx.doi.org/10.1109/SFCS.1992.267823>).
- Arora, S.; Safra, S. (1998), "Probabilistic checking of proofs: A new characterization of NP", *Journal of the ACM* **45** (1): 70–122, doi: 10.1145/273865.273901 (<http://dx.doi.org/10.1145/273865.273901>). Originally presented at the 1992 Symposium on Foundations of Computer Science, doi: 10.1109/SFCS.1992.267824 (<http://dx.doi.org/10.1109/SFCS.1992.267824>).
- Balas, E.; Yu, C. S. (1986), "Finding a maximum clique in an arbitrary graph", *SIAM Journal on Computing* **15** (4): 1054–1068, doi: 10.1137/0215075 (<http://dx.doi.org/10.1137/0215075>).
- Battiti, R.; Protasi, M. (2001), "Reactive local search for the maximum clique problem", *Algorithmica* **29** (4): 610–637, doi: 10.1007/s004530010074 (<http://dx.doi.org/10.1007/s004530010074>).
- Bollobás, Béla (1976), "Complete subgraphs are elusive", *Journal of Combinatorial Theory, Series B* **21** (1): 1–7, doi: 10.1016/0095-8956(76)90021-6 ([http://dx.doi.org/10.1016/0095-8956\(76\)90021-6](http://dx.doi.org/10.1016/0095-8956(76)90021-6)), ISSN 0095-8956 (<http://www.worldcat.org/issn/0095-8956>).
- Bomze, I. M.; Budinich, M.; Pardalos, P. M.; Pelillo, M. (1999), "The maximum clique problem", *Handbook of Combinatorial Optimization* **4**, Kluwer Academic Publishers, pp. 1–74, CiteSeerX: 10.1.1.48.4074 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.4074>).
- Boppana, R.; Halldórsson, M. M. (1992), "Approximating maximum independent sets by excluding subgraphs", *BIT* **32** (2): 180–196, doi: 10.1007/BF01994876 (<http://dx.doi.org/10.1007/BF01994876>).
- Bron, C.; Kerbosch, J. (1973), "Algorithm 457: finding all cliques of an undirected graph", *Communications of the ACM* **16** (9): 575–577, doi: 10.1145/362342.362367 (<http://dx.doi.org/10.1145/362342.362367>).
- Carraghan, R.; Pardalos, P. M. (1990), "An exact algorithm for the maximum clique problem" (http://www.inf.ufpr.br/renato/download/An_Exact_Algorithm_for_the_Maximum_Clique_Problem.pdf), *Operations Research Letters* **9** (6): 375–382, doi: 10.1016/0167-6377(90)90057-C ([http://dx.doi.org/10.1016/0167-6377\(90\)90057-C](http://dx.doi.org/10.1016/0167-6377(90)90057-C)).

- Cazals, F.; Karande, C. (2008), "A note on the problem of reporting maximal cliques" (<ftp://ftp-sop.inria.fr/geometrica/fcazals/papers/ncliques.pdf>), *Theoretical Computer Science* **407** (1): 564–568, doi: [10.1016/j.tcs.2008.05.010](https://doi.org/10.1016/j.tcs.2008.05.010) (<http://dx.doi.org/10.1016/j.tcs.2008.05.010>).
- Chen, Jianer; Huang, Xiuzhen; Kanj, Iyad A.; Xia, Ge (2006), "Strong computational lower bounds via parameterized complexity", *J. Comput. Syst. Sci.* **72** (8): 1346–1367, doi: [10.1016/j.jcss.2006.04.007](https://doi.org/10.1016/j.jcss.2006.04.007) (<http://dx.doi.org/10.1016/j.jcss.2006.04.007>)
- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM Journal on Computing* **14** (1): 210–223, doi: [10.1137/0214017](https://doi.org/10.1137/0214017) (<http://dx.doi.org/10.1137/0214017>).
- Childs, A. M.; Farhi, E.; Goldstone, J.; Gutmann, S. (2002), "Finding cliques by quantum adiabatic evolution", *Quantum Information and Computation* **2** (3): 181–191, arXiv: quant-ph/0012104 (<http://arxiv.org/abs/quant-ph/0012104>).
- Childs, A. M.; Eisenberg, J. M. (2005), "Quantum algorithms for subset finding", *Quantum Information and Computation* **5** (7): 593–604, arXiv: quant-ph/0311038 (<http://arxiv.org/abs/quant-ph/0311038>).
- Clark, Brent N.; Colbourn, Charles J.; Johnson, David S. (1990), "Unit disk graphs", *Discrete Mathematics* **86** (1–3): 165–177, doi: [10.1016/0012-365X\(90\)90358-O](https://doi.org/10.1016/0012-365X(90)90358-O) ([http://dx.doi.org/10.1016/0012-365X\(90\)90358-O](http://dx.doi.org/10.1016/0012-365X(90)90358-O))
- Cook, S. A. (1971), "The complexity of theorem-proving procedures" (<http://4mhz.de/cook.html>), *Proc. 3rd ACM Symposium on Theory of Computing*, pp. 151–158, doi: [10.1145/800157.805047](https://doi.org/10.1145/800157.805047) (<http://dx.doi.org/10.1145/800157.805047>).
- Downey, R. G.; Fellows, M. R. (1995), "Fixed-parameter tractability and completeness. II. On completeness for W[1]", *Theoretical Computer Science* **141** (1–2): 109–131, doi: [10.1016/0304-3975\(94\)00097-3](https://doi.org/10.1016/0304-3975(94)00097-3) ([http://dx.doi.org/10.1016/0304-3975\(94\)00097-3](http://dx.doi.org/10.1016/0304-3975(94)00097-3)).
- Downey, R. G.; Fellows, M. R. (1999), *Parameterized complexity*, Springer-Verlag, ISBN 0-387-94883-X.
- Eisenbrand, F.; Grandoni, F. (2004), "On the complexity of fixed parameter clique and dominating set", *Theoretical Computer Science* **326** (1–3): 57–67, doi: [10.1016/j.tcs.2004.05.009](https://doi.org/10.1016/j.tcs.2004.05.009) (<http://dx.doi.org/10.1016/j.tcs.2004.05.009>).
- Eppstein, David; Löffler, Maarten; Strash, Darren (2010), "Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time", in Cheong, Otfried; Chwa, Kyung-Yong; Park, Kunsoo, *21st International Symposium on Algorithms and Computation (ISAAC 2010)*, Jeju, Korea, Lecture Notes in Computer Science **6506**, Springer-Verlag, pp. 403–414, arXiv: 1006.5440 (<http://arxiv.org/abs/1006.5440>), doi: [10.1007/978-3-642-17517-6_36](https://doi.org/10.1007/978-3-642-17517-6_36) (http://dx.doi.org/10.1007/978-3-642-17517-6_36), ISBN 978-3-642-17516-9.
- Eppstein, David; Strash, Darren (2011), "Listing all maximal cliques in large sparse real-world graphs", *10th International Symposium on Experimental Algorithms*, arXiv: 1103.0318 (<http://arxiv.org/abs/1103.0318>).
- Erdős, Paul; Szekeres, George (1935), "A combinatorial problem in geometry" (http://www.renyi.hu/~p_erdos/1935-01.pdf), *Compositio Mathematica* **2**: 463–470.
- Even, S.; Pnueli, A.; Lempel, A. (1972), "Permutation graphs and transitive graphs", *Journal of the ACM* **19** (3): 400–410, doi: [10.1145/321707.321710](https://doi.org/10.1145/321707.321710) (<http://dx.doi.org/10.1145/321707.321710>).
- Fahle, T. (2002), "Simple and Fast: Improving a Branch-And-Bound Algorithm for Maximum Clique", *Proc. 10th European Symposium on Algorithms*, Lecture Notes in Computer Science **2461**, Springer-Verlag, pp. 47–86, doi: [10.1007/3-540-45749-6_44](https://doi.org/10.1007/3-540-45749-6_44) (http://dx.doi.org/10.1007/3-540-45749-6_44), ISBN 978-3-540-44180-9.
- Feige, U. (2004), "Approximating maximum clique by removing subgraphs", *SIAM Journal on Discrete Mathematics* **18** (2): 219–225, doi: [10.1137/S089548010240415X](https://doi.org/10.1137/S089548010240415X) (<http://dx.doi.org/10.1137/S089548010240415X>).
- Feige, U.; Goldwasser, S.; Lovász, L.; Safra, S.; Szegedy, M. (1991), "Approximating clique is almost NP-complete", *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pp. 2–12, doi: [10.1109/SFCS.1991.185341](https://doi.org/10.1109/SFCS.1991.185341) (<http://dx.doi.org/10.1109/SFCS.1991.185341>), ISBN 0-8186-2445-0.

- Feige, U.; Krauthgamer, R. (2000), "Finding and certifying a large hidden clique in a semirandom graph", *Random Structures and Algorithms* **16** (2): 195–208, doi: 10.1002/(SICI)1098-2418(200003)16:2<195::AID-RSA5>3.0.CO;2-A ([http://dx.doi.org/10.1002/\(SICI\)1098-2418\(200003\)16:2<195::AID-RSA5>3.0.CO;2-A](http://dx.doi.org/10.1002/(SICI)1098-2418(200003)16:2<195::AID-RSA5>3.0.CO;2-A)).
- Garey, M. R.; Johnson, D. S. (1978), ""Strong" NP-completeness results: motivation, examples and implications", *Journal of the ACM* **25** (3): 499–508, doi: 10.1145/322077.322090 (<http://dx.doi.org/10.1145/322077.322090>).
- Gavril, F. (1973), "Algorithms for a maximum clique and a maximum independent set of a circle graph", *Networks* **3** (3): 261–273, doi: 10.1002/net.3230030305 (<http://dx.doi.org/10.1002/net.3230030305>).
- Goldmann, M.; Håstad, J. (1992), "A simple lower bound for monotone clique using a communication game", *Information Processing Letters* **41** (4): 221–226, doi: 10.1016/0020-0190(92)90184-W ([http://dx.doi.org/10.1016/0020-0190\(92\)90184-W](http://dx.doi.org/10.1016/0020-0190(92)90184-W)).
- Golumbic, M. C. (1980), *Algorithmic Graph Theory and Perfect Graphs*, Computer Science and Applied Mathematics, Academic Press, ISBN 0-444-51530-5.
- Gröger, Hans Dietmar (1992), "On the randomized complexity of monotone graph properties" (http://www.inf.u-szeged.hu/actacybernetica/edb/vol10n3/pdf/Groger_1992_ActaCybernetica.pdf), *Acta Cybernetica* **10** (3): 119–127, retrieved 2009-10-02
- Grossi, A.; Locatelli, M.; Della Croce, F. (2004), "Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem", *Journal of Heuristics* **10** (2): 135–152, doi: 10.1023/B:HEUR.0000026264.51747.7f (<http://dx.doi.org/10.1023/B:HEUR.0000026264.51747.7f>).
- Grötschel, M.; Lovász, L.; Schrijver, A. (1988), "9.4 Coloring Perfect Graphs", *Geometric Algorithms and Combinatorial Optimization*, Algorithms and Combinatorics **2**, Springer–Verlag, pp. 296–298, ISBN 0-387-13624-X.
- Gutin, G. (2004), "5.3 Independent sets and cliques", in Gross, J. L.; Yellen, J., *Handbook of graph theory*, Discrete Mathematics & Its Applications, CRC Press, pp. 389–402, ISBN 978-1-58488-090-5.
- Halldórsson, M. M. (2000), "Approximations of Weighted Independent Set and Hereditary Subset Problems" (<http://jgaa.info/accepted/00/Halldorsson00.4.1.pdf>), *Journal of Graph Algorithms and Applications* **4** (1): 1–16, doi: 10.7155/jgaa.00020 (<http://dx.doi.org/10.7155/jgaa.00020>).
- Harary, F.; Ross, I. C. (1957), "A procedure for clique detection using the group matrix", *Sociometry* (American Sociological Association) **20** (3): 205–215, doi: 10.2307/2785673 (<http://dx.doi.org/10.2307/2785673>), JSTOR 2785673 (<http://www.jstor.org/stable/2785673>), MR 0110590 (<http://www.ams.org/mathscinet-getitem?mr=0110590>).
- Håstad, J. (1999), "Clique is hard to approximate within $n^{1 - \varepsilon}$ ", *Acta Mathematica* **182** (1): 105–142, doi: 10.1007/BF02392825 (<http://dx.doi.org/10.1007/BF02392825>).
- Impagliazzo, R.; Paturi, R.; Zane, F. (2001), "Which problems have strongly exponential complexity?", *Journal of Computer and System Sciences* **63** (4): 512–530, doi: 10.1006/jcss.2001.1774 (<http://dx.doi.org/10.1006/jcss.2001.1774>).
- Itai, A.; Rodeh, M. (1978), "Finding a minimum circuit in a graph", *SIAM Journal on Computing* **7** (4): 413–423, doi: 10.1137/0207033 (<http://dx.doi.org/10.1137/0207033>).
- Jerrum, M. (1992), "Large cliques elude the Metropolis process", *Random Structures and Algorithms* **3** (4): 347–359, doi: 10.1002/rsa.3240030402 (<http://dx.doi.org/10.1002/rsa.3240030402>).
- Jian, T (1986), "An $O(2^{0.304n})$ Algorithm for Solving Maximum Independent Set Problem", *IEEE Transactions on Computers* (IEEE Computer Society) **35** (9): 847–851, doi: 10.1109/TC.1986.1676847 (<http://dx.doi.org/10.1109/TC.1986.1676847>), ISSN 0018-9340 (<http://www.worldcat.org/issn/0018-9340>).
- Johnson, D. S.; Trick, M. A., eds. (1996), *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, October 11–13, 1993* (<http://dimacs.rutgers.edu/Volumes/Vol26.html>), DIMACS Series in Discrete Mathematics and Theoretical Computer Science **26**, American Mathematical Society,

- ISBN 0-8218-6609-5.
- Johnson, D. S.; Yannakakis, M. (1988), "On generating all maximal independent sets", *Information Processing Letters* **27** (3): 119–123, doi: 10.1016/0020-0190(88)90065-8 ([http://dx.doi.org/10.1016/0020-0190\(88\)90065-8](http://dx.doi.org/10.1016/0020-0190(88)90065-8)).
 - Karp, Richard M. (1972), "Reducibility among combinatorial problems" (<http://www.cs.berkeley.edu/~luca/cs172/karp.pdf>), in Miller, R. E.; Thatcher, J. W., *Complexity of Computer Computations*, New York: Plenum, pp. 85–103.
 - Karp, Richard M. (1976), "Probabilistic analysis of some combinatorial search problems", in Traub, J. F., *Algorithms and Complexity: New Directions and Recent Results*, New York: Academic Press, pp. 1–19.
 - Katayama, K.; Hamamoto, A.; Narihisa, H. (2005), "An effective local search for the maximum clique problem", *Information Processing Letters* **95** (5): 503–511, doi: 10.1016/j IPL.2005.05.010 (<http://dx.doi.org/10.1016/j.IPL.2005.05.010>).
 - Khot, S. (2001), "Improved inapproximability results for MaxClique, chromatic number and approximate graph coloring", *Proc. 42nd IEEE Symp. Foundations of Computer Science*, pp. 600–609, doi: 10.1109/SFCS.2001.959936 (<http://dx.doi.org/10.1109/SFCS.2001.959936>), ISBN 0-7695-1116-3.
 - Kloks, T.; Kratsch, D.; Müller, H. (2000), "Finding and counting small induced subgraphs efficiently", *Information Processing Letters* **74** (3–4): 115–121, doi: 10.1016/S0020-0190(00)00047-8 ([http://dx.doi.org/10.1016/S0020-0190\(00\)00047-8](http://dx.doi.org/10.1016/S0020-0190(00)00047-8)).
 - Konc, J.; Janežič, D. (2007), "An improved branch and bound algorithm for the maximum clique problem" (<http://www.sicmm.org/~konc/articles/match2007.pdf>), *MATCH Communications in Mathematical and in Computer Chemistry* **58** (3): 569–590. Source code (<http://www.sicmm.org/~konc/maxclique>)
 - Lipton, R. J.; Tarjan, R. E. (1980), "Applications of a planar separator theorem", *SIAM Journal on Computing* **9** (3): 615–627, doi: 10.1137/0209046 (<http://dx.doi.org/10.1137/0209046>).
 - Luce, R. Duncan; Perry, Albert D. (1949), "A method of matrix analysis of group structure", *Psychometrika* **14** (2): 95–116, doi: 10.1007/BF02289146 (<http://dx.doi.org/10.1007/BF02289146>), PMID 18152948 (<http://www.ncbi.nlm.nih.gov/pubmed/18152948>).
 - Magniez, Frédéric; Santha, Miklos; Szegedy, Mario (2007), "Quantum algorithms for the triangle problem", *SIAM Journal on Computing* **37** (2): 413–424, arXiv: quant-ph/0310134 (<http://arxiv.org/abs/quant-ph/0310134>), doi: 10.1137/050643684 (<http://dx.doi.org/10.1137/050643684>).
 - Makino, K.; Uno, T. (2004), "New algorithms for enumerating all maximal cliques" (<http://www.springerlink.com/content/p9qbl6y1v5t3xc1w/>), *Algorithm Theory: SWAT 2004*, Lecture Notes in Computer Science **3111**, Springer-Verlag, pp. 260–272.
 - Moon, J. W.; Moser, L. (1965), "On cliques in graphs", *Israel Journal of Mathematics* **3**: 23–28, doi: 10.1007/BF02760024 (<http://dx.doi.org/10.1007/BF02760024>), MR 0182577 (<http://www.ams.org/mathscinet-getitem?mr=0182577>).
 - Nešetřil, J.; Poljak, S. (1985), "On the complexity of the subgraph problem", *Commentationes Mathematicae Universitatis Carolinae* **26** (2): 415–419.
 - Östergård, P. R. J. (2002), "A fast algorithm for the maximum clique problem", *Discrete Applied Mathematics* **120** (1–3): 197–207, doi: 10.1016/S0166-218X(01)00290-6 ([http://dx.doi.org/10.1016/S0166-218X\(01\)00290-6](http://dx.doi.org/10.1016/S0166-218X(01)00290-6)).
 - Ouyang, Q.; Kaplan, P. D.; Liu, S.; Libchaber, A. (1997), "DNA solution of the maximal clique problem", *Science* **278** (5337): 446–449, doi: 10.1126/science.278.5337.446 (<http://dx.doi.org/10.1126/science.278.5337.446>), PMID 9334300 (<http://www.ncbi.nlm.nih.gov/pubmed/9334300>).
 - Pardalos, P. M.; Rogers, G. P. (1992), "A branch and bound algorithm for the maximum clique problem", *Computers & Operations Research* **19** (5): 363–375, doi: 10.1016/0305-0548(92)90067-F ([http://dx.doi.org/10.1016/0305-0548\(92\)90067-F](http://dx.doi.org/10.1016/0305-0548(92)90067-F)).

- Razborov, A. A. (1985), "Lower bounds for the monotone complexity of some Boolean functions", *Proceedings of the USSR Academy of Sciences* (in Russian) **281**: 798–801. English translation in *Sov. Math. Dokl.* **31** (1985): 354–357.
- Régin, J.-C. (2003), "Using constraint programming to solve the maximum clique problem" (<http://www.springerlink.com/content/8p1980dfmrt3agyp/>), *Proc. 9th Int. Conf. Principles and Practice of Constraint Programming – CP 2003*, Lecture Notes in Computer Science **2833**, Springer-Verlag, pp. 634–648.
- Robson, J. M. (1986), "Algorithms for maximum independent sets", *Journal of Algorithms* **7** (3): 425–440, doi: [10.1016/0196-6774\(86\)90032-5](https://doi.org/10.1016/0196-6774(86)90032-5) ([http://dx.doi.org/10.1016/0196-6774\(86\)90032-5](http://dx.doi.org/10.1016/0196-6774(86)90032-5)).
- Robson, J. M. (2001), *Finding a maximum independent set in time $O(2^{n/4})$* (<http://www.labri.fr/perso/robson/mis/techrep.html>).
- Rosgen, B; Stewart, L (2007), "Complexity results on graphs with few cliques" (<http://www.dmtcs.org/dmtcs-ojs/index.php/dmtcs/article/view/707/1817>), *Discrete Mathematics and Theoretical Computer Science* **9** (1): 127–136.
- Sipser, M. (1996), *Introduction to the Theory of Computation*, International Thompson Publishing, ISBN 0-534-94728-X.
- Tarjan, R. E.; Trojanowski, A. E. (1977), "Finding a maximum independent set" (<ftp://db.stanford.edu/pub/cstr.old/reports/cs/tr/76/550/CS-TR-76-550.pdf>), *SIAM Journal on Computing* **6** (3): 537–546, doi: [10.1137/0206038](https://doi.org/10.1137/0206038) (<http://dx.doi.org/10.1137/0206038>).
- Tomita, E.; Kameda, T. (2007), "An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments", *Journal of Global Optimization* **37** (1): 95–111, doi: [10.1007/s10898-006-9039-7](https://doi.org/10.1007/s10898-006-9039-7) (<http://dx.doi.org/10.1007/s10898-006-9039-7>).
- Tomita, E.; Seki, T. (2003), "An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique", *Discrete Mathematics and Theoretical Computer Science*, Lecture Notes in Computer Science **2731**, Springer-Verlag, pp. 278–289, doi: [10.1007/3-540-45066-1_22](https://doi.org/10.1007/3-540-45066-1_22) (http://dx.doi.org/10.1007/3-540-45066-1_22), ISBN 978-3-540-40505-4.
- Tomita, E.; Tanaka, A.; Takahashi, H. (2006), "The worst-case time complexity for generating all maximal cliques and computational experiments", *Theoretical Computer Science* **363** (1): 28–42, doi: [10.1016/j.tcs.2006.06.015](https://doi.org/10.1016/j.tcs.2006.06.015) (<http://dx.doi.org/10.1016/j.tcs.2006.06.015>).
- Tsukiyama, S.; Ide, M.; Ariyoshi, I.; Shirakawa, I. (1977), "A new algorithm for generating all the maximal independent sets", *SIAM Journal on Computing* **6** (3): 505–517, doi: [10.1137/0206036](https://doi.org/10.1137/0206036) (<http://dx.doi.org/10.1137/0206036>).
- Valiant, L. G. (1983), "Exponential lower bounds for restricted monotone circuits", *Proc. 15th ACM Symposium on Theory of Computing*, pp. 110–117, doi: [10.1145/800061.808739](https://doi.org/10.1145/800061.808739) (<http://dx.doi.org/10.1145/800061.808739>), ISBN 0-89791-099-0.
- Vassilevska, V.; Williams, R. (2009), "Finding, minimizing, and counting weighted subgraphs", *Proc. 41st ACM Symposium on Theory of Computing*, pp. 455–464, doi: [10.1145/1536414.1536477](https://doi.org/10.1145/1536414.1536477) (<http://dx.doi.org/10.1145/1536414.1536477>), ISBN 978-1-60558-506-2.
- Wegener, I. (1988), "On the complexity of branching programs and decision trees for clique functions", *Journal of the ACM* **35** (2): 461–472, doi: [10.1145/42282.46161](https://doi.org/10.1145/42282.46161) (<http://dx.doi.org/10.1145/42282.46161>).
- Yuster, R. (2006), "Finding and counting cliques and independent sets in r -uniform hypergraphs", *Information Processing Letters* **99** (4): 130–134, doi: [10.1016/j.ipl.2006.04.005](https://doi.org/10.1016/j.ipl.2006.04.005) (<http://dx.doi.org/10.1016/j.ipl.2006.04.005>).
- Zuckerman, D. (2006), "Linear degree extractors and the inapproximability of max clique and chromatic number", *Proc. 38th ACM Symp. Theory of Computing*, pp. 681–690, doi: [10.1145/1132516.1132612](https://doi.org/10.1145/1132516.1132612) (<http://dx.doi.org/10.1145/1132516.1132612>), ISBN 1-59593-134-1, ECCC TR05-100 (<http://eccc.uni-trier.de/report/2005/100/>).

Bron–Kerbosch algorithm for listing all maximal cliques

In computer science, the **Bron–Kerbosch algorithm** is an algorithm for finding maximal cliques in an undirected graph. That is, it lists all subsets of vertices with the two properties that each pair of vertices in one of the listed subsets is connected by an edge, and no listed subset can have any additional vertices added to it while preserving its complete connectivity. The Bron–Kerbosch algorithm was designed by Dutch scientists Joep Kerbosch and Coenraad Bron, who published its description in 1973. Although other algorithms for solving the clique problem have running times that are, in theory, better on inputs that have few maximal independent sets, the Bron–Kerbosch algorithm and subsequent improvements to it are frequently reported as being more efficient in practice than the alternatives. It is well-known and widely used in application areas of graph algorithms such as computational chemistry.

A contemporaneous algorithm of Akkoyunlu (1973), although presented in different terms, can be viewed as being the same as the Bron–Kerbosch algorithm, as it generates the same recursive search tree.

Without pivoting

The basic form of the Bron–Kerbosch algorithm is a recursive backtracking algorithm that searches for all maximal cliques in a given graph G . More generally, given three sets R , P , and X , it finds the maximal cliques that include all of the vertices in R , some of the vertices in P , and none of the vertices in X . In each call to the algorithm, P and X are disjoint sets whose union consists of those vertices that form cliques when added to R . In other words, $P \cup X$ is the set of vertices which are joined to every element of R . When P and X are both empty there are no further elements that can be added to R , so R is a maximal clique and the algorithm outputs R .

The recursion is initiated by setting R and X to be the empty set and P to be the vertex set of the graph. Within each recursive call, the algorithm considers the vertices in P in turn; if there are no such vertices, it either reports R as a maximal clique (if X is empty), or backtracks. For each vertex v chosen from P , it makes a recursive call in which v is added to R and in which P and X are restricted to the neighbor set $N(v)$ of v , which finds and reports all clique extensions of R that contain v . Then, it moves v from P to X to exclude it from consideration in future cliques and continues with the next vertex in P .

That is, in pseudocode, the algorithm performs the following steps:

```
BronKerbosch1(R, P, X) :
    if P and X are both empty:
        report R as a maximal clique
    for each vertex v in P:
        BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

With pivoting

The basic form of the algorithm, described above, is inefficient in the case of graphs with many non-maximal cliques: it makes a recursive call for every clique, maximal or not. To save time and allow the algorithm to backtrack more quickly in branches of the search that contain no maximal cliques, Bron and Kerbosch introduced a variant of the algorithm involving a "pivot vertex" u , chosen from P (or more generally, as later investigators realized,^[1] from $P \cup X$). Any maximal clique must include either u or one of its non-neighbors, for otherwise the clique could be augmented by adding u to it. Therefore, only u and its non-neighbors need to be tested as the choices for the vertex v

that is added to R in each recursive call to the algorithm. In pseudocode:

```
BronKerbosch2(R, P, X) :
    if P and X are both empty:
        report R as a maximal clique
    choose a pivot vertex u in P ∪ X
    for each vertex v in P \ N(u):
        BronKerbosch2(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

If the pivot is chosen to minimize the number of recursive calls made by the algorithm, the savings in running time compared to the non-pivoting version of the algorithm can be significant.^[2]

With vertex ordering

An alternative method for improving the basic form of the Bron–Kerbosch algorithm involves forgoing pivoting at the outermost level of recursion, and instead choosing the ordering of the recursive calls carefully in order to minimize the sizes of the sets P of candidate vertices within each recursive call.

The degeneracy of a graph G is the smallest number d such that every subgraph of G has a vertex with degree d or less. Every graph has a *degeneracy ordering*, an ordering of the vertices such that each vertex has d or fewer neighbors that come later in the ordering; a degeneracy ordering may be found in linear time by repeatedly selecting the vertex of minimum degree among the remaining vertices. If the order of the vertices v that the Bron–Kerbosch algorithm loops through is a degeneracy ordering, then the set P of candidate vertices in each call (the neighbors of v that are later in the ordering) will be guaranteed to have size at most d . The set X of excluded vertices will consist of all earlier neighbors of v , and may be much larger than d . In recursive calls to the algorithm below the topmost level of the recursion, the pivoting version can still be used.

In pseudocode, the algorithm performs the following steps:

```
BronKerbosch3(G) :
    P = V(G)
    R = X = empty
    for each vertex v in a degeneracy ordering of G:
        BronKerbosch2(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

This variant of the algorithm can be proven to be efficient for graphs of small degeneracy, and experiments show that it also works well in practice for large sparse social networks and other real-world graphs.

Example

In the example graph shown, the algorithm is initially called with $R = \emptyset$, $P = \{1,2,3,4,5,6\}$, and $X = \emptyset$. The pivot u should be chosen as one of the degree-three vertices, to minimize the number of recursive calls; for instance, suppose that u is chosen to be vertex 2. Then there are three remaining vertices in $P \setminus N(u)$: vertices 2, 4, and 6.

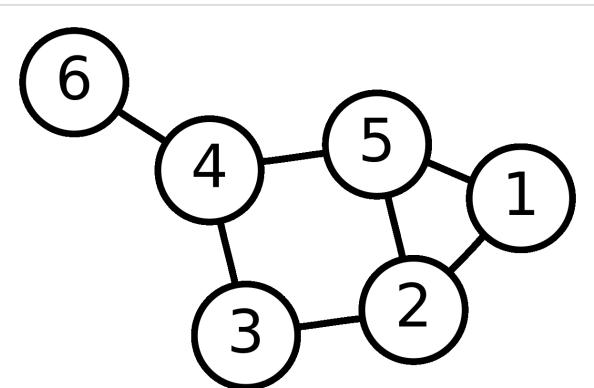
The iteration of the inner loop of the algorithm for $v = 2$ makes a recursive call to the algorithm with $R = \{2\}$, $P = \{1,3,5\}$, and $X = \emptyset$. Within this recursive call, one of 1 or 5 will be chosen as a pivot, and there will be two second-level recursive calls, one for vertex 3 and the other for whichever vertex was not chosen as pivot. These two calls will eventually report the two cliques $\{1,2,5\}$ and $\{2,3\}$. After returning from these recursive calls, vertex 2 is added to X and removed from P .

The iteration of the inner loop of the algorithm for $v = 4$ makes a recursive call to the algorithm with $R = \{4\}$, $P = \{3,5,6\}$, and $X = \emptyset$ (although vertex 2 belongs to the set X in the outer call to the algorithm, it is not a neighbor of v and is excluded from the subset of X passed to the recursive call). This recursive call will end up making three second-level recursive calls to the algorithm that report the three cliques $\{3,4\}$, $\{4,5\}$, and $\{4,6\}$. Then, vertex 4 is added to X and removed from P .

In the third and final iteration of the inner loop of the algorithm, for $v = 6$, there is a recursive call to the algorithm with $R = \{6\}$, $P = \emptyset$, and $X = \{4\}$. Because this recursive call has P empty and X non-empty, it immediately backtracks without reporting any more cliques, as there can be no maximal clique that includes vertex 6 and excludes vertex 4.

The call tree for the algorithm, therefore, looks like:

```
BronKerbosch2(∅, {1,2,3,4,5,6}, ∅)
  BronKerbosch2({2}, {1,3,5}, ∅)
    BronKerbosch2({2,3}, ∅, ∅): output {2, 3}
    BronKerbosch2({2,5}, {1}, ∅)
      BronKerbosch2({1,2,5}, ∅, ∅): output {1,2,5}
  BronKerbosch2({4}, {3,5,6}, ∅)
    BronKerbosch2({3,4}, ∅, ∅): output {3,4}
    BronKerbosch2({4,5}, ∅, ∅): output {4,5}
    BronKerbosch2({4,6}, ∅, ∅): output {4,6}
  BronKerbosch2({6}, ∅, {4}): no output
```



A graph with five maximal cliques: four edges and a triangle

The graph in the example has degeneracy two; one possible degeneracy ordering is 6,4,3,1,2,5. If the vertex-ordering version of the Bron–Kerbosch algorithm is applied to the vertices, in this order, the call tree looks like

```
BronKerbosch3(G)
  BronKerbosch2({6}, {4}, ∅)
    BronKerbosch2({6,4}, ∅, ∅): output {6,4}
  BronKerbosch2({4}, {3,5}, {6})
    BronKerbosch2({4,3}, ∅, ∅): output {4,3}
    BronKerbosch2({4,5}, ∅, ∅): output {4,5}
  BronKerbosch2({3}, {2}, {4})
```

```

BronKerbosch2({3,2}, ∅, ∅): output {3,2}
BronKerbosch2({1}, {2,5}, ∅)
    BronKerbosch2({1,2}, {5}, ∅)
        BronKerbosch2({1,2,5}, ∅, ∅): output {1,2,5}
BronKerbosch2({2}, {5}, {1,3}): no output
BronKerbosch2({5}, ∅, {1,2,4}): no output

```

Worst-case analysis

The Bron–Kerbosch algorithm is not an output-sensitive algorithm: unlike some other algorithms for the clique problem, it does not run in polynomial time per maximal clique generated. However, it is efficient in a worst-case sense: by a result of Moon & Moser (1965), any n -vertex graph has at most $3^{n/3}$ maximal cliques, and the worst-case running time of the Bron–Kerbosch algorithm (with a pivot strategy that minimizes the number of recursive calls made at each step) is $O(3^{n/3})$, matching this bound.

For sparse graphs, tighter bounds are possible. In particular the vertex-ordering version of the Bron–Kerbosch algorithm can be made to run in time $O(dn3^{d/3})$, where d is the degeneracy of the graph, a measure of its sparseness. There exist d -degenerate graphs for which the total number of maximal cliques is $(n - d)3^{d/3}$, so this bound is close to tight.

Notes

[1] :.
[2] ;;.

References

- Akkoyunlu, E. A. (1973), "The enumeration of maximal cliques of large graphs", *SIAM Journal on Computing* **2**: 1–6, doi: 10.1137/0202001 (<http://dx.doi.org/10.1137/0202001>).
- Chen, Lingran (2004), "Substructure and maximal common substructure searching", in Bultinck, Patrick, *Computational Medicinal Chemistry for Drug Discovery*, CRC Press, pp. 483–514, ISBN 978-0-8247-4774-9.
- Bron, Coen; Kerbosch, Joep (1973), "Algorithm 457: finding all cliques of an undirected graph", *Commun. ACM (ACM)* **16** (9): 575–577, doi: 10.1145/362342.362367 (<http://dx.doi.org/10.1145/362342.362367>).
- Cazals, F.; Karande, C. (2008), "A note on the problem of reporting maximal cliques" (<ftp://ftp-sop.inria.fr/geometrica/fcazals/papers/ncliques.pdf>), *Theoretical Computer Science* **407** (1): 564–568, doi: 10.1016/j.tcs.2008.05.010 (<http://dx.doi.org/10.1016/j.tcs.2008.05.010>).
- Eppstein, David; Löffler, Maarten; Strash, Darren (2010), "Listing all maximal cliques in sparse graphs in near-optimal time", in Cheong, Otfried; Chwa, Kyung-Yong; Park, Kunsoo, *21st International Symposium on Algorithms and Computation (ISAAC 2010), Jeju, Korea*, Lecture Notes in Computer Science **6506**, Springer-Verlag, pp. 403–414, arXiv: 1006.5440 (<http://arxiv.org/abs/1006.5440>), doi: 10.1007/978-3-642-17517-6_36 (http://dx.doi.org/10.1007/978-3-642-17517-6_36).
- Eppstein, David; Strash, Darren (2011), "Listing all maximal cliques in large sparse real-world graphs", *10th International Symposium on Experimental Algorithms*, arXiv: 1103.0318 (<http://arxiv.org/abs/1103.0318>).
- Johnston, H. C. (1976), "Cliques of a graph—variations on the Bron–Kerbosch algorithm", *International Journal of Parallel Programming* **5** (3): 209–238, doi: 10.1007/BF00991836 (<http://dx.doi.org/10.1007/BF00991836>).
- Koch, Ina (2001), "Enumerating all connected maximal common subgraphs in two graphs", *Theoretical Computer Science* **250** (1–2): 1–30, doi: 10.1016/S0304-3975(00)00286-3 ([http://dx.doi.org/10.1016/S0304-3975\(00\)00286-3](http://dx.doi.org/10.1016/S0304-3975(00)00286-3)).

- Moon, J. W.; Moser, L. (1965), "On cliques in graphs", *Israel J. Math.* **3**: 23–28, doi: 10.1007/BF02760024 (<http://dx.doi.org/10.1007/BF02760024>), MR 0182577 (<http://www.ams.org/mathscinet-getitem?mr=0182577>).
- Tomita, Etsushi; Tanaka, Akira; Takahashi, Haruhisa (2006), "The worst-case time complexity for generating all maximal cliques and computational experiments", *Theoretical Computer Science* **363** (1): 28–42, doi: 10.1016/j.tcs.2006.06.015 (<http://dx.doi.org/10.1016/j.tcs.2006.06.015>).

External links

- Review of the Bron-Kerbosch algorithm and variations (<http://www.dcs.gla.ac.uk/~pat/jchoco/clique/enumeration/report.pdf>) by Alessio Conte
- Bron-Kerbosch algorithm implementation in Python (<http://www.kuchaev.com/files/graph.py>)
- Bron-Kerbosch algorithm with vertex ordering implementation in Python (<https://gist.github.com/abhinav8304062>)
- Finding all cliques of an undirected graph (http://www.dfki.de/~neumann/ie-seminar/presentations/finding_cliques.pdf). Seminar notes by Michaela Regneri, January 11, 2007.

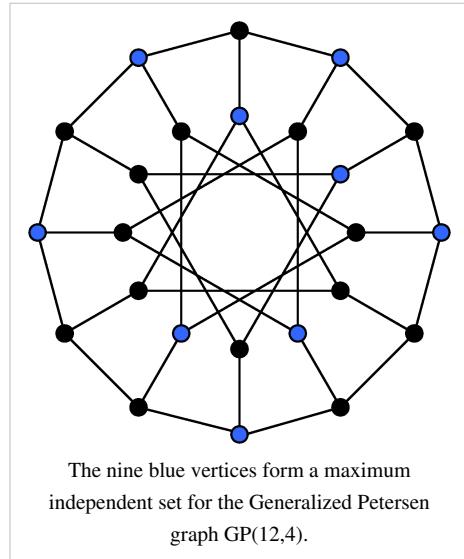
Independent set problem

In graph theory, an **independent set** or **stable set** is a set of vertices in a graph, no two of which are adjacent. That is, it is a set I of vertices such that for every two vertices in I , there is no edge connecting the two. Equivalently, each edge in the graph has at most one endpoint in I . The size of an independent set is the number of vertices it contains. Independent sets have also been called internally stable sets.

A maximal independent set is either an independent set such that adding any other vertex to the set forces the set to contain an edge or the set of all vertices of the empty graph.

A **maximum independent set** is an independent set of largest possible size for a given graph G . This size is called the **independence number** of G , and denoted $\alpha(G)$.^[1] The problem of finding such a set is called the **maximum independent set problem** and is an NP-hard optimization problem. As such, it is unlikely that there exists an efficient algorithm for finding a maximum independent set of a graph.

Every maximum independent set also is maximal, but the converse implication does not necessarily hold.



Properties

Relationship to other graph parameters

A set is independent if and only if it is a clique in the graph's complement, so the two concepts are complementary. In fact, sufficiently large graphs with no large cliques have large independent sets, a theme that is explored in Ramsey theory.

A set is independent if and only if its complement is a vertex cover.^[2] Therefore, the sum of the size of the largest independent set $\alpha(G)$, and the size of a minimum vertex cover $\beta(G)$, is equal to the number of vertices in the graph.

A vertex coloring of a graph G corresponds to a partition of its vertex set into independent subsets. Hence the minimal number of colors needed in a vertex coloring, the *chromatic number* $\chi(G)$, is at least the quotient of the number of vertices in G and the independent number $\alpha(G)$.

In a bipartite graph with no isolated vertices, the number of vertices in a maximum independent set equals the number of edges in a minimum edge covering; this is König's theorem.

Maximal independent set

Main article: Maximal independent set

An independent set that is not the subset of another independent set is called *maximal*. Such sets are dominating sets. Every graph contains at most $3^{n/3}$ maximal independent sets, but many graphs have far fewer. The number of maximal independent sets in n -vertex cycle graphs is given by the Perrin numbers, and the number of maximal independent sets in n -vertex path graphs is given by the Padovan sequence. Therefore, both numbers are proportional to powers of 1.324718, the plastic number.

Finding independent sets

Further information: Clique problem

In computer science, several computational problems related to independent sets have been studied.

- In the **maximum independent set** problem, the input is an undirected graph, and the output is a maximum independent set in the graph. If there are multiple maximum independent sets, only one need be output. This problem is sometimes referred to as "**vertex packing**".
- In the **maximum-weight independent set** problem, the input is an undirected graph with weights on its vertices and the output is an independent set with maximum total weight. The maximum independent set problem is the special case in which all weights are one.
- In the **maximal independent set listing** problem, the input is an undirected graph, and the output is a list of all its maximal independent sets. The maximum independent set problem may be solved using as a subroutine an algorithm for the maximal independent set listing problem, because the maximum independent set must be included among all the maximal independent sets.
- In the **independent set decision** problem, the input is an undirected graph and a number k , and the output is a Boolean value: true if the graph contains an independent set of size k , and false otherwise.

The first three of these problems are all important in practical applications; the independent set decision problem is not, but is necessary in order to apply the theory of NP-completeness to problems related to independent sets.

Maximum independent sets and maximum cliques

The independent set problem and the clique problem are complementary: a clique in G is an independent set in the complement graph of G and vice versa. Therefore, many computational results may be applied equally well to either problem. For example, the results related to the clique problem have the following corollaries:

- The independent set decision problem is NP-complete, and hence it is not believed that there is an efficient algorithm for solving it.
- The maximum independent set problem is NP-hard and it is also hard to approximate.

Despite the close relationship between maximum cliques and maximum independent sets in arbitrary graphs, the independent set and clique problems may be very different when restricted to special classes of graphs. For instance, for sparse graphs (graphs in which the number of edges is at most a constant times the number of vertices in any subgraph), the maximum clique has bounded size and may be found exactly in linear time; however, for the same classes of graphs, or even for the more restricted class of bounded degree graphs, finding the maximum independent set is MAXSNP-complete, implying that, for some constant c (depending on the degree) it is NP-hard to find an approximate solution that comes within a factor of c of the optimum.

Finding maximum independent sets

Further information: Clique problem § Finding maximum cliques in arbitrary graphs

Exact algorithms

The maximum independent set problem is NP-hard. However, it can be solved more efficiently than the $O(n^2 2^n)$ time that would be given by a naive brute force algorithm that examines every vertex subset and checks whether it is an independent set.

An algorithm of Robson (1986) solves the problem in time $O(1.2108^n)$ using exponential space. When restricted to polynomial space, there is a time $O(1.2127^n)$ algorithm, which improves upon a simpler $O(1.2209^n)$ algorithm.

In some classes of graphs, including claw-free graphs and perfect graphs, the maximum independent set may be found in polynomial time.^[3]

In a bipartite graph, all nodes that are not in the minimum vertex cover can be included in maximum independent set; see König's theorem. Therefore, minimum vertex covers can be found using a bipartite matching algorithm.

Approximation algorithms

The general, the maximum independent set problem cannot be approximated to a constant factor in polynomial time (unless P=NP). However, there are efficient approximation algorithms for restricted classes of graphs.

In planar graphs, the maximum independent set may be approximated to within any approximation ratio $c < 1$ in polynomial time; similar polynomial-time approximation schemes exist in any family of graphs closed under taking minors.^[4]

In bounded degree graphs, effective approximation algorithms are known with worse than constant approximation ratios; for instance, a greedy algorithm that forms a maximal independent set by, at each step, choosing the minimum degree vertex in the graph and removing its neighbors, achieves an approximation ratio of $(\Delta+2)/3$ on graphs with maximum degree Δ .

Independent sets in interval intersection graphs

Main article: Interval scheduling

An interval graph is a graph in which the nodes are 1-dimensional intervals (e.g. time intervals) and there is an edge between two intervals iff they intersect. An independent set in an interval graph is just a set of non-overlapping intervals. The problem of finding maximum independent sets in interval graphs has been studied, for example, in the context of job scheduling: given a set of jobs that has to be executed on a computer, find a maximum set of jobs that can be executed without interfering with each other. This problem can be solved exactly in polynomial time using earliest deadline first scheduling.

Independent sets in geometric intersection graphs

Main article: Maximum disjoint set

A geometric intersection graph is a graph in which the nodes are geometric shapes and there is an edge between two shapes iff they intersect. An independent set in a geometric intersection graph is just a set of disjoint (non-overlapping) shapes. The problem of finding maximum independent sets in geometric intersection graphs has been studied, for example, in the context of Automatic label placement: given a set of locations in a map, find a maximum set of disjoint rectangular labels near these locations.

Finding a maximum independent set in intersection graphs is still NP-complete, but it is easier to approximate than the general maximum independent set problem. A recent survey can be found in the introduction of Chan & Har-Peled (2012).

Finding maximal independent sets

Main article: Maximal independent set

The problem of finding a maximal independent set can be solved in polynomial time by a trivial greedy algorithm. All maximal independent sets can be found in time $O(3^{n/3}) = O(1.4423^n)$.

Software for searching maximum independent set

Name	License	API language	Brief info
igraph ^[5]	GPL	C, Python, R, Ruby	exact solution. "The current implementation was ported to igraph from the Very Nauty Graph Library by Keith Briggs and uses the algorithm from the paper S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. SIAM J Computing, 6:505--517, 1977".
NetworkX	BSD	Python	approximate solution, see the routine <code>maximum_independent_set</code> ^[6]
OpenOpt	BSD	Python	exact and approximate solutions, possibility to specify nodes that have to be included / excluded; see STAB ^[7] class for more details and examples

Notes

- [1] , p. 3.
- [2] PROOF: A set V of vertices is an independent set IFF every edge in the graph is adjacent to at most one member of V IFF every edge in the graph is adjacent to at least one member not in V IFF the complement of V is a vertex cover.
- [3] For claw-free graphs, see . For perfect graphs, see .
- [4] ; .
- [5] <http://igraph.sourceforge.net/doc/html/ch15s02.html>
- [6] http://networkx.lanl.gov/reference/generated/networkx.algorithms.approximation.independent_set.maximum_independent_set.html
- [7] <http://openopt.org/STAB>

References

- Baker, Brenda S. (1994), "Approximation algorithms for NP-complete problems on planar graphs", *Journal of the ACM* **41** (1): 153–180, doi: 10.1145/174644.174650 (<http://dx.doi.org/10.1145/174644.174650>).
- Berman, Piotr; Fujito, Toshihiro (1995), "On approximation properties of the Independent set problem for degree 3 graphs", *Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **955**, Springer-Verlag, pp. 449–460, doi: 10.1007/3-540-60220-8_84 (http://dx.doi.org/10.1007/3-540-60220-8_84).
- Bourgeois, Nicolas; Escoffier, Bruno; Paschos, Vangelis Th.; van Rooij, Johan M. M. (2010), "A bottom-up method and fast algorithms for MAX INDEPENDENT SET", *Algorithm theory—SWAT 2010*, Lecture Notes in Computer Science **6139**, Berlin: Springer, pp. 62–73, doi: 10.1007/978-3-642-13731-0_7 (http://dx.doi.org/10.1007/978-3-642-13731-0_7), MR 2678485 (<http://www.ams.org/mathscinet-getitem?mr=2678485>).
- Chan, T. M. (2003), "Polynomial-time approximation schemes for packing and piercing fat objects", *Journal of Algorithms* **46** (2): 178–189, doi: 10.1016/s0196-6774(02)00294-8 ([http://dx.doi.org/10.1016/s0196-6774\(02\)00294-8](http://dx.doi.org/10.1016/s0196-6774(02)00294-8))
- Chan, T. M.; Har-Peled, S. (2012), "Approximation algorithms for maximum independent set of pseudo-disks", *Discrete & Computational Geometry* **48** (2): 373, doi: 10.1007/s00454-012-9417-5 (<http://dx.doi.org/10.1007/s00454-012-9417-5>)
- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM Journal on Computing* **14** (1): 210–223, doi: 10.1137/0214017 (<http://dx.doi.org/10.1137/0214017>).
- Erlebach, T.; Jansen, K.; Seidel, E. (2005), "Polynomial-Time Approximation Schemes for Geometric Intersection Graphs", *SIAM Journal on Computing* **34** (6): 1302, doi: 10.1137/s0097539702402676 (<http://dx.doi.org/10.1137/s0097539702402676>)
- Fomin, Fedor V.; Grandoni, Fabrizio; Kratsch, Dieter (2009), "A measure & conquer approach for the analysis of exact algorithms", *Journal of ACM* **56** (5): 1–32, doi: 10.1145/1552285.1552286 (<http://dx.doi.org/10.1145/1552285.1552286>), article no. 25.
- Füredi, Z. (1987), "The number of maximal independent sets in connected graphs", *Journal of Graph Theory* **11** (4): 463–470, doi: 10.1002/jgt.3190110403 (<http://dx.doi.org/10.1002/jgt.3190110403>).
- Godsil, Chris; Royle, Gordon (2001), *Algebraic Graph Theory*, New York: Springer, ISBN 0-387-95220-9.
- Grohe, Martin (2003), "Local tree-width, excluded minors, and approximation algorithms", *Combinatorica* **23** (4): 613–632, doi: 10.1007/s00493-003-0037-9 (<http://dx.doi.org/10.1007/s00493-003-0037-9>).
- Grötschel, M.; Lovász, L.; Schrijver, A. (1988), "9.4 Coloring Perfect Graphs", *Geometric Algorithms and Combinatorial Optimization*, Algorithms and Combinatorics **2**, Springer-Verlag, pp. 296–298, ISBN 0-387-13624-X.
- Halldórsson, M. M.; Radhakrishnan, J. (1997), "Greed is good: Approximating independent sets in sparse and bounded-degree graphs", *Algorithmica* **18** (1): 145–163, doi: 10.1007/BF02523693 (<http://dx.doi.org/10.1007/BF02523693>).
- Luby, Michael (1986), "A simple parallel algorithm for the maximal independent set problem", *SIAM Journal on Computing* **15** (4): 1036–1053, doi: 10.1137/0215074 (<http://dx.doi.org/10.1137/0215074>), MR 861369

(<http://www.ams.org/mathscinet-getitem?mr=861369>).

- Moon, J. W.; Moser, Leo (1965), "On cliques in graphs", *Israel Journal of Mathematics* **3** (1): 23–28, doi: [10.1007/BF02760024](https://doi.org/10.1007/BF02760024) (<http://dx.doi.org/10.1007/BF02760024>), MR 0182577 (<http://www.ams.org/mathscinet-getitem?mr=0182577>).
- Robson, J. M. (1986), "Algorithms for maximum independent sets", *Journal of Algorithms* **7** (3): 425–440, doi: [10.1016/0196-6774\(86\)90032-5](https://doi.org/10.1016/0196-6774(86)90032-5) ([http://dx.doi.org/10.1016/0196-6774\(86\)90032-5](http://dx.doi.org/10.1016/0196-6774(86)90032-5)).
- Sbihi, Najiba (1980), "Algorithme de recherche d'un stable de cardinalité maximum dans un graphe sans étoile", *Discrete Mathematics* (in French) **29** (1): 53–76, doi: [10.1016/0012-365X\(90\)90287-R](https://doi.org/10.1016/0012-365X(90)90287-R) ([http://dx.doi.org/10.1016/0012-365X\(90\)90287-R](http://dx.doi.org/10.1016/0012-365X(90)90287-R)), MR 553650 (<http://www.ams.org/mathscinet-getitem?mr=553650>).
- Korshunov, A.D. (1974), "Coefficient of Internal Stability", *Kibernetika* (in Ukrainian) **10** (1): 17–28, doi: [10.1007/BF01069014](https://doi.org/10.1007/BF01069014) (<http://dx.doi.org/10.1007/BF01069014>).

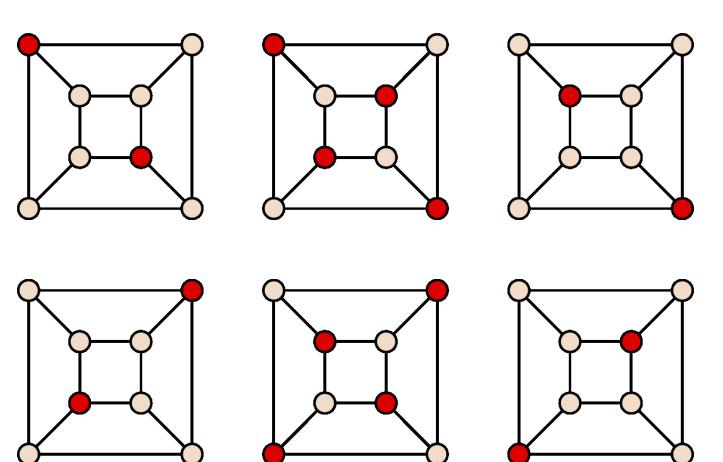
External links

- Weisstein, Eric W., "Maximal Independent Vertex Set" (<http://mathworld.wolfram.com/MaximalIndependentVertexSet.html>), *MathWorld*.
- Challenging Benchmarks for Maximum Clique, Maximum Independent Set, Minimum Vertex Cover and Vertex Coloring (<http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm>)
- Independent Set and Vertex Cover (<http://www.hananayad.com/teaching/syde423/IndependentSet.pdf>), Hanan Ayad.

Maximal independent set

This article is about the combinatorial aspects of maximal independent sets of vertices in a graph. For other aspects of independent vertex sets in graph theory, see Independent set (graph theory). For other kinds of independent sets, see Independent set (disambiguation).

In graph theory, a **maximal independent set** or **maximal stable set** is an independent set that is not a subset of any other independent set. That is, it is a set S such that every edge of the graph has at least one endpoint not in S and every vertex not in S has at least one neighbor in S . A maximal independent set is also a dominating set in the graph, and every dominating set that is independent must be maximal independent, so maximal independent sets are also called **independent dominating sets**. A graph may have many maximal independent sets of widely varying sizes;^[1] a largest maximal independent set is called a maximum independent set.



The graph of the cube has six different maximal independent sets, shown as the red vertices.

For example, in the graph P_3 , a path with three vertices a , b , and c , and two edges ab and bc , the sets $\{b\}$ and $\{a,c\}$ are both maximally independent. The set $\{a\}$ is independent, but is not maximal independent, because it is a subset of the larger independent set $\{a,c\}$. In this same graph, the maximal cliques are the sets $\{a,b\}$ and $\{b,c\}$.

The phrase "maximal independent set" is also used to describe maximal subsets of independent elements in mathematical structures other than graphs, and in particular in vector spaces and matroids.

Related vertex sets

If S is a maximal independent set in some graph, it is a **maximal clique** or **maximal complete subgraph** in the complementary graph. A maximal clique is a set of vertices that induces a complete subgraph, and that is not a subset of the vertices of any larger complete subgraph. That is, it is a set S such that every pair of vertices in S is connected by an edge and every vertex not in S is missing an edge to at least one vertex in S . A graph may have many maximal cliques, of varying sizes; finding the largest of these is the maximum clique problem.

Some authors include maximality as part of the definition of a clique, and refer to maximal cliques simply as cliques. The complement of a maximal independent set, that is, the set of vertices not belonging to the independent set, forms a **minimal vertex cover**. That is, the complement is a vertex cover, a set of vertices that includes at least one endpoint of each edge, and is minimal in the sense that none of its vertices can be removed while preserving the property that it is a cover. Minimal vertex covers have been studied in statistical mechanics in connection with the hard-sphere lattice gas model, a mathematical abstraction of fluid-solid state transitions.

Every maximal independent set is a dominating set, a set of vertices such that every vertex in the graph either belongs to the set or is adjacent to the set. A set of vertices is a maximal independent set if and only if it is an independent dominating set.

Graph family characterizations

Certain graph families have also been characterized in terms of their maximal cliques or maximal independent sets. Examples include the maximal-clique irreducible and hereditary maximal-clique irreducible graphs. A graph is said to be *maximal-clique irreducible* if every maximal clique has an edge that belongs to no other maximal clique, and *hereditary maximal-clique irreducible* if the same property is true for every induced subgraph.^[2] Hereditary maximal-clique irreducible graphs include triangle-free graphs, bipartite graphs, and interval graphs.

Cographs can be characterized as graphs in which every maximal clique intersects every maximal independent set, and in which the same property is true in all induced subgraphs.

Bounding the number of sets

Moon & Moser (1965) showed that any graph with n vertices has at most $3^{n/3}$ maximal cliques. Complementarily, any graph with n vertices also has at most $3^{n/3}$ maximal independent sets. A graph with exactly $3^{n/3}$ maximal independent sets is easy to construct: simply take the disjoint union of $n/3$ triangle graphs. Any maximal independent set in this graph is formed by choosing one vertex from each triangle. The complementary graph, with exactly $3^{n/3}$ maximal cliques, is a special type of Turán graph; because of their connection with Moon and Moser's bound, these graphs are also sometimes called Moon-Moser graphs. Tighter bounds are possible if one limits the size of the maximal independent sets: the number of maximal independent sets of size k in any n -vertex graph is at most

$$\lfloor n/k \rfloor^{k-(n \bmod k)} \lfloor n/k + 1 \rfloor^{n \bmod k}.$$

The graphs achieving this bound are again Turán graphs.^[3]

Certain families of graphs may, however, have much more restrictive bounds on the numbers of maximal independent sets or maximal cliques. If all n -vertex graphs in a family of graphs have $O(n)$ edges, and if every subgraph of a graph in the family also belongs to the family, then each graph in the family can have at most $O(n)$ maximal cliques, all of which have size $O(1)$.^[4] For instance, these conditions are true for the planar graphs: every n -vertex planar graph has at most $3n - 6$ edges, and a subgraph of a planar graph is always planar, from which it follows that each planar graph has $O(n)$ maximal cliques (of size at most four). Interval graphs and chordal graphs

also have at most n maximal cliques, even though they are not always sparse graphs.

The number of maximal independent sets in n -vertex cycle graphs is given by the Perrin numbers, and the number of maximal independent sets in n -vertex path graphs is given by the Padovan sequence.^[5] Therefore, both numbers are proportional to powers of 1.324718, the plastic number.

Set listing algorithms

Further information: Clique problem § Listing all maximal cliques

An algorithm for listing all maximal independent sets or maximal cliques in a graph can be used as a subroutine for solving many NP-complete graph problems. Most obviously, the solutions to the maximum independent set problem, the maximum clique problem, and the minimum independent dominating problem must all be maximal independent sets or maximal cliques, and can be found by an algorithm that lists all maximal independent sets or maximal cliques and retains the ones with the largest or smallest size. Similarly, the minimum vertex cover can be found as the complement of one of the maximal independent sets. Lawler (1976) observed that listing maximal independent sets can also be used to find 3-colorings of graphs: a graph can be 3-colored if and only if the complement of one of its maximal independent sets is bipartite. He used this approach not only for 3-coloring but as part of a more general graph coloring algorithm, and similar approaches to graph coloring have been refined by other authors since.^[6] Other more complex problems can also be modeled as finding a clique or independent set of a specific type. This motivates the algorithmic problem of listing all maximal independent sets (or equivalently, all maximal cliques) efficiently.

It is straightforward to turn a proof of Moon and Moser's $3^{n/3}$ bound on the number of maximal independent sets into an algorithm that lists all such sets in time $O(3^{n/3})$.^[7] For graphs that have the largest possible number of maximal independent sets, this algorithm takes constant time per output set. However, an algorithm with this time bound can be highly inefficient for graphs with more limited numbers of independent sets. For this reason, many researchers have studied algorithms that list all maximal independent sets in polynomial time per output set.^[8] The time per maximal independent set is proportional to that for matrix multiplication in dense graphs, or faster in various classes of sparse graphs.^[9]

Notes

[1] shows that the number of different sizes of maximal independent sets in an n -vertex graph may be as large as $n - \log n - O(\log \log n)$ and is never larger than $n - \log n$.

[2] Information System on Graph Class Inclusions: maximal clique irreducible graphs (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_749.html) and hereditary maximal clique irreducible graphs (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_750.html).

[3] . For related earlier results see and .

[4] . Chiba and Nishizeki express the condition of having $O(n)$ edges equivalently, in terms of the arboricity of the graphs in the family being constant.

[5] ; ; .

[6] ; .

[7] . For a matching bound for the widely used Bron–Kerbosch algorithm, see .

[8] ; ; ; ; ; ; ; .

[9] ; .

References

- Bisdorff, R.; Marichal, J.-L. (2007), *Counting non-isomorphic maximal independent sets of the n-cycle graph*, arXiv: math.CO/0701647 (<http://arxiv.org/abs/math.CO/0701647>).
- Bomze, I. M.; Budinich, M.; Pardalos, P. M.; Pelillo, M. (1999), "The maximum clique problem", *Handbook of Combinatorial Optimization* 4, Kluwer Academic Publishers, pp. 1–74, CiteSeerX: 10.1.1.48.4074 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.4074>).
- Byskov, J. M. (2003), "Algorithms for k -colouring and finding maximal independent sets" (<http://portal.acm.org/citation.cfm?id=644182>), *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 456–457.
- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM J. on Computing* 14 (1): 210–223, doi: 10.1137/0214017 (<http://dx.doi.org/10.1137/0214017>).
- Croitoru, C. (1979), "On stables in graphs", *Proc. Third Coll. Operations Research*, Babeş-Bolyai University, Cluj-Napoca, Romania, pp. 55–60.
- Eppstein, D. (2003), "Small maximal independent sets and faster exact graph coloring" (<http://www.cs.brown.edu/publications/jgaa/accepted/2003/Eppstein2003.7.2.pdf>), *Journal of Graph Algorithms and Applications* 7 (2): 131–140, arXiv: cs.DS/0011009 (<http://arxiv.org/abs/cs.DS/0011009>), doi: 10.7155/jgaa.00064 (<http://dx.doi.org/10.7155/jgaa.00064>).
- Eppstein, D. (2005), "All maximal independent sets and dynamic dominance for sparse graphs", *Proc. Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 451–459, arXiv: cs.DS/0407036 (<http://arxiv.org/abs/cs.DS/0407036>).
- Erdős, P. (1966), "On cliques in graphs", *Israel J. Math.* 4 (4): 233–234, doi: 10.1007/BF02771637 (<http://dx.doi.org/10.1007/BF02771637>), MR 0205874 (<http://www.ams.org/mathscinet-getitem?mr=0205874>).
- Euler, R. (2005), "The Fibonacci number of a grid graph and a new class of integer sequences", *Journal of Integer Sequences* 8 (2): 05.2.6.
- Füredi, Z. (1987), "The number of maximal independent sets in connected graphs", *Journal of Graph Theory* 11 (4): 463–470, doi: 10.1002/jgt.3190110403 (<http://dx.doi.org/10.1002/jgt.3190110403>).
- Jennings, E.; Motycková, L. (1992), "A distributed algorithm for finding all maximal cliques in a network graph", *Proc. First Latin American Symposium on Theoretical Informatics*, Lecture Notes in Computer Science 583, Springer-Verlag, pp. 281–293
- Johnson, D. S.; Yannakakis, M.; Papadimitriou, C. H. (1988), "On generating all maximal independent sets", *Information Processing Letters* 27 (3): 119–123, doi: 10.1016/0020-0190(88)90065-8 ([http://dx.doi.org/10.1016/0020-0190\(88\)90065-8](http://dx.doi.org/10.1016/0020-0190(88)90065-8)).
- Lawler, E. L. (1976), "A note on the complexity of the chromatic number problem", *Information Processing Letters* 5 (3): 66–67, doi: 10.1016/0020-0190(76)90065-X ([http://dx.doi.org/10.1016/0020-0190\(76\)90065-X](http://dx.doi.org/10.1016/0020-0190(76)90065-X)).
- Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G. (1980), "Generating all maximal independent sets: NP-hardness and polynomial time algorithms", *SIAM Journal on Computing* 9 (3): 558–565, doi: 10.1137/0209042 (<http://dx.doi.org/10.1137/0209042>).
- Leung, J. Y.-T. (1984), "Fast algorithms for generating all maximal independent sets of interval, circular-arc and chordal graphs", *Journal of Algorithms* 5: 22–35, doi: 10.1016/0196-6774(84)90037-3 ([http://dx.doi.org/10.1016/0196-6774\(84\)90037-3](http://dx.doi.org/10.1016/0196-6774(84)90037-3)).
- Liang, Y. D.; Dhall, S. K.; Lakshmivarahan, S. (1991), "On the problem of finding all maximum weight independent sets in interval and circular arc graphs", *Proc. Symp. Applied Computing*, pp. 465–470
- Makino, K.; Uno, T. (2004), "New algorithms for enumerating all maximal cliques" (<http://www.springerlink.com/content/p9ql6y1v5t3xc1w/>), *Proc. Ninth Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Compute Science 3111, Springer-Verlag, pp. 260–272.

- Mishra, N.; Pitt, L. (1997), "Generating all maximal independent sets of bounded-degree hypergraphs", *Proc. Tenth Conf. Computational Learning Theory*, pp. 211–217, doi: 10.1145/267460.267500 (<http://dx.doi.org/10.1145/267460.267500>), ISBN 0-89791-891-6.
- Moon, J. W.; Moser, L. (1965), "On cliques in graphs", *Israel Journal of Mathematics* **3**: 23–28, doi: 10.1007/BF02760024 (<http://dx.doi.org/10.1007/BF02760024>), MR 0182577 (<http://www.ams.org/mathscinet-getitem?mr=0182577>).
- Stix, V. (2004), "Finding all maximal cliques in dynamic graphs", *Computational Optimization Appl.* **27** (2): 173–186, doi: 10.1023/B:COAP.0000008651.28952.b6 (<http://dx.doi.org/10.1023/B:COAP.0000008651.28952.b6>).
- Tomita, E.; Tanaka, A.; Takahashi, H. (2006), "The worst-case time complexity for generating all maximal cliques and computational experiments", *Theoretical Computer Science* **363** (1): 28–42, doi: 10.1016/j.tcs.2006.06.015 (<http://dx.doi.org/10.1016/j.tcs.2006.06.015>).
- Tsukiyama, S.; Ide, M.; Ariyoshi, H.; Shirakawa, I. (1977), "A new algorithm for generating all the maximal independent sets", *SIAM J. on Computing* **6** (3): 505–517, doi: 10.1137/0206036 (<http://dx.doi.org/10.1137/0206036>).
- Weigt, Martin; Hartmann, Alexander K. (2001), "Minimal vertex covers on finite-connectivity random graphs: A hard-sphere lattice-gas picture", *Phys. Rev. E* **63** (5): 056127, arXiv: cond-mat/0011446 (<http://arxiv.org/abs/cond-mat/0011446>), doi: 10.1103/PhysRevE.63.056127 (<http://dx.doi.org/10.1103/PhysRevE.63.056127>).
- Yu, C.-W.; Chen, G.-H. (1993), "Generate all maximal independent sets in permutation graphs", *Internat. J. Comput. Math.* **47**: 1–8, doi: 10.1080/00207169308804157 (<http://dx.doi.org/10.1080/00207169308804157>).

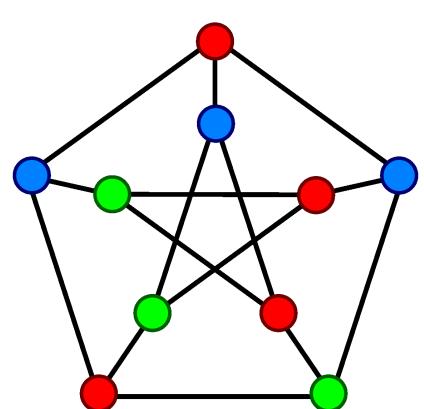
Graph coloring

In graph theory, **graph coloring** is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a **vertex coloring**. Similarly, an **edge coloring** assigns a color to each edge so that no two adjacent edges share the same color, and a **face coloring** of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.

Vertex coloring is the starting point of the subject, and other coloring problems can be transformed into a vertex version. For example, an edge coloring of a graph is just a vertex coloring of its line graph, and a face coloring of a plane graph is just a vertex coloring of its dual. However, non-vertex coloring problems are often stated and studied *as is*. That is partly for perspective, and partly because some problems are best studied in non-vertex form, as for instance is edge coloring.

The convention of using colors originates from coloring the countries of a map, where each face is literally colored. This was generalized to coloring the faces of a graph embedded in the plane. By planar duality it became coloring the vertices, and in this form it generalizes to all graphs. In mathematical and computer representations, it is typical to use the first few positive or nonnegative integers as the "colors". In general, one can use any finite set as the "color set". The nature of the coloring problem depends on the number of colors but not on what they are.

Graph coloring enjoys many practical applications as well as theoretical challenges. Beside the classical types of problems, different limitations can also be set on the graph, or on the way a color is assigned, or even on the color



A proper vertex coloring of the Petersen graph with 3 colors, the minimum number possible.

itself. It has even reached popularity with the general public in the form of the popular number puzzle Sudoku. Graph coloring is still a very active field of research.

Note: Many terms used in this article are defined in Glossary of graph theory.

History

See also: History of the four color theorem and History of graph theory

The first results about graph coloring deal almost exclusively with planar graphs in the form of the coloring of *maps*. While trying to color a map of the counties of England, Francis Guthrie postulated the four color conjecture, noting that four colors were sufficient to color the map so that no regions sharing a common border received the same color. Guthrie's brother passed on the question to his mathematics teacher Augustus de Morgan at University College, who mentioned it in a letter to William Hamilton in 1852. Arthur Cayley raised the problem at a meeting of the London Mathematical Society in 1879. The same year, Alfred Kempe published a paper that claimed to establish the result, and for a decade the four color problem was considered solved. For his accomplishment Kempe was elected a Fellow of the Royal Society and later President of the London Mathematical Society.^[1]

In 1890, Heawood pointed out that Kempe's argument was wrong. However, in that paper he proved the five color theorem, saying that every planar map can be colored with no more than *five* colors, using ideas of Kempe. In the following century, a vast amount of work and theories were developed to reduce the number of colors to four, until the four color theorem was finally proved in 1976 by Kenneth Appel and Wolfgang Haken. The proof went back to the ideas of Heawood and Kempe and largely disregarded the intervening developments. The proof of the four color theorem is also noteworthy for being the first major computer-aided proof.

In 1912, George David Birkhoff introduced the chromatic polynomial to study the coloring problems, which was generalised to the Tutte polynomial by Tutte, important structures in algebraic graph theory. Kempe had already drawn attention to the general, non-planar case in 1879,^[2] and many results on generalisations of planar graph coloring to surfaces of higher order followed in the early 20th century.

In 1960, Claude Berge formulated another conjecture about graph coloring, the *strong perfect graph conjecture*, originally motivated by an information-theoretic concept called the zero-error capacity of a graph introduced by Shannon. The conjecture remained unresolved for 40 years, until it was established as the celebrated strong perfect graph theorem by Chudnovsky, Robertson, Seymour, and Thomas in 2002.

Graph coloring has been studied as an algorithmic problem since the early 1970s: the chromatic number problem is one of Karp's 21 NP-complete problems from 1972, and at approximately the same time various exponential-time algorithms were developed based on backtracking and on the deletion-contraction recurrence of Zykov (1949). One of the major applications of graph coloring, register allocation in compilers, was introduced in 1981.

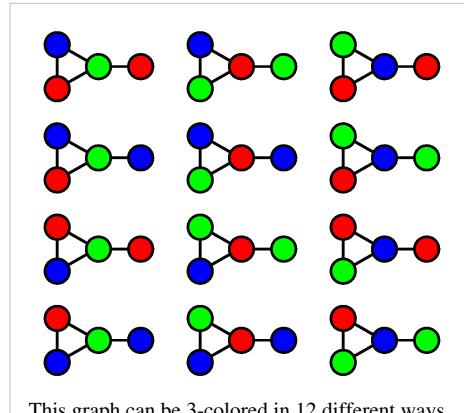
Definition and terminology

Vertex coloring

When used without any qualification, a **coloring** of a graph is almost always a *proper vertex coloring*, namely a labelling of the graph's vertices with colors such that no two vertices sharing the same edge have the same color. Since a vertex with a loop (i.e. a connection directly back to itself) could never be properly colored, it is understood that graphs in this context are loopless.

The terminology of using *colors* for vertex labels goes back to map coloring. Labels like *red* and *blue* are only used when the number of colors is small, and normally it is understood that the labels are drawn from the integers $\{1, 2, 3, \dots\}$.

A coloring using at most k colors is called a (proper) **k -coloring**. The smallest number of colors needed to color a graph G is called its **chromatic number**, and is often denoted $\chi(G)$. Sometimes $\gamma(G)$ is used, since $\chi(G)$ is also used to denote the Euler characteristic of a graph. A graph that can be assigned a (proper) k -coloring is **k -colorable**, and it is **k -chromatic** if its chromatic number is exactly k . A subset of vertices assigned to the same color is called a **color class**, every such class forms an independent set. Thus, a k -coloring is the same as a partition of the vertex set into k independent sets, and the terms **k -partite** and **k -colorable** have the same meaning.

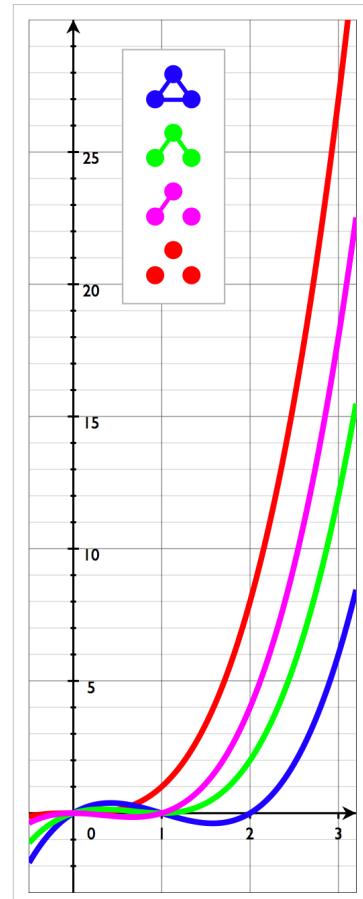


This graph can be 3-colored in 12 different ways.

Chromatic polynomial

Main article: Chromatic polynomial

The **chromatic polynomial** counts the number of ways a graph can be colored using no more than a given number of colors. For example, using three colors, the graph in the image to the right can be colored in 12 ways. With only two colors, it cannot be colored at all. With four colors, it can be colored in $24 + 4 \cdot 12 = 72$ ways: using all four colors, there are $4! = 24$ valid colorings (*every* assignment of four colors to *any* 4-vertex graph is a proper coloring); and for every choice of three of the four colors, there are 12 valid 3-colorings. So, for the graph in the example, a table of the number of valid colorings would start like this:



All nonisomorphic graphs on 3 vertices and their chromatic polynomials. The empty graph E_3 (red) admits a 1-coloring, the others admit no such colorings. The green graph admits 12 colorings with 3 colors.

Available colors	1	2	3	4	...
Number of colorings	0	0	12	72	...

The chromatic polynomial is a function $P(G, t)$ that counts the number of t -colorings of G . As the name indicates, for a given G the function is indeed a polynomial in t . For the example graph, $P(G, t) = t(t - 1)^2(t - 2)$, and indeed $P(G, 4) = 72$.

The chromatic polynomial includes at least as much information about the colorability of G as does the chromatic number. Indeed, χ is the smallest positive integer that is not a root of the chromatic polynomial

$$\chi(G) = \min\{k : P(G, k) > 0\}.$$

Chromatic polynomials for certain graphs

Triangle K_3	$t(t-1)(t-2)$
Complete graph K_n	$t(t-1)(t-2) \cdots (t-(n-1))$
Tree with n vertices	$t(t-1)^{n-1}$
Cycle C_n	$(t-1)^n + (-1)^n(t-1)$
Petersen graph	$t(t-1)(t-2)(t^7 - 12t^6 + 67t^5 - 230t^4 + 529t^3 - 814t^2 + 775t - 352)$

Edge coloring

Main article: Edge coloring

An **edge coloring** of a graph is a proper coloring of the *edges*, meaning an assignment of colors to edges so that no vertex is incident to two edges of the same color. An edge coloring with k colors is called a k -edge-coloring and is equivalent to the problem of partitioning the edge set into k matchings. The smallest number of colors needed for an edge coloring of a graph G is the **chromatic index**, or **edge chromatic number**, $\chi'(G)$. A **Tait coloring** is a 3-edge coloring of a cubic graph. The four color theorem is equivalent to the assertion that every planar cubic bridgeless graph admits a Tait coloring.

Total coloring

Main article: Total coloring

Total coloring is a type of coloring on the vertices *and* edges of a graph. When used without any qualification, a total coloring is always assumed to be proper in the sense that no adjacent vertices, no adjacent edges, and no edge and its endvertices are assigned the same color. The total chromatic number $\chi''(G)$ of a graph G is the least number of colors needed in any total coloring of G .

Unlabeled coloring

An **unlabeled coloring** of a graph is an orbit of a coloring under the action of the automorphism group of the graph. If we interpret a coloring of a graph on d vertices as a vector in \mathbb{Z}^d , the action of an automorphism is a permutation of the coefficients of the coloring. There are analogues of the chromatic polynomials which count the number of unlabeled colorings of a graph from a given finite color set.

Properties

Bounds on the chromatic number

Assigning distinct colors to distinct vertices always yields a proper coloring, so

$$1 \leq \chi(G) \leq n.$$

The only graphs that can be 1-colored are edgeless graphs. A complete graph K_n of n vertices requires $\chi(K_n) = n$ colors. In an optimal coloring there must be at least one of the graph's m edges between every pair of color classes, so

$$\chi(G)(\chi(G) - 1) \leq 2m.$$

If G contains a clique of size k , then at least k colors are needed to color that clique; in other words, the chromatic number is at least the clique number:

$$\chi(G) \geq \omega(G).$$

For interval graphs this bound is tight.

The 2-colorable graphs are exactly the bipartite graphs, including trees and forests. By the four color theorem, every planar graph can be 4-colored.

A greedy coloring shows that every graph can be colored with one more color than the maximum vertex degree,

$$\chi(G) \leq \Delta(G) + 1.$$

Complete graphs have $\chi(G) = n$ and $\Delta(G) = n - 1$, and odd cycles have $\chi(G) = 3$ and $\Delta(G) = 2$, so for these graphs this bound is best possible. In all other cases, the bound can be slightly improved; Brooks' theorem states that

Brooks' theorem: $\chi(G) \leq \Delta(G)$ for a connected, simple graph G , unless G is a complete graph or an odd cycle.

Graphs with high chromatic number

Graphs with large cliques have a high chromatic number, but the opposite is not true. The Grötzsch graph is an example of a 4-chromatic graph without a triangle, and the example can be generalised to the Mycielskians.

Mycielski's Theorem (Alexander Zykov 1949, Jan Mycielski 1955): There exist triangle-free graphs with arbitrarily high chromatic number.

From Brooks's theorem, graphs with high chromatic number must have high maximum degree. Another local property that leads to high chromatic number is the presence of a large clique. But colorability is not an entirely local phenomenon: A graph with high girth looks locally like a tree, because all cycles are long, but its chromatic number need not be 2:

Theorem (Erdős): There exist graphs of arbitrarily high girth and chromatic number.

Bounds on the chromatic index

An edge coloring of G is a vertex coloring of its line graph $L(G)$, and vice versa. Thus,

$$\chi'(G) = \chi(L(G)).$$

There is a strong relationship between edge colorability and the graph's maximum degree $\Delta(G)$. Since all edges incident to the same vertex need their own color, we have

$$\chi'(G) \geq \Delta(G).$$

Moreover,

König's theorem: $\chi'(G) = \Delta(G)$ if G is bipartite.

In general, the relationship is even stronger than what Brooks's theorem gives for vertex coloring:

Vizing's Theorem: A graph of maximal degree Δ has edge-chromatic number Δ or $\Delta + 1$.

Other properties

A graph has a k -coloring if and only if it has an acyclic orientation for which the longest path has length at most k ; this is the Gallai–Hasse–Roy–Vitaver theorem (Nešetřil & Ossona de Mendez 2012).

For planar graphs, vertex colorings are essentially dual to nowhere-zero flows.

About infinite graphs, much less is known. The following are two of the few results about infinite graph coloring:

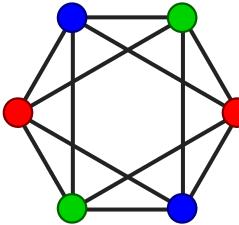
- If all finite subgraphs of an infinite graph G are k -colorable, then so is G , under the assumption of the axiom of choice. This is the de Bruijn–Erdős theorem of de Bruijn & Erdős (1951).
- If a graph admits a full n -coloring for every $n \geq n_0$, it admits an infinite full coloring (Fawcett 1978).

Open problems

The chromatic number of the plane, where two points are adjacent if they have unit distance, is unknown, although it is one of 4, 5, 6, or 7. Other open problems concerning the chromatic number of graphs include the Hadwiger conjecture stating that every graph with chromatic number k has a complete graph on k vertices as a minor, the Erdős–Faber–Lovász conjecture bounding the chromatic number of unions of complete graphs that have at exactly one vertex in common to each pair, and the Albertson conjecture that among k -chromatic graphs the complete graphs are the ones with smallest crossing number.

When Birkhoff and Lewis introduced the chromatic polynomial in their attack on the four-color theorem, they conjectured that for planar graphs G , the polymomial $P(G, t)$ has no zeros in the region $[4, \infty)$. Although it is known that such a chromatic polynomial has no zeros in the region $[5, \infty)$ and that $P(G, 4) \neq 0$, their conjecture is still unresolved. It also remains an unsolved problem to characterize graphs which have the same chromatic polynomial and to determine which polynomials are chromatic.

Algorithms

Graph coloring	
	
Decision	
Name	Graph coloring, vertex coloring, k -coloring
Input	Graph G with n vertices. Integer k
Output	Does G admit a proper vertex coloring with k colors?
Running time	$O(2^n n)$
Complexity	NP-complete
Reduction from	3-Satisfiability
Garey–Johnson	GT4
Optimisation	
Name	Chromatic number
Input	Graph G with n vertices.
Output	$\chi(G)$
Complexity	NP-hard
Approximability	$O(n (\log n)^{-3} (\log \log n)^2)$
Inapproximability	$O(n^{1-\epsilon})$ unless $P = NP$
Counting problem	
Name	Chromatic polynomial
Input	Graph G with n vertices. Integer k
Output	The number $P(G, k)$ of proper k -colorings of G

Running time	$O(2^n n)$
Complexity	#P-complete
Approximability	FPRAS for restricted cases
Inapproximability	No PTAS unless P = NP

Polynomial time

Determining if a graph can be colored with 2 colors is equivalent to determining whether or not the graph is bipartite, and thus computable in linear time using breadth-first search. More generally, the chromatic number and a corresponding coloring of perfect graphs can be computed in polynomial time using semidefinite programming. Closed formulas for chromatic polynomial are known for many classes of graphs, such as forests, chordal graphs, cycles, wheels, and ladders, so these can be evaluated in polynomial time.

If the graph is planar and has low branchwidth (or is nonplanar but with a known branch decompositon), then it can be solved in polynomial time using dynamic programming. In general, the time required is polynomial in the graph size, but exponential in the branchwidth.

Exact algorithms

Brute-force search for a k -coloring considers each of the k^n assignments of k colors to n vertices and checks for each if it is legal. To compute the chromatic number and the chromatic polynomial, this procedure is used for every $k = 1, \dots, n - 1$, impractical for all but the smallest input graphs.

Using dynamic programming and a bound on the number of maximal independent sets, k -colorability can be decided in time and space $O(2.445^n)$. Using the principle of inclusion–exclusion and Yates's algorithm for the fast zeta transform, k -colorability can be decided in time $O(2^n n)$ for any k . Faster algorithms are known for 3- and 4-colorability, which can be decided in time $O(1.3289^n)$ and $O(1.7272^n)$, respectively.

Contraction

The contraction G/uv of graph G is the graph obtained by identifying the vertices u and v , removing any edges between them, and replacing them with a single vertex w where any edges that were incident on u or v are redirected to w . This operation plays a major role in the analysis of graph coloring.

The chromatic number satisfies the recurrence relation:

$$\chi(G) = \min\{\chi(G + uv), \chi(G/uv)\}$$

due to Zykov (1949), where u and v are nonadjacent vertices, $G + uv$ is the graph with the edge uv added.

Several algorithms are based on evaluating this recurrence, the resulting computation tree is sometimes called a Zykov tree. The running time is based on the heuristic for choosing the vertices u and v .

The chromatic polynomial satisfies following recurrence relation

$$P(G - uv, k) = P(G/uv, k) + P(G, k)$$

where u and v are adjacent vertices and $G - uv$ is the graph with the edge uv removed. $P(G - uv, k)$ represents the number of possible proper colorings of the graph, when the vertices may have same or different colors. The number of proper colorings therefore come from the sum of two graphs. If the vertices u and v have different colors, then we can as well consider a graph, where u and v are adjacent. If u and v have the same colors, we may as well consider a graph, where u and v are contracted. Tutte's curiosity about which other graph properties satisfied this recurrence led him to discover a bivariate generalization of the chromatic polynomial, the Tutte polynomial.

The expressions give rise to a recursive procedure, called the *deletion–contraction algorithm*, which forms the basis of many algorithms for graph coloring. The running time satisfies the same recurrence relation as the Fibonacci numbers, so in the worst case, the algorithm runs in time within a polynomial factor of $((1 + \sqrt{5})/2)^{n+m} = O(1.6180^{n+m})$ for n vertices and m edges. The analysis can be improved to within a polynomial factor of the number $t(G)$ of spanning trees of the input graph. In practice, branch and bound strategies and graph isomorphism rejection are employed to avoid some recursive calls, the running time depends on the heuristic used to pick the vertex pair.

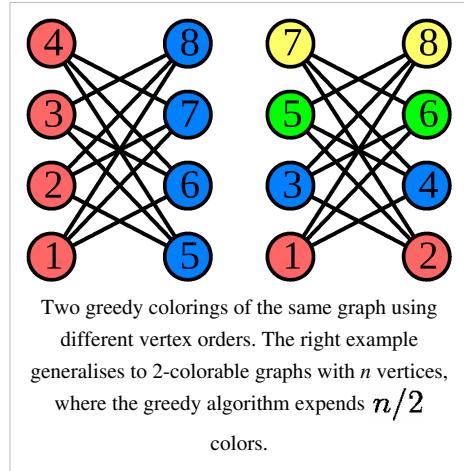
Greedy coloring

Main article: Greedy coloring

The greedy algorithm considers the vertices in a specific order v_1, \dots, v_n and assigns to v_i the smallest available color not used by v_i 's neighbours among v_1, \dots, v_{i-1} , adding a fresh color if needed. The quality of the resulting coloring depends on the chosen ordering. There exists an ordering that leads to a greedy coloring with the optimal number of $\chi(G)$ colors. On the other hand, greedy colorings can be arbitrarily bad; for example, the crown graph on n vertices can be 2-colored, but has an ordering that leads to a greedy coloring with $n/2$ colors.

For chordal graphs, and for special cases of chordal graphs such as interval graphs and indifference graphs, the greedy coloring algorithm can be used to find optimal colorings in polynomial time, by choosing the vertex ordering to be the reverse of a perfect elimination ordering for the graph. The perfectly orderable graphs generalize this property, but it is NP-hard to find a perfect ordering of these graphs.

If the vertices are ordered according to their degrees, the resulting greedy coloring uses at most $\max_i \min\{d(x_i) + 1, i\}$ colors, at most one more than the graph's maximum degree. This heuristic is sometimes called the Welsh–Powell algorithm. Another heuristic due to Brélaz establishes the ordering dynamically while the algorithm proceeds, choosing next the vertex adjacent to the largest number of different colors. Many other graph coloring heuristics are similarly based on greedy coloring for a specific static or dynamic strategy of ordering the vertices, these algorithms are sometimes called **sequential coloring** algorithms.



Parallel and distributed algorithms

In the field of distributed algorithms, graph coloring is closely related to the problem of symmetry breaking. The current state-of-the-art randomized algorithms are faster for sufficiently large maximum degree Δ than deterministic algorithms. The fastest randomized algorithms employ the multi-trials technique by Schneider et al.

In a symmetric graph, a deterministic distributed algorithm cannot find a proper vertex coloring. Some auxiliary information is needed in order to break symmetry. A standard assumption is that initially each node has a *unique identifier*, for example, from the set $\{1, 2, \dots, n\}$. Put otherwise, we assume that we are given an n -coloring. The challenge is to *reduce* the number of colors from n to, e.g., $\Delta + 1$. The more colors are employed, e.g. $O(\Delta)$ instead of $\Delta + 1$, the fewer communication rounds are required.

A straightforward distributed version of the greedy algorithm for $(\Delta + 1)$ -coloring requires $\Theta(n)$ communication rounds in the worst case – information may need to be propagated from one side of the network to another side.

The simplest interesting case is an n -cycle. Richard Cole and Uzi Vishkin^[3] show that there is a distributed algorithm that reduces the number of colors from n to $O(\log n)$ in one synchronous communication step. By iterating the same procedure, it is possible to obtain a 3-coloring of an n -cycle in $O(\log^* n)$ communication steps (assuming

that we have unique node identifiers).

The function \log^* , iterated logarithm, is an extremely slowly growing function, "almost constant". Hence the result by Cole and Vishkin raised the question of whether there is a *constant-time* distribute algorithm for 3-coloring an n -cycle. Linial (1992) showed that this is not possible: any deterministic distributed algorithm requires $\Omega(\log^* n)$ communication steps to reduce an n -coloring to a 3-coloring in an n -cycle.

The technique by Cole and Vishkin can be applied in arbitrary bounded-degree graphs as well; the running time is $\text{poly}(\Delta) + O(\log^* n)$. The technique was extended to unit disk graphs by Schneider et al. The fastest deterministic algorithms for $(\Delta + 1)$ -coloring for small Δ are due to Leonid Barenboim, Michael Elkin and Fabian Kuhn. The algorithm by Barenboim et al. runs in time $O(\Delta) + \log^*(n)/2$, which is optimal in terms of n since the constant factor 1/2 cannot be improved due to Linial's lower bound. Panconesi et al. use network decompositions to compute a $\Delta+1$ coloring in time $2^{O(\sqrt{\log n})}$.

The problem of edge coloring has also been studied in the distributed model. Panconesi & Rizzi (2001) achieve a $(2\Delta - 1)$ -coloring in $O(\Delta + \log^* n)$ time in this model. The lower bound for distributed vertex coloring due to Linial (1992) applies to the distributed edge coloring problem as well.

Decentralized algorithms

Decentralized algorithms are ones where no message passing is allowed (in contrast to distributed algorithms where local message passing takes places), and efficient decentralized algorithms exist that will color a graph if a proper coloring exists. These assume that a vertex is able to sense whether any of its neighbors are using the same color as the vertex i.e., whether a local conflict exists. This is a mild assumption in many applications e.g. in wireless channel allocation it is usually reasonable to assume that a station will be able to detect whether other interfering transmitters are using the same channel (e.g. by measuring the SINR). This sensing information is sufficient to allow algorithms based on learning automata to find a proper graph coloring with probability one, e.g. see Leith (2006) and Duffy (2008).

Computational complexity

Graph coloring is computationally hard. It is NP-complete to decide if a given graph admits a k -coloring for a given k except for the cases $k = 1$ and $k = 2$. In particular, it is NP-hard to compute the chromatic number.^[4] The 3-coloring problem remains NP-complete even on planar graphs of degree 4.

The best known approximation algorithm computes a coloring of size at most within a factor $O(n(\log n)^{-3}(\log \log n)^2)$ of the chromatic number. For all $\varepsilon > 0$, approximating the chromatic number within $n^{1-\varepsilon}$ is NP-hard.

It is also NP-hard to color a 3-colorable graph with 4 colors and a k -colorable graph with $k^{(\log k)/25}$ colors for sufficiently large constant k .

Computing the coefficients of the chromatic polynomial is #P-hard. In fact, even computing the value of $\chi(G, k)$ is #P-hard at any rational point k except for $k = 1$ and $k = 2$. There is no FPRAS for evaluating the chromatic polynomial at any rational point $k \geq 1.5$ except for $k = 2$ unless $\text{NP} = \text{RP}$.

For edge coloring, the proof of Vizing's result gives an algorithm that uses at most $\Delta+1$ colors. However, deciding between the two candidate values for the edge chromatic number is NP-complete. In terms of approximation algorithms, Vizing's algorithm shows that the edge chromatic number can be approximated to within 4/3, and the hardness result shows that no $(4/3 - \varepsilon)$ -algorithm exists for any $\varepsilon > 0$ unless $\text{P} = \text{NP}$. These are among the oldest results in the literature of approximation algorithms, even though neither paper makes explicit use of that notion.

Applications

Scheduling

Vertex coloring models to a number of scheduling problems. In the cleanest form, a given set of jobs need to be assigned to time slots, each job requires one such slot. Jobs can be scheduled in any order, but pairs of jobs may be in *conflict* in the sense that they may not be assigned to the same time slot, for example because they both rely on a shared resource. The corresponding graph contains a vertex for every job and an edge for every conflicting pair of jobs. The chromatic number of the graph is exactly the minimum *makespan*, the optimal time to finish all jobs without conflicts.

Details of the scheduling problem define the structure of the graph. For example, when assigning aircraft to flights, the resulting conflict graph is an interval graph, so the coloring problem can be solved efficiently. In bandwidth allocation to radio stations, the resulting conflict graph is a unit disk graph, so the coloring problem is 3-approximable.

Register allocation

Main article: Register allocation

A compiler is a computer program that translates one computer language into another. To improve the execution time of the resulting code, one of the techniques of compiler optimization is register allocation, where the most frequently used values of the compiled program are kept in the fast processor registers. Ideally, values are assigned to registers so that they can all reside in the registers when they are used.

The textbook approach to this problem is to model it as a graph coloring problem. The compiler constructs an *interference graph*, where vertices are variables and an edge connects two vertices if they are needed at the same time. If the graph can be colored with k colors then any set of variables needed at the same time can be stored in at most k registers.

Other applications

The problem of coloring a graph has found a number of applications, including pattern matching.

The recreational puzzle Sudoku can be seen as completing a 9-coloring on given specific graph with 81 vertices.

Other colorings

Ramsey theory

Main article: Ramsey theory

An important class of *improper* coloring problems is studied in Ramsey theory, where the graph's edges are assigned to colors, and there is no restriction on the colors of incident edges. A simple example is the friendship theorem, which states that in any coloring of the edges of K_6 the complete graph of six vertices there will be a monochromatic triangle; often illustrated by saying that any group of six people either has three mutual strangers or three mutual acquaintances. Ramsey theory is concerned with generalisations of this idea to seek regularity amid disorder, finding general conditions for the existence of monochromatic subgraphs with given structure.

Other colorings

List coloring	Rank coloring
Each vertex chooses from a list of colors	If two vertices have the same color i , then every path between them contain a vertex with color greater than i
List edge-coloring	Interval edge-coloring
Each edge chooses from a list of colors	A color of edges meeting in a common vertex must be contiguous
Total coloring	Circular coloring
Vertices and edges are colored	Motivated by task systems in which production proceeds in a cyclic way
Harmonious coloring	Path coloring
Every pair of colors appears on at most one edge	Models a routing problem in graphs
Complete coloring	Fractional coloring
Every pair of colors appears on at least one edge	Vertices may have multiple colors, and on each edge the sum of the color parts of each vertex is not greater than one
Exact coloring	Oriented coloring
Every pair of colors appears on exactly one edge	Takes into account orientation of edges of the graph
Acyclic coloring	Cocoloring
Every 2-chromatic subgraph is acyclic	An improper vertex coloring where every color class induces an independent set or a clique
Star coloring	Subcoloring
Every 2-chromatic subgraph is a disjoint collection of stars	An improper vertex coloring where every color class induces a union of cliques
Strong coloring	Defective coloring
Every color appears in every partition of equal size exactly once	An improper vertex coloring where every color class induces a bounded degree subgraph.
Strong edge coloring	Weak coloring
Edges are colored such that each color class induces a matching (equivalent to coloring the square of the line graph)	An improper vertex coloring where every non-isolated node has at least one neighbor with a different color
Equitable coloring	Sum-coloring
The sizes of color classes differ by at most one	The criterion of minimalization is the sum of colors
T-coloring	Adjacent-vertex-distinguishing-total coloring
Distance between two colors of adjacent vertices must not belong to fixed set T	A total coloring with the additional restriction that any two adjacent vertices have different color sets
	Centered coloring
	Every connected induced subgraph has a color that is used exactly once

Coloring can also be considered for signed graphs and gain graphs.

Notes

- [1] M. Kubale, *History of graph coloring*, in
- [2] , p. 2
- [3] , see also
- [4] ; .

References

- Barenboim, L.; Elkin, M. (2009), "Distributed $(\Delta + 1)$ -coloring in linear (in Δ) time", *Proceedings of the 41st Symposium on Theory of Computing*, pp. 111–120, doi: 10.1145/1536414.1536432 (<http://dx.doi.org/10.1145/1536414.1536432>), ISBN 978-1-60558-506-2
- Panconesi, A.; Srinivasan, A. (1996), "On the complexity of distributed network decomposition", *Journal of Algorithms* **20**
- Schneider, J. (2010), "A new technique for distributed symmetry breaking" (http://www.dcg.ethz.ch/publications/podcfp107_schneider_188.pdf), *Proceedings of the Symposium on Principles of Distributed Computing*
- Schneider, J. (2008), "A log-star distributed maximal independent set algorithm for growth-bounded graphs" (<http://www.dcg.ethz.ch/publications/podc08SW.pdf>), *Proceedings of the Symposium on Principles of Distributed Computing*
- Beigel, R.; Eppstein, D. (2005), "3-coloring in time $O(1.3289^n)$ ", *Journal of Algorithms* **54** (2): 168–204, doi: 10.1016/j.jalgor.2004.06.008 (<http://dx.doi.org/10.1016/j.jalgor.2004.06.008>)
- Björklund, A.; Husfeldt, T.; Koivisto, M. (2009), "Set partitioning via inclusion–exclusion", *SIAM Journal on Computing* **39** (2): 546–563, doi: 10.1137/070683933 (<http://dx.doi.org/10.1137/070683933>)
- Brélaz, D. (1979), "New methods to color the vertices of a graph", *Communications of the ACM* **22** (4): 251–256, doi: 10.1145/359094.359101 (<http://dx.doi.org/10.1145/359094.359101>)
- Brooks, R. L.; Tutte, W. T. (1941), "On colouring the nodes of a network", *Proceedings of the Cambridge Philosophical Society* **37** (2): 194–197, doi: 10.1017/S030500410002168X (<http://dx.doi.org/10.1017/S030500410002168X>)
- de Bruijn, N. G.; Erdős, P. (1951), "A colour problem for infinite graphs and a problem in the theory of relations" (http://www.math-inst.hu/~p_erdos/1951-01.pdf), *Nederl. Akad. Wetensch. Proc. Ser. A* **54**: 371–373 (= *Indag. Math.* **13**)
- Byskov, J.M. (2004), "Enumerating maximal independent sets with applications to graph colouring", *Operations Research Letters* **32** (6): 547–556, doi: 10.1016/j.orl.2004.03.002 (<http://dx.doi.org/10.1016/j.orl.2004.03.002>)
- Chaitin, G. J. (1982), "Register allocation & spilling via graph colouring", *Proc. 1982 SIGPLAN Symposium on Compiler Construction*, pp. 98–105, doi: 10.1145/800230.806984 (<http://dx.doi.org/10.1145/800230.806984>), ISBN 0-89791-074-5
- Cole, R.; Vishkin, U. (1986), "Deterministic coin tossing with applications to optimal parallel list ranking", *Information and Control* **70** (1): 32–53, doi: 10.1016/S0019-9958(86)80023-7 ([http://dx.doi.org/10.1016/S0019-9958\(86\)80023-7](http://dx.doi.org/10.1016/S0019-9958(86)80023-7))
- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. (1990), *Introduction to Algorithms* (1st ed.), The MIT Press
- Dailey, D. P. (1980), "Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete", *Discrete Mathematics* **30** (3): 289–293, doi: 10.1016/0012-365X(80)90236-8 ([http://dx.doi.org/10.1016/0012-365X\(80\)90236-8](http://dx.doi.org/10.1016/0012-365X(80)90236-8))
- Duffy, K.; O'Connell, N.; Sapozhnikov, A. (2008), "Complexity analysis of a decentralised graph colouring algorithm" (http://www.hamilton.ie/ken_duffy/Downloads/cfl.pdf), *Information Processing Letters* **107** (2): 60–63, doi: 10.1016/j.ipl.2008.01.002 (<http://dx.doi.org/10.1016/j.ipl.2008.01.002>)

- Fawcett, B. W. (1978), "On infinite full colourings of graphs", *Can. J. Math.* **XXX**: 455–457
- Fomin, F.V.; Gaspers, S.; Saurabh, S. (2007), "Improved Exact Algorithms for Counting 3- and 4-Colorings", *Proc. 13th Annual International Conference, COCOON 2007, Lecture Notes in Computer Science* **4598**, Springer, pp. 65–74, doi: 10.1007/978-3-540-73545-8_9 (http://dx.doi.org/10.1007/978-3-540-73545-8_9), ISBN 978-3-540-73544-1
- Garey, M. R.; Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, ISBN 0-7167-1045-5
- Garey, M. R.; Johnson, D. S.; Stockmeyer, L. (1974), "Some simplified NP-complete problems" (<http://portal.acm.org/citation.cfm?id=803884>), *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pp. 47–63, doi: 10.1145/800119.803884 (<http://dx.doi.org/10.1145/800119.803884>)
- Goldberg, L. A.; Jerrum, M. (July 2008), "Inapproximability of the Tutte polynomial", *Information and Computation* **206** (7): 908–929, doi: 10.1016/j.ic.2008.04.003 (<http://dx.doi.org/10.1016/j.ic.2008.04.003>)
- Goldberg, A. V.; Plotkin, S. A.; Shannon, G. E. (1988), "Parallel symmetry-breaking in sparse graphs", *SIAM Journal on Discrete Mathematics* **1** (4): 434–446, doi: 10.1137/0401044 (<http://dx.doi.org/10.1137/0401044>)
- Guruswami, V.; Khanna, S. (2000), "On the hardness of 4-coloring a 3-colorable graph", *Proceedings of the 15th Annual IEEE Conference on Computational Complexity*, pp. 188–197, doi: 10.1109/CCC.2000.856749 (<http://dx.doi.org/10.1109/CCC.2000.856749>), ISBN 0-7695-0674-7
- Halldórsson, M. M. (1993), "A still better performance guarantee for approximate graph coloring", *Information Processing Letters* **45**: 19–23, doi: 10.1016/0020-0190(93)90246-6 ([http://dx.doi.org/10.1016/0020-0190\(93\)90246-6](http://dx.doi.org/10.1016/0020-0190(93)90246-6))
- Holyer, I. (1981), "The NP-completeness of edge-coloring", *SIAM Journal on Computing* **10** (4): 718–720, doi: 10.1137/0210055 (<http://dx.doi.org/10.1137/0210055>)
- Crescenzi, P.; Kann, V. (December 1998), "How to find the best approximation results — a follow-up to Garey and Johnson", *ACM SIGACT News* **29** (4): 90, doi: 10.1145/306198.306210 (<http://dx.doi.org/10.1145/306198.306210>)
- Jaeger, F.; Vertigan, D. L.; Welsh, D. J. A. (1990), "On the computational complexity of the Jones and Tutte polynomials", *Mathematical Proceedings of the Cambridge Philosophical Society* **108**: 35–53, doi: 10.1017/S0305004100068936 (<http://dx.doi.org/10.1017/S0305004100068936>)
- Jensen, T. R.; Toft, B. (1995), *Graph Coloring Problems*, Wiley-Interscience, New York, ISBN 0-471-02865-7
- Khot, S. (2001), "Improved inapproximability results for MaxClique, chromatic number and approximate graph coloring", *Proc. 42nd Annual Symposium on Foundations of Computer Science*, pp. 600–609, doi: 10.1109/SFCS.2001.959936 (<http://dx.doi.org/10.1109/SFCS.2001.959936>), ISBN 0-7695-1116-3
- Kubale, M. (2004), *Graph Colorings*, American Mathematical Society, ISBN 0-8218-3458-4
- Kuhn, F. (2009), "Weak graph colorings: distributed algorithms and applications", *Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures*, pp. 138–144, doi: 10.1145/1583991.1584032 (<http://dx.doi.org/10.1145/1583991.1584032>), ISBN 978-1-60558-606-9
- Lawler, E.L. (1976), "A note on the complexity of the chromatic number problem", *Information Processing Letters* **5** (3): 66–67, doi: 10.1016/0020-0190(76)90065-X ([http://dx.doi.org/10.1016/0020-0190\(76\)90065-X](http://dx.doi.org/10.1016/0020-0190(76)90065-X))
- Leith, D.J.; Clifford, P. (2006), "A Self-Managed Distributed Channel Selection Algorithm for WLAN" (<http://www.hamilton.ie/peterc/downloads/rawnet06.pdf>), *Proc. RAWNET 2006, Boston, MA*
- Linial, N. (1992), "Locality in distributed graph algorithms", *SIAM Journal on Computing* **21** (1): 193–201, doi: 10.1137/0221015 (<http://dx.doi.org/10.1137/0221015>)
- van Lint, J. H.; Wilson, R. M. (2001), *A Course in Combinatorics* (2nd ed.), Cambridge University Press, ISBN 0-521-80340-3
- Marx, Dániel (2004), "Graph colouring problems and their applications in scheduling", *Periodica Polytechnica, Electrical Engineering* **48** (1–2), pp. 11–16, CiteSeerX: 10.1.1.95.4268 (<http://citeseerx.ist.psu.edu/viewdoc/>)

- summary?doi=10.1.1.95.4268)
- Mycielski, J. (1955), "Sur le coloriage des graphes" (<http://matwbn.icm.edu.pl/ksiazki/cm/cm3/cm3119.pdf>), *Colloq. Math.* **3**: 161–162.
 - Nešetřil, Jaroslav; Ossona de Mendez, Patrice (2012), "Theorem 3.13", *Sparsity: Graphs, Structures, and Algorithms*, Algorithms and Combinatorics **28**, Heidelberg: Springer, p. 42, doi: 10.1007/978-3-642-27875-4 (<http://dx.doi.org/10.1007/978-3-642-27875-4>), ISBN 978-3-642-27874-7, MR 2920058 (<http://www.ams.org/mathscinet-getitem?mr=2920058>).
 - Panconesi, Alessandro; Rizzi, Romeo (2001), "Some simple distributed algorithms for sparse networks", *Distributed Computing* (Berlin, New York: Springer-Verlag) **14** (2): 97–100, doi: 10.1007/PL00008932 (<http://dx.doi.org/10.1007/PL00008932>), ISSN 0178-2770 (<http://www.worldcat.org/issn/0178-2770>)
 - Sekine, K.; Imai, H.; Tani, S. (1995), "Computing the Tutte polynomial of a graph of moderate size", *Proc. 6th International Symposium on Algorithms and Computation (ISAAC 1995)*, Lecture Notes in Computer Science **1004**, Springer, pp. 224–233, doi: 10.1007/BFb0015427 (<http://dx.doi.org/10.1007/BFb0015427>), ISBN 3-540-60573-8
 - Welsh, D. J. A.; Powell, M. B. (1967), "An upper bound for the chromatic number of a graph and its application to timetabling problems", *The Computer Journal* **10** (1): 85–86, doi: 10.1093/comjnl/10.1.85 (<http://dx.doi.org/10.1093/comjnl/10.1.85>)
 - West, D. B. (1996), *Introduction to Graph Theory*, Prentice-Hall, ISBN 0-13-227828-6
 - Wilf, H. S. (1986), *Algorithms and Complexity*, Prentice-Hall
 - Zuckerman, D. (2007), "Linear degree extractors and the inapproximability of Max Clique and Chromatic Number", *Theory of Computing* **3**: 103–128, doi: 10.4086/toc.2007.v003a006 (<http://dx.doi.org/10.4086/toc.2007.v003a006>)
 - Zykov, A. A. (1949), "О некоторых свойствах линейных комплексов (On some properties of linear complexes)" (<http://mi.mathnet.ru/eng/msb5974>), *Math. Sbornik*. (in Russian), 24(66) (2): 163–188
 - Jensen, Tommy R.; Toft, Bjarne (1995), *Graph Coloring Problems*, John Wiley & Sons, ISBN 9780471028659
 - Normann, Per (2014), *Parallel Graph Coloring* (<http://uu.diva-portal.org/smash/record.jsf?pid=diva2:730761>), DIVA, ISSN 1401-5757 (<http://www.worldcat.org/issn/1401-5757>)

External links

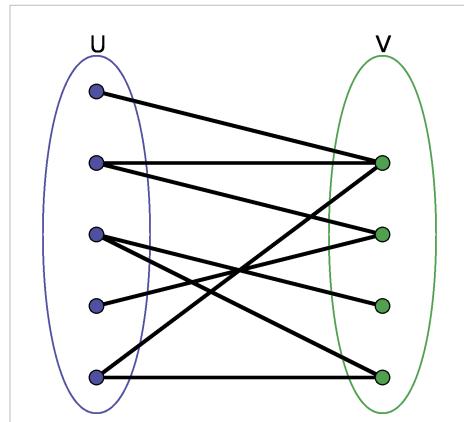
- *Graph Coloring Page* (<http://www.cs.ualberta.ca/~joe/Coloring/index.html>) by Joseph Culberson (graph coloring programs)
- *CoLoRaTiOn* (<http://vispo.com/software>) by Jim Andrews and Mike Fellows is a graph coloring puzzle
- Links to Graph Coloring source codes (http://www.adaptivebox.net/research/bookmark/gcpcodes_link.html)
- Code for efficiently computing Tutte, Chromatic and Flow Polynomials (<http://www.mcs.vuw.ac.nz/~djp/tutte/>) by Gary Haggard, David J. Pearce and Gordon Royle
- Graph Coloring Web Application (<http://graph-coloring.appspot.com/>)

Bipartite graph

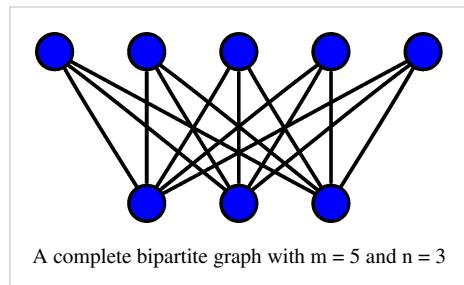
In the mathematical field of graph theory, a **bipartite graph** (or **bigraph**) is a graph whose vertices can be divided into two disjoint sets U and V (that is, U and V are each independent sets) such that every edge connects a vertex in U to one in V . Vertex set U and V are often denoted as *partite sets*. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.

The two sets U and V may be thought of as a coloring of the graph with two colors: if one colors all nodes in U blue, and all nodes in V green, each edge has endpoints of differing colors, as is required in the graph coloring problem. In contrast, such a coloring is impossible in the case of a non-bipartite graph, such as a triangle: after one node is colored blue and another green, the third vertex of the triangle is connected to vertices of both colors, preventing it from being assigned either color.

One often writes $G = (U, V, E)$ to denote a bipartite graph whose partition has the parts U and V , with E denoting the edges of the graph. If a bipartite graph is not connected, it may have more than one bipartition; in this case, the (U, V, E) notation is helpful in specifying one particular bipartition that may be of importance in an application. If $|U| = |V|$, that is, if the two subsets have equal cardinality, then G is called a *balanced* bipartite graph.^[1] If all vertices on the same side of the bipartition have the same degree, then G is called biregular.



Example of a bipartite graph without cycles



A complete bipartite graph with $m = 5$ and $n = 3$

Examples

When modelling relations between two different classes of objects, bipartite graphs very often arise naturally. For instance, a graph of football players and clubs, with an edge between a player and a club if the player has played for that club, is a natural example of an *affiliation network*, a type of bipartite graph used in social network analysis.

Another example where bipartite graphs appear naturally is in the (NP-complete) railway optimization problem, in which the input is a schedule of trains and their stops, and the goal is to find a set of train stations as small as possible such that every train visits at least one of the chosen stations. This problem can be modeled as a dominating set problem in a bipartite graph that has a vertex for each train and each station and an edge for each pair of a station and a train that stops at that station.

More abstract examples include the following:

- Every tree is bipartite.
- Cycle graphs with an even number of vertices are bipartite.
- Every planar graph whose faces all have even length is bipartite.^[1] Special cases of this are grid graphs and squaregraphs, in which every inner face consists of 4 edges and every inner vertex has four or more neighbors.
- The complete bipartite graph on m and n vertices, denoted by $K_{n,m}$ is the bipartite graph $G = (U, V, E)$, where U and V are disjoint sets of size m and n , respectively, and E connects every vertex in U with all vertices in V . It follows that $K_{m,n}$ has mn edges.^[2] Closely related to the complete bipartite graphs are the crown graphs, formed from complete bipartite graphs by removing the edges of a perfect matching.

- Hypercube graphs, partial cubes, and median graphs are bipartite. In these graphs, the vertices may be labeled by bitvectors, in such a way that two vertices are adjacent if and only if the corresponding bitvectors differ in a single position. A bipartition may be formed by separating the vertices whose bitvectors have an even number of ones from the vertices with an odd number of ones. Trees and squaregraphs form examples of median graphs, and every median graph is a partial cube.^[3]

Properties

Characterization

Bipartite graphs may be characterized in several different ways:

- A graph is bipartite if and only if it does not contain an odd cycle.^[4]
- A graph is bipartite if and only if it is 2-colorable, (i.e. its chromatic number is less than or equal to 2).
- The spectrum of a graph is symmetric if and only if it's a bipartite graph.

König's theorem and perfect graphs

In bipartite graphs, the size of minimum vertex cover is equal to the size of the maximum matching; this is König's theorem. An alternative and equivalent form of this theorem is that the size of the maximum independent set plus the size of the maximum matching is equal to the number of vertices. In any graph without isolated vertices the size of the minimum edge cover plus the size of a maximum matching equals the number of vertices. Combining this equality with König's theorem leads to the facts that, in bipartite graphs, the size of the minimum edge cover is equal to the size of the maximum independent set, and the size of the minimum edge cover plus the size of the minimum vertex cover is equal to the number of vertices.

Another class of related results concerns perfect graphs: every bipartite graph, the complement of every bipartite graph, the line graph of every bipartite graph, and the complement of the line graph of every bipartite graph, are all perfect. Perfection of bipartite graphs is easy to see (their chromatic number is two and their maximum clique size is also two) but perfection of the complements of bipartite graphs is less trivial, and is another restatement of König's theorem. This was one of the results that motivated the initial definition of perfect graphs. Perfection of the complements of line graphs of perfect graphs is yet another restatement of König's theorem, and perfection of the line graphs themselves is a restatement of an earlier theorem of König, that every bipartite graph has an edge coloring using a number of colors equal to its maximum degree.

According to the strong perfect graph theorem, the perfect graphs have a forbidden graph characterization resembling that of bipartite graphs: a graph is bipartite if and only if it has no odd cycle as a subgraph, and a graph is perfect if and only if it has no odd cycle or its complement as an induced subgraph. The bipartite graphs, line graphs of bipartite graphs, and their complements form four out of the five basic classes of perfect graphs used in the proof of the strong perfect graph theorem.

Degree

For a vertex, the number of adjacent vertices is called the degree of the vertex and is denoted $\deg(v)$. The *degree sum formula* for a bipartite graph states that

$$\sum_{v \in V} \deg(v) = \sum_{u \in U} \deg(u) = |E|.$$

The degree sequence of a bipartite graph is the pair of lists each containing the degrees of the two partite sets U and V . For example, the complete bipartite graph $K_{3,5}$ has degree sequence $(5, 5, 5), (3, 3, 3, 3, 3)$. Isomorphic bipartite graphs have the same degree sequence. However, the degree sequence does not, in general, uniquely identify a bipartite graph; in some cases, non-isomorphic bipartite graphs may have the same degree sequence.

The bipartite realization problem is the problem of finding a simple bipartite graph with the degree sequence being two given lists of natural numbers. (Trailing zeros may be ignored since they are trivially realized by adding an appropriate number of isolated vertices to the digraph.)

Relation to hypergraphs and directed graphs

The biadjacency matrix of a bipartite graph (U, V, E) is a $(0, 1)$ -matrix of size $|U| \times |V|$ that has a one for each pair of adjacent vertices and a zero for nonadjacent vertices.^[5] Biadjacency matrices may be used to describe equivalences between bipartite graphs, hypergraphs, and directed graphs.

A hypergraph is a combinatorial structure that, like an undirected graph, has vertices and edges, but in which the edges may be arbitrary sets of vertices rather than having to have exactly two endpoints. A bipartite graph (U, V, E) may be used to model a hypergraph in which U is the set of vertices of the hypergraph, V is the set of hyperedges, and E contains an edge from a hypergraph vertex v to a hypergraph edge e exactly when v is one of the endpoints of e . Under this correspondence, the biadjacency matrices of bipartite graphs are exactly the incidence matrices of the corresponding hypergraphs. As a special case of this correspondence between bipartite graphs and hypergraphs, any multigraph (a graph in which there may be two or more edges between the same two vertices) may be interpreted as a hypergraph in which some hyperedges have equal sets of endpoints, and represented by a bipartite graph that does not have multiple adjacencies and in which the vertices on one side of the bipartition all have degree two.

A similar reinterpretation of adjacency matrices may be used to show a one-to-one correspondence between directed graphs (on a given number of labeled vertices, allowing self-loops) and balanced bipartite graphs, with the same number of vertices on both sides of the bipartition. For, the adjacency matrix of a directed graph with n vertices can be any $(0, 1)$ -matrix of size $n \times n$, which can then be reinterpreted as the adjacency matrix of a bipartite graph with n vertices on each side of its bipartition.^[6] In this construction, the bipartite graph is the bipartite double cover of the directed graph.

Algorithms

Testing bipartiteness

It is possible to test whether a graph is bipartite, and to return either a two-coloring (if it is bipartite) or an odd cycle (if it is not) in linear time, using depth-first search. The main idea is to assign to each vertex the color that differs from the color of its parent in the depth-first search tree, assigning colors in a preorder traversal of the depth-first-search tree. This will necessarily provide a two-coloring of the spanning tree consisting of the edges connecting vertices to their parents, but it may not properly color some of the non-tree edges. In a depth-first search tree, one of the two endpoints of every non-tree edge is an ancestor of the other endpoint, and when the depth first search discovers an edge of this type it should check that these two vertices have different colors. If they do not, then the path in the tree from ancestor to descendant, together with the miscolored edge, form an odd cycle, which is returned from the algorithm together with the result that the graph is not bipartite. However, if the algorithm terminates without detecting an odd cycle of this type, then every edge must be properly colored, and the algorithm returns the coloring together with the result that the graph is bipartite.

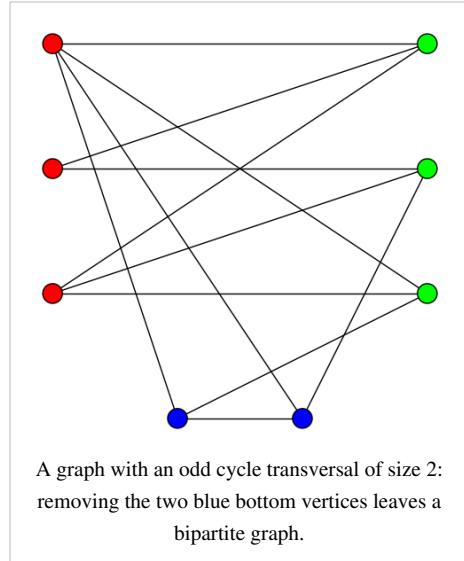
Alternatively, a similar procedure may be used with breadth-first search in place of depth-first search. Again, each node is given the opposite color to its parent in the search tree, in breadth-first order. If, when a vertex is colored, there exists an edge connecting it to a previously-colored vertex with the same color, then this edge together with the paths in the breadth-first search tree connecting its two endpoints to their lowest common ancestor forms an odd cycle. If the algorithm terminates without finding an odd cycle in this way, then it must have found a proper coloring, and can safely conclude that the graph is bipartite.

For the intersection graphs of n line segments or other simple shapes in the Euclidean plane, it is possible to test whether the graph is bipartite and return either a two-coloring or an odd cycle in time $O(n \log n)$, even though the graph itself may have as many as $\Omega(n^2)$ edges.

Odd cycle transversal

Odd cycle transversal is an NP-complete algorithmic problem that asks, given a graph $G = (V, E)$ and a number k , whether there exists a set of k vertices whose removal from G would cause the resulting graph to be bipartite. The problem is fixed-parameter tractable, meaning that there is an algorithm whose running time can be bounded by a polynomial function of the size of the graph multiplied by a larger function of k . More specifically, the time for this algorithm is $O(3^k \cdot mn)$, although this was not stated in that paper, but later given by Hüffner after a new analysis of the algorithm. The name *odd cycle transversal* comes from the fact that a graph is bipartite if and only if it has no odd cycles. Hence, to delete vertices from a graph in order to obtain a bipartite graph, one needs to "hit all odd cycle", or find a so-called odd cycle transversal set. In the illustration, one can observe that every odd cycle in the graph contains the blue (the bottommost) vertices, hence removing those vertices kills all odd cycles and leaves a bipartite graph.

The *edge bipartization* problem is the algorithmic problem of deleting as few edges as possible to make a graph bipartite.



Matching

A matching in a graph is a subset of its edges, no two of which share an endpoint. Polynomial time algorithms are known for many algorithmic problems on matchings, including maximum matching (finding a matching that uses as many edges as possible), maximum weight matching, and stable marriage. In many cases, matching problems are simpler to solve on bipartite graphs than on non-bipartite graphs,^[7] and many matching algorithms such as the Hopcroft–Karp algorithm for maximum cardinality matching work correctly only on bipartite inputs.

As a simple example, suppose that a set P of people are all seeking jobs from among a set of J jobs, with not all people suitable for all jobs. This situation can be modeled as a bipartite graph (P, J, E) where an edge connects each job-seeker with each suitable job.^[8] A perfect matching describes a way of simultaneously satisfying all job-seekers and filling all jobs; the marriage theorem provides a characterization of the bipartite graphs which allow perfect matchings. The National Resident Matching Program applies graph matching methods to solve this problem for U.S. medical student job-seekers and hospital residency jobs.

The Dulmage–Mendelsohn decomposition is a structural decomposition of bipartite graphs that is useful in finding maximum matchings.

Additional applications

Bipartite graphs are extensively used in modern coding theory, especially to decode codewords received from the channel. Factor graphs and Tanner graphs are examples of this. A Tanner graph is a bipartite graph in which the vertices on one side of the bipartition represent digits of a codeword, and the vertices on the other side represent combinations of digits that are expected to sum to zero in a codeword without errors. A factor graph is a closely related belief network used for probabilistic decoding of LDPC and turbo codes.^[9]

In computer science, a Petri net is a mathematical modeling tool used in analysis and simulations of concurrent systems. A system is modeled as a bipartite directed graph with two sets of nodes: A set of "place" nodes that contain resources, and a set of "event" nodes which generate and/or consume resources. There are additional constraints on the nodes and edges that constrain the behavior of the system. Petri nets utilize the properties of bipartite directed graphs and other properties to allow mathematical proofs of the behavior of systems while also allowing easy implementation of simulations of the system.

In projective geometry, Levi graphs are a form of bipartite graph used to model the incidences between points and lines in a configuration. Corresponding to the geometric property of points and lines that every two lines meet in at most one point and every two points be connected with a single line, Levi graphs necessarily do not contain any cycles of length four, so their girth must be six or more.

References

- [1] . This result has sometimes been called the "two color theorem"; Soifer credits it to a famous 1879 paper of Alfred Kempe containing a false proof of the four color theorem.
- [2] , p. 11.
- [3] . See especially Chapter 5, "Partial Cubes", pp. 127–181.
- [4] , Theorem 2.1.3, p. 8. Asratian et al. attribute this characterization to a 1916 paper by Dénes Kőnig.
- [5] , p. 17.
- [6] . Brualdi et al. credit the idea for this equivalence to .
- [7] , p. 463: "Nonbipartite matching problems are more difficult to solve because they do not reduce to standard network flow problems."
- [8] , Application 12.1 Bipartite Personnel Assignment, pp. 463–464.
- [9] , p. 686.

External links

- Information System on Graph Classes and their Inclusions (<http://www.graphclasses.org/>): bipartite graph (http://www.graphclasses.org/classes/gc_69.html)
- Weisstein, Eric W., "Bipartite Graph" (<http://mathworld.wolfram.com/BipartiteGraph.html>), *MathWorld*.

Greedy coloring

In the study of graph coloring problems in mathematics and computer science, a **greedy coloring** is a coloring of the vertices of a graph formed by a greedy algorithm that considers the vertices of the graph in sequence and assigns each vertex its first available color. Greedy colorings do not in general use the minimum number of colors possible; however they have been used in mathematics as a technique for proving other results about colorings and in computer science as a heuristic to find colorings with few colors.

Greed is not always good

A crown graph (a complete bipartite graph $K_{n,n}$, with the edges of a perfect matching removed) is a particularly bad case for greedy coloring: if the vertex ordering places two vertices consecutively whenever they belong to one of the pairs of the removed matching, then a greedy coloring will use n colors, while the optimal number of colors for this graph is two. There also exist graphs such that with high probability a randomly chosen vertex ordering leads to a number of colors much larger than the minimum. Therefore, it is of some importance in greedy coloring to choose the vertex ordering carefully.

It is NP-complete to determine, for a given graph G and number k , whether there exists an ordering of the vertices of G that forces the greedy algorithm to use k or more colors. In particular, this means that it is difficult to find the worst ordering for G .

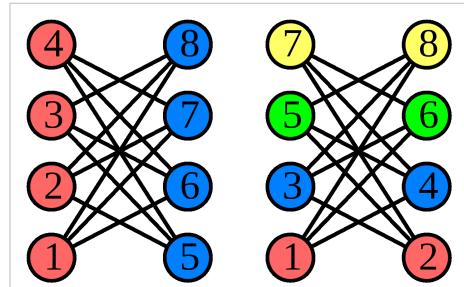
Optimal ordering

The vertices of any graph may always be ordered in such a way that the greedy algorithm produces an optimal coloring. For, given any optimal coloring in which the smallest color set is maximal, the second color set is maximal with respect to the first color set, etc., one may order the vertices by their colors. Then when one uses a greedy algorithm with this order, the resulting coloring is automatically optimal. More strongly, perfectly orderable graphs (which include chordal graphs, comparability graphs, and distance-hereditary graphs) have an ordering that is optimal both for the graph itself and for all of its induced subgraphs. However, finding an optimal ordering for an arbitrary graph is NP-hard (because it could be used to solve the NP-complete graph coloring problem), and recognizing perfectly orderable graphs is also NP-complete. For this reason, heuristics have been used that attempt to reduce the number of colors while not guaranteeing an optimal number of colors.

Heuristic ordering strategies

A commonly used ordering for greedy coloring is to choose a vertex v of minimum degree, order the remaining vertices, and then place v last in the ordering. If every subgraph of a graph G contains a vertex of degree at most d , then the greedy coloring for this ordering will use at most $d + 1$ colors.^[1] The smallest such d is commonly known as the degeneracy of the graph.

For a graph of maximum degree Δ , any greedy coloring will use at most $\Delta + 1$ colors. Brooks' theorem states that with two exceptions (cliques and odd cycles) at most Δ colors are needed. One proof of Brooks' theorem involves finding a vertex ordering in which the first two vertices are adjacent to the final vertex but not adjacent to each other, and each subsequent vertex has at least one earlier neighbor. For an ordering with this property, the greedy coloring algorithm uses at most Δ colors.



Two greedy colorings of the same graph using different vertex orders. The right example generalises to 2-colorable graphs with n vertices, where the greedy algorithm expends $n/2$ colors.

Alternative color selection schemes

It is possible to define a greedy coloring algorithm in which the vertices of the given graph are colored in a given sequence but in which the color chosen for each vertex is not necessarily the first available color; alternative color selection strategies have been studied within the framework of online algorithms. In the online graph-coloring problem, vertices of a graph are presented one at a time in an arbitrary order to a coloring algorithm; the algorithm must choose a color for each vertex, based only on the colors of and adjacencies among already-processed vertices. In this context, one measures the quality of a color selection strategy by its competitive ratio, the ratio between the number of colors it uses and the optimal number of colors for the given graph.

If no additional restrictions on the graph are given, the optimal competitive ratio is only slightly sublinear.^[2] However, for interval graphs, a constant competitive ratio is possible, while for bipartite graphs and sparse graphs a logarithmic ratio can be achieved. Indeed, for sparse graphs, the standard greedy coloring strategy of choosing the first available color achieves this competitive ratio, and it is possible to prove a matching lower bound on the competitive ratio of any online coloring algorithm.

Notes

[1] ; ; ; ..

[2] ; Sz, .

References

- Chvátal, Václav (1984), "Perfectly orderable graphs", in Berge, Claude; Chvátal, Václav, *Topics in Perfect Graphs*, Annals of Discrete Mathematics **21**, Amsterdam: North-Holland, pp. 63–68. As cited by Maffray (2003).
- Irani, Sandy (1994), "Coloring inductive graphs on-line", *Algorithmica* **11** (1): 53–72, doi: 10.1007/BF01294263 (<http://dx.doi.org/10.1007/BF01294263>).
- Kierstead, H. A.; Trotter, W. A. (1981), "An extremal problem in recursive combinatorics", *Congress. Numer.* **33**: 143–153. As cited by Irani (1994).
- Kučera, Luděk (1991), "The greedy coloring is a bad probabilistic algorithm", *Journal of Algorithms* **12** (4): 674–684, doi: 10.1016/0196-6774(91)90040-6 ([http://dx.doi.org/10.1016/0196-6774\(91\)90040-6](http://dx.doi.org/10.1016/0196-6774(91)90040-6)).
- Johnson, D. S. (1979), "Worst case behavior of graph coloring algorithms", *Proc. 5th Southeastern Conf. Combinatorics, Graph Theory and Computation*, Winnipeg: Utilitas Mathematica, pp. 513–527. As cited by Maffray (2003).
- Lovász, L. (1975), "Three short proofs in graph theory", *Journal of Combinatorial Theory, Series B* **19** (3): 269–271, doi: 10.1016/0095-8956(75)90089-1 ([http://dx.doi.org/10.1016/0095-8956\(75\)90089-1](http://dx.doi.org/10.1016/0095-8956(75)90089-1)).
- Lovász, L.; Saks, M. E.; Trotter, W. A. (1989), "An on-line graph coloring algorithm with sublinear performance ratio", *Discrete Mathematics* **75** (1–3): 319–325, doi: 10.1016/0012-365X(89)90096-4 ([http://dx.doi.org/10.1016/0012-365X\(89\)90096-4](http://dx.doi.org/10.1016/0012-365X(89)90096-4)).
- Maffray, Frédéric (2003), "On the coloration of perfect graphs", in Reed, Bruce A.; Sales, Cláudia L., *Recent Advances in Algorithms and Combinatorics*, CMS Books in Mathematics **11**, Springer-Verlag, pp. 65–84, doi: 10.1007/0-387-22444-0_3 (http://dx.doi.org/10.1007/0-387-22444-0_3), ISBN 0-387-95434-1.
- Middendorf, Matthias; Pfeiffer, Frank (1990), "On the complexity of recognizing perfectly orderable graphs", *Discrete Mathematics* **80** (3): 327–333, doi: 10.1016/0012-365X(90)90251-C ([http://dx.doi.org/10.1016/0012-365X\(90\)90251-C](http://dx.doi.org/10.1016/0012-365X(90)90251-C)).
- Sysło, Maciej M. (1989), "Sequential coloring versus Welsh-Powell bound", *Discrete Mathematics* **74** (1–2): 241–243, doi: 10.1016/0012-365X(89)90212-4 ([http://dx.doi.org/10.1016/0012-365X\(89\)90212-4](http://dx.doi.org/10.1016/0012-365X(89)90212-4)).
- Vishwanathan, S. (1990), "Randomized online graph coloring", *Proc. 31st IEEE Symp. Foundations of Computer Science (FOCS '90)* **2**, pp. 464–469, doi: 10.1109/FSCS.1990.89567 (<http://dx.doi.org/10.1109/FSCS.1990.89567>), ISBN 0-8186-2082-X.

- Welsh, D. J. A.; Powell, M. B. (1967), "An upper bound for the chromatic number of a graph and its application to timetabling problems", *The Computer Journal* **10** (1): 85–86, doi: 10.1093/comjnl/10.1.85 (<http://dx.doi.org/10.1093/comjnl/10.1.85>).
- Zaker, Manouchehr (2006), "Results on the Grundy chromatic number of graphs", *Discrete Mathematics* **306** (2–3): 3166–3173, doi: 10.1016/j.disc.2005.06.044 (<http://dx.doi.org/10.1016/j.disc.2005.06.044>).

Application: Register allocation

In compiler optimization, **register allocation** is the process of assigning a large number of target program variables onto a small number of CPU registers. Register allocation can happen over a basic block (*local register allocation*), over a whole function/procedure (*global register allocation*), or across function boundaries traversed via call-graph (*interprocedural register allocation*). When done per function/procedure the calling convention may require insertion of save/restore around each call-site.

Introduction

In many programming languages, the programmer has the illusion of allocating arbitrarily many variables. However, during compilation, the compiler must decide how to allocate these variables to a small, finite set of registers. Not all variables are in use (or "live") at the same time, so some registers may be assigned to more than one variable. However, two variables in use at the same time cannot be assigned to the same register without corrupting its value. Variables which cannot be assigned to some register must be kept in RAM and loaded in/out for every read/write, a process called *spilling*. Accessing RAM is significantly slower than accessing registers and slows down the execution speed of the compiled program, so an optimizing compiler aims to assign as many variables to registers as possible. *Register pressure* is the term used when there are fewer hardware registers available than would have been optimal; higher pressure usually means that more spills and reloads are needed.

In addition, programs can be further optimized by assigning the same register to a source and destination of a `move` instruction whenever possible. This is especially important if the compiler is using other optimizations such as SSA analysis, which artificially generates additional `move` instructions in the intermediate code.

Isomorphism to graph colorability

Through liveness analysis, compilers can determine which sets of variables are live at the same time, as well as variables which are involved in `move` instructions. Using this information, the compiler can construct a graph such that every vertex represents a unique variable in the program. *Interference edges* connect pairs of vertices which are live at the same time, and *preference edges* connect pairs of vertices which are involved in move instructions. Register allocation can then be reduced to the problem of K-coloring the resulting graph, where K is the number of registers available on the target architecture. No two vertices sharing an interference edge may be assigned the same color, and vertices sharing a preference edge should be assigned the same color if possible. Some of the vertices may be precolored to begin with, representing variables which must be kept in certain registers due to calling conventions or communication between modules. As graph coloring in general is NP-complete, so is register allocation. However, good algorithms exist which balance performance with quality of compiled code.

It may be the case that the graph coloring algorithm fails to find a coloring of the interference graph. In this case, some of the variables must be spilled to memory in order to enable the remaining variables to be allocated to registers. This may be accomplished by a recursive search that tries spilling one variable and then recursively either colors the remaining set of variables or continues spilling recursively until all remaining unspilled variables can be colored and assigned to registers.

Spilling

In most register allocators, each variable is assigned to either a CPU register or to main memory. The advantage of using a register is speed. Computers have a limited number of registers, so not all variables can be assigned to registers. A "spilled variable" is a variable in main memory rather than in a CPU register. The operation of moving a variable from a register to memory is called *spilling*, while the reverse operation of moving a variable from memory to a register is called *filling*. For example, a 32-bit variable spilled to memory gets 32 bits of stack space allocated and all references to the variable are then to that memory. Such a variable has a much slower processing speed than a variable in a register. When deciding which variables to spill, multiple factors are considered: execution time, code space, data space.

Iterated Register Coalescing

Register allocators have several types, with Iterated Register Coalescing (IRC) being a more common one. IRC was invented by LAL George and Andrew Appel in 1996, building on earlier work by Gregory Chaitin. IRC works based on a few principles. First, if there are any non-move related vertices in the graph with degree less than K the graph can be simplified by removing those vertices, since once those vertices are added back in it is guaranteed that a color can be found for them (simplification). Second, two vertices sharing a preference edge whose adjacency sets combined have a degree less than K can be combined into a single vertex, by the same reasoning (coalescing). If neither of the two steps can simplify the graph, simplification can be run again on move-related vertices (freezing). Finally, if nothing else works, vertices can be marked for potential spilling and removed from the graph (spill). Since all of these steps reduce the degrees of vertices in the graph, vertices may transform from being high-degree (degree > K) to low-degree during the algorithm, enabling them to be simplified or coalesced. Thus, the stages of the algorithm are iterated to ensure aggressive simplification and coalescing. The pseudo-code is thus:

```

function IRC_color g K :
repeat
    if  $\exists v$  s.t. !moveRelated(v)  $\wedge$  degree(v) < K then simplify v
    else if  $\exists e$  s.t. cardinality(neighbors(first e)  $\cup$  neighbors(second e)) < K then coalesce e
    else if  $\exists v$  s.t. moveRelated(v) then deletePreferenceEdges v
    else if  $\exists v$  s.t. !precolored(v) then spill v
    else return
loop

```

The coalescing done in IRC is conservative, because aggressive coalescing may introduce spills into the graph. However, additional coalescing heuristics such as George coalescing may coalesce more vertices while still ensuring that no additional spills are added. Work-lists are used in the algorithm to ensure that each iteration of IRC requires sub-quadratic time.

Recent developments

Graph coloring allocators produce efficient code, but their allocation time is high. In cases of static compilation, allocation time is not a significant concern. In cases of dynamic compilation, such as just-in-time (JIT) compilers, fast register allocation is important. An efficient technique proposed by Poletto and Sarkar is linear scan allocation [1]. This technique requires only a single pass over the list of variable live ranges. Ranges with short lifetimes are assigned to registers, whereas those with long lifetimes tend to be spilled, or reside in memory. The results are on average only 12% less efficient than graph coloring allocators.

The linear scan algorithm follows:

1. Perform dataflow analysis to gather liveness information. Keep track of all variables' live intervals, the interval when a variable is live, in a list sorted in order of increasing start point (note that this ordering is free if the list is built when computing liveness.) We consider variables and their intervals to be interchangeable in this algorithm.
2. Iterate through liveness start points and allocate a register from the available register pool to each live variable.
3. At each step maintain a list of active intervals sorted by the end point of the live intervals. (Note that insertion sort into a balanced binary tree can be used to maintain this list at linear cost.) Remove any expired intervals from the active list and free the expired interval's register to the available register pool.
4. In the case where the active list is size R we cannot allocate a register. In this case add the current interval to the active pool without allocating a register. Spill the interval from the active list with the furthest end point. Assign the register from the spilled interval to the current interval or, if the current interval is the one spilled, do not change register assignments.

Cooper and Dasgupta recently developed a "lossy" Chaitin-Briggs graph coloring algorithm suitable for use in a JIT.^[2] The "lossy" moniker refers to the imprecision the algorithm introduces into the interference graph. This optimization reduces the costly graph building step of Chaitin-Briggs making it suitable for runtime compilation. Experiments indicate that this lossy register allocator outperforms linear scan on the majority of tests used.

"Optimal" register allocation algorithms based on Integer Programming have been developed by Goodwin and Wilken for regular architectures. These algorithms have been extended to irregular architectures by Kong and Wilken.

While the worst case execution time is exponential, the experimental results show that the actual time is typically of order $O(n^{2.5})$ of the number of constraints n .^[3]

The possibility of doing register allocation on SSA-form programs is a focus of recent research.^[4] The interference graphs of SSA-form programs are chordal, and as such, they can be colored in polynomial time. To clarify the sources of NP-completeness, recent research has examined register allocation in a broader context.^[5]

References

- [1] <http://www.cs.ucla.edu/~palsberg/course/cs132/linearscan.pdf>
- [2] Cooper, Dasgupta, "Tailoring Graph-coloring Register Allocation For Runtime Compilation", <http://llvm.org/pubs/2006-04-04-CGO-GraphColoring.html>
- [3] Kong, Wilken, "Precise Register Allocation for Irregular Architectures", http://www.ece.ucdavis.edu/cerl/cerl_arch/irreg.pdf
- [4] Brisk, Hack, Palsberg, Pereira, Rastello, "SSA-Based Register Allocation", ESWEK Tutorial <http://thedude.cc.gt.atl.ga.us/tutorials/1/>
- [5] ; also <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.130.7256> Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing, 2006, pages 2–4.

Covering and domination

Vertex cover

In the mathematical discipline of graph theory, a **vertex cover** (sometimes **node cover**) of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. The problem of finding a **minimum vertex cover** is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm. Its decision version, the **vertex cover problem**, was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in computational complexity theory. Furthermore, the vertex cover problem is fixed-parameter tractable and a central problem in parameterized complexity theory.

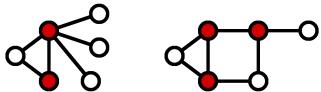
The minimum vertex cover problem can be formulated as a half-integral linear program whose dual linear program is the maximum matching problem.

Definition

Formally, a vertex-cover of an undirected graph $G = (V, E)$ is a subset V' of V such that if edge (u, v) is an edge of G , then u is in V' , v is in V' , or both. The set V' is said to "cover" the edges of G . The following figure shows examples of vertex covers in two graphs (and the set V' is marked with red).



A minimum vertex cover is a vertex cover of smallest possible size. The vertex cover number τ is the size of a minimum vertex cover. The following figure shows examples of minimum vertex covers in the previous graphs.



Examples

- The set of all vertices is a vertex cover.
- The endpoints of any maximal matching form a vertex cover.
- The complete bipartite graph $K_{m,n}$ has a minimum vertex cover of size $\tau(K_{m,n}) = \min(m, n)$.

Properties

- A set of vertices is a vertex cover, if and only if its complement is an independent set. An immediate consequence is:
- The number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set (Gallai 1959).

Computational problem

The **minimum vertex cover problem** is the optimization problem of finding a smallest vertex cover in a given graph.

INSTANCE: Graph G

OUTPUT: Smallest number k such that G has a vertex cover of size k .

If the problem is stated as a decision problem, it is called the **vertex cover problem**:

INSTANCE: Graph G and positive integer k .

QUESTION: Does G have a vertex cover of size at most k ?

The vertex cover problem is an NP-complete problem: it was one of Karp's 21 NP-complete problems. It is often used in computational complexity theory as a starting point for NP-hardness proofs.

ILP formulation

Assume that every vertex has an associated cost of $c(v) \geq 0$. The (weighted) minimum vertex cover problem can be formulated as the following integer linear program (ILP).

$$\begin{aligned} & \text{minimize} && \sum_{v \in V} c(v)x_v && \text{(minimize the total cost)} \\ & \text{subject to} && x_u + x_v \geq 1 \quad \text{for all } \{u, v\} \in E && \text{(cover every edge of the graph)} \\ & && x_v \in \{0, 1\} && \text{for all } v \in V. \end{aligned}$$

(every vertex is either in the vertex cover or not)

This ILP belongs to the more general class of ILPs for covering problems. The integrality gap of this ILP is 2, so its relaxation gives a factor- 2 approximation algorithm for the minimum vertex cover problem. Furthermore, the linear programming relaxation of that ILP is *half-integral*, that is, there exists an optimal solution for which each entry x_v is either 0, 1/2, or 1.

Exact evaluation

The decision variant of the vertex cover problem is NP-complete, which means it is unlikely that there is an efficient algorithm to solve it exactly. NP-completeness can be proven by reduction from 3-satisfiability or, as Karp did, by reduction from the clique problem. Vertex cover remains NP-complete even in cubic graphs and even in planar graphs of degree at most 3.^[1]

For bipartite graphs, the equivalence between vertex cover and maximum matching described by König's theorem allows the bipartite vertex cover problem to be solved in polynomial time.

For tree graphs, a greedy algorithm finds a minimal vertex cover in polynomial time.

Fixed-parameter tractability

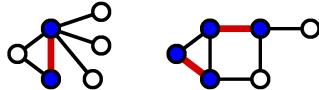
An exhaustive search algorithm can solve the problem in time $2^k n^{O(1)}$. Vertex cover is therefore fixed-parameter tractable, and if we are only interested in small k , we can solve the problem in polynomial time. One algorithmic technique that works here is called *bounded search tree algorithm*, and its idea is to repeatedly choose some vertex and recursively branch, with two cases at each step: place either the current vertex or all its neighbours into the vertex cover. The algorithm for solving vertex cover that achieves the best asymptotic dependence on the parameter runs in time $1.2738^k \cdot n^{O(1)}$. Under reasonable complexity-theoretic assumptions, namely the exponential time hypothesis, this running time cannot be improved to $2^{o(k)} n^{O(1)}$.

However, for planar graphs, and more generally, for graphs excluding some fixed graph as a minor, a vertex cover of size k can be found in time $2^{O(\sqrt{k})} n^{O(1)}$, i.e., the problem is subexponential fixed-parameter tractable. This algorithm is again optimal, in the sense that, under the exponential time hypothesis, no algorithm can solve vertex

cover on planar graphs in time $2^{o(\sqrt{k})} n^{O(1)}$.

Approximate evaluation

One can find a factor-2 approximation by repeatedly taking *both* endpoints of an edge into the vertex cover, then removing them from the graph. Put otherwise, we find a maximal matching M with a greedy algorithm and construct a vertex cover C that consists of all endpoints of the edges in M . In the following figure, a maximal matching M is marked with red, and the vertex cover C is marked with blue.



The set C constructed this way is a vertex cover: suppose that an edge e is not covered by C ; then $M \cup \{e\}$ is a matching and $e \notin M$, which is a contradiction with the assumption that M is maximal. Furthermore, if $e = \{u, v\} \in M$, then any vertex cover – including an optimal vertex cover – must contain u or v (or both); otherwise the edge e is not covered. That is, an optimal cover contains at least *one* endpoint of each edge in M ; in total, the set C is at most 2 times as large as the optimal vertex cover.

This simple algorithm was discovered independently by Fanica Gavril and Mihalis Yannakakis.^[2]

More involved techniques show that there are approximation algorithms with a slightly better approximation factor. For example, an approximation algorithm with an approximation factor of $2 - \Theta\left(1/\sqrt{\log |V|}\right)$ is known.

Inapproximability

No better constant-factor approximation algorithm than the above one is known. The minimum vertex cover problem is APX-complete, that is, it cannot be approximated arbitrarily well unless $\mathbf{P} = \mathbf{NP}$. Using techniques from the PCP theorem, Dinur and Safra proved in 2005 that minimum vertex cover cannot be approximated within a factor of 1.3606 for any sufficiently large vertex degree unless $\mathbf{P} = \mathbf{NP}$. Moreover, if the unique games conjecture is true then minimum vertex cover cannot be approximated within any constant factor better than 2.

Although finding the minimum-size vertex cover is equivalent to finding the maximum-size independent set, as described above, the two problems are not equivalent in an approximation-preserving way: The Independent Set problem has *no* constant-factor approximation unless $\mathbf{P} = \mathbf{NP}$.

Vertex cover in hypergraphs

Given a collection of sets, a set which intersects all sets in the collection in at least one element is called a **hitting set**, and a simple NP-hard problem is to find a hitting set of smallest size or **minimum hitting set**. By mapping the sets in the collection onto hyperedges, this can be understood as a natural generalization of the vertex cover problem to hypergraphs which is often just called **vertex cover for hypergraphs** and, in a more combinatorial context, **transversal**. The notions of hitting set and set cover are equivalent.

Formally, let $H = (V, E)$ be a hypergraph with vertex set V and hyperedge set E . Then a set $S \subseteq V$ is called *hitting set* of H if, for all edges $e \in E$, it holds $S \cap e \neq \emptyset$. The computational problems *minimum hitting set* and *hitting set* are defined as in the case of graphs. Note that we get back the case of vertex covers for simple graphs if the maximum size of the hyperedges is 2.

If that size is restricted to d , the problem of finding a minimum d -hitting set permits a d -approximation algorithm. Assuming the unique games conjecture, this is the best constant-factor algorithm that is possible and otherwise there is the possibility of improving the approximation to $d - 1$.

Fixed-parameter tractability

For the hitting set problem, different parameterizations make sense. The hitting set problem is $W[2]$ -complete for the parameter OPT , that is, it is unlikely that there is an algorithm that runs in time $f(\text{OPT})n^{O(1)}$ where OPT is the cardinality of the smallest hitting set. The hitting set problem is fixed-parameter tractable for the parameter $\text{OPT} + d$, where d is the size of the largest edge of the hypergraph. More specifically, there is an algorithm for hitting set that runs in time $d^{\text{OPT}} n^{O(1)}$.

Hitting set and set cover

The hitting set problem is equivalent to the set cover problem: An instance of set cover can be viewed as an arbitrary bipartite graph, with sets represented by vertices on the left, elements of the universe represented by vertices on the right, and edges representing the inclusion of elements in sets. The task is then to find a minimum cardinality subset of left-vertices which covers all of the right-vertices. In the hitting set problem, the objective is to cover the left-vertices using a minimum subset of the right vertices. Converting from one problem to the other is therefore achieved by interchanging the two sets of vertices.

Applications

An example of a practical application involving the hitting set problem arises in efficient dynamic detection of race conditions. In this case, each time global memory is written, the current thread and set of locks held by that thread are stored. Under lockset-based detection, if later another thread writes to that location and there is *not* a race, it must be because it holds at least one lock in common with each of the previous writes. Thus the size of the hitting set represents the minimum lock set size to be race-free. This is useful in eliminating redundant write events, since large lock sets are considered unlikely in practice.

Notes

[1] ; , pp. 190 and 195.

[2] , p. 432, mentions both Gavril and Yannakakis. , p. 134, cites Gavril.

References

- Chen, Jianer; Kanj, Iyad A.; Xia, Ge (2006). "Improved Parameterized Upper Bounds for Vertex Cover". *Mfcs 2006*. Lecture Notes in Computer Science **4162**: 238–249. doi: 10.1007/11821069_21 (http://dx.doi.org/10.1007/11821069_21). ISBN 978-3-540-37791-7.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms*. Cambridge, Mass.: MIT Press and McGraw-Hill. pp. 1024–1027. ISBN 0-262-03293-7.
- Demaine, Erik; Fomin, Fedor V.; Hajiaghayi, Mohammad Taghi; Thilikos, Dimitrios M. (2005). "Subexponential parameterized algorithms on bounded-genus graphs and H-minor-free graphs" (http://erikdemaine.org/papers/HMinorFree_JACM/). *Journal of the ACM* **52** (6): 866–893. doi: 10.1145/1101821.1101823 (<http://dx.doi.org/10.1145/1101821.1101823>). Retrieved 2010-03-05.
- Dinur, Irit; Safra, Samuel (2005). "On the hardness of approximating minimum vertex cover". *Annals of Mathematics* **162** (1): 439–485. doi: 10.4007/annals.2005.162.439 (<http://dx.doi.org/10.4007/annals.2005.162.439>). CiteSeerX: 10.1.1.125.334 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.125.334>).
- Flum, Jörg; Grohe, Martin (2006). *Parameterized Complexity Theory* (<http://www.springer.com/east/home/generic/search/results?SGWID=5-40109-22-141358322-0>). Springer. ISBN 978-3-540-29952-3. Retrieved 2010-03-05.
- Garey, Michael R.; Johnson, David S. (1977). "The rectilinear Steiner tree problem is NP-complete". *SIAM Journal on Applied Mathematics* **32** (4): 826–834. doi: 10.1137/0132071 (<http://dx.doi.org/10.1137/0132071>).

- Garey, Michael R.; Johnson, David S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5. A1.1: GT1, pg.190.
- Garey, Michael R.; Johnson, David S.; Stockmeyer, Larry (1974). "Some simplified NP-complete problems" (<http://portal.acm.org/citation.cfm?id=803884>). *Proceedings of the sixth annual ACM symposium on Theory of computing*. pp. 47–63. doi: 10.1145/800119.803884 (<http://dx.doi.org/10.1145/800119.803884>).
- Gallai, Tibor "Über extreme Punkt- und Kantenmengen." *Ann. Univ. Sci. Budapest, Eötvös Sect. Math.* **2**, 133-138, 1959.
- Karakostas, George (2004). "A better approximation ratio for the Vertex Cover problem". *ECCC TR04-084* (<http://eccc.uni-trier.de/report/2004/084/>).
- Khot, Subhash; Regev, Oded (2008). "Vertex cover might be hard to approximate to within $2-\epsilon$ ". *Journal of Computer and System Sciences* **74** (3): 335–349. doi: 10.1016/j.jcss.2007.06.019 (<http://dx.doi.org/10.1016/j.jcss.2007.06.019>).
- O'Callahan, Robert; Choi, Jong-Deok (2003). *Hybrid dynamic data race detection* (<http://portal.acm.org/citation.cfm?id=781528>). "Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP 2003) and workshop on partial evaluation and semantics-based program manipulation (PEPM 2003)". *ACM SIGPLAN Notices* **38** (10): 167–178. doi: 10.1145/966049.781528 (<http://dx.doi.org/10.1145/966049.781528>).
- Papadimitriou, Christos H.; Steiglitz, Kenneth (1998). *Combinatorial Optimization: Algorithms and Complexity*. Dover.
- Vazirani, Vijay V. (2001). *Approximation Algorithms*. Springer-Verlag. ISBN 3-540-65367-8.

External links

- Weisstein, Eric W., "Vertex Cover" (<http://mathworld.wolfram.com/VertexCover.html>), *MathWorld*.
- Weisstein, Eric W., "Minimum Vertex Cover" (<http://mathworld.wolfram.com/MinimumVertexCover.html>), *MathWorld*.
- Weisstein, Eric W., "Vertex Cover Number" (<http://mathworld.wolfram.com/VertexCoverNumber.html>), *MathWorld*.

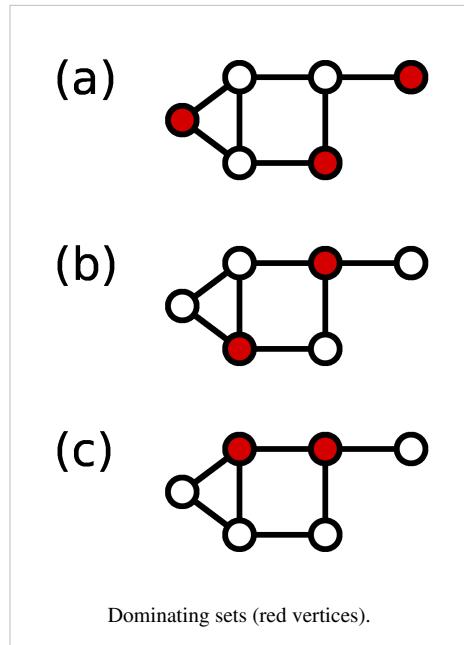
Dominating set

For Dominator in control flow graphs, see Dominator (graph theory).

In graph theory, a **dominating set** for a graph $G = (V, E)$ is a subset D of V such that every vertex not in D is adjacent to at least one member of D . The **domination number** $\gamma(G)$ is the number of vertices in a smallest dominating set for G .

The **dominating set problem** concerns testing whether $\gamma(G) \leq K$ for a given graph G and input K ; it is a classical NP-complete decision problem in computational complexity theory (Garey & Johnson 1979). Therefore it is believed that there is no efficient algorithm that finds a smallest dominating set for a given graph.

Figures (a)–(c) on the right show three examples of dominating sets for a graph. In each example, each white vertex is adjacent to at least one red vertex, and it is said that the white vertex is *dominated* by the red vertex. The domination number of this graph is 2: the examples (b) and (c) show that there is a dominating set with 2 vertices, and it can be checked that there is no dominating set with only 1 vertex for this graph.



History

As Hedetniemi & Laskar (1990) note, the domination problem was studied from the 1950s onwards, but the rate of research on domination significantly increased in the mid-1970s. Their bibliography lists over 300 papers related to domination in graphs.

Bounds

Let G be a graph with $n \geq 1$ vertices and let Δ be the maximum degree of the graph. The following bounds on $\gamma(G)$ are known (Haynes, Hedetniemi & Slater 1998a, Chapter 2):

- One vertex can dominate at most Δ other vertices; therefore $\gamma(G) \geq n/(1 + \Delta)$.
- The set of all vertices is a dominating set in any graph; therefore $\gamma(G) \leq n$.
- If there are no isolated vertices in G , then there are two disjoint dominating sets in G ; see domatic partition for details. Therefore in any graph without isolated vertices it holds that $\gamma(G) \leq n/2$.

Independent domination

Dominating sets are closely related to independent sets: an independent set is also a dominating set if and only if it is a maximal independent set, so any maximal independent set in a graph is necessarily also a minimal dominating set. Thus, the smallest maximal independent set is also the smallest independent dominating set. The **independent domination number** $i(G)$ of a graph G is the size of the smallest independent dominating set (or, equivalently, the size of the smallest maximal independent set).

The minimum dominating set in a graph will not necessarily be independent, but the size of a minimum dominating set is always less than or equal to the size of a minimum maximal independent set, that is, $\gamma(G) \leq i(G)$.

There are graph families in which a minimum maximal independent set is a minimum dominating set. For example, Allan & Laskar (1978) show that $\gamma(G) = i(G)$ if G is a claw-free graph.

A graph G is called a **domination-perfect graph** if $\gamma(H) = i(H)$ in every induced subgraph H of G . Since an induced subgraph of a claw-free graph is claw-free, it follows that every claw-free graphs is also domination-perfect (Faudree, Flandrin & Ryjáček 1997).

Examples

Figures (a) and (b) are independent dominating sets, while figure (c) illustrates a dominating set that is not an independent set.

For any graph G , its line graph $L(G)$ is claw-free, and hence a minimum maximal independent set in $L(G)$ is also a minimum dominating set in $L(G)$. An independent set in $L(G)$ corresponds to a matching in G , and a dominating set in $L(G)$ corresponds to an edge dominating set in G . Therefore a minimum maximal matching has the same size as a minimum edge dominating set.

Algorithms and computational complexity

There exists a pair of polynomial-time L-reductions between the minimum dominating set problem and the set cover problem (Kann 1992, pp. 108–109). These reductions (see below) show that an efficient algorithm for the minimum dominating set problem would provide an efficient algorithm for the set cover problem and vice versa. Moreover, the reductions preserve the approximation ratio: for any α , a polynomial-time α -approximation algorithm for minimum dominating sets would provide a polynomial-time α -approximation algorithm for the set cover problem and vice versa.

The set cover problem is a well-known NP-hard problem – the decision version of set covering was one of Karp's 21 NP-complete problems, which were shown to be NP-complete already in 1972. Hence the reductions show that the dominating set problem is NP-hard as well.

The approximability of set covering is also well understood: a logarithmic approximation factor can be found by using a simple greedy algorithm, and finding a sublogarithmic approximation factor is NP-hard. More specifically, the greedy algorithm provides a factor $1 + \log |V|$ approximation of a minimum dominating set, and Raz & Safra (1997) show that no algorithm can achieve an approximation factor better than $c \log |V|$ for some $c > 0$ unless $P = NP$.

L-reductions

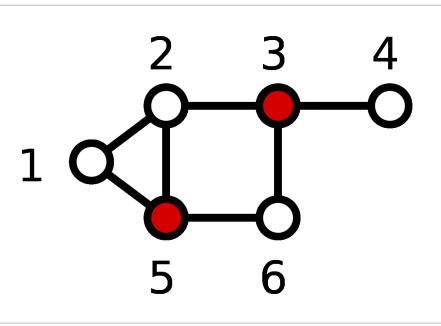
The following pair of reductions (Kann 1992, pp. 108–109) shows that the minimum dominating set problem and the set cover problem are equivalent under L-reductions: given an instance of one problem, we can construct an equivalent instance of the other problem.

From dominating set to set covering. Given a graph $G = (V, E)$ with $V = \{1, 2, \dots, n\}$, construct a set cover instance (S, U) as follows: the universe U is V , and the family of subsets is $S = \{S_1, S_2, \dots, S_n\}$ such that S_v consists of the vertex v and all vertices adjacent to v in G .

Now if D is a dominating set for G , then $C = \{S_v : v \in D\}$ is a feasible solution of the set cover problem, with $|C| = |D|$. Conversely, if $C = \{S_v : v \in D\}$ is a feasible solution of the set cover problem, then D is a dominating set for G , with $|D| = |C|$.

Hence the size of a minimum dominating set for G equals the size of a minimum set cover for (S, U) . Furthermore, there is a simple algorithm that maps a dominating set to a set cover of the same size and vice versa. In particular, an efficient α -approximation algorithm for set covering provides an efficient α -approximation algorithm for minimum dominating sets.

For example, given the graph G shown on the right, we construct a set cover instance with the universe $U = \{1, 2, \dots, 6\}$ and the subsets $S_1 = \{1, 2, 5\}$, $S_2 = \{1, 2, 3, 5\}$, $S_3 = \{2, 3, 4, 6\}$, $S_4 = \{3, 4\}$, $S_5 = \{1, 2, 5, 6\}$, and $S_6 = \{3, 5, 6\}$. In this example, $D = \{3, 5\}$ is a dominating set for G – this corresponds to the set cover $C = \{S_3, S_5\}$. For example, the vertex $4 \in V$ is dominated by the vertex $3 \in D$, and the element $4 \in U$ is contained in the set $S_3 \in C$.



From set covering to dominating set. Let (S, U) be an instance of the set cover problem with the universe U and the family of subsets $S = \{S_i : i \in I\}$; we assume that U and the index set I are disjoint. Construct a graph $G = (V, E)$ as follows: the set of vertices is $V = I \cup U$, there is an edge $\{i, j\} \in E$ between each pair $i, j \in I$, and there is also an edge $\{i, u\}$ for each $i \in I$ and $u \in S_i$. That is, G is a split graph: I is a clique and U is an independent set.

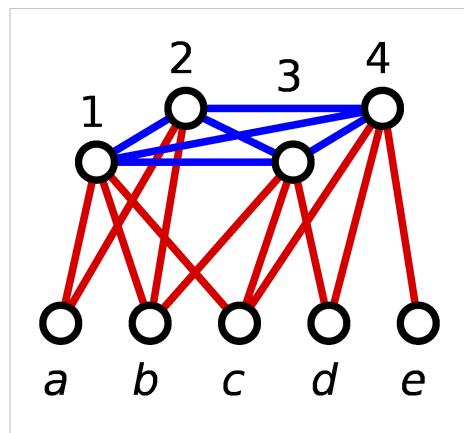
Now if $C = \{S_i : i \in D\}$ is a feasible solution of the set cover problem for some subset $D \subseteq I$, then D is a dominating set for G , with $|D| = |C|$: First, for each $u \in U$ there is an $i \in D$ such that $u \in S_i$, and by construction, u and i are adjacent in G ; hence u is dominated by i . Second, since D must be nonempty, each $i \in I$ is adjacent to a vertex in D .

Conversely, let D be a dominating set for G . Then it is possible to construct another dominating set X such that $|X| \leq |D|$ and $X \subseteq I$: simply replace each $u \in D \cap U$ by a neighbour $i \in I$ of u . Then $C = \{S_i : i \in X\}$ is a feasible solution of the set cover problem, with $|C| = |X| \leq |D|$.

The illustration on the right shows the construction for $U = \{a, b, c, d, e\}$, $I = \{1, 2, 3, 4\}$, $S_1 = \{a, b, c\}$, $S_2 = \{a, b\}$, $S_3 = \{b, c, d\}$, and $S_4 = \{c, d, e\}$.

In this example, $C = \{S_1, S_4\}$ is a set cover; this corresponds to the dominating set $D = \{1, 4\}$.

$D = \{a, 3, 4\}$ is another dominating set for the graph G . Given D , we can construct a dominating set $X = \{1, 3, 4\}$ which is not larger than D and which is a subset of I . The dominating set X corresponds to the set cover $C = \{S_1, S_3, S_4\}$.



Special cases

If the graph has maximum degree Δ , then the greedy approximation algorithm finds an $O(\log \Delta)$ -approximation of a minimum dominating set. The problem admits a PTAS for special cases such as unit disk graphs and planar graphs (Crescenzi et al. 2000). A minimum dominating set can be found in linear time in series-parallel graphs (Takamizawa, Nishizeki & Saito 1982).

Exact algorithms

A minimum dominating set of an n -vertex graph can be found in time $O(2^n n)$ by inspecting all vertex subsets. Fomin, Grandoni & Kratsch (2009) show how to find a minimum dominating set in time $O(1.5137^n)$ and exponential space, and in time $O(1.5264^n)$ and polynomial space. A faster algorithm, using $O(1.5048^n)$ time was found by van Rooij, Nederlof & van Dijk (2009), who also show that the number of minimum dominating sets can be computed in this time. The number of minimal dominating sets is at most 1.7159^n and all such sets can be listed in time $O(1.7159^n)$ (Fomin et al. 2008).

Parameterized complexity

Finding a dominating set of size k plays a central role in the theory of parameterized complexity. It is the most well-known problem complete for the class W[2] and used in many reductions to show intractability of other problems. In particular, the problem is not fixed-parameter tractable in the sense that no algorithm with running time $f(k)n^{O(1)}$ for any function f exists unless the W-hierarchy collapses to FPT=W[2]. On the other hand, if the input graph is planar, the problem remains NP-hard, but a fixed-parameter algorithm is known. In fact, the problem has a kernel of size linear in k (Alber, Fellows & Niedermeier 2004), and running times that are exponential in \sqrt{k} and cubic in n may be obtained by applying dynamic programming to a branch-decomposition of the kernel (Fomin & Thilikos 2006).

Variants

Vizing's conjecture relates the domination number of a cartesian product of graphs to the domination number of its factors.

There has been much work on connected dominating sets. If S is a connected dominating set, one can form a spanning tree of G in which S forms the set of non-leaf vertices of the tree; conversely, if T is any spanning tree in a graph with more than two vertices, the non-leaf vertices of T form a connected dominating set. Therefore, finding minimum connected dominating sets is equivalent to finding spanning trees with the maximum possible number of leaves.

A *total dominating set* is a set of vertices such that all vertices in the graph (including the vertices in the dominating set themselves) have a neighbor in the dominating set. Figure (c) above shows a dominating set that is a connected dominating set and a total dominating set; the examples in figures (a) and (b) are neither.

A *k -tuple dominating set* is a set of vertices such that each vertex in the graph has at least k neighbors in the set.

A domatic partition is a partition of the vertices into disjoint dominating sets. The domatic number is the maximum size of a domatic partition.

Software for searching minimum dominating set

Name (alphabetically)	License	API language	Brief info
OpenOpt	BSD	Python	Uses NetworkX graphs, can use free and commercial solvers, included / excluded vertices, see its dominating set [1] page for details and examples

References

- Alber, Jochen; Fellows, Michael R; Niedermeier, Rolf (2004), "Polynomial-time data reduction for dominating set", *Journal of the ACM* **51** (3): 363–384, doi:10.1145/990308.990309 [2].
- Allan, Robert B.; Laskar, Renu (1978), "On domination and independent domination numbers of a graph", *Discrete Mathematics* **23** (2): 73–76, doi:10.1016/0012-365X(78)90105-X [3].
- Crescenzi, Pierluigi; Kann, Viggo; Halldórsson, Magnús; Karpinski, Marek; Woeginger, Gerhard (2000), "Minimum dominating set" [4], *A Compendium of NP Optimization Problems*.
- Faudree, Ralph; Flandrin, Evelyne; Ryjáček, Zdeněk (1997), "Claw-free graphs — A survey", *Discrete Mathematics* **164** (1–3): 87–147, doi:10.1016/S0012-365X(96)00045-3 [5], MR 1432221 [6].
- Fomin, Fedor V.; Grandoni, Fabrizio; Kratsch, Dieter (2009), "A measure & conquer approach for the analysis of exact algorithms", *Journal of the ACM* **56** (5): 25:1–32, doi:10.1145/1552285.1552286 [7].

- Fomin, Fedor V.; Grandoni, Fabrizio; Pyatkin, Artem; Stepanov, Alexey (2008), "Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications", *ACM Transactions on Algorithms* **5** (1): 9:1–17, doi:10.1145/1435375.1435384^[8].
- Fomin, Fedor V.; Thilikos, Dimitrios M. (2006), "Dominating sets in planar graphs: branch-width and exponential speed-up", *SIAM Journal on Computing* **36** (2): 281, doi:10.1137/S0097539702419649^[9].
- Garey, Michael R.; Johnson, David S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, ISBN 0-7167-1045-5, p. 190, problem GT2.
- Grandoni, F. (2006), "A note on the complexity of minimum dominating set", *Journal of Discrete Algorithms* **4** (2): 209–214, doi:10.1016/j.jda.2005.03.002^[10].
- Guha, S.; Khuller, S. (1998), "Approximation algorithms for connected dominating sets", *Algorithmica* **20** (4): 374–387, doi:10.1007/PL00009201^[11].
- Haynes, Teresa W.; Hedetniemi, Stephen; Slater, Peter (1998a), *Fundamentals of Domination in Graphs*, Marcel Dekker, ISBN 0-8247-0033-3, OCLC 37903553^[12].
- Haynes, Teresa W.; Hedetniemi, Stephen; Slater, Peter (1998b), *Domination in Graphs: Advanced Topics*, Marcel Dekker, ISBN 0-8247-0034-1, OCLC 38201061^[13].
- Hedetniemi, S. T.; Laskar, R. C. (1990), "Bibliography on domination in graphs and some basic definitions of domination parameters", *Discrete Mathematics* **86** (1–3): 257–277, doi:10.1016/0012-365X(90)90365-O^[14].
- Kann, Viggo (1992), *On the Approximability of NP-complete Optimization Problems*^[15]. PhD thesis, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm.
- Raz, R.; Safra, S. (1997), "A sub-constant error-probability low-degree test, and sub-constant error-probability PCP characterization of NP", *Proc. 29th Annual ACM Symposium on Theory of Computing*, ACM, pp. 475–484, doi:10.1145/258533.258641^[16], ISBN 0-89791-888-6.
- Takamizawa, K.; Nishizeki, T.; Saito, N. (1982), "Linear-time computability of combinatorial problems on series-parallel graphs", *Journal of the ACM* **29** (3): 623–641, doi:10.1145/322326.322328^[17].
- van Rooij, J. M. M.; Nederlof, J.; van Dijk, T. C. (2009), "Inclusion/Exclusion Meets Measure and Conquer: Exact Algorithms for Counting Dominating Sets", *Proc. 17th Annual European Symposium on Algorithms, ESA 2009*, Lecture Notes in Computer Science **5757**, Springer, pp. 554–565, doi:10.1007/978-3-642-04128-0_50^[18], ISBN 978-3-642-04127-3.

References

- [1] <http://openopt.org/DSP>
- [2] <http://dx.doi.org/10.1145%2F990308.990309>
- [3] <http://dx.doi.org/10.1016%2F0012-365X%2878%2990105-X>
- [4] <http://www.nada.kth.se/~viggo/wwwcompendium/node11.html>
- [5] <http://dx.doi.org/10.1016%2FS0012-365X%2896%2900045-3>
- [6] <http://www.ams.org/mathscinet-getitem?mr=1432221>
- [7] <http://dx.doi.org/10.1145%2F1552285.1552286>
- [8] <http://dx.doi.org/10.1145%2F1435375.1435384>
- [9] <http://dx.doi.org/10.1137%2FS0097539702419649>
- [10] <http://dx.doi.org/10.1016%2Fj.jda.2005.03.002>
- [11] <http://dx.doi.org/10.1007%2FPL00009201>
- [12] <http://www.worldcat.org/oclc/37903553>
- [13] <http://www.worldcat.org/oclc/38201061>
- [14] <http://dx.doi.org/10.1016%2F0012-365X%2890%2990365-O>
- [15] <http://www.csc.kth.se/~viggo/papers/phdthesis.pdf>
- [16] <http://dx.doi.org/10.1145%2F258533.258641>
- [17] <http://dx.doi.org/10.1145%2F322326.322328>
- [18] http://dx.doi.org/10.1007%2F978-3-642-04128-0_50

Feedback vertex set

In the mathematical discipline of graph theory, a **feedback vertex set** of a graph is a set of vertices whose removal leaves a graph without cycles. In other words, each feedback vertex set contains at least one vertex of any cycle in the graph. The **feedback vertex set problem** is an NP-complete problem in computational complexity theory. It was among the first problems shown to be NP-complete. It has wide applications in operating systems, database systems, and VLSI chip design.

Definition

The decision problem is as follows:

INSTANCE: An (undirected or directed) graph $G = (V, E)$ and a positive integer k .

QUESTION: Is there a subset $X \subseteq V$ with $|X| \leq k$ such that G with the vertices from X deleted is cycle-free?

The graph $G[V \setminus X]$ that remains after removing X from G is an induced forest (resp. an induced directed acyclic graph in the case of directed graphs). Thus, finding a minimum feedback vertex set in a graph is equivalent to finding a maximum induced forest (resp. maximum induced directed acyclic graph in the case of directed graphs).

NP-hardness

Karp (1972) showed that the feedback vertex set problem for directed graphs is NP-complete. The problem remains NP-complete on directed graphs with maximum in-degree and out-degree two, and on directed planar graphs with maximum in-degree and out-degree three.^[1] Karp's reduction also implies the NP-completeness of the feedback vertex set problem on undirected graphs, where the problem stays NP-hard on graphs of maximum degree four. The feedback vertex set problem can be solved in polynomial time on graphs of maximum degree at most three.

Note that the problem of deleting as few *edges* as possible to make the graph cycle-free is equivalent to finding a spanning tree, which can be done in polynomial time. In contrast, the problem of deleting edges from a directed graph to make it acyclic, the feedback arc set problem, is NP-complete.

Exact algorithms

The corresponding NP optimization problem of finding the size of a minimum feedback vertex set can be solved in time $O(1.7347^n)$, where n is the number of vertices in the graph. This algorithm actually computes a maximum induced forest, and when such a forest is obtained, its complement is a minimum feedback vertex set. The number of minimal feedback vertex sets in a graph is bounded by $O(1.8638^n)$.^[2] The directed feedback vertex set problem can still be solved in time $O^*(1.9977^n)$, where n is the number of vertices in the given directed graph.^[3] The parameterized versions of the directed and undirected problems are both fixed-parameter tractable.^[4]

Approximation

The problem is APX-complete, which directly follows from the APX-completeness of the vertex cover problem, and the existence of an approximation preserving L-reduction from the vertex cover problem to it. The best known approximation algorithm on undirected graphs is by a factor of two.^[5]

Bounds

According to the Erdős–Pósa theorem, the size of a minimum feedback vertex set is within a logarithmic factor of the maximum number of vertex-disjoint cycles in the given graph.^[6]

Applications

In operating systems, feedback vertex sets play a prominent role in the study of deadlock recovery. In the wait-for graph of an operating system, each directed cycle corresponds to a deadlock situation. In order to resolve all deadlocks, some blocked processes have to be aborted. A minimum feedback vertex set in this graph corresponds to a minimum number of processes that one needs to abort.^[7]

Furthermore, the feedback vertex set problem has applications in VLSI chip design.^[8]

Notes

- [1] unpublished results due to Garey and Johnson, cf. : GT7
- [2] Fomin et al. (2008).
- [3] Razgon (2007).
- [4] Chen et al. (2008).
- [5] . See also for an alternative approximation algorithm with the same approximation ratio.
- [6] Erdős & Pósa (1965).
- [7] Silberschatz, Galvin & Gagne (2008).
- [8] Festa, Pardalos & Resende (2000).

References

Research articles

- Bafna, Vineet; Berman, Piotr; Fujito, Toshihiro (1999), "A 2-approximation algorithm for the undirected feedback vertex set problem", *SIAM Journal on Discrete Mathematics* **12** (3): 289–297 (electronic), doi: 10.1137/S0895480196305124 (<http://dx.doi.org/10.1137/S0895480196305124>), MR 1710236 (<http://www.ams.org/mathscinet-getitem?mr=1710236>).
- Becker, Ann; Bar-Yehuda, Reuven; Geiger, Dan (2000), "Randomized algorithms for the loop cutset problem", *Journal of Artificial Intelligence Research* **12**: 219–234, arXiv: 1106.0225 (<http://arxiv.org/abs/1106.0225>), doi: 10.1613/jair.638 (<http://dx.doi.org/10.1613/jair.638>), MR 1765590 (<http://www.ams.org/mathscinet-getitem?mr=1765590>)
- Becker, Ann; Geiger, Dan (1996), "Optimization of Pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem.", *Artificial Intelligence* **83** (1): 167–188, doi: 10.1016/0004-3702(95)00004-6 ([http://dx.doi.org/10.1016/0004-3702\(95\)00004-6](http://dx.doi.org/10.1016/0004-3702(95)00004-6))
- Cao, Yixin; Chen, Jianer; Liu, Yang (2010), "On feedback vertex set: new measure and new structures", in Kaplan, Haim, *Proc. 12th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2010), Bergen, Norway, June 21-23, 2010*, Lecture Notes in Computer Science **6139**, pp. 93–104, doi: 10.1007/978-3-642-13731-0_10 (http://dx.doi.org/10.1007/978-3-642-13731-0_10)
- Chen, Jianer; Fomin, Fedor V.; Liu, Yang; Lu, Songjian; Villanger, Yngve (2008), "Improved algorithms for feedback vertex set problems", *Journal of Computer and System Sciences* **74** (7): 1188–1198, doi:

- 10.1016/j.jcss.2008.05.002 (<http://dx.doi.org/10.1016/j.jcss.2008.05.002>), MR 2454063 (<http://www.ams.org/mathscinet-getitem?mr=2454063>)
- Chen, Jianer; Liu, Yang; Lu, Songjian; O'Sullivan, Barry; Razgon, Igor (2008), "A fixed-parameter algorithm for the directed feedback vertex set problem", *Journal of the ACM* **55** (5), Art. 21, doi: 10.1145/1411509.1411511 (<http://dx.doi.org/10.1145/1411509.1411511>), MR 2456546 (<http://www.ams.org/mathscinet-getitem?mr=2456546>)
 - Dinur, Irit; Safra, Samuel (2005), "On the hardness of approximating minimum vertex cover" (<http://www.cs.huji.ac.il/~dinuri/mypapers/vc.pdf>), *Annals of Mathematics*, Second Series **162** (1): 439–485, doi: 10.4007/annals.2005.162.439 (<http://dx.doi.org/10.4007/annals.2005.162.439>), MR 2178966 (<http://www.ams.org/mathscinet-getitem?mr=2178966>)
 - Erdős, Paul; Pósa, Lajos (1965), "On independent circuits contained in a graph" (http://www.renyi.hu/~p_erdos/1965-05.pdf), *Canadian Journal of Mathematics* **17**: 347–352, doi: 10.4153/CJM-1965-035-8 (<http://dx.doi.org/10.4153/CJM-1965-035-8>)
 - Fomin, Fedor V.; Gaspers, Serge; Pyatkin, Artem; Razgon, Igor (2008), "On the minimum feedback vertex set problem: exact and enumeration algorithms.", *Algorithmica* **52** (2): 293–307, doi: 10.1007/s00453-007-9152-0 (<http://dx.doi.org/10.1007/s00453-007-9152-0>)
 - Fomin, Fedor V.; Villanger, Yngve (2010), "Finding induced subgraphs via minimal triangulations", *Proc. 27th International Symposium on Theoretical Aspects of Computer Science (STACS 2010)*, Leibniz International Proceedings in Informatics (LIPIcs) **5**, pp. 383–394, doi: 10.4230/LIPIcs.STACS.2010.2470 (<http://dx.doi.org/10.4230/LIPIcs.STACS.2010.2470>)
 - Karp, Richard M. (1972), "Reducibility Among Combinatorial Problems", *Proc. Symposium on Complexity of Computer Computations*, IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., New York: Plenum, pp. 85–103
 - Li, Deming; Liu, Yanpei (1999), "A polynomial algorithm for finding the minimum feedback vertex set of a 3-regular simple graph", *Acta Mathematica Scientia* **19** (4): 375–381, MR 1735603 (<http://www.ams.org/mathscinet-getitem?mr=1735603>)
 - Razgon, I. (2007), "Computing minimum directed feedback vertex set in $O^*(1.9977^n)$ " (<http://www.cs.le.ac.uk/people/ir45/papers/ictesIgorCamera.pdf>), in Italiano, Giuseppe F.; Moggi, Eugenio; Laura, Luigi, *Proceedings of the 10th Italian Conference on Theoretical Computer Science*, World Scientific, pp. 70–81
 - Ueno, Shuichi; Kajitani, Yoji; Gotoh, Shin'ya (1988), "On the nonseparating independent set problem and feedback set problem for graphs with no vertex degree exceeding three", *Discrete Mathematics* **72** (1-3): 355–360, doi: 10.1016/0012-365X(88)90226-9 ([http://dx.doi.org/10.1016/0012-365X\(88\)90226-9](http://dx.doi.org/10.1016/0012-365X(88)90226-9)), MR 975556 (<http://www.ams.org/mathscinet-getitem?mr=975556>)

Textbooks and survey articles

- Festa, P.; Pardalos, P. M.; Resende, M.G.C. (2000), "Feedback set problems" (<http://www.research.att.com/~mgcr/doc/sfsp.pdf>), in Du, D.-Z.; Pardalos, P. M., *Handbook of Combinatorial Optimization, Supplement vol. A*, Kluwer Academic Publishers, pp. 209–259
- Garey, Michael R.; Johnson, David S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, A1.1: GT7, p. 191, ISBN 0-7167-1045-5
- Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2008), *Operating System Concepts* (8th ed.), John Wiley & Sons. Inc, ISBN 978-0-470-12872-5

Feedback arc set

In graph theory, a directed graph may contain directed cycles, a one-way loop of edges. In some applications, such cycles are undesirable, and we wish to eliminate them and obtain a directed acyclic graph (DAG). One way to do this is simply to drop edges from the graph to break the cycles. A **feedback arc set (FAS)** or **feedback edge set** is a set of edges which, when removed from the graph, leave a DAG. Put another way, it's a set containing at least one edge of every cycle in the graph.

Closely related are the feedback vertex set, which is a set of vertices containing at least one vertex from every cycle in the directed graph, and the minimum spanning tree, which is the undirected variant of the feedback arc set problem.

A minimal feedback arc set (one that can not be reduced in size by removing any edges) has the additional property that, if the edges in it are reversed rather than removed, then the graph remains acyclic. Finding a small edge set with this property is a key step in layered graph drawing.

Example

As a simple example, consider the following hypothetical situation:

- George says he will give you a piano, but only in exchange for a lawnmower.
- Harry says he will give you a lawnmower, but only in exchange for a microwave.
- Jane says she will give you a microwave, but only in exchange for a piano.

We can express this as a graph problem. Let each vertex represent an item, and add an edge from A to B if you must have A to obtain B. Your goal is to get the lawnmower. Unfortunately, you don't have any of the three items, and because this graph is cyclic, you can't get any of them either.

However, suppose you offer George \$100 for his piano. If he accepts, this effectively removes the edge from the lawnmower to the piano, because you no longer need the lawnmower to get the piano. Consequently, the cycle is broken, and you can trade twice to get the lawnmower. This one edge constitutes a feedback arc set.

Computational Complexity

As in the above example, there is usually some cost associated with removing an edge. For this reason, we'd like to remove as few edges as possible. Removing one edge suffices in a simple cycle, but in general figuring out the minimum number of edges to remove is an NP-hard problem called the **minimum feedback arc set** problem. It is particularly difficult in k -edge-connected graphs for large k , where each edge falls in many different cycles. The decision version of the problem, which is NP-complete, asks whether all cycles can be broken by removing at most k edges; this was one of Richard M. Karp's 21 NP-complete problems, shown by reducing from the vertex cover problem.

Although NP-complete, the feedback arc set problem is fixed-parameter tractable: there exists an algorithm for solving it whose running time is a fixed polynomial in the size of the input graph (independent of the number of edges in the set) but exponential in the number of edges in the feedback arc set.

Viggo Kann showed in 1992 that the minimum feedback arc set problem is APX-hard, which means that there is a constant c , such that, assuming P \neq NP, there is no polynomial-time approximation algorithm that always finds an edge set at most c times bigger than the optimal result. As of 2006[1], the highest value of c for which such an impossibility result is known is $c = 1.3606$.^[2] The best known approximation algorithm has ratio $O(\log n \log \log n)$. For the dual problem, of approximating the maximum number of edges in an acyclic subgraph, an approximation somewhat better than 1/2 is possible.

If the input digraphs are restricted to be tournaments, the resulting problem is known as the *minimum feedback arc set problem on tournaments (FAST)*. This restricted problem does admit a polynomial-time approximation scheme (PTAS); and this still holds for a restricted weighted version of the problem.^[3] A subexponential fixed parameter algorithm for the weighted FAST was given by Karpinski & Schudy (2010).

On the other hand, if the edges are undirected, the problem of deleting edges to make the graph cycle-free is equivalent to finding a minimum spanning tree, which can be done easily in polynomial time.

References

- [1] http://en.wikipedia.org/w/index.php?title=Feedback_arc_set&action=edit
- [2] . (Preliminary version in STOC 2002, titled "The importance of being biased", .)
- [3] . See also author's extended version (http://www.cs.brown.edu/people/ws/papers/fast_journal.pdf).

Tours

Eulerian path

In graph theory, a **Eulerian trail** (or **Eulerian path**) is a trail in a graph which visits every edge exactly once. Similarly, an **Eulerian circuit** or **Eulerian cycle** is an Eulerian trail which starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. Mathematically the problem can be stated like this:

Given the graph on the right, is it possible to construct a path (or a cycle, i.e. a path starting and ending on the same vertex) which visits each edge exactly once?

Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published posthumously in 1873 by Carl Hierholzer.^[1]

The term **Eulerian graph** has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.

For the existence of Eulerian trails it is necessary that zero or two vertices have an odd degree; this means the Königsberg graph is *not* Eulerian. If there are no vertices of odd degree, all Eulerian trails are circuits. If there are exactly two vertices of odd degree, all Eulerian trails start at one of them and end at the other. A graph that has an Eulerian trail but not an Eulerian circuit is called **semi-Eulerian**.

Definition

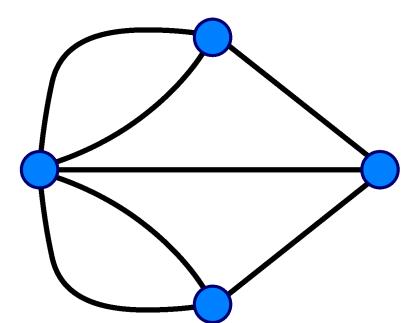
An **Eulerian trail**,^[2] or **Euler walk** in an undirected graph is a walk that uses each edge exactly once. If such a walk exists, the graph is called **traversable** or **semi-eulerian**.^[3]

An **Eulerian cycle**, **Eulerian circuit** or **Euler tour** in an undirected graph is a cycle that uses each edge exactly once. If such a cycle exists, the graph is called **Eulerian** or **unicursal**.^[4] The term "Eulerian graph" is also sometimes used in a weaker sense to denote a graph where every vertex has even degree. For finite connected graphs the two definitions are equivalent, while a possibly unconnected graph is Eulerian in the weaker sense if and only if each connected component has an Eulerian cycle.

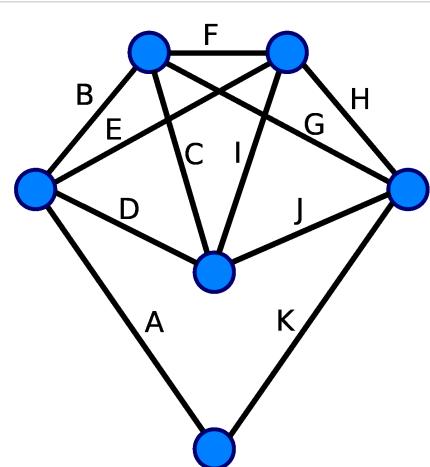
For directed graphs, "path" has to be replaced with *directed path* and "cycle" with *directed cycle*.

The definition and properties of Eulerian trails, cycles and graphs are valid for multigraphs as well.

An **Eulerian orientation** of an undirected graph G is an assignment of a direction to each edge of G such that, at each vertex v , the indegree of v equals the outdegree of v . Such an orientation exists for any undirected graph in



The Königsberg Bridges graph. This graph is not Eulerian, therefore, a solution does not exist.



Every vertex of this graph has an even degree, therefore this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

which every vertex has even degree, and may be found by constructing an Euler tour in each connected component of G and then orienting the edges according to the tour. Every Eulerian orientation of a connected graph is a strong orientation, an orientation that makes the resulting directed graph strongly connected.

Properties

- An undirected graph has an Eulerian cycle if and only if every vertex has even degree, and all of its vertices with nonzero degree belong to a single connected component.
- An undirected graph can be decomposed into edge-disjoint cycles if and only if all of its vertices have even degree. So, a graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint cycles and its nonzero-degree vertices belong to a single connected component.
- An undirected graph has an Eulerian trail if and only if at most two vertices have odd degree, and if all of its vertices with nonzero degree belong to a single connected component.
- A directed graph has an Eulerian cycle if and only if every vertex has equal in degree and out degree, and all of its vertices with nonzero degree belong to a single strongly connected component. Equivalently, a directed graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint directed cycles and all of its vertices with nonzero degree belong to a single strongly connected component.
- A directed graph has an Eulerian trail if and only if at most one vertex has $(\text{out-degree}) - (\text{in-degree}) = 1$, at most one vertex has $(\text{in-degree}) - (\text{out-degree}) = 1$, every other vertex has equal in-degree and out-degree, and all of its vertices with nonzero degree belong to a single connected component of the underlying undirected graph.

Constructing Eulerian trails and circuits

Fleury's algorithm

Fleury's algorithm is an elegant but inefficient algorithm which dates to 1883. Consider a graph known to have all edges in the same component and at most two vertices of odd degree. The algorithm starts at a vertex of odd degree, or, if the graph has none, it starts with an arbitrarily chosen vertex. At each step it chooses the next edge in the path to be one whose deletion would not disconnect the graph, unless there is no such edge, in which case it picks the remaining edge left at the current vertex. It then moves to the other endpoint of that vertex and deletes the chosen edge. At the end of the algorithm there are no edges left, and the sequence from which the edges were chosen forms an Eulerian cycle if the graph has no vertices of odd degree, or an Eulerian trail if there are exactly two vertices of odd degree.

While the *graph traversal* in Fleury's algorithm is linear in the number of edges, i.e. $O(|E|)$, we also need to factor in the complexity of detecting bridges. If we are to re-run Tarjan's linear time bridge-finding algorithm after the removal of every edge, Fleury's algorithm will have a time complexity of $O(|E|^2)$. A dynamic bridge-finding algorithm of Thorup (2000) allows this to be improved to $O(|E| \log^3 |E| \log \log |E|)$ but this is still significantly slower than alternative algorithms.

Hierholzer's algorithm

Hierholzer's 1873 paper provides a different method for finding Euler cycles that is more efficient than Fleury's algorithm:

- Choose any starting vertex v , and follow a trail of edges from that vertex until returning to v . It is not possible to get stuck at any vertex other than v , because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w . The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
- As long as there exists a vertex v that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from v , following unused edges until returning to v , and join the tour formed in this way to the

previous tour.

By using a data structure such as a doubly linked list to maintain the set of unused edges incident to each vertex, to maintain the list of vertices on the current tour that have unused edges, and to maintain the tour itself, the individual operations of the algorithm (finding unused edges exiting each vertex, finding a new starting vertex for a tour, and connecting two tours that share a vertex) may be performed in constant time each, so the overall algorithm takes linear time.

Counting Eulerian circuits

Complexity issues

The number of Eulerian circuits in *digraphs* can be calculated using the so-called **BEST theorem**, named after de Bruijn, van Aardenne-Ehrenfest, Smith and Tutte. The formula states that the number of Eulerian circuits in a digraph is the product of certain degree factorials and the number of rooted arborescences. The latter can be computed as a determinant, by the matrix tree theorem, giving a polynomial time algorithm.

BEST theorem is first stated in this form in a "note added in proof" to the Aardenne-Ehrenfest and de Bruijn paper (1951). The original proof was bijective and generalized the de Bruijn sequences. It is a variation on an earlier result by Smith and Tutte (1941).

Counting the number of Eulerian circuits on *undirected* graphs is much more difficult. This problem is known to be #P-complete.^[5] In a positive direction, a Markov chain Monte Carlo approach, via the *Kotzig transformations* (introduced by Anton Kotzig in 1968) is believed to give a sharp approximation for the number of Eulerian circuits in a graph, though as yet there is no proof of this fact (even for graphs of bounded degree).

Special cases

The asymptotic formula for the number of Eulerian circuits in the complete graphs was determined by McKay and Robinson (1995):^[6]

$$ec(K_n) = 2^{(n+1)/2} \pi^{1/2} e^{-n^2/2 + 11/12} n^{(n-2)(n+1)/2} (1 + O(n^{-1/2+\epsilon})).$$

A similar formula was later obtained by M.I. Isaev (2009) for complete bipartite graphs:

$$ec(K_{n,n}) = (n/2 - 1)!^{2n} 2^{n^2 - n + 1/2} \pi^{-n + 1/2} n^{n-1} (1 + O(n^{-1/2+\epsilon})).$$

Applications

Eulerian trails are used in bioinformatics to reconstruct the DNA sequence from its fragments. They are also used in CMOS circuit design to find an optimal logic gate ordering.

Notes

- [1] N. L. Biggs, E. K. Lloyd and R. J. Wilson, Graph Theory 1736–1936, Clarendon Press, Oxford, 1976, 8–9, ISBN 0-19-853901-0.
- [2] Some people reserve the terms *path* and *cycle* to mean *non-self-intersecting* path and cycle. A (potentially) self-intersecting path is known as a **trail** or an **open walk**; and a (potentially) self-intersecting cycle, a **circuit** or a **closed walk**. This ambiguity can be avoided by using the terms Eulerian trail and Eulerian circuit when self-intersection is allowed.
- [3] Jun-ichi Yamaguchi, Introduction of Graph Theory (<http://jwilson.coe.uga.edu/EMAT6680/Yamaguchi/emat6690/essay1/GT.html>).
- [4] Schaum's outline of theory and problems of graph theory By V. K. Balakrishnan (<http://books.google.co.uk/books?id=1NTPbSehvWsC&lpg=PA60&dq=unicursal&pg=PA60#v=onepage&q=unicursal&f=false>).
- [5] Brightwell and Winkler, " Note on Counting Eulerian Circuits (<http://www.cdam.lse.ac.uk/Reports/Files/cdam-2004-12.pdf>)", 2004.
- [6] Brendan McKay and Robert W. Robinson, Asymptotic enumeration of eulerian circuits in the complete graph (<http://cs.anu.edu.au/~bdm/papers/euler.pdf>), *Combinatorica*, 10 (1995), no. 4, 367–377.

References

- Euler, L., "Solutio problematis ad geometriam situs pertinentis (<http://www.math.dartmouth.edu/~euler/pages/E053.html>)", *Comment. Academiae Sci. I. Petropolitanae* **8** (1736), 128–140.
- Hierholzer, Carl (1873), "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren", *Mathematische Annalen* **6** (1): 30–32, doi: 10.1007/BF01442866 (<http://dx.doi.org/10.1007/BF01442866>).
- Lucas, E., *Récréations Mathématiques IV*, Paris, 1921.
- Fleury, "Deux problèmes de géométrie de situation", *Journal de mathématiques élémentaires* (1883), 257–261.
- T. van Aardenne-Ehrenfest and N. G. de Bruijn (1951) "Circuits and trees in oriented linear graphs", *Simon Stevin* 28: 203–217.
- Thorup, Mikkel (2000), "Near-optimal fully-dynamic graph connectivity", *Proc. 32nd ACM Symposium on Theory of Computing*, pp. 343–350, doi: 10.1145/335305.335345 (<http://dx.doi.org/10.1145/335305.335345>)
- W. T. Tutte and C. A. B. Smith (1941) "On Unicursal Paths in a Network of Degree 4", *American Mathematical Monthly* 48: 233–237.

External links



Wikimedia Commons has media related to **Eulerian paths**.

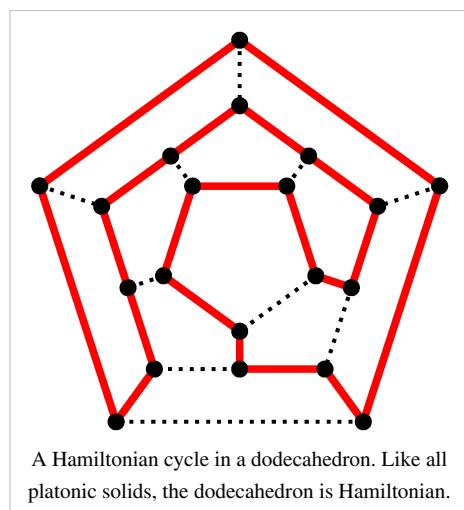
- Discussion of early mentions of Fleury's algorithm (<http://mathforum.org/kb/message.jspa?messageID=3648262&tstart=135>)

Hamiltonian path

This article is about the overall graph theory concept of a Hamiltonian path. For the specific problem of determining whether a Hamiltonian path or cycle exists in a given graph, see Hamiltonian path problem.

In the mathematical field of graph theory, a **Hamiltonian path** (or **traceable path**) is a path in an undirected or directed graph that visits each vertex exactly once. A **Hamiltonian cycle** (or **Hamiltonian circuit**) is a Hamiltonian path that is a cycle. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

Hamiltonian paths and cycles and cycle paths are named after William Rowan Hamilton who invented the icosian game, now also known as *Hamilton's puzzle*, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron. Hamilton solved this problem using the icosian calculus, an algebraic structure based on roots of unity with many similarities to the quaternions (also invented by Hamilton). This solution does not generalize to arbitrary graphs. However, despite being named after Hamilton, Hamiltonian cycles in polyhedra had also been studied a year earlier by Thomas Kirkman.



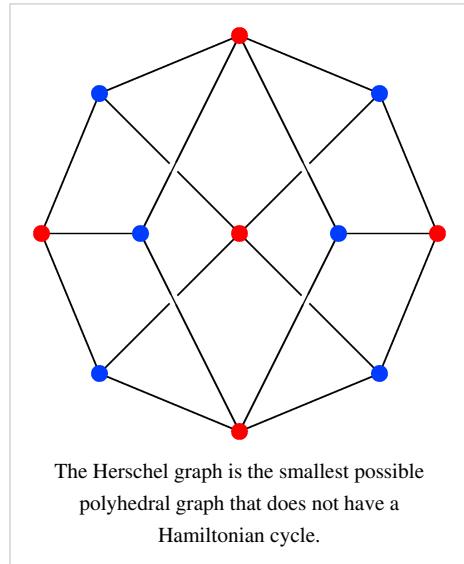
Definitions

A *Hamiltonian path* or *traceable path* is a path that visits each vertex exactly once. A graph that contains a Hamiltonian path is called a **traceable graph**. A graph is **Hamiltonian-connected** if for every pair of vertices there is a Hamiltonian path between the two vertices.

A *Hamiltonian cycle*, *Hamiltonian circuit*, *vertex tour* or *graph cycle* is a cycle that visits each vertex exactly once (except for the vertex that is both the start and end, which is visited twice). A graph that contains a Hamiltonian cycle is called a **Hamiltonian graph**.

Similar notions may be defined for *directed graphs*, where each edge (arc) of a path or cycle can only be traced in a single direction (i.e., the vertices are connected with arrows and the edges traced "tail-to-head").

A **Hamiltonian decomposition** is an edge decomposition of a graph into Hamiltonian circuits.



Examples

- a complete graph with more than two vertices is Hamiltonian
- every cycle graph is Hamiltonian
- every tournament has an odd number of Hamiltonian paths (Rédei 1934)
- every platonic solid, considered as a graph, is Hamiltonian^[1]

Properties

Any Hamiltonian cycle can be converted to a Hamiltonian path by removing one of its edges, but a Hamiltonian path can be extended to Hamiltonian cycle only if its endpoints are adjacent.

All Hamiltonian graphs are biconnected, but a biconnected graph need not be Hamiltonian (see, for example, the Petersen graph).

An Eulerian graph G (a connected graph in which every vertex has even degree) necessarily has an Euler tour, a closed walk passing through each edge of G exactly once. This tour corresponds to a Hamiltonian cycle in the line graph $L(G)$, so the line graph of every Eulerian graph is Hamiltonian. Line graphs may have other Hamiltonian cycles that do not correspond to Euler tours, and in particular the line graph $L(G)$ of every Hamiltonian graph G is itself Hamiltonian, regardless of whether the graph G is Eulerian.

A tournament (with more than two vertices) is Hamiltonian if and only if it is strongly connected.

The number of different Hamiltonian cycles in a complete undirected graph on n vertices is $(n - 1)! / 2$ and in a complete directed graph on n vertices is $(n - 1)!$.

Bondy–Chvátal theorem

The best vertex degree characterization of Hamiltonian graphs was provided in 1972 by the Bondy–Chvátal theorem, which generalizes earlier results by G. A. Dirac (1952) and Øystein Ore. In fact, both Dirac's and Ore's theorems are less powerful than what can be derived from Pósa's theorem (1962). Dirac and Ore's theorems basically state that a graph is Hamiltonian if it has *enough edges*. First we have to define the closure of a graph.

Given a graph G with n vertices, the **closure** $\text{cl}(G)$ is uniquely constructed from G by repeatedly adding a new edge uv connecting a nonadjacent pair of vertices u and v with $\text{degree}(v) + \text{degree}(u) \geq n$ until no more pairs with this property can be found.

Bondy–Chvátal theorem

A graph is Hamiltonian if and only if its closure is Hamiltonian.

As complete graphs are Hamiltonian, all graphs whose closure is complete are Hamiltonian, which is the content of the following earlier theorems by Dirac and Ore.

Dirac (1952)

A simple graph with n vertices ($n \geq 3$) is Hamiltonian if every vertex has degree $n / 2$ or greater.

Ore (1960)

A graph with n vertices ($n \geq 3$) is Hamiltonian if, for every pair of non-adjacent vertices, the sum of their degrees is n or greater (see Ore's theorem).

The following theorems can be regarded as directed versions:

Ghouila-Houiri (1960)

A strongly connected simple directed graph with n vertices is Hamiltonian if every vertex has a full degree greater than or equal to n .

Meyniel (1973)

A strongly connected simple directed graph with n vertices is Hamiltonian if the sum of full degrees of every pair of distinct non-adjacent vertices is greater than or equal to $2n - 1$.

The number of vertices must be doubled because each undirected edge corresponds to two directed arcs and thus the degree of a vertex in the directed graph is twice the degree in the undirected graph.

Theorems on existence of Hamiltonian cycles in planar graphs

Whitney (1931)

A 4-connected planar triangulation has a Hamiltonian cycle.

Tutte (1956)

A 4-connected planar graph has a Hamiltonian cycle.

Notes

- [1] Gardner, M. "Mathematical Games: About the Remarkable Similarity between the Icosian Game and the Towers of Hanoi." *Sci. Amer.* 196, 150–156, May 1957 (http://booklists.narod.ru/M_Mathematics/MGe_Eencyclopaediae/Weisstein_Concise_encyclopedia_of_mathematics_CRC_T_3236s_75.htm)

References

- Berge, Claude; Ghouila-Houiri, A. (1962), *Programming, games and transportation networks*, New York: Sons, Inc.
- DeLeon, Melissa (2000), "A study of sufficient conditions for Hamiltonian cycles" (<http://www.rose-hulman.edu/mathjournal/archives/2000/vol1-n1/paper4/v1n1-4pd.PDF>), *Rose-Hulman Undergraduate Math Journal* **1** (1).
- Dirac, G. A. (1952), "Some theorems on abstract graphs", *Proceedings of the London Mathematical Society*, 3rd Ser. **2**: 69–81, doi: 10.1112/plms/s3-2.1.69 (<http://dx.doi.org/10.1112/plms/s3-2.1.69>), MR 0047308 (<http://www.ams.org/mathscinet-getitem?mr=0047308>).
- Hamilton, William Rowan (1856), "Memorandum respecting a new system of roots of unity", *Philosophical Magazine* **12**: 446.
- Hamilton, William Rowan (1858), "Account of the Icosian Calculus", *Proceedings of the Royal Irish Academy* **6**: 415–416.
- Meyniel, M. (1973), "Une condition suffisante d'existence d'un circuit hamiltonien dans un graphe orienté", *Journal of Combinatorial Theory, Series B* **14** (2): 137–147, doi: 10.1016/0095-8956(73)90057-9 ([http://dx.doi.org/10.1016/0095-8956\(73\)90057-9](http://dx.doi.org/10.1016/0095-8956(73)90057-9)), MR 0317997 (<http://www.ams.org/mathscinet-getitem?mr=0317997>).
- Ore, Øystein (1960), "Note on Hamilton circuits", *The American Mathematical Monthly* **67**: 55, JSTOR 2308928 (<http://www.jstor.org/stable/2308928>), MR 0118683 (<http://www.ams.org/mathscinet-getitem?mr=0118683>).
- Pósa, L. (1962), "A theorem concerning Hamilton lines", *Magyar Tud. Akad. Mat. Kutató Int. Közl.* **7**: 225–226, MR 0184876 (<http://www.ams.org/mathscinet-getitem?mr=0184876>).
- Whitney, Hassler (1931), "A theorem on graphs", *Annals of Mathematics*, Second Series **32** (2): 378–390, doi: 10.2307/1968197 (<http://dx.doi.org/10.2307/1968197>), MR 1503003 (<http://www.ams.org/mathscinet-getitem?mr=1503003>).
- Tutte, W. T. (1956), "A theorem on planar graphs", *Trans. American Math. Soc.* **82**: 99–116, doi: 10.1090/s0002-9947-1956-0081471-8 (<http://dx.doi.org/10.1090/s0002-9947-1956-0081471-8>).

External links

- Weisstein, Eric W., "Hamiltonian Cycle" (<http://mathworld.wolfram.com/HamiltonianCycle.html>), *MathWorld*.
- Euler tour and Hamilton cycles (<http://www.graph-theory.net/euler-tour-and-hamilton-cycles/>)

Hamiltonian path problem

This article is about the specific problem of determining whether a Hamiltonian path or cycle exists in a given graph. For the general graph theory concepts, see Hamiltonian path.

In the mathematical field of graph theory the **Hamiltonian path problem** and the **Hamiltonian cycle problem** are problems of determining whether a Hamiltonian path (a path in an undirected or directed graph that visits each vertex exactly once) or a Hamiltonian cycle exists in a given graph (whether directed or undirected). Both problems are NP-complete.^[1]

There is a simple relation between the problems of finding a Hamiltonian path and a Hamiltonian cycle. In one direction, the Hamiltonian path problem for graph G is equivalent to the Hamiltonian cycle problem in a graph H obtained from G by adding a new vertex and connecting it to all vertices of G. Thus, finding a Hamiltonian path cannot be significantly slower (in the worst case, as a function of the number of vertices) than finding a Hamiltonian cycle. In the other direction, a graph G has a Hamiltonian cycle using edge uv if and only if the graph H obtained from G by replacing the edge by a pair of vertices of degree 1, one connected to u and one connected to v, has a Hamiltonian path. Therefore, by trying this replacement for all edges incident to some chosen vertex of G, the Hamiltonian cycle problem can be solved by at most n Hamiltonian path computations, where n is the number of vertices in the graph. The Hamiltonian cycle problem is also a special case of the travelling salesman problem, obtained by setting the distance between two cities to one if they are adjacent and two otherwise, and verifying that the total distance travelled is equal to n (if so, the route is a Hamiltonian circuit; if there is no Hamiltonian circuit then the shortest route will be longer).

Algorithms

There are $n!$ different sequences of vertices that *might* be Hamiltonian paths in a given n -vertex graph (and are, in a complete graph), so a brute force search algorithm that tests all possible sequences would be very slow. There are several faster approaches. A search procedure by Frank Rubin divides the edges of the graph into three classes: those that must be in the path, those that cannot be in the path, and undecided. As the search proceeds, a set of decision rules classifies the undecided edges, and determines whether to halt or continue the search. The algorithm divides the graph into components that can be solved separately. Also, a dynamic programming algorithm of Bellman, Held, and Karp can be used to solve the problem in time $O(n^2 2^n)$. In this method, one determines, for each set S of vertices and each vertex v in S , whether there is a path that covers exactly the vertices in S and ends at v . For each choice of S and v , a path exists for (S, v) if and only if v has a neighbor w such that a path exists for $(S - v, w)$, which can be looked up from already-computed information in the dynamic program.

Andreas Björklund provided an alternative approach using the inclusion–exclusion principle to reduce the problem of counting the number of Hamiltonian cycles to a simpler counting problem, of counting cycle covers, which can be solved by computing certain matrix determinants. Using this method, he showed how to solve the Hamiltonian cycle problem in arbitrary n -vertex graphs by a Monte Carlo algorithm in time $O(1.657^n)$; for bipartite graphs this algorithm can be further improved to time $O(1.414^n)$.

For graphs of maximum degree three, a careful backtracking search can find a Hamiltonian cycle (if one exists) in time $O(1.251^n)$.

Because of the difficulty of solving the Hamiltonian path and cycle problems on conventional computers, they have also been studied in unconventional models of computing. For instance, Leonard Adleman showed that the Hamiltonian path problem may be solved using a DNA computer. Exploiting the parallelism inherent in chemical reactions, the problem may be solved using a number of chemical reaction steps linear in the number of vertices of the graph; however, it requires a factorial number of DNA molecules to participate in the reaction.

Complexity

The problem of finding a Hamiltonian cycle or path is in FNP; the analogous decision problem is to test whether a Hamiltonian cycle or path exists. The directed and undirected Hamiltonian cycle problems were two of Karp's 21 NP-complete problems. They remain NP-complete even for undirected planar graphs of maximum degree three, for directed planar graphs with indegree and outdegree at most two, for bridgeless undirected planar 3-regular bipartite graphs, and for 3-connected 3-regular bipartite graphs. However, putting all of these conditions together, it remains open whether 3-connected 3-regular bipartite planar graphs must always contain a Hamiltonian cycle, in which case the problem restricted to those graphs could not be NP-complete; see Barnette's conjecture.

In graphs in which all vertices have odd degree, an argument related to the handshaking lemma shows that the number of Hamiltonian cycles through any fixed edge is always even, so if one Hamiltonian cycle is given, then a second one must also exist. However, finding this second cycle does not seem to be an easy computational task. Papadimitriou defined the complexity class PPA to encapsulate problems such as this one.

References

- [1] A1.3: GT37–39, pp. 199–200.

Travelling salesman problem

The **travelling salesman problem** (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

TSP is a special case of the travelling purchaser problem.

In the theory of computational complexity, the decision version of the TSP (where, given a length L , the task is to decide whether the graph has any tour shorter than L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (or perhaps exponentially) with the number of cities.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.^[1]

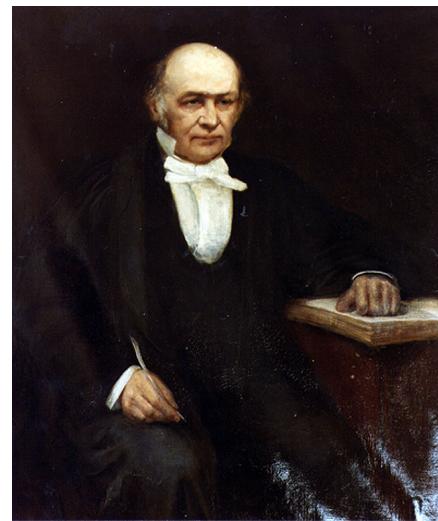
The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows may be imposed.

History

The origins of the travelling salesman problem are unclear. A handbook for travelling salesmen from 1832 mentions the problem and includes example tours through Germany and Switzerland, but contains no mathematical treatment.^[2]

The travelling salesman problem was mathematically formulated in the 1800s by the Irish mathematician W. R. Hamilton and by the British mathematician Thomas Kirkman. Hamilton's Icosian Game was a recreational puzzle based on finding a Hamiltonian cycle.^[3] The general form of the TSP appears to have been first studied by mathematicians during the 1930s in Vienna and at Harvard, notably by Karl Menger, who defines the problem, considers the obvious brute-force algorithm, and observes the non-optimality of the nearest neighbour heuristic:

We denote by *messenger problem* (since in practice this question should be solved by each postman, anyway also by many travelers) the task to find, for finitely many points whose pairwise distances are known, the shortest route connecting the points. Of course, this problem is solvable by finitely many trials. Rules which would push the number of trials below the number of permutations of the given points, are not known. The rule that one first should go from the starting point to the closest point, then to the point closest to this, etc., in general does not yield the shortest route.^[4]



William Rowan Hamilton

Hassler Whitney at Princeton University introduced the name *travelling salesman problem* soon after.^[5]

In the 1950s and 1960s, the problem became increasingly popular in scientific circles in Europe and the USA. Notable contributions were made by George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson at the RAND Corporation in Santa Monica, who expressed the problem as an integer linear program and developed the cutting plane method for its solution. With these new methods they solved an instance with 49 cities to optimality by constructing a tour and proving that no other tour could be shorter. In the following decades, the problem was studied by many researchers from mathematics, computer science, chemistry, physics, and other sciences.

Richard M. Karp showed in 1972 that the Hamiltonian cycle problem was NP-complete, which implies the NP-hardness of TSP. This supplied a mathematical explanation for the apparent computational difficulty of finding optimal tours.

Great progress was made in the late 1970s and 1980, when Grötschel, Padberg, Rinaldi and others managed to exactly solve instances with up to 2392 cities, using cutting planes and branch-and-bound.

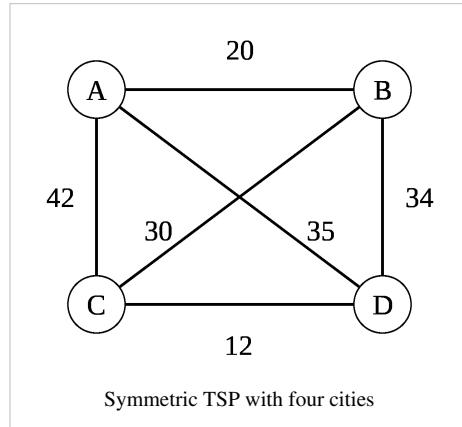
In the 1990s, Applegate, Bixby, Chvátal, and Cook developed the program *Concorde* that has been used in many recent record solutions. Gerhard Reinelt published the TSPLIB in 1991, a collection of benchmark instances of varying difficulty, which has been used by many research groups for comparing results. In 2006, Cook and others computed an optimal tour through an 85,900-city instance given by a microchip layout problem, currently the largest solved TSPLIB instance. For many other instances with millions of cities, solutions can be found that are guaranteed to be within 2–3% of an optimal tour.

The problem is sometimes, especially in newer publications, referred to as *Travelling Salesperson Problem*.

Description

As a graph problem

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (*i.e.* each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.



Asymmetric and symmetric

In the *symmetric TSP*, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the *asymmetric TSP*, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

Related problems

- An equivalent formulation in terms of graph theory is: Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight.
- The requirement of returning to the starting city does not change the computational complexity of the problem, see Hamiltonian path problem.
- Another related problem is the bottleneck traveling salesman problem (bottleneck TSP): Find a Hamiltonian cycle in a weighted graph with the minimal weight of the weightiest edge. The problem is of considerable practical importance, apart from evident transportation and logistics areas. A classic example is in printed circuit manufacturing: scheduling of a route of the drill machine to drill holes in a PCB. In robotic machining or drilling applications, the "cities" are parts to machine or holes (of different sizes) to drill, and the "cost of travel" includes time for retooling the robot (single machine job sequencing problem).
- The generalized traveling salesman problem deals with "states" that have (one or more) "cities" and the salesman has to visit exactly one "city" from each "state". Also known as the "traveling politician problem". One application is encountered in ordering a solution to the cutting stock problem in order to minimise knife changes. Another is concerned with drilling in semiconductor manufacturing, see e.g., U.S. Patent 7,054,798^[6]. Surprisingly, Behzad and Modarres demonstrated that the generalised traveling salesman problem can be transformed into a standard traveling salesman problem with the same number of cities, but a modified distance matrix.
- The sequential ordering problem deals with the problem of visiting a set of cities where precedence relations between the cities exist.
- The traveling purchaser problem deals with a purchaser who is charged with purchasing a set of products. He can purchase these products in several cities, but at different prices and not all cities offer the same products. The objective is to find a route between a subset of the cities, which minimizes total cost (travel cost + purchasing cost) and which enables the purchase of all required products.

Integer linear programming formulation

TSP can be formulated as an integer linear program.^{[7][8][9]} Label the cities with the numbers 0, ..., n and define:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

For $i = 1, \dots, n$, let u_i be an artificial variable, and finally take c_{ij} to be the distance from city i to city j . Then TSP can be written as the following integer linear programming problem:

$$\begin{aligned} \min \sum_{i=0}^n \sum_{j \neq i, j=0}^n c_{ij} x_{ij} \\ 0 \leq x_{ij} \leq 1 & \quad i, j = 0, \dots, n \\ u_i \in \mathbf{Z} & \quad i = 0, \dots, n \\ \sum_{i=0, i \neq j}^n x_{ij} = 1 & \quad j = 0, \dots, n \\ \sum_{j=0, j \neq i}^n x_{ij} = 1 & \quad i = 0, \dots, n \\ u_i - u_j + nx_{ij} \leq n - 1 & \quad 1 \leq i \neq j \leq n \end{aligned}$$

The first set of equalities requires that each city be arrived at from exactly one other city, and the second set of equalities requires that from each city there is a departure to exactly one other city. The last constraints enforce that there is only a single tour covering all cities, and not two or more disjointed tours that only collectively cover all cities. To prove this, it is shown below (1) that every feasible solution contains only one closed sequence of cities, and (2) that for every single tour covering all cities, there are values for the dummy variables u_i that satisfy the constraints.

To prove that every feasible solution contains only one closed sequence of cities, it suffices to show that every subtour in a feasible solution passes through city 0 (noting that the equalities ensure there can only be one such tour). For if we sum all the inequalities corresponding to $x_{ij} = 1$ for any subtour of k steps not passing through city 0, we obtain:

$$nk \leq (n-1)k,$$

which is a contradiction.

It now must be shown that for every single tour covering all cities, there are values for the dummy variables u_i that satisfy the constraints.

Without loss of generality, define the tour as originating (and ending) at city 0. Choose $u_i = t$ if city i is visited in step t ($i, t = 1, 2, \dots, n$). Then

$$u_i - u_j \leq n - 1,$$

since u_i can be no greater than n and u_j can be no less than 1; hence the constraints are satisfied whenever $x_{ij} = 0$. For $x_{ij} = 1$, we have:

$$u_i - u_j + nx_{ij} = (t) - (t+1) + n = n - 1,$$

satisfying the constraint.

Computing a solution

The traditional lines of attack for the NP-hard problems are the following:

- Devising algorithms for finding exact solutions (they will work reasonably fast only for small problem sizes).
- Devising "suboptimal" or heuristic algorithms, i.e., algorithms that deliver either seemingly or probably good solutions, but which could not be proved to be optimal.
- Finding special cases for the problem ("subproblems") for which either better or exact heuristics are possible.

Computational complexity

The problem has been shown to be NP-hard (more precisely, it is complete for the complexity class FP^{NP} ; see function problem), and the decision problem version ("given the costs and a number x , decide whether there is a round-trip route cheaper than x ") is NP-complete. The bottleneck travelling salesman problem is also NP-hard. The problem remains NP-hard even for the case when the cities are in the plane with Euclidean distances, as well as in a number of other restrictive cases. Removing the condition of visiting each city "only once" does not remove the NP-hardness, since it is easily seen that in the planar case there is an optimal tour that visits each city only once (otherwise, by the triangle inequality, a shortcut that skips a repeated visit would not increase the tour length).

Complexity of approximation

In the general case, finding a shortest travelling salesman tour is NPO-complete. If the distance measure is a metric and symmetric, the problem becomes APX-complete and Christofides's algorithm approximates it within 1.5.

If the distances are restricted to 1 and 2 (but still are a metric) the approximation ratio becomes $8/7$.^[10] In the asymmetric, metric case, only logarithmic performance guarantees are known, the best current algorithm achieves performance ratio $0.814 \log(n)$; it is an open question if a constant factor approximation exists.

The corresponding maximization problem of finding the *longest* travelling salesman tour is approximable within $63/38$. If the distance function is symmetric, the longest tour can be approximated within $4/3$ by a deterministic algorithm and within $\frac{1}{25}(33 + \varepsilon)$ by a randomised algorithm.

Exact algorithms

The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute force search). The running time for this approach lies within a polynomial factor of $O(n!)$, the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. One of the earliest applications of dynamic programming is the Held–Karp algorithm that solves the problem in time $O(n^2 2^n)$.^[11]

Improving these time bounds seems to be difficult. For example, it has not been determined whether an exact algorithm for TSP that runs in time $O(1.9999^n)$ exists.

Other approaches include:

- Various branch-and-bound algorithms, which can be used to process TSPs containing 40–60 cities.
- Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities.
- Implementations of branch-and-bound and problem-specific cut generation (branch-and-cut); this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85,900 cities, see Applegate et al. (2006).

An exact solution for 15,112 German towns from TSPLIB was found in 2001 using the cutting-plane method proposed by George Dantzig, Ray Fulkerson, and Selmer M. Johnson in 1954, based on linear programming. The computations were performed on a network of 110 processors located at Rice University and Princeton University (see the Princeton external link). The total computation time was equivalent to 22.6 years on a single 500 MHz Alpha processor. In May 2004, the travelling salesman problem of visiting all 24,978 towns in Sweden was solved: a

tour of length approximately 72,500 kilometers was found and it was proven that no shorter tour exists.^[12]

In March 2005, the travelling salesman problem of visiting all 33,810 points in a circuit board was solved using *Concorde TSP Solver*: a tour of length 66,048,945 units was found and it was proven that no shorter tour exists. The computation took approximately 15.7 CPU-years (Cook et al. 2006). In April 2006 an instance with 85,900 points was solved using *Concorde TSP Solver*, taking over 136 CPU-years, see Applegate et al. (2006).

Heuristic and approximation algorithms

Various heuristics and approximation algorithms, which quickly yield good solutions have been devised. Modern methods can find solutions for extremely large problems (millions of cities) within a reasonable time which are with a high probability just 2–3% away from the optimal solution.

Several categories of heuristics are recognized.

Constructive heuristics

The nearest neighbor (NN) algorithm (or so-called greedy algorithm) lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. For N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path.^[13] However, there exist many specially arranged city distributions which make the NN algorithm give the worst route (Gutin, Yeo, and Zverovich, 2002). This is true for both asymmetric and symmetric TSPs (Gutin and Yeo, 2007). Rosenkrantz et al. [1977] showed that the NN algorithm has the approximation factor $\Theta(\log |V|)$ for instances satisfying the triangle inequality. A variation of NN algorithm, called Nearest Fragment (NF) operator, which connects a group (fragment) of nearest unvisited cities, can find shorter route with successive iterations.^[14] The NF operator can also be applied on an initial solution obtained by NN algorithm for further improvement in an elitist model, where only better solutions are accepted.

Constructions based on a minimum spanning tree have an approximation ratio of 2. The Christofides algorithm achieves a ratio of 1.5.

The bitonic tour of a set of points is the minimum-perimeter monotone polygon that has the points as its vertices; it can be computed efficiently by dynamic programming.

Another constructive heuristic, Match Twice and Stitch (MTS) (Kahng, Reda 2004^[15]), performs two sequential matchings, where the second matching is executed after deleting all the edges of the first matching, to yield a set of cycles. The cycles are then stitched to produce the final tour.

Iterative improvement

Pairwise exchange

The pairwise exchange or *2-opt* technique involves iteratively removing two edges and replacing these with two different edges that reconnect the fragments created by edge removal into a new and shorter tour. This is a special case of the *k-opt* method. Note that the label *Lin-Kernighan* is an often heard misnomer for 2-opt. Lin-Kernighan is actually the more general *k-opt* method.

k-opt heuristic, or Lin-Kernighan heuristics

Take a given tour and delete *k* mutually disjoint edges. Reassemble the remaining fragments into a tour, leaving no disjoint subtours (that is, don't connect a fragment's endpoints together). This in effect simplifies the TSP under consideration into a much simpler problem. Each fragment endpoint can be connected to $2k - 2$ other possibilities: of $2k$ total fragment endpoints available, the two endpoints of the fragment under consideration are disallowed. Such a constrained $2k$ -city TSP can then be solved with brute force methods to find the least-cost recombination of the original fragments. The *k-opt* technique is a special case of the *V-opt* or variable-opt technique. The most popular of the *k-opt* methods are 3-opt, and these were introduced by Shen

Lin of Bell Labs in 1965. There is a special case of 3-opt where the edges are not disjoint (two of the edges are adjacent to one another). In practice, it is often possible to achieve substantial improvement over 2-opt without the combinatorial cost of the general 3-opt by restricting the 3-changes to this special subset where two of the removed edges are adjacent. This so-called two-and-a-half-opt typically falls roughly midway between 2-opt and 3-opt, both in terms of the quality of tours achieved and the time required to achieve those tours.

V-opt heuristic

The variable-opt method is related to, and a generalization of the k -opt method. Whereas the k -opt methods remove a fixed number (k) of edges from the original tour, the variable-opt methods do not fix the size of the edge set to remove. Instead they grow the set as the search process continues. The best known method in this family is the Lin–Kernighan method (mentioned above as a misnomer for 2-opt). Shen Lin and Brian Kernighan first published their method in 1972, and it was the most reliable heuristic for solving travelling salesman problems for nearly two decades. More advanced variable-opt methods were developed at Bell Labs in the late 1980s by David Johnson and his research team. These methods (sometimes called Lin–Kernighan–Johnson) build on the Lin–Kernighan method, adding ideas from tabu search and evolutionary computing. The basic Lin–Kernighan technique gives results that are guaranteed to be at least 3-opt. The Lin–Kernighan–Johnson methods compute a Lin–Kernighan tour, and then perturb the tour by what has been described as a mutation that removes at least four edges and reconnecting the tour in a different way, then V-opting the new tour. The mutation is often enough to move the tour from the local minimum identified by Lin–Kernighan. V-opt methods are widely considered the most powerful heuristics for the problem, and are able to address special cases, such as the Hamilton Cycle Problem and other non-metric TSPs that other heuristics fail on. For many years Lin–Kernighan–Johnson had identified optimal solutions for all TSPs where an optimal solution was known and had identified the best known solutions for all other TSPs on which the method had been tried.

Randomised improvement

Optimized Markov chain algorithms which use local searching heuristic sub-algorithms can find a route extremely close to the optimal route for 700 to 800 cities.

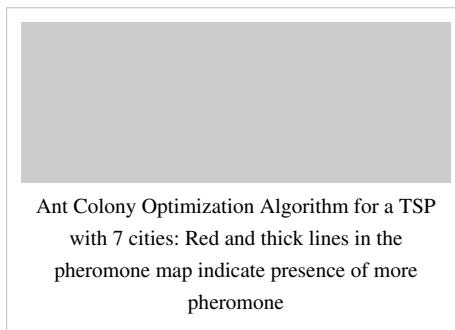
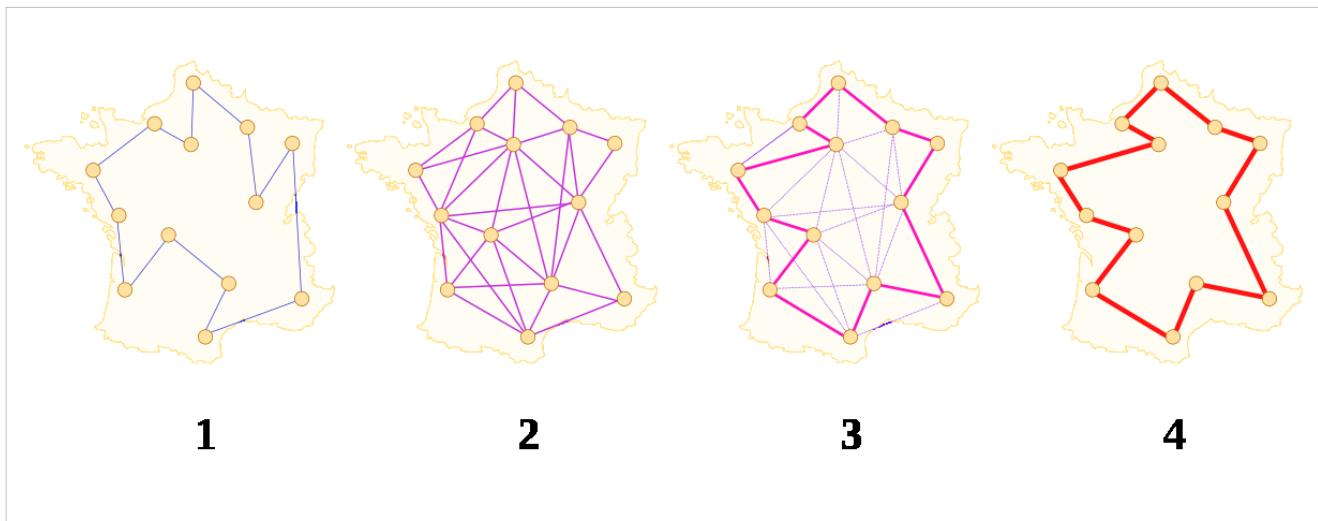
TSP is a touchstone for many general heuristics devised for combinatorial optimization such as genetic algorithms, simulated annealing, Tabu search, ant colony optimization, river formation dynamics (see swarm intelligence) and the cross entropy method.

Ant colony optimization

Main article: Ant colony optimization algorithms

Artificial intelligence researcher Marco Dorigo described in 1997 a method of heuristically generating "good solutions" to the TSP using a simulation of an ant colony called ACS (Ant Colony System).^[16] It models behavior observed in real ants to find short paths between food sources and their nest, an emergent behaviour resulting from each ant's preference to follow trail pheromones deposited by other ants.

ACS sends out a large number of virtual ant agents to explore many possible routes on the map. Each ant probabilistically chooses the next city to visit based on a heuristic combining the distance to the city and the amount of virtual pheromone deposited on the edge to the city. The ants explore, depositing pheromone on each edge that they cross, until they have all completed a tour. At this point the ant which completed the shortest tour deposits virtual pheromone along its complete tour route (*global trail updating*). The amount of pheromone deposited is inversely proportional to the tour length: the shorter the tour, the more it deposits.



Special cases

Metric TSP

In the *metric TSP*, also known as *delta-TSP* or Δ -TSP, the intercity distances satisfy the triangle inequality.

A very natural restriction of the TSP is to require that the distances between cities form a metric to satisfy the triangle inequality; that is the direct connection from A to B is never farther than the route via intermediate C :

$$d_{AB} \leq d_{AC} + d_{CB}.$$

The edge spans then build a metric on the set of vertices. When the cities are viewed as points in the plane, many natural distance functions are metrics, and so many natural instances of TSP satisfy this constraint.

The following are some examples of metric TSPs for various metrics.

- In the Euclidean TSP (see below) the distance between two cities is the Euclidean distance between the corresponding points.
- In the rectilinear TSP the distance between two cities is the sum of the differences of their x - and y -coordinates. This metric is often called the Manhattan distance or city-block metric.
- In the maximum metric, the distance between two points is the maximum of the absolute values of differences of their x - and y -coordinates.

The last two metrics appear for example in routing a machine that drills a given set of holes in a printed circuit board. The Manhattan metric corresponds to a machine that adjusts first one co-ordinate, and then the other, so the time to move to a new point is the sum of both movements. The maximum metric corresponds to a machine that adjusts both co-ordinates simultaneously, so the time to move to a new point is the slower of the two movements.

In its definition, the TSP does not allow cities to be visited twice, but many applications do not need this constraint. In such cases, a symmetric, non-metric instance can be reduced to a metric one. This replaces the original graph with a complete graph in which the inter-city distance d_{AB} is replaced by the shortest path between A and B in the original graph.

The span of the minimum spanning tree of the network G is a natural lower bound for the span of the optimal route, because deleting any edge of the optimal route yields a Hamiltonian path, which is a spanning tree in G . In the TSP with triangle inequality case it is possible to prove upper bounds in terms of the minimum spanning tree and design an algorithm that has a provable upper bound on the span of the route. The first published (and the simplest) example follows:

1. Construct a minimum spanning tree T for G .
2. Duplicate all edges of T . That is, wherever there is an edge from u to v , add a second edge from v to u . This gives us an Eulerian graph H .
3. Find an Eulerian circuit C in H . Clearly, its span is twice the span of the tree.
4. Convert the Eulerian circuit C of H into a Hamiltonian cycle of G in the following way: walk along C , and each time you are about to come into an already visited vertex, skip it and try to go to the next one (along C).

It is easy to prove that the last step works. Moreover, thanks to the triangle inequality, each skipping at Step 4 is in fact a shortcut; i.e., the length of the cycle does not increase. Hence it gives us a TSP tour no more than twice as long as the optimal one.

The Christofides algorithm follows a similar outline but combines the minimum spanning tree with a solution of another problem, minimum-weight perfect matching. This gives a TSP tour which is at most 1.5 times the optimal. The Christofides algorithm was one of the first approximation algorithms, and was in part responsible for drawing attention to approximation algorithms as a practical approach to intractable problems. As a matter of fact, the term "algorithm" was not commonly extended to approximation algorithms until later; the Christofides algorithm was initially referred to as the Christofides heuristic.

Euclidean TSP

The **Euclidean TSP**, or **planar TSP**, is the TSP with the distance being the ordinary Euclidean distance.

The Euclidean TSP is a particular case of the metric TSP, since distances in a plane obey the triangle inequality.

Like the general TSP, the Euclidean TSP is NP-hard. With discretized metric (distances rounded up to an integer), the problem is NP-complete.^[17] However, in some respects it seems to be easier than the general metric TSP. For example, the minimum spanning tree of the graph associated with an instance of the Euclidean TSP is a Euclidean minimum spanning tree, and so can be computed in expected $O(n \log n)$ time for n points (considerably less than the number of edges). This enables the simple 2-approximation algorithm for TSP with triangle inequality above to operate more quickly.

In general, for any $c > 0$, where d is the number of dimensions in the Euclidean space, there is a polynomial-time algorithm that finds a tour of length at most $(1 + 1/c)$ times the optimal for geometric instances of TSP in

$$O\left(n(\log n)^{(O(c\sqrt{d}))^{d-1}}\right),$$

time; this is called a polynomial-time approximation scheme (PTAS).^[18] Sanjeev Arora and Joseph S. B. Mitchell were awarded the Gödel Prize in 2010 for their concurrent discovery of a PTAS for the Euclidean TSP.

In practice, heuristics with weaker guarantees continue to be used.

Asymmetric TSP

In most cases, the distance between two nodes in the TSP network is the same in both directions. The case where the distance from A to B is not equal to the distance from B to A is called asymmetric TSP. A practical application of an asymmetric TSP is route optimisation using street-level routing (which is made asymmetric by one-way streets, slip-roads, motorways, etc.).

Solving by conversion to symmetric TSP

Solving an asymmetric TSP graph can be somewhat complex. The following is a 3×3 matrix containing all possible path weights between the nodes A , B and C . One option is to turn an asymmetric matrix of size N into a symmetric matrix of size $2N$.^[19]

	A	B	C
A		1	2
B	6		3
C	5	4	

†+ Asymmetric path weights

To double the size, each of the nodes in the graph is duplicated, creating a second *ghost node*. Using duplicate points with very low weights, such as $-\infty$, provides a cheap route "linking" back to the real node and allowing symmetric evaluation to continue. The original 3×3 matrix shown above is visible in the bottom left and the inverse of the original in the top-right. Both copies of the matrix have had their diagonals replaced by the low-cost hop paths, represented by $-\infty$.

	A	B	C	A'	B'	C'
A				$-\infty$	6	5
B				1	$-\infty$	4
C				2	3	$-\infty$
A'	$-\infty$	1	2			
B'	6	$-\infty$	3			
C'	5	4	$-\infty$			

†+ Symmetric path weights

The original 3×3 matrix would produce two Hamiltonian cycles (a path that visits every node once), namely $A-B-C-A$ [score 9] and $A-C-B-A$ [score 12]. Evaluating the 6×6 symmetric version of the same problem now produces many paths, including $A-A'-B-B'-C-C'-A$, $A-B'-C-A'-A$, $A-A'-B-C'-A$ [all score $9 - \infty$].

The important thing about each new sequence is that there will be an alternation between dashed (A', B', C') and un-dashed nodes (A, B, C) and that the link to "jump" between any related pair ($A-A'$) is effectively free. A version of the algorithm could use any weight for the $A-A'$ path, as long as that weight is *lower* than all other path weights present in the graph. As the path weight to "jump" must effectively be "free", the value zero (0) could be used to represent this cost—if zero is not being used for another purpose already (such as designating invalid paths). In the two examples above, non-existent paths between nodes are shown as a blank square.

Benchmarks

For benchmarking of TSP algorithms, **TSPLIB**^[20] is a library of sample instances of the TSP and related problems is maintained, see the TSPLIB external reference. Many of them are lists of actual cities and layouts of actual printed circuits.

Human performance on TSP

The TSP, in particular the Euclidean variant of the problem, has attracted the attention of researchers in cognitive psychology. It has been observed that humans are able to produce good quality solutions quickly. These results suggest that computer performance on the TSP may be improved by understanding and emulating the methods used by humans for these problems, and have also led to new insights into the mechanisms of human thought. The first issue of the *Journal of Problem Solving* was devoted to the topic of human performance on TSP,^[21] and a 2011 review listed dozens of papers on the subject.

TSP path length for random sets of points in a square

Suppose X_1, \dots, X_n are n independent random variables with uniform distribution in the square $[0, 1]^2$, and let L_n^* be the shortest path length (i.e. TSP solution) for this set of points, according to the usual Euclidean distance. It is known that, almost surely,

$$\frac{L_n^*}{\sqrt{n}} \rightarrow \beta \quad \text{when } n \rightarrow \infty,$$

where β is a positive constant that is not known explicitly. Since $L_n^* \leq 2\sqrt{n} + 2$ (see below), it follows from bounded convergence theorem that $\beta = \lim_{n \rightarrow \infty} \mathbb{E}[L_n^*]/\sqrt{n}$, hence lower and upper bounds on β follow from bounds on $\mathbb{E}[L_n^*]$.

Upper bound

- One has $L^* \leq 2\sqrt{n} + 2$, and therefore $\beta \leq 2$, by using a naive path which visits monotonically the points inside each of \sqrt{n} slices of width $1/\sqrt{n}$ in the square.
- Few proved $L_n^* \leq \sqrt{2n} + 1.75$, hence $\beta \leq \sqrt{2}$, later improved by Karloff (1987): $\beta \leq 0.984\sqrt{2}$.

Lower bound

- By observing that $\mathbb{E}[L_n^*]$ is greater than n times the distance between X_0 and the closest point $X_i \neq X_0$, one gets (after a short computation)

$$\mathbb{E}[L_n^*] \geq \frac{1}{2}\sqrt{n}.$$

- A better lower bound is obtained by observing that $\mathbb{E}[L_n^*]$ is greater than $\frac{1}{2}n$ times the sum of the distances between X_0 and the closest and second closest points $X_i, X_j \neq X_0$, which gives

$$\mathbb{E}[L_n^*] \geq \left(\frac{1}{4} + \frac{3}{8}\right)\sqrt{n} = \frac{5}{8}\sqrt{n},$$

hence $\beta \geq \frac{5}{8}$.

- Held and Karp gave a polynomial-time algorithm that provides numerical lower bounds for L_n^* , and thus for $\beta (\simeq L_n^*/\sqrt{n})$ which seem to be good up to more or less 1%. In particular, David S. Johnson^[22] obtained a lower bound by computer experiment:

$$L_n^* \gtrsim 0.7080\sqrt{n} + 0.522,$$

where 0.522 comes from the points near square boundary which have fewer neighbors, and Christine L. Valenzuela and Antonia J. Jones^[23] obtained the following other numerical lower bound:

$$L_n^* \gtrsim 0.7078\sqrt{n} + 0.551.$$

Analyst's travelling salesman problem

There is an analogous problem in geometric measure theory which asks the following: under what conditions may a subset E of Euclidean space be contained in a rectifiable curve (that is, when is there a curve with finite length that visits every point in E)? This problem is known as the analyst's travelling salesman problem or the geometric travelling salesman problem.

Free software for solving TSP

Name (alphabetically)	License	API language	Brief info
Concorde [24]	free for academic	only executable	requires a linear solver installation for its MILP subproblem
DynOpt [25]	?	C	an ANSI C implementation a dynamic programming based algorithm developed by Balas and Simonetti, approximate solution
LKH [26]	research only	C	an effective implementation of the Lin-Kernighan heuristic for Euclidean traveling salesman problem
OpenOpt	BSD	Python	exact and approximate solvers, STSP / ATSP, can handle multigraphs, constraints, multiobjective problems, see its TSP [27] page for details and examples
OptaPlanner	Apache License	Java	Open Source Java constraint solver with TSP and VRP examples.
R TSP package [28]	GPL	R	infrastructure and solvers for STSP / ATSP, interface to Concorde
TSP Solver and Generator [29]	GPL	C++	branch and bound algorithm
TSPGA [30]	?	C	approximate solution of the STSP using the "pgapack" package

Popular culture

Travelling Salesman, by director Timothy Lanzone, is the story of four mathematicians hired by the U.S. government to solve the most elusive problem in computer-science history: P vs. NP.

Notes

- [1] See the TSP world tour problem which has already been solved to within 0.05% of the optimal solution. (<http://www.math.uwaterloo.ca/tsp/world/>)
- [2] "Der Handlungsreisende – wie er sein soll und was er zu tun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein – von einem alten Commis-Voyageur" (The traveling salesman — how he must be and what he should do in order to get commissions and be sure of the happy success in his business — by an old *commis-voyageur*)
- [3] A discussion of the early work of Hamilton and Kirkman can be found in Graph Theory 1736–1936
- [4] Cited and English translation in . Original German: "Wir bezeichnen als *Botenproblem* (weil diese Frage in der Praxis von jedem Postboten, übrigens auch von vielen Reisenden zu lösen ist) die Aufgabe, für endlich viele Punkte, deren paarweise Abstände bekannt sind, den kürzesten die Punkte verbindenden Weg zu finden. Dieses Problem ist natürlich stets durch endlich viele Versuche lösbar. Regeln, welche die Anzahl der Versuche unter die Anzahl der Permutationen der gegebenen Punkte herunterdrücken würden, sind nicht bekannt. Die Regel, man solle vom Ausgangspunkt erst zum nächstgelegenen Punkt, dann zu dem diesem nächstgelegenen Punkt gehen usw., liefert im allgemeinen nicht den kürzesten Weg."
- [5] A detailed treatment of the connection between Menger and Whitney as well as the growth in the study of TSP can be found in Alexander Schrijver's 2005 paper "On the history of combinatorial optimization (till 1960). Handbook of Discrete Optimization (K. Aardal, G.L. Nemhauser, R. Weismantel, eds.), Elsevier, Amsterdam, 2005, pp. 1–68. PS (<http://homepages.cwi.nl/~lex/files/histco.ps>), PDF (<http://homepages.cwi.nl/~lex/files/histco.pdf>)
- [6] <http://www.google.com/patents/US7054798>

- [7] , pp.308-309.
- [8] Tucker, A. W. (1960), "On Directed Graphs and Integer Programs", IBM Mathematical research Project (Princeton University)
- [9] Dantzig, George B. (1963), *Linear Programming and Extensions*, Princeton, NJ: PrincetonUP, pp. 545–7, ISBN 0-691-08000-3, sixth printing, 1974.
- [10] Berman & Karpinski (2006).
- [11] ,
- [12] Work by David Applegate, AT&T Labs – Research, Robert Bixby, ILOG and Rice University, Vašek Chvátal, Concordia University, William Cook, University of Waterloo, and Keld Helsgaun, Roskilde University is discussed on their project web page hosted by the University of Waterloo and last updated in June 2004, here (<http://www.math.uwaterloo.ca/tsp/sweden/>)
- [13] Johnson, D.S. and McGeoch, L.A.. "The traveling salesman problem: A case study in local optimization", Local search in combinatorial optimization, 1997, 215-310
- [14] S. S. Ray, S. Bandyopadhyay and S. K. Pal, "Genetic Operators for Combinatorial Optimization in TSP and Microarray Gene Ordering," Applied Intelligence, 2007, 26(3). pp. 183-195.
- [15] A. B. Kahng and S. Reda, "Match Twice and Stitch: A New TSP Tour Construction Heuristic," Operations Research Letters, 2004, 32(6). pp. 499–509. <http://dx.doi.org/10.1016/j.orl.2004.04.001>
- [16] Marco Dorigo. Ant Colonies for the Traveling Salesman Problem. IRIDIA, Université Libre de Bruxelles. IEEE Transactions on Evolutionary Computation, 1(1):53–66. 1997. <http://citeseer.ist.psu.edu/86357.html>
- [17] Papadimitriou (1977).
- [18] Arora (1998).
- [19] Roy Jonker and Ton Volgenant. "Transforming asymmetric into symmetric traveling salesman problems". *Operations Research Letters* 2:161–163, 1983.
- [20] <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>
- [21] *Journal of Problem Solving* 1(1) (<http://docs.lib.psu.edu/jps/vol1/iss1/>), 2006, retrieved 2014-06-06.
- [22] David S. Johnson (<http://www.research.att.com/~dsj/papers/HKsoda.pdf>)
- [23] Christine L. Valenzuela and Antonia J. Jones (<http://users.cs.cf.ac.uk/Antonia.J.Jones/Papers/EJORHeldKarp/HeldKarp.pdf>)
- [24] <http://www.math.uwaterloo.ca/tsp/concorde.html>
- [25] <http://www.andrew.cmu.edu/user/neils/tsp/>
- [26] <http://www.akira.ruc.dk/~keld/research/LKH/>
- [27] <http://openopt.org/TSP>
- [28] <http://tsp.r-forge.r-project.org/>
- [29] <http://tspg.info/>
- [30] <http://www.rz.uni-karlsruhe.de/~lh71/>

References

- Applegate, D. L.; Bixby, R. M.; Chvátal, V.; Cook, W. J. (2006), *The Traveling Salesman Problem*, ISBN 0-691-12993-2.
- Arora, Sanjeev (1998), "Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems", *Journal of the ACM* **45** (5): 753–782, doi: 10.1145/290179.290180 (<http://dx.doi.org/10.1145/290179.290180>), MR 1668147 (<http://www.ams.org/mathscinet-getitem?mr=1668147>).
- Beardwood, J.; Halton, J.H.; Hammersley, J.M. (1959), "The Shortest Path Through Many Points", *Proceedings of the Cambridge Philosophical Society* **55**: 299–327.
- Bellman, R. (1960), "Combinatorial Processes and Dynamic Programming", in Bellman, R., Hall, M., Jr. (eds.), *Combinatorial Analysis, Proceedings of Symposia in Applied Mathematics 10*, American Mathematical Society, pp. 217–249.
- Bellman, R. (1962), "Dynamic Programming Treatment of the Travelling Salesman Problem", *J. Assoc. Comput. Mach.* **9**: 61–63, doi: 10.1145/321105.321111 (<http://dx.doi.org/10.1145/321105.321111>).
- Berman, Piotr; Karpinski, Marek (2006), "8/7-approximation algorithm for (1,2)-TSP", *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA '06)*, pp. 641–648, doi: 10.1145/1109557.1109627 (<http://dx.doi.org/10.1145/1109557.1109627>), ISBN 0898716055, ECCC TR05-069 (<http://eccc.uni-trier.de/report/2005/069/>).
- Christofides, N. (1976), *Worst-case analysis of a new heuristic for the travelling salesman problem*, Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh.

- Hassin, R.; Rubinstein, S. (2000), "Better approximations for max TSP", *Information Processing Letters* **75** (4): 181–186, doi: 10.1016/S0020-0190(00)00097-1 ([http://dx.doi.org/10.1016/S0020-0190\(00\)00097-1](http://dx.doi.org/10.1016/S0020-0190(00)00097-1)).
- Held, M.; Karp, R. M. (1962), "A Dynamic Programming Approach to Sequencing Problems", *Journal of the Society for Industrial and Applied Mathematics* **10** (1): 196–210, doi: 10.1137/0110015 (<http://dx.doi.org/10.1137/0110015>).
- Kaplan, H.; Lewenstein, L.; Shafir, N.; Sviridenko, M. (2004), "Approximation Algorithms for Asymmetric TSP by Decomposing Directed Regular Multigraphs", *In Proc. 44th IEEE Symp. on Foundations of Comput. Sci.*, pp. 56–65.
- Kosaraju, S. R.; Park, J. K.; Stein, C. (1994), "Long tours and short superstrings'", *Proc. 35th Ann. IEEE Symp. on Foundations of Comput. Sci.*, IEEE Computer Society, pp. 166–177.
- Orponen, P.; Mannila, H. (1987), "On approximation preserving reductions: Complete problems and robust measures'", *Technical Report C-1987-28, Department of Computer Science, University of Helsinki*.
- Padberg, M.; Rinaldi, G. (1991), "A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems", *Siam Review*: 60–100, doi: 10.1137/1033004 (<http://dx.doi.org/10.1137/1033004>).
- Papadimitriou, Christos H. (1977), "The Euclidean traveling salesman problem is NP-complete", *Theoretical Computer Science* **4** (3): 237–244, doi: 10.1016/0304-3975(77)90012-3 ([http://dx.doi.org/10.1016/0304-3975\(77\)90012-3](http://dx.doi.org/10.1016/0304-3975(77)90012-3)), MR 0455550 (<http://www.ams.org/mathscinet-getitem?mr=0455550>).
- Papadimitriou, C. H.; Yannakakis, M. (1993), "The traveling salesman problem with distances one and two", *Math. Oper. Res.* **18**: 1–11, doi: 10.1287/moor.18.1.1 (<http://dx.doi.org/10.1287/moor.18.1.1>).
- Serdyukov, A. I. (1984), "An algorithm with an estimate for the traveling salesman problem of the maximum", *Upravlyayemye Sistemy* **25**: 80–86.
- Woeginger, G.J. (2003), "Exact Algorithms for NP-Hard Problems: A Survey", *Combinatorial Optimization – Eureka, You Shrink! Lecture notes in computer science*, vol. 2570, Springer, pp. 185–207.

Further reading

- Adleman, Leonard (1994), "Molecular Computation of Solutions To Combinatorial Problems" (<http://www.usc.edu/dept/molecular-science/papers/fp-sci94.pdf>), *Science* **266** (5187): 1021–4, Bibcode: 1994Sci...266.1021A (<http://adsabs.harvard.edu/abs/1994Sci...266.1021A>), doi: 10.1126/science.7973651 (<http://dx.doi.org/10.1126/science.7973651>), PMID 7973651 (<http://www.ncbi.nlm.nih.gov/pubmed/7973651>)
- Arora, S. (1998), "Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems" (<http://graphics.stanford.edu/courses/cs468-06-winter/Papers/arora-tsp.pdf>), *Journal of the ACM* **45** (5): 753–782, doi: 10.1145/290179.290180 (<http://dx.doi.org/10.1145/290179.290180>).
- Babin, Gilbert; Deneault, Stéphanie; Laporte, Gilbert (2005), *Improvements to the Or-opt Heuristic for the Symmetric Traveling Salesman Problem* (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.9953>), Cahiers du GERAD, G-2005-02, Montreal: Group for Research in Decision Analysis.
- Cook, William (2011), *In Pursuit of the Travelling Salesman: Mathematics at the Limits of Computation*, Princeton University Press, ISBN 978-0-691-15270-7.
- Cook, William; Espinoza, Daniel; Goycoolea, Marcos (2007), "Computing with domino-parity inequalities for the TSP", *INFORMS Journal on Computing* **19** (3): 356–365, doi: 10.1287/ijoc.1060.0204 (<http://dx.doi.org/10.1287/ijoc.1060.0204>).
- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), "35.2: The traveling-salesman problem", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 1027–1033, ISBN 0-262-03293-7.
- Dantzig, G. B.; Fulkerson, R.; Johnson, S. M. (1954), "Solution of a large-scale traveling salesman problem", *Operations Research* **2** (4): 393–410, doi: 10.1287/opre.2.4.393 (<http://dx.doi.org/10.1287/opre.2.4.393>), JSTOR 166695 (<http://www.jstor.org/stable/166695>).

- Garey, M. R.; Johnson, D. S. (1979), "A2.3: ND22–24", *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, pp. 211–212, ISBN 0-7167-1045-5.
- Goldberg, D. E. (1989), "Genetic Algorithms in Search, Optimization & Machine Learning", *Reading: Addison-Wesley* (New York: Addison-Wesley), Bibcode: 1989gaso.book.....G (<http://adsabs.harvard.edu/abs/1989gaso.book.....G>), ISBN 0-201-15767-5.
- Gutin, G.; Yeo, A.; Zverovich, A. (2002), "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP", *Discrete Applied Mathematics* **117** (1–3): 81–86, doi: [10.1016/S0166-218X\(01\)00195-0](https://doi.org/10.1016/S0166-218X(01)00195-0) ([http://dx.doi.org/10.1016/S0166-218X\(01\)00195-0](http://dx.doi.org/10.1016/S0166-218X(01)00195-0)).
- Gutin, G.; Punnen, A. P. (2006), *The Traveling Salesman Problem and Its Variations*, Springer, ISBN 0-387-44459-9.
- Johnson, D. S.; McGeoch, L. A. (1997), "The Traveling Salesman Problem: A Case Study in Local Optimization", in Aarts, E. H. L.; Lenstra, J. K., *Local Search in Combinatorial Optimisation*, John Wiley and Sons Ltd, pp. 215–310.
- Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G.; Shmoys, D. B. (1985), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, ISBN 0-471-90413-9.
- MacGregor, J. N.; Ormerod, T. (1996), "Human performance on the traveling salesman problem" (<http://www.psych.lancs.ac.uk/people/uploads/TomOrmerod20030716T112601.pdf>), *Perception & Psychophysics* **58** (4): 527–539, doi: [10.3758/BF03213088](https://doi.org/10.3758/BF03213088) (<http://dx.doi.org/10.3758/BF03213088>).
- Mitchell, J. S. B. (1999), "Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k -MST, and related problems" (<http://citeseer.ist.psu.edu/622594.html>), *SIAM Journal on Computing* **28** (4): 1298–1309, doi: [10.1137/S0097539796309764](https://doi.org/10.1137/S0097539796309764) (<http://dx.doi.org/10.1137/S0097539796309764>).
- Rao, S.; Smith, W. (1998), "Approximating geometrical graphs via 'spanners' and 'banyans'", *Proc. 30th Annual ACM Symposium on Theory of Computing*, pp. 540–550.
- Rosenkrantz, Daniel J.; Stearns, Richard E.; Lewis, Philip M., II (1977), "An Analysis of Several Heuristics for the Traveling Salesman Problem", *SIAM Journal on Computing* **6** (5): 563–581, doi: [10.1137/0206041](https://doi.org/10.1137/0206041) (<http://dx.doi.org/10.1137/0206041>).
- Vickers, D.; Butavicius, M.; Lee, M.; Medvedev, A. (2001), "Human performance on visually presented traveling salesman problems", *Psychological Research* **65** (1): 34–45, doi: [10.1007/s004260000031](https://doi.org/10.1007/s004260000031) (<http://dx.doi.org/10.1007/s004260000031>), PMID 11505612 (<http://www.ncbi.nlm.nih.gov/pubmed/11505612>).
- Walshaw, Chris (2000), *A Multilevel Approach to the Travelling Salesman Problem*, CMS Press.
- Walshaw, Chris (2001), *A Multilevel Lin-Kernighan-Helsgaun Algorithm for the Travelling Salesman Problem*, CMS Press.

External links



Wikimedia Commons has media related to *Traveling salesman problem*.

- Traveling Salesman Problem (<http://www.math.uwaterloo.ca/tsp/index.html>) at University of Waterloo
- TSPLIB (<http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>) at the University of Heidelberg
- *Traveling Salesman Problem* (<http://demonstrations.wolfram.com/TravelingSalesmanProblem/>) by Jon McLoone at the Wolfram Demonstrations Project
- Source code library for the travelling salesman problem (http://www.adaptivebox.net/CILib/code/tspcodes_link.html)
- TSP solvers in R (<http://tsp.r-forge.r-project.org/>) for symmetric and asymmetric TSPs. Implements various insertion, nearest neighbor and 2-opt heuristics and an interface to Concorde and Chained Lin-Kernighan

heuristics.

- TSP solver with BING Maps (<https://www.multiroute.de/?locale=en>) solves TSP and TSPTW.
- Traveling Salesman movie (on IMDB) (<http://www.imdb.com/title/tt1801123/>)
- "Traveling Salesman in Python and Linear Optimization, IBM Developerworks with Source Code (<http://www.ibm.com/developerworks/cloud/library/cl-optimizepythoncloud1/index.html>)" by Noah Gift

Bottleneck traveling salesman problem

The **Bottleneck traveling salesman problem** (bottleneck TSP) is a problem in discrete or combinatorial optimization. It is stated as follows: Find the Hamiltonian cycle in a weighted graph which minimizes the weight of the most weighty edge of the cycle.^[1]

The problem is known to be NP-hard. The decision problem version of this, "for a given length x , is there a Hamiltonian cycle in a graph g with no edge longer than x ? ", is NP-complete.

In an **asymmetric bottleneck TSP**, there are cases where the weight from node A to B is different from the weight from B to A (e. g. travel time between two cities with a traffic jam in one direction).

Euclidean bottleneck TSP, or planar bottleneck TSP, is the bottleneck TSP with the distance being the ordinary Euclidean distance. The problem still remains NP-hard, however many heuristics work better.

If the graph is a metric space then there is an efficient approximation algorithm that finds a Hamiltonian cycle with maximum edge weight being no more than twice the optimum.^[2]

References

[1] A2.3: ND24, pg.212.

[2] 2(6):269–272

Christofides' heuristic for the TSP

The goal of the **Christofides approximation algorithm** (named after Nicos Christofides) is to find a solution to the instances of the traveling salesman problem where the edge weights satisfy the triangle inequality. Let $G = (V, w)$ be an instance of TSP, i.e. G is a complete graph on the set V of vertices with weight function w assigning a nonnegative real weight to every edge of G .

Algorithm

In pseudo-code:

1. Create a minimum spanning tree T of G .
2. Let O be the set of vertices with odd degree in T and find a perfect matching M with minimum weight in the complete graph over the vertices from O .
3. Combine the edges of M and T to form a multigraph H .
4. Form an Eulerian circuit in H (H is Eulerian because it is connected, with only even-degree vertices).
5. Make the circuit found in previous step Hamiltonian by skipping visited nodes (*shortcutting*).

Approximation ratio

The cost of the solution produced by the algorithm is within $3/2$ of the optimum.

The proof is as follows:

Let A denote the edge set of the optimal solution of TSP for G . Because (V, A) is connected, it contains some spanning tree T and thus $w(A) \geq w(T)$. Further let B denote the edge set of the optimal solution of TSP for the complete graph over vertices from O . Because the edge weights are triangular (so visiting more nodes cannot reduce total cost), we know that $w(A) \geq w(B)$. We show that there is a perfect matching of vertices from O with weight under $w(B)/2 \leq w(A)/2$ and therefore we have the same upper bound for M (because M is a perfect matching of minimum cost). Because O must contain an even number of vertices, a perfect matching exists. Let e_1, \dots, e_{2k} be the (only) Eulerian path in (O, B) . Clearly both $e_1, e_3, \dots, e_{2k-1}$ and e_2, e_4, \dots, e_{2k} are perfect matchings and the weight of at least one of them is less than or equal to $w(B)/2$. Thus $w(M) + w(T) \leq w(A) + w(A)/2$ and from the triangle inequality it follows that the algorithm is $3/2$ -approximative.

Example

	Given: metric graph $G = (V, E)$ with edge weights
	Calculate minimum spanning tree T .

	Calculate the set of vertices V' with odd degree in T .
	Reduce G to the vertices of V' ($G _{V'}$).
	Calculate matching M with minimum weight in $G _{V'}$.
	Unite matching and spanning tree ($T \cup M$).
	Calculate Euler tour on $T \cup M$ (A-B-C-A-D-E-A).
	Remove reoccurring vertices and replace by direct connections (A-B-C-D-E-A). In metric graphs, this step can not lengthen the tour. This tour is the algorithms output.

References

- NIST Christofides Algorithm Definition [1]
- Nicos Christofides, Worst-case analysis of a new heuristic for the travelling salesman problem, Report 388, Graduate School of Industrial Administration, CMU, 1976.

References

[1] <http://www.nist.gov/dads/HTML/christofides.html>

Route inspection problem

In graph theory, a branch of mathematics, the **Chinese postman problem (CPP), postman tour or route inspection problem** is to find a shortest closed path or circuit that visits every edge of a (connected) undirected graph. When the graph has an Eulerian circuit (a closed walk that covers every edge once), that circuit is an optimal solution.

Alan Goldman of the U.S. National Bureau of Standards first coined the name 'Chinese Postman Problem' for this problem, as it was originally studied by the Chinese mathematician Kwan Mei-Ko in 1962.

Eulerian paths and circuits

In order for a graph to have an Eulerian circuit, it will certainly have to be connected.

Suppose we have a connected graph $G = (V, E)$, The following statements are equivalent:

1. All vertices in G have even degree.
 2. G consists of the edges from a disjoint union of some cycles, and the vertices from these cycles.
 3. G has an Eulerian circuit.
- $1 \rightarrow 2$ can be shown by induction on the number of cycles.
 - $2 \rightarrow 3$ can also be shown by induction on the number of cycles, and
 - $3 \rightarrow 1$ should be immediate.

An Eulerian path (a walk which is not closed but uses all edges of G just once) exists if and only if G is connected and exactly two vertices have odd valence.

T -joins

Let T be a subset of the vertex set of a graph. An edge set is called a **T -join** if in the induced subgraph of this edge set, the collection of all the odd-degree vertices is T . (In a connected graph, a T -join exists if and only if $|T|$ is even.) The **T -join problem** is to find a smallest T -join. When T is the set of all odd-degree vertices, a smallest T -join leads to a solution of the postman problem. For any T , a smallest T -join necessarily consists of $\frac{1}{2}|T|$ paths, no two having an edge in common, that join the vertices of T in pairs. The paths will be such that the total length of all of them is as small as possible. A minimum T -join can be obtained using a weighted matching algorithm that uses $O(n^3)$ computational steps.^[1]

Solution

If a graph has an Eulerian circuit (or an Eulerian path), then an Eulerian circuit (or path) visits every edge, and so the solution is to choose any Eulerian circuit (or path).

If the graph is not Eulerian, it must contain vertices of odd degree. By the handshaking lemma, there must be an even number of these vertices. To solve the postman problem we first find a smallest T -join. We make the graph Eulerian by doubling of the T -join. The solution to the postman problem in the original graph is obtained by finding an Eulerian circuit for the new graph.

Applications

Various combinatorial problems are reduced to the Chinese Postman Problem, including finding a maximum cut in a planar graph and a minimum-mean length circuit in an undirected graph^[2].

Variants

A few variants of the Chinese Postman Problem have been studied and shown to be NP-complete.

- Min Chinese postman problem for mixed graphs: for this problem, some of the edges may be directed and can therefore only be visited from one direction. When the problem is minimal traversal of a digraph it is known as the "New York Street Sweeper problem."
- Min k -Chinese postman problem: find k cycles all starting at a designated location such that each edge is traversed by at least one cycle. The goal is to minimize the cost of the most expensive cycle.
- Rural postman problem: Given is also a subset of the edges. Find the cheapest Hamiltonian cycle containing each of these edges (and possibly others). This is a special case of the minimum general routing problem which specifies precisely which vertices the cycle must contain.

References

- [1] J. Edmonds and E.L. Johnson, Matching Euler tours and the Chinese postman problem, *Math. Program.* (1973).
[2] A. Schrijver, *Combinatorial Optimization, Polyhedra and Efficiency*, Volume A, Springer. (2002).

External links

- Chinese Postman Problem (<http://mathworld.wolfram.com/ChinesePostmanProblem.html>)

Matching

Matching

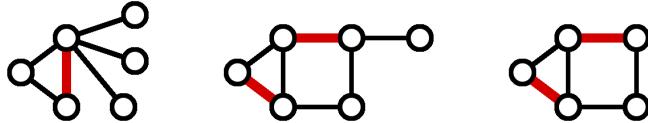
In the mathematical discipline of graph theory, a **matching** or **independent edge set** in a graph is a set of edges without common vertices. It may also be an entire graph consisting of edges without common vertices.

Definition

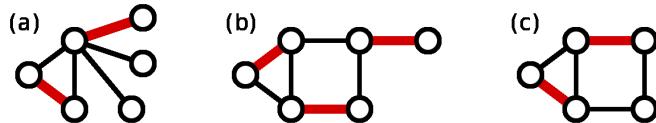
Given a graph $G = (V, E)$, a **matching** M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.

A vertex is **matched** (or **saturated**) if it is an endpoint of one of the edges in the matching. Otherwise the vertex is **unmatched**.

A **maximal matching** is a matching M of a graph G with the property that if any edge not in M is added to M , it is no longer a matching, that is, M is maximal if it is not a proper subset of any other matching in graph G . In other words, a matching M of a graph G is maximal if every edge in G has a non-empty intersection with at least one edge in M . The following figure shows examples of maximal matchings (red) in three graphs.



A **maximum matching** (also known as maximum-cardinality matching^[1]) is a matching that contains the largest possible number of edges. There may be many maximum matchings. The **matching number** $\nu(G)$ of a graph G is the size of a maximum matching. Note that every maximum matching is maximal, but not every maximal matching is a maximum matching. The following figure shows examples of maximum matchings in the same three graphs.



A **perfect matching** (a.k.a. 1-factor) is a matching which matches all vertices of the graph. That is, every vertex of the graph is incident to exactly one edge of the matching. Figure (b) above is an example of a perfect matching. Every perfect matching is maximum and hence maximal. In some literature, the term **complete matching** is used. In the above figure, only part (b) shows a perfect matching. A perfect matching is also a minimum-size edge cover. Thus, $\nu(G) \leq \rho(G)$, that is, the size of a maximum matching is no larger than the size of a minimum edge cover.

A **near-perfect matching** is one in which exactly one vertex is unmatched. This can only occur when the graph has an odd number of vertices, and such a matching must be maximum. In the above figure, part (c) shows a near-perfect matching. If, for every vertex in a graph, there is a near-perfect matching that omits only that vertex, the graph is also called factor-critical.

Given a matching M ,

- an **alternating path** is a path in which the edges belong alternatively to the matching and not to the matching.
- an **augmenting path** is an alternating path that starts from and ends on free (unmatched) vertices.

One can prove that a matching is maximum if and only if it does not have any augmenting path. (This result is sometimes called Berge's lemma.)

Properties

In any graph without isolated vertices, the sum of the matching number and the edge covering number equals the number of vertices. If there is a perfect matching, then both the matching number and the edge cover number are $|V| / 2$.

If A and B are two maximal matchings, then $|A| \leq 2|B|$ and $|B| \leq 2|A|$. To see this, observe that each edge in $B \setminus A$ can be adjacent to at most two edges in $A \setminus B$ because A is a matching; moreover each edge in $A \setminus B$ is adjacent to an edge in $B \setminus A$ by maximality of B , hence

$$|A \setminus B| \leq 2|B \setminus A|.$$

Further we get that

$$|A| = |A \cap B| + |A \setminus B| \leq 2|B \cap A| + 2|B \setminus A| = 2|B|.$$

In particular, this shows that any maximal matching is a 2-approximation of a maximum matching and also a 2-approximation of a minimum maximal matching. This inequality is tight: for example, if G is a path with 3 edges and 4 nodes, the size of a minimum maximal matching is 1 and the size of a maximum matching is 2.

Matching polynomials

Main article: Matching polynomial

A generating function of the number of k -edge matchings in a graph is called a matching polynomial. Let G be a graph and m_k be the number of k -edge matchings. One matching polynomial of G is

$$\sum_{k \geq 0} m_k x^k.$$

Another definition gives the matching polynomial as

$$\sum_{k \geq 0} (-1)^k m_k x^{n-2k},$$

where n is the number of vertices in the graph. Each type has its uses; for more information see the article on matching polynomials.

Algorithms and computational complexity

In unweighted bipartite graphs

Matching problems are often concerned with bipartite graphs. Finding a **maximum bipartite matching** (often called a **maximum cardinality bipartite matching**) in a bipartite graph $G = (V = (X, Y), E)$ is perhaps the simplest problem.

The Augmenting path algorithm finds it by finding an augmenting path from each $x \in X$ to Y and adding it to the matching if it exists. As each path can be found in $O(E)$ time, the running time is $O(VE)$. This solution is equivalent to adding a *super source* s with edges to all vertices in X , and a *super sink* t with edges from all vertices in Y , and finding a maximal flow from s to t . All edges with flow from X to Y then constitute a maximum matching.

An improvement over this is the Hopcroft–Karp algorithm, which runs in $O(\sqrt{V}E)$ time. Another approach is based on the fast matrix multiplication algorithm and gives $O(V^{2.376})$ complexity, which is better in theory for sufficiently dense graphs, but in practice the algorithm is slower. Finally, for sparse graphs, $\tilde{O}(E^{10/7})$ is possible with Madry's algorithm based on electric flows.

In weighted bipartite graphs

In a weighted bipartite graph, each edge has an associated value. A **maximum weighted bipartite matching** is defined as a matching where the sum of the values of the edges in the matching have a maximal value. If the graph is not complete bipartite, missing edges are inserted with value zero. Finding such a matching is known as the assignment problem. The remarkable Hungarian algorithm solves the assignment problem and it was one of the beginnings of combinatorial optimization algorithms. It uses a modified shortest path search in the augmenting path algorithm. If the Bellman–Ford algorithm is used for this step, the running time of the Hungarian algorithm becomes $O(V^2E)$, or the edge cost can be shifted with a potential to achieve $O(V^2 \log V + VE)$ running time with the Dijkstra algorithm and Fibonacci heap.

In general graphs

Main article: Edmonds's matching algorithm

There is a polynomial time algorithm to find a maximum matching or a maximum weight matching in a graph that is not bipartite; it is due to Jack Edmonds, is called the *paths, trees, and flowers* method or simply Edmonds's algorithm, and uses bidirected edges. A generalization of the same technique can also be used to find maximum independent sets in claw-free graphs. Edmonds' algorithm has subsequently been improved to run in time $O(\sqrt{VE})$ time, matching the time for bipartite maximum matching.

Another (randomized) algorithm by Mucha and Sankowski, based on the fast matrix multiplication algorithm, gives $O(V^{2.376})$ complexity.

Maximal matchings

A maximal matching can be found with a simple greedy algorithm. A maximum matching is also a maximal matching, and hence it is possible to find a *largest* maximal matching in polynomial time. However, no polynomial-time algorithm is known for finding a **minimum maximal matching**, that is, a maximal matching that contains the *smallest* possible number of edges.

Note that a maximal matching with k edges is an edge dominating set with k edges. Conversely, if we are given a minimum edge dominating set with k edges, we can construct a maximal matching with k edges in polynomial time. Therefore the problem of finding a minimum maximal matching is essentially equal to the problem of finding a minimum edge dominating set. Both of these two optimisation problems are known to be NP-hard; the decision versions of these problems are classical examples of NP-complete problems.^[2] Both problems can be approximated within factor 2 in polynomial time: simply find an arbitrary maximal matching M .^[3]

Counting problems

Main article: Hosoya index

The number of matchings in a graph is known as the Hosoya index of the graph. It is #P-complete to compute this quantity. It remains #P-complete in the special case of counting the number of perfect matchings in a given bipartite graph, because computing the permanent of an arbitrary 0–1 matrix (another #P-complete problem) is the same as computing the number of perfect matchings in the bipartite graph having the given matrix as its biadjacency matrix. However, there exists a fully polynomial time randomized approximation scheme for counting the number of bipartite matchings. A remarkable theorem of Kasteleyn states that the number of perfect matchings in a planar graph can be computed exactly in polynomial time via the FKT algorithm.

The number of perfect matchings in a complete graph K_n (with n even) is given by the double factorial $(n - 1)!!$. The numbers of matchings in complete graphs, without constraining the matchings to be perfect, are given by the telephone numbers.

Finding all maximally-matchable edges

One of the basic problems in matching theory is to find in a given graph all edges that may be extended to a maximum matching in the graph. (Such edges are called **maximally-matchable** edges, or **allowed** edges.) The best deterministic algorithm for solving this problem in general graphs runs in time $O(VE)$. There exists a randomized algorithm that solves this problem in time $\tilde{O}(V^{2.376})$. In the case of bipartite graphs, it is possible to find a single maximum matching and then use it in order to find all maximally-matchable edges in linear time; the resulting overall runtime is $O(V^{1/2}E)$ for general bipartite graphs and $O((V/\log V)^{1/2}E)$ for dense bipartite graphs with $E = \Theta(V^2)$. In cases where one of the maximum matchings is known upfront, the overall runtime of the algorithm is $O(V + E)$.

Characterizations and notes

König's theorem states that, in bipartite graphs, the maximum matching is equal in size to the minimum vertex cover. Via this result, the minimum vertex cover, maximum independent set, and maximum vertex biclique problems may be solved in polynomial time for bipartite graphs.

The marriage theorem (or Hall's Theorem) provides a characterization of bipartite graphs which have a perfect matching and the Tutte theorem provides a characterization for arbitrary graphs.

A perfect matching is a spanning 1-regular subgraph, a.k.a. a 1-factor. In general, a spanning k -regular subgraph is a k -factor.

Applications

A **Kekulé structure** of an aromatic compound consists of a perfect matching of its carbon skeleton, showing the locations of double bonds in the chemical structure. These structures are named after Friedrich August Kekulé von Stradonitz, who showed that benzene (in graph theoretical terms, a 6-vertex cycle) can be given such a structure.^[4]

The Hosoya index is the number of non-empty matchings plus one; it is used in computational chemistry and mathematical chemistry investigations for organic compounds.

References

- [1] Alan Gibbons, Algorithmic Graph Theory, Cambridge University Press, 1985, Chapter 5.
- [2] Edge dominating set (decision version) is discussed under the dominating set problem, which is the problem GT2 in Appendix A1.1. Minimum maximal matching (decision version) is the problem GT10 in Appendix A1.1.
- [3] Minimum edge dominating set (optimisation version) is the problem GT3 in Appendix B (page 370). Minimum maximal matching (optimisation version) is the problem GT10 in Appendix B (page 374). See also Minimum Edge Dominating Set (<http://www.nada.kth.se/~viggo/wwwcompendium/node13.html>) and Minimum Maximal Matching (<http://www.nada.kth.se/~viggo/wwwcompendium/node21.html>) in the web compendium (<http://www.nada.kth.se/~viggo/wwwcompendium/>).
- [4] See, e.g., .

Further reading

1. László Lovász; M. D. Plummer (1986), *Matching Theory*, North-Holland, ISBN 0-444-87916-1
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2001), *Introduction to Algorithms* (second ed.), MIT Press and McGraw–Hill, Chapter 26, pp. 643–700, ISBN 0-262-53196-8
3. András Frank (2004). *On Kuhn's Hungarian Method – A tribute from Hungary* (<http://www.cs.elte.hu/egres/tr/egres-04-14.pdf>) (Technical report). Egerváry Research Group.
4. Michael L. Fredman and Robert E. Tarjan (1987), "Fibonacci heaps and their uses in improved network optimization algorithms", *Journal of the ACM* **34** (3): 595–615, doi: 10.1145/28869.28874 (<http://dx.doi.org/10.1145/28869.28874>).
5. S. J. Cyvin and Ivan Gutman (1988), *Kekulé Structures in Benzenoid Hydrocarbons*, Springer-Verlag

6. Marek Karpinski and Wojciech Rytter (1998), *Fast Parallel Algorithms for Graph Matching Problems*, Oxford University Press, ISBN 978-0-19-850162-6

External links

- A graph library with Hopcroft–Karp and Push–Relabel-based maximum cardinality matching implementation (<http://lemon.cs.elte.hu/>)

Hopcroft–Karp algorithm for maximum matching in bipartite graphs

In computer science, the **Hopcroft–Karp algorithm** is an algorithm that takes as input a bipartite graph and produces as output a maximum cardinality matching – a set of as many edges as possible with the property that no two edges share an endpoint. It runs in $O(|E|\sqrt{|V|})$ time in the worst case, where E is set of edges in the graph, and V is set of vertices of the graph. In the case of dense graphs the time bound becomes $O(|V|^{2.5})$, and for random graphs it runs in near-linear time.

The algorithm was found by John Hopcroft and Richard Karp (1973). As in previous methods for matching such as the Hungarian algorithm and the work of Edmonds (1965), the Hopcroft–Karp algorithm repeatedly increases the size of a partial matching by finding augmenting paths. However, instead of finding just a single augmenting path per iteration, the algorithm finds a maximal set of shortest augmenting paths. As a result only $O(\sqrt{n})$ iterations are needed. The same principle has also been used to develop more complicated algorithms for non-bipartite matching with the same asymptotic running time as the Hopcroft–Karp algorithm.

Augmenting paths

A vertex that is not the endpoint of an edge in some partial matching M is called a *free vertex*. The basic concept that the algorithm relies on is that of an *augmenting path*, a path that starts at a free vertex, ends at a free vertex, and alternates between unmatched and matched edges within the path. If M is a matching, and P is an augmenting path relative to M , then the symmetric difference of the two sets of edges, $M \oplus P$, would form a matching with size $|M| + 1$. Thus, by finding augmenting paths, an algorithm may increase the size of the matching.

Conversely, suppose that a matching M is not optimal, and let P be the symmetric difference $M \oplus M^*$ where M^* is an optimal matching. Then P must form a collection of disjoint augmenting paths and cycles or paths in which matched and unmatched edges are of equal number; the difference in size between M and M^* is the number of augmenting paths in P . Thus, if no augmenting path can be found, an algorithm may safely terminate, since in this case M must be optimal.

An augmenting path in a matching problem is closely related to the augmenting paths arising in maximum flow problems, paths along which one may increase the amount of flow between the terminals of the flow. It is possible to transform the bipartite matching problem into a maximum flow instance, such that the alternating paths of the matching problem become augmenting paths of the flow problem.^[1] In fact, a generalization of the technique used in Hopcroft–Karp algorithm to arbitrary flow networks is known as Dinic's algorithm.

Input: Bipartite graph $G(U \cup V, E)$

Output: Matching $M \subseteq E$

$M \leftarrow \emptyset$

repeat

$\mathcal{P} \leftarrow \{P_1, P_2, \dots, P_k\}$ maximal set of vertex-disjoint shortest augmenting paths

```

 $M \leftarrow M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ 
until  $\mathcal{P} = \emptyset$ 

```

Algorithm

Let U and V be the two sets in the bipartition of G , and let the matching from U to V at any time be represented as the set M .

The algorithm is run in phases. Each phase consists of the following steps.

- A breadth-first search partitions the vertices of the graph into layers. The free vertices in U are used as the starting vertices of this search, and form the first layer of the partition. At the first level of the search, only unmatched edges may be traversed (since the free vertices in U are by definition not adjacent to any matched edges); at subsequent levels of the search, the traversed edges are required to alternate between matched and unmatched. That is, when searching for successors from a vertex in U , only unmatched edges may be traversed, while from a vertex in V only matched edges may be traversed. The search terminates at the first layer k where one or more free vertices in V are reached.
- All free vertices in V at layer k are collected into a set F . That is, a vertex v is put into F if and only if it ends a shortest augmenting path.
- The algorithm finds a maximal set of *vertex disjoint* augmenting paths of length k . This set may be computed by depth first search from F to the free vertices in U , using the breadth first layering to guide the search: the depth first search is only allowed to follow edges that lead to an unused vertex in the previous layer, and paths in the depth first search tree must alternate between matched and unmatched edges. Once an augmenting path is found that involves one of the vertices in F , the depth first search is continued from the next starting vertex.
- Every one of the paths found in this way is used to enlarge M .

The algorithm terminates when no more augmenting paths are found in the breadth first search part of one of the phases.

Analysis

Each phase consists of a single breadth first search and a single depth first search. Thus, a single phase may be implemented in linear time. Therefore, the first $\sqrt{|V|}$ phases, in a graph with $|V|$ vertices and $|E|$ edges, take time $O(|E|\sqrt{|V|})$.

It can be shown that each phase increases the length of the shortest augmenting path by at least one: the phase finds a maximal set of augmenting paths of the given length, so any remaining augmenting path must be longer. Therefore, once the initial $\sqrt{|V|}$ phases of the algorithm are complete, the shortest remaining augmenting path has at least $\sqrt{|V|}$ edges in it. However, the symmetric difference of the eventual optimal matching and of the partial matching M found by the initial phases forms a collection of vertex-disjoint augmenting paths and alternating cycles. If each of the paths in this collection has length at least $\sqrt{|V|}$, there can be at most $\sqrt{|V|}$ paths in the collection, and the size of the optimal matching can differ from the size of M by at most $\sqrt{|V|}$ edges. Since each phase of the algorithm increases the size of the matching by at least one, there can be at most $\sqrt{|V|}$ additional phases before the algorithm terminates.

Since the algorithm performs a total of at most $2\sqrt{|V|}$ phases, it takes a total time of $O(|E|\sqrt{|V|})$ in the worst case.

In many instances, however, the time taken by the algorithm may be even faster than this worst case analysis indicates. For instance, in the average case for sparse bipartite random graphs, Bast et al. (2006) (improving a

previous result of Motwani 1994) showed that with high probability all non-optimal matchings have augmenting paths of logarithmic length. As a consequence, for these graphs, the Hopcroft–Karp algorithm takes $O(\log |V|)$ phases and $O(|E|)$ time.

Comparison with other bipartite matching algorithms

For sparse graphs, the Hopcroft–Karp algorithm continues to have the best known worst-case performance, but for dense graphs a more recent algorithm by Alt et al. (1991) achieves a slightly better time bound, $O\left(n^{1.5}\sqrt{\frac{m}{\log n}}\right)$.

Their algorithm is based on using a push-relabel maximum flow algorithm and then, when the matching created by this algorithm becomes close to optimum, switching to the Hopcroft–Karp method.

Several authors have performed experimental comparisons of bipartite matching algorithms. Their results in general tend to show that the Hopcroft–Karp method is not as good in practice as it is in theory: it is outperformed both by simpler breadth-first and depth-first strategies for finding augmenting paths, and by push-relabel techniques.^[2]

Non-bipartite graphs

The same idea of finding a maximal set of shortest augmenting paths works also for finding maximum cardinality matchings in non-bipartite graphs, and for the same reasons the algorithms based on this idea take $O(\sqrt{|V|})$ phases. However, for non-bipartite graphs, the task of finding the augmenting paths within each phase is more difficult. Building on the work of several slower predecessors, Micali & Vazirani (1980) showed how to implement a phase in linear time, resulting in a non-bipartite matching algorithm with the same time bound as the Hopcroft–Karp algorithm for bipartite graphs. The Micali–Vazirani technique is complex, and its authors did not provide full proofs of their results; subsequently, a "clear exposition" was published by Peterson & Loui (1988) and alternative methods were described by other authors.^[3] In 2012, Vazirani offered a new simplified proof of the Micali-Vazirani algorithm.

Pseudocode

```
/*
G = G1 ∪ G2 ∪ {NIL}
where G1 and G2 are partition of graph and NIL is a special null vertex
*/

function BFS ()
    for v in G1
        if Pair_G1[v] == NIL
            Dist[v] = 0
            Enqueue(Q, v)
        else
            Dist[v] = ∞
    Dist[NIL] = ∞
    while Empty(Q) == false
        v = Dequeue(Q)
        if Dist[v] < Dist[NIL]
            for each u in Adj[v]
                if Dist[ Pair_G2[u] ] == ∞
                    Dist[ Pair_G2[u] ] = Dist[v] + 1
```

```

        Enqueue(Q, Pair_G2[u])
    return Dist[NIL] != ∞

function DFS (v)
    if v != NIL
        for each u in Adj[v]
            if Dist[ Pair_G2[u] ] == Dist[v] + 1
                if DFS(Pair_G2[u]) == true
                    Pair_G2[u] = v
                    Pair_G1[v] = u
                    return true
            Dist[v] = ∞
        return false
    return true

function Hopcroft-Karp
    for each v in G
        Pair_G1[v] = NIL
        Pair_G2[v] = NIL
    matching = 0
    while BFS() == true
        for each v in G1
            if Pair_G1[v] == NIL
                if DFS(v) == true
                    matching = matching + 1
    return matching

```

Notes

[1] , section 12.3, bipartite cardinality matching problem, pp. 469–470.

[2] ;;;.

[3] and .

References

- Ahuja, Ravindra K.; Magnanti, Thomas L.; Orlin, James B. (1993), *Network Flows: Theory, Algorithms and Applications*, Prentice-Hall.
- Alt, H.; Blum, N.; Mehlhorn, K.; Paul, M. (1991), "Computing a maximum cardinality matching in a bipartite graph in time $O\left(n^{1.5} \sqrt{\frac{m}{\log n}}\right)$ ", *Information Processing Letters* **37** (4): 237–240, doi: 10.1016/0020-0190(91)90195-N ([http://dx.doi.org/10.1016/0020-0190\(91\)90195-N](http://dx.doi.org/10.1016/0020-0190(91)90195-N)).
- Bast, Holger; Mehlhorn, Kurt; Schafer, Guido; Tamaki, Hisao (2006), "Matching algorithms are fast in sparse random graphs", *Theory of Computing Systems* **39** (1): 3–14, doi: 10.1007/s00224-005-1254-y (<http://dx.doi.org/10.1007/s00224-005-1254-y>).
- Blum, Norbert (2001), *A Simplified Realization of the Hopcroft-Karp Approach to Maximum Matching in General Graphs* (<http://theory.cs.uni-bonn.de/ftp/reports/cs-reports/2001/85232-CS.ps.gz>), Tech. Rep. 85232-CS, Computer Science Department, Univ. of Bonn.
- Chang, S. Frank; McCormick, S. Thomas (1990), *A faster implementation of a bipartite cardinality matching algorithm*, Tech. Rep. 90-MSC-005, Faculty of Commerce and Business Administration, Univ. of British

Columbia. As cited by Setubal (1996).

- Darby-Dowman, Kenneth (1980), *The exploitation of sparsity in large scale linear programming problems – Data structures and restructuring algorithms*, Ph.D. thesis, Brunel University. As cited by Setubal (1996).
- Edmonds, Jack (1965), "Paths, Trees and Flowers", *Canadian J. Math* **17**: 449–467, doi: 10.4153/CJM-1965-045-4 (<http://dx.doi.org/10.4153/CJM-1965-045-4>), MR 0177907 (<http://www.ams.org/mathscinet-getitem?mr=0177907>).
- Gabow, Harold N.; Tarjan, Robert E. (1991), "Faster scaling algorithms for general graph matching problems", *Journal of the ACM* **38** (4): 815–853, doi: 10.1145/115234.115366 (<http://dx.doi.org/10.1145/115234.115366>).
- Hopcroft, John E.; Karp, Richard M. (1973), "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs", *SIAM Journal on Computing* **2** (4): 225–231, doi: 10.1137/0202019 (<http://dx.doi.org/10.1137/0202019>).
- Micali, S.; Vazirani, V. V. (1980), "An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs", *Proc. 21st IEEE Symp. Foundations of Computer Science*, pp. 17–27, doi: 10.1109/SFCS.1980.12 (<http://dx.doi.org/10.1109/SFCS.1980.12>).
- Peterson, Paul A.; Loui, Michael C. (1988), "The general maximum matching algorithm of Micali and Vazirani", *Algorithmica* **3** (1-4): 511–533, doi: 10.1007/BF01762129 (<http://dx.doi.org/10.1007/BF01762129>).
- Motwani, Rajeev (1994), "Average-case analysis of algorithms for matchings and related problems", *Journal of the ACM* **41** (6): 1329–1356, doi: 10.1145/195613.195663 (<http://dx.doi.org/10.1145/195613.195663>).
- Setubal, João C. (1993), "New experimental results for bipartite matching", *Proc. Netflow93*, Dept. of Informatics, Univ. of Pisa, pp. 211–216. As cited by Setubal (1996).
- Setubal, João C. (1996), *Sequential and parallel experimental results with bipartite matching algorithms* (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.3539>), Tech. Rep. IC-96-09, Inst. of Computing, Univ. of Campinas.
- Vazirani, Vijay (2012), *An Improved Definition of Blossoms and a Simpler Proof of the MV Matching Algorithm* (<http://arxiv.org/abs/1210.4594>), CoRR abs/1210.4594.

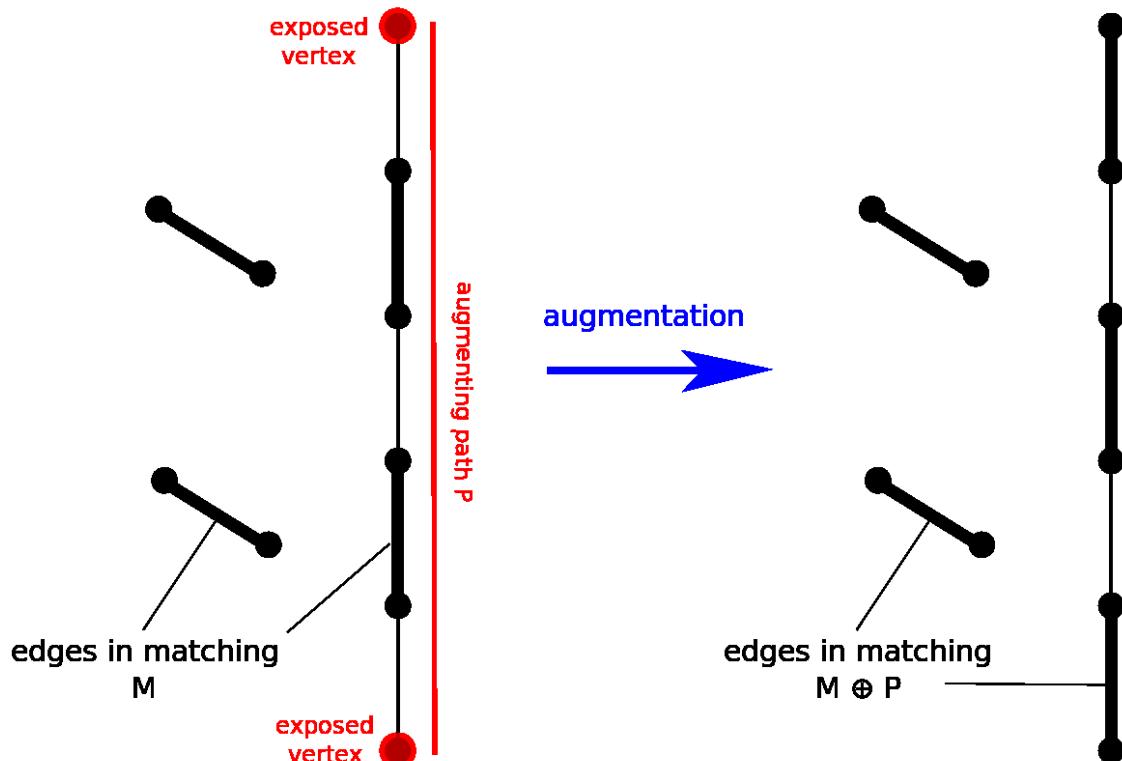
Edmonds's algorithm for maximum matching in non-bipartite graphs

The **blossom algorithm** is an algorithm in graph theory for constructing maximum matchings on graphs. The algorithm was developed by Jack Edmonds in 1961, and published in 1965. Given a general graph $G = (V, E)$, the algorithm finds a matching M such that each vertex in V is incident with at most one edge in M and $|M|$ is maximized. The matching is constructed by iteratively improving an initial empty matching along augmenting paths in the graph. Unlike bipartite matching, the key new idea is that an odd-length cycle in the graph (blossom) is contracted to a single vertex, with the search continuing iteratively in the contracted graph.

A major reason that the blossom algorithm is important is that it gave the first proof that a maximum-size matching could be found using a polynomial amount of computation time. Another reason is that it led to a linear programming polyhedral description of the matching polytope, yielding an algorithm for min-weight matching. As elaborated by Alexander Schrijver, further significance of the result comes from the fact that this was the first polytope whose proof of integrality "does not simply follow just from total unimodularity, and its description was a breakthrough in polyhedral combinatorics."

Augmenting paths

Given $G = (V, E)$ and a matching M of G , a vertex v is **exposed**, if no edge of M is incident with v . A path in G is an **alternating path**, if its edges are alternately not in M and in M (or in M and not in M). An **augmenting path** P is an alternating path that starts and ends at two distinct exposed vertices. A **matching augmentation** along an augmenting path P is the operation of replacing M with a new matching $M_1 = M \oplus P = (M \setminus P) \cup (P \setminus M)$.



It may be proven that a matching M is maximum if and only if there is no M -augmenting path in G . Hence, either a matching is maximum, or it can be augmented. Thus, starting from an initial matching, we can compute a maximum matching by augmenting the current matching with augmenting paths as long as we can find them, and return

whenever no augmenting paths are left. We can formalize the algorithm as follows:

```

INPUT: Graph  $G$ , initial matching  $M$  on  $G$ 
OUTPUT: maximum matching  $M^*$  on  $G$ 

A1 function find_maximum_matching(  $G$ ,  $M$  ) :  $M^*$ 
A2      $P \leftarrow \text{find\_augmenting\_path}( G, M )$ 
A3     if  $P$  is non-empty then
A4         return find_maximum_matching(  $G$ , augment  $M$  along  $P$  )
A5     else
A6         return  $M$ 
A7     end if
A8 end function

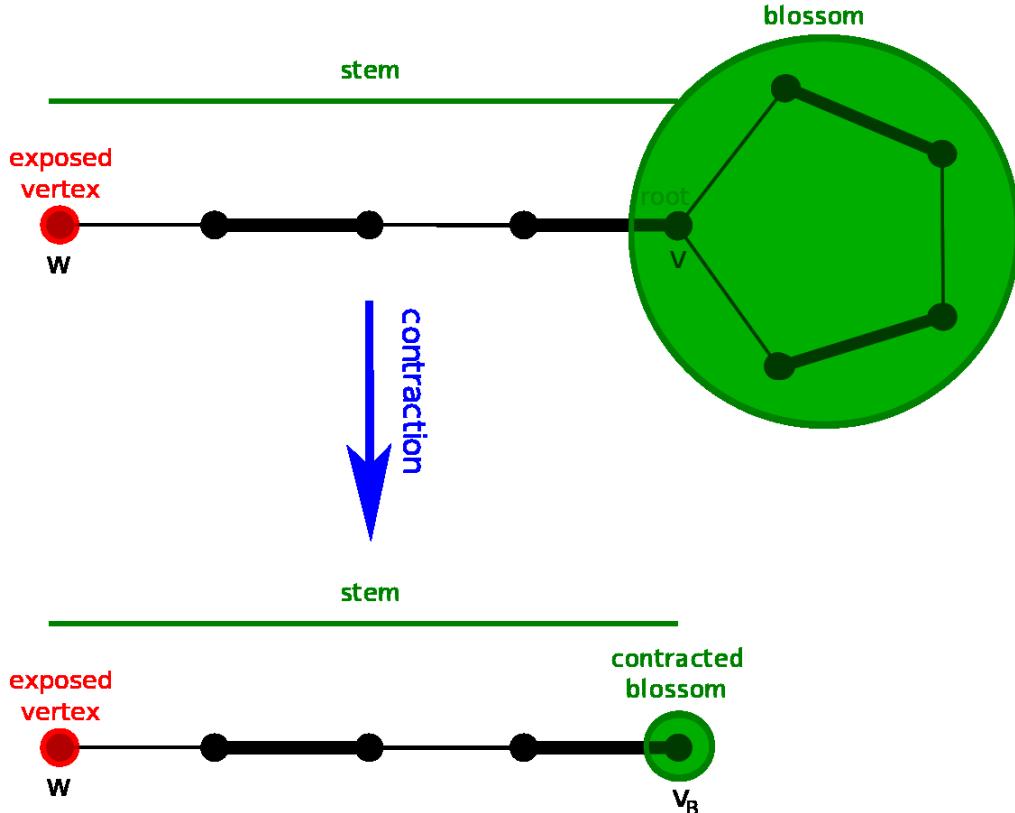
```

We still have to describe how augmenting paths can be found efficiently. The subroutine to find them uses blossoms and contractions.

Blossoms and contractions

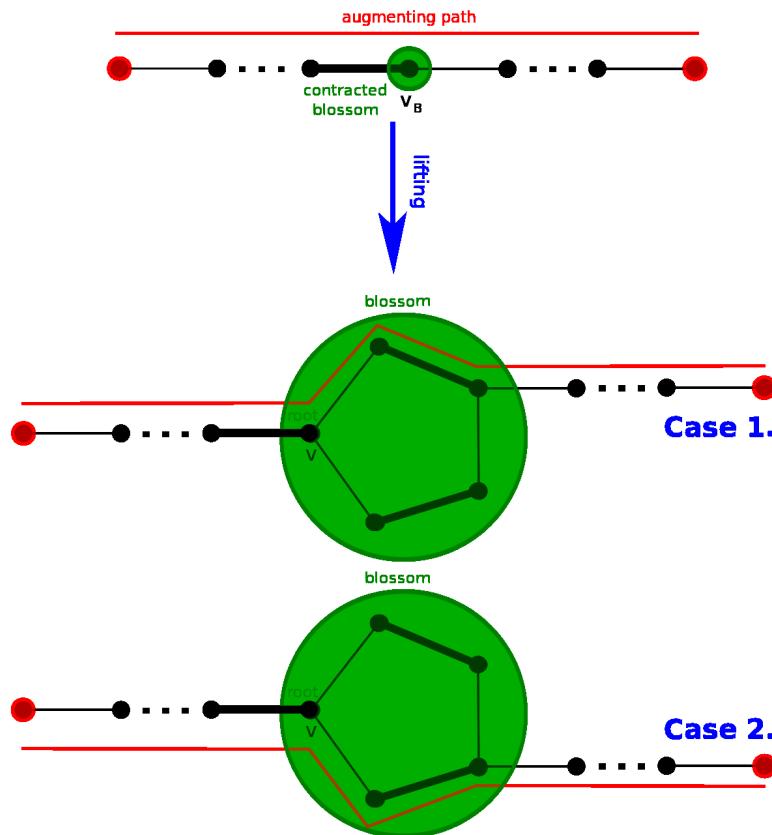
Given $G = (V, E)$ and a matching M of G , a *blossom* B is a cycle in G consisting of $2k + 1$ edges of which exactly k belong to M , and where one of the vertices v of the cycle (the *base*) is such that there exists an alternating path of even length (the *stem*) from v to an exposed vertex w .

We define the **contracted graph** G' as the graph obtained from G by contracting every edge of B , and define the **contracted matching** M' as the matching of G' corresponding to M .

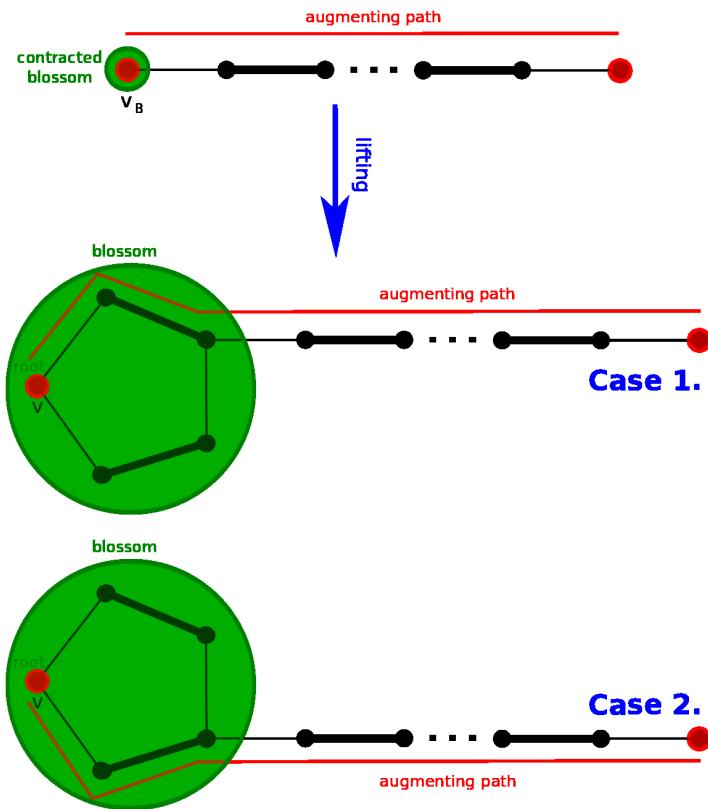


It can be shown that G' has an M' -augmenting path iff G has an M -augmenting path, and that any M' -augmenting path P' in G' can be **lifted** to a M -augmenting path in G by undoing the contraction by B so that the segment of P' (if any) traversing through v_B is replaced by an appropriate segment traversing through B . In more detail:

- if P' traverses through a segment $u \rightarrow v_B \rightarrow w$ in G' , then this segment is replaced with the segment $u \rightarrow (u' \rightarrow \dots \rightarrow w') \rightarrow w$ in G , where blossom vertices u' and w' and the side of B , ($u' \rightarrow \dots \rightarrow w'$), going from u' to w' are chosen to ensure that the new path is still alternating (u' is exposed with respect to $M \cap B$, $\{w', w\} \in E \setminus M$).



- if P' has an endpoint v_B , then the path segment $u \rightarrow v_B$ in G' is replaced with the segment $u \rightarrow (u' \rightarrow \dots \rightarrow v')$ in G , where blossom vertices u' and v' and the side of B , ($u' \rightarrow \dots \rightarrow v'$), going from u' to v' are chosen to ensure that the path is alternating (v' is exposed, $\{u', u\} \in E \setminus M$).



Thus blossoms can be contracted and search performed in the contracted graphs. This reduction is at the heart of Edmonds' algorithm.

Finding an augmenting path

The search for an augmenting path uses an auxiliary data structure consisting of a forest F whose individual trees correspond to specific portions of the graph G . In fact, the forest F is the same that would be used to find maximum matchings in bipartite graphs (without need for shrinking blossoms). In each iteration the algorithm either (1) finds an augmenting path, (2) finds a blossom and recurses onto the corresponding contracted graph, or (3) concludes there are no augmenting paths. The auxiliary structure is built by an incremental procedure discussed next.

The construction procedure considers vertices v and edges e in G and incrementally updates F as appropriate. If v is in a tree T of the forest, we let $\text{root}(v)$ denote the root of T . If both u and v are in the same tree T in F , we let $\text{distance}(u,v)$ denote the length of the unique path from u to v in T .

```

INPUT: Graph  $G$ , matching  $M$  on  $G$ 
OUTPUT: augmenting path  $P$  in  $G$  or empty path if none found

B01 function find_augmenting_path( G, M ) :  $P$ 
B02    $F \leftarrow$  empty forest
B03   unmark all vertices and edges in  $G$ , mark all edges of  $M$ 
B05   for each exposed vertex  $v$  do
B06     create a singleton tree  $\{ v \}$  and add the tree to  $F$ 
B07   end for
B08   while there is an unmarked vertex  $v$  in  $F$  with  $\text{distance}( v, \text{root}( v ) )$  even do
B09     while there exists an unmarked edge  $e = \{ v, w \}$  do
B10       if  $w$  is not in  $F$  then
B11         //  $w$  is matched, so add  $e$  and  $w$ 's matched edge to  $F$ 
B12          $x \leftarrow$  vertex matched to  $w$  in  $M$ 
```

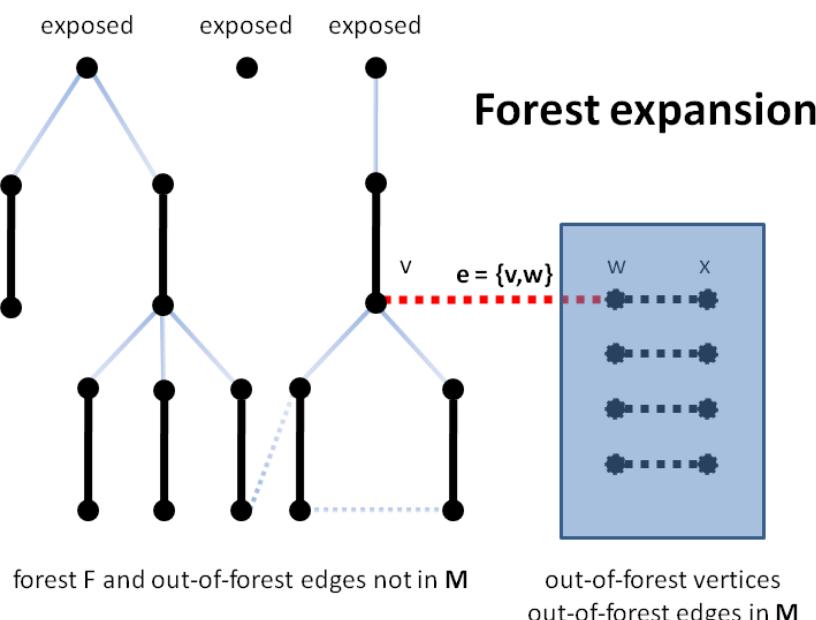
```

B12           add edges { v, w } and { w, x } to the tree of v
B13   else
B14       if distance( w, root( w ) ) is odd then
          // Do nothing.
B15   else
B16       if root( v ) ≠ root( w ) then
          // Report an augmenting path in F ∪ { e }.
B17       P ← path ( root( v ) → ... → v ) → ( w → ... → root( w ) )
B18       return P
B19   else
          // Contract a blossom in G and look for the path in the contracted graph.
B20   B ← blossom formed by e and edges on the path v → w in T
B21   G', M' ← contract G and M by B
B22   P' ← find_augmenting_path( G', M' )
B23   P ← lift P' to G
B24   return P
B25   end if
B26   end if
B27   end if
B28   mark edge e
B29   end while
B30   mark vertex v
B31 end while
B32 return empty path
B33 end function

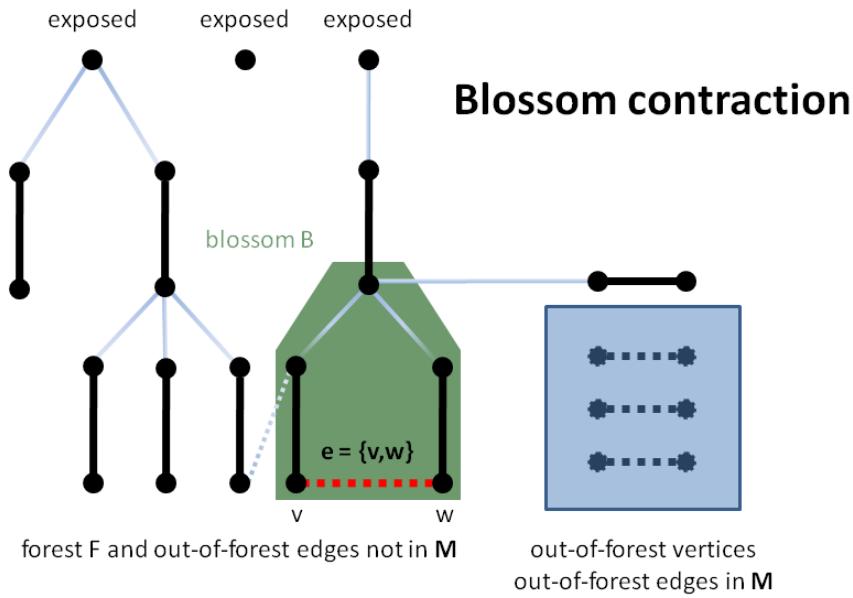
```

Examples

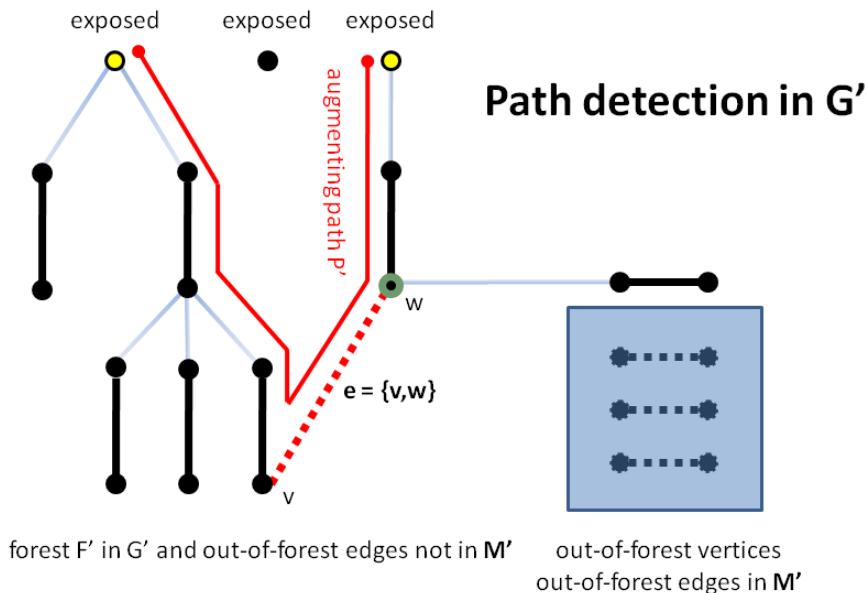
The following four figures illustrate the execution of the algorithm. Dashed lines indicate edges that are currently not present in the forest. First, the algorithm processes an out-of-forest edge that causes the expansion of the current forest (lines B10 – B12).

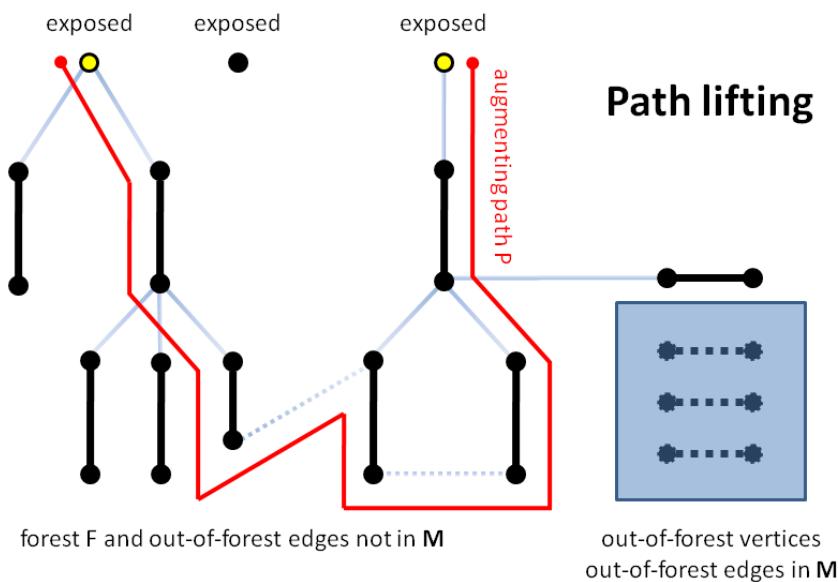


Next, it detects a blossom and contracts the graph (lines B20 – B21).



Finally, it locates an augmenting path P' in the contracted graph (line B22) and lifts it to the original graph (line B23). Note that the ability of the algorithm to contract blossoms is crucial here; the algorithm can not find P in the original graph directly because only out-of-forest edges between vertices at even distances from the roots are considered on line B17 of the algorithm.





Analysis

The forest F constructed by the `find_augmenting_path()` function is an alternating forest.

- a tree T in G is an **alternating tree** with respect to M , if
 - T contains exactly one exposed vertex r called the tree root
 - every vertex at an odd distance from the root has exactly two incident edges in T , and
 - all paths from r to leaves in T have even lengths, their odd edges are not in M and their even edges are in M .
- a forest F in G is an **alternating forest** with respect to M , if
 - its connected components are alternating trees, and
 - every exposed vertex in G is a root of an alternating tree in F .

Each iteration of the loop starting at line B09 either adds to a tree T in F (line B10) or finds an augmenting path (line B17) or finds a blossom (line B20). It is easy to see that the running time is $O(|V|^4)$. Micali and Vazirani show an algorithm that constructs maximum matching in $O(|E||V|^{1/2})$ time.

Bipartite matching

The algorithm reduces to the standard algorithm for matching in bipartite graphs when G is bipartite. As there are no odd cycles in G in that case, blossoms will never be found and one can simply remove lines B20 – B24 of the algorithm.

Weighted matching

The matching problem can be generalized by assigning weights to edges in G and asking for a set M that produces a matching of maximum (minimum) total weight. The weighted matching problem can be solved by a combinatorial algorithm that uses the unweighted Edmonds's algorithm as a subroutine. Kolmogorov provides an efficient C++ implementation of this.

References

Assignment problem

The **assignment problem** is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics. It consists of finding a maximum weight matching in a weighted bipartite graph.

In its most general form, the problem is as follows:

There are a number of *agents* and a number of *tasks*. Any agent can be assigned to perform any task, incurring some *cost* that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task and exactly one task to each agent in such a way that the *total cost* of the assignment is minimized.

If the numbers of agents and tasks are equal and the total cost of the assignment for all tasks is equal to the sum of the costs for each agent (or the sum of the costs for each task, which is the same thing in this case), then the problem is called the *linear assignment problem*. Commonly, when speaking of the *assignment problem* without any additional qualification, then the *linear assignment problem* is meant.

Algorithms and generalizations

The Hungarian algorithm is one of many algorithms that have been devised that solve the linear assignment problem within time bounded by a polynomial expression of the number of agents.

The assignment problem is a special case of the transportation problem, which is a special case of the minimum cost flow problem, which in turn is a special case of a linear program. While it is possible to solve any of these problems using the simplex algorithm, each specialization has more efficient algorithms designed to take advantage of its special structure. If the cost function involves quadratic inequalities it is called the quadratic assignment problem.

Example

Suppose that a taxi firm has three taxis (the agents) available, and three customers (the tasks) wishing to be picked up as soon as possible. The firm prides itself on speedy pickups, so for each taxi the "cost" of picking up a particular customer will depend on the time taken for the taxi to reach the pickup point. The solution to the assignment problem will be whichever combination of taxis and customers results in the least total cost.

However, the assignment problem can be made rather more flexible than it first appears. In the above example, suppose that there are four taxis available, but still only three customers. Then a fourth dummy task can be invented, perhaps called "sitting still doing nothing", with a cost of 0 for the taxi assigned to it. The assignment problem can then be solved in the usual way and still give the best solution to the problem.

Similar tricks can be played in order to allow more tasks than agents, tasks to which multiple agents must be assigned (for instance, a group of more customers than will fit in one taxi), or maximizing profit rather than minimizing cost.

Formal mathematical definition

The formal definition of the **assignment problem** (or **linear assignment problem**) is

Given two sets, A and T , of equal size, together with a weight function $C : A \times T \rightarrow \mathbf{R}$. Find a bijection $f : A \rightarrow T$ such that the cost function:

$$\sum_{a \in A} C(a, f(a))$$

is minimized.

Usually the weight function is viewed as a square real-valued matrix C , so that the cost function is written down as:

$$\sum_{a \in A} C_{a,f(a)}$$

The problem is "linear" because the cost function to be optimized as well as all the constraints contain only linear terms.

The problem can be expressed as a standard linear program with the objective function

$$\sum_{i \in A} \sum_{j \in T} C(i, j) x_{ij}$$

subject to the constraints

$$\sum_{j \in T} x_{ij} = 1 \text{ for } i \in A,$$

$$\sum_{i \in A} x_{ij} = 1 \text{ for } j \in T,$$

$$x_{ij} \geq 0 \text{ for } i, j \in A, T.$$

The variable x_{ij} represents the assignment of agent i to task j , taking value 1 if the assignment is done and 0 otherwise. This formulation allows also fractional variable values, but there is always an optimal solution where the variables take integer values. This is because the constraint matrix is totally unimodular. The first constraint requires that every agent is assigned to exactly one task, and the second constraint requires that every task is assigned exactly one agent.

Further reading

- Munkres, James. "Algorithms for the Assignment and Transportation Problems" ^[1]. *Journal of the Society for Industrial and Applied Mathematics Vol. 5, No. 1 (Mar., 1957)*, pp. 32-38.
- Brualdi, Richard A. (2006). *Combinatorial matrix classes*. Encyclopedia of Mathematics and Its Applications **108**. Cambridge: Cambridge University Press. ISBN 0-521-86565-4. Zbl 1106.05001 ^[2].
- Burkard, Rainer; M. Dell'Amico; S. Martello (2012). *Assignment Problems (Revised reprint)*. SIAM. ISBN 978-1-61197-222-1.

References

- [1] <http://www.jstor.org/stable/2098689>
[2] <http://www.zentralblatt-math.org/zmath/en/search/?format=complete&q=an:1106.05001>

Hungarian algorithm for the assignment problem

The **Hungarian method** is a combinatorial optimization algorithm that solves the assignment problem in polynomial time and which anticipated later primal-dual methods. It was developed and published by Harold Kuhn in 1955, who gave the name "Hungarian method" because the algorithm was largely based on the earlier works of two Hungarian mathematicians: Dénes König and Jenő Egerváry.^{[1][2]}

James Munkres reviewed the algorithm in 1957 and observed that it is (strongly) polynomial.^[3] Since then the algorithm has been known also as **Kuhn–Munkres algorithm** or **Munkres assignment algorithm**. The time complexity of the original algorithm was $O(n^4)$, however Edmonds and Karp, and independently Tomizawa noticed that it can be modified to achieve an $O(n^3)$ running time. Ford and Fulkerson extended the method to general transportation problems. In 2006, it was discovered that Carl Gustav Jacobi had solved the assignment problem in the 19th century, and the solution had been published posthumously in 1890 in Latin.^[4]

Layman's Explanation of the Assignment Problem

Say you have three workers: **Jim**, **Steve**, and **Alan**. You need to have one of them clean the bathroom, another sweep the floors, and the third wash the windows. What's the best (minimum-cost) way to assign the jobs? First we need a matrix of the costs of the workers doing the jobs.

	Clean bathroom	Sweep floors	Wash windows
Jim	\$3	\$3	\$3
Steve	\$3	\$2	\$3
Alan	\$3	\$3	\$2

Then the Hungarian method, when applied to the above table would give us the minimum cost it can be done with: Jim cleans the bathroom, Steve sweeps the floors, and Alan washes the windows.

Setting

We are given a nonnegative $n \times n$ matrix, where the element in the i -th row and j -th column represents the cost of assigning the j -th job to the i -th worker. We have to find an assignment of the jobs to the workers that has minimum cost. If the goal is to find the assignment that yields the maximum cost, the problem can be altered to fit the setting by replacing each cost with the maximum cost subtracted by the cost.

The algorithm is easier to describe if we formulate the problem using a bipartite graph. We have a complete bipartite graph $G=(S, T; E)$ with n worker vertices (S) and n job vertices (T), and each edge has a nonnegative cost $c(i,j)$. We want to find a perfect matching with minimum cost.

Let us call a function $y : (S \cup T) \mapsto \mathbb{R}$ a **potential** if $y(i) + y(j) \leq c(i, j)$ for each $i \in S, j \in T$. The value of potential y is $\sum_{v \in S \cup T} y(v)$. It can be seen that the cost of each perfect matching is at least the value of each

potential. The Hungarian method finds a perfect matching and a potential with equal cost/value which proves the optimality of both. In fact it finds a perfect matching of **tight edges**: an edge ij is called tight for a potential y if $y(i) + y(j) = c(i, j)$. Let us denote the subgraph of tight edges by G_y . The cost of a perfect matching in G_y (if there is one) equals the value of y .

The algorithm in terms of bipartite graphs

During the algorithm we maintain a potential y and an orientation of G_y (denoted by $\overrightarrow{G_y}$) which has the property that the edges oriented from T to S form a matching M . Initially, y is 0 everywhere, and all edges are oriented from S to T (so M is empty). In each step, either we modify y so that its value increases, or modify the orientation to obtain a matching with more edges. We maintain the invariant that all the edges of M are tight. We are done if M is a perfect matching.

In a general step, let $R_S \subseteq S$ and $R_T \subseteq T$ be the vertices not covered by M (so R_S consists of the vertices in S with no incoming edge and R_T consists of the vertices in T with no outgoing edge). Let Z be the set of vertices reachable in $\overrightarrow{G_y}$ from R_S by a directed path only following edges that are tight. This can be computed by breadth-first search.

If $R_T \cap Z$ is nonempty, then reverse the orientation of a directed path in $\overrightarrow{G_y}$ from R_S to R_T . Thus the size of the corresponding matching increases by 1.

If $R_T \cap Z$ is empty, then let $\Delta := \min\{c(i, j) - y(i) - y(j) : i \in Z \cap S, j \in T \setminus Z\}$. Δ is positive because there are no tight edges between $Z \cap S$ and $T \setminus Z$. Increase y by Δ on the vertices of $Z \cap S$ and decrease y by Δ on the vertices of $Z \cap T$. The resulting y is still a potential. The graph G_y changes, but it still contains M . We orient the new edges from S to T . By the definition of Δ the set Z of vertices reachable from R_S increases (note that the number of tight edges does not necessarily increase). We repeat these steps until M is a perfect matching, in which case it gives a minimum cost assignment. The running time of this version of the method is $O(n^4)$: M is augmented n times, and in a phase where M is unchanged, there are at most n potential changes (since Z increases every time). The time needed for a potential change is $O(n^2)$.

Matrix interpretation

Given n workers and tasks, and an $n \times n$ matrix containing the cost of assigning each worker to a task, find the cost minimizing assignment.

First the problem is written in the form of a matrix as given below

$$\begin{bmatrix} a1 & a2 & a3 & a4 \\ b1 & b2 & b3 & b4 \\ c1 & c2 & c3 & c4 \\ d1 & d2 & d3 & d4 \end{bmatrix}$$

where a, b, c and d are the workers who have to perform tasks 1, 2, 3 and 4. a_1, a_2, a_3, a_4 denote the penalties incurred when worker "a" does task 1, 2, 3, 4 respectively. The same holds true for the other symbols as well. The matrix is square, so each worker can perform only one task.

Step 1

Then we perform row operations on the matrix. To do this, the lowest of all a_i (i belonging to 1-4) is taken and is subtracted from each element in that row. This will lead to at least one zero in that row (We get multiple zeros when there are two equal elements which also happen to be the lowest in that row). This procedure is repeated for all rows. We now have a matrix with at least one zero per row. Now we try to assign tasks to agents such that each agent is doing only one task and the penalty incurred in each case is zero. This is illustrated below.

0	a2'	0'	a4'
b1'	b2'	b3'	0'
0'	c2'	c3'	c4'
d1'	0'	d3'	d4'

The zeros that are indicated as 0' are the assigned tasks.

Step 2

Sometimes it may turn out that the matrix at this stage cannot be used for assigning, as is the case in for the matrix below.

0	a2'	a3'	a4'
b1'	b2'	b3'	0
0	c2'	c3'	c4'
d1'	0	d3'	d4'

In the above case, no assignment can be made. Note that task 1 is done efficiently by both agent a and c. Both can't be assigned the same task. Also note that no one does task 3 efficiently. To overcome this, we repeat the above procedure for all columns (i.e. the minimum element in each column is subtracted from all the elements in that column) and then check if an assignment is possible.

Step 3

In most situations this will give the result, but if it is still not possible to assign then all zeros in the matrix must be covered by marking as few rows and/or columns as possible. The following procedure is one way to accomplish this:

Initially assign as many tasks as possible then do the following (assign tasks in rows 2, 3 and 4)

0	a2'	a3'	a4'
b1'	b2'	b3'	0'
0'	c2'	c3'	c4'
d1'	0'	0	d4'

Mark all rows having no assignments (row 1). Then mark all columns having zeros in marked row(s) (column 1). Then mark all rows having assignments in marked columns (row 3). Repeat this until a closed loop is obtained.

x				
0	a2'	a3'	a4'	x
b1'	b2'	b3'	0'	
0'	c2'	c3'	c4'	x
d1'	0'	0	d4'	

Now draw lines through all marked columns and unmarked rows.

x					
0	a2'	a3'	a4'	x	
b1'	b2'	b3'	0'		
0'	c2'	c3'	c4'	x	
d1'	0'	0	d4'		

The aforementioned detailed description is just one way to draw the minimum number of lines to cover all the 0's. Other methods work as well.

Step 4

From the elements that are left, find the lowest value. Subtract this from every unmarked element and add it to every element covered by two lines.

Repeat the procedure (steps 3–4) until an assignment is possible; this is when the minimum number of lines used to cover all the 0's is equal to the max(number of people, number of assignments), assuming dummy variables (usually the max cost) are used to fill in when the number of people is greater than the number of assignments.

Basically you find the second minimum cost among the two rows. The procedure is repeated until you are able to distinguish among the workers in terms of least cost.

Bibliography

- R.E. Burkard, M. Dell'Amico, S. Martello: *Assignment Problems* (Revised reprint). SIAM, Philadelphia (PA.) 2012. ISBN 978-1-61197-222-1
- M. Fischetti, "Lezioni di Ricerca Operativa", Edizioni Libreria Progetto Padova, Italia, 1995.
- R. Ahuja, T. Magnanti, J. Orlin, "Network Flows", Prentice Hall, 1993.
- S. Martello, "Jeno Egerváry: from the origins of the Hungarian algorithm to satellite communication". Central European Journal of Operations Research 18, 47–58, 2010

References

- [1] Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterly*, **2**:83–97, 1955. Kuhn's original publication.
- [2] Harold W. Kuhn, "Variants of the Hungarian method for assignment problems", *Naval Research Logistics Quarterly*, **3**: 253–258, 1956.
- [3] J. Munkres, "Algorithms for the Assignment and Transportation Problems", *Journal of the Society for Industrial and Applied Mathematics*, **5**(1):32–38, 1957 March.
- [4] <http://www.lix.polytechnique.fr/~ollivier/JACOBI/jacobiEngl.htm>

External links

- Bruff, Derek, "The Assignment Problem and the Hungarian Method", (http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf)
- Mordecai J. Golin, Bipartite Matching and the Hungarian Method (<http://www.cse.ust.hk/~golin/COMP572/Notes/Matching.pdf>), Course Notes, Hong Kong University of Science and Technology.
- R. A. Pilgrim, *Munkres' Assignment Algorithm. Modified for Rectangular Matrices* (<http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html>), Course notes, Murray State University.
- Mike Dawes, *The Optimal Assignment Problem* (<http://www.math.uwo.ca/~mdawes/courses/344/kuhn-munkres.pdf>), Course notes, University of Western Ontario.
- On Kuhn's Hungarian Method – A tribute from Hungary (<http://www.cs.elte.hu/egres/tr/egres-04-14.pdf>), András Frank, Egerváry Research Group, Pazmany P. setany 1/C, H1117, Budapest, Hungary.

- Lecture: Fundamentals of Operations Research - Assignment Problem - Hungarian Algorithm (<https://www.youtube.com/watch?v=BUGIhEecipE>), Prof. G. Srinivasan, Department of Management Studies, IIT Madras.
- Extension: Assignment sensitivity analysis (with $O(n^4)$ time complexity) (<http://www.roboticsproceedings.org/rss06/p16.html>), Liu, Shell.
- Solve any Assignment Problem online (<http://www.hungarianalgorithm.com/solve.php>), provides a step by step explanation of the Hungarian Algorithm.

Implementations

(Note that not all of these satisfy the $O(n^3)$ time constraint.)

- C implementation with $O(n^3)$ time complexity (<https://github.com/maandree/hungarian-algorithm-n3/blob/master/hungarian.c>)
- Java implementation of $O(n^3)$ time variant (https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/optimization/HungarianAlgorithm.java)
- Python implementation (<http://software.clapper.org/munkres/>) (see also here (<https://github.com/xtof-durr/makeSimple/blob/master/Munkres/kuhnMunkres.py>))
- Ruby implementation with unit tests (<https://github.com/evansenter/gene/blob/f515fd73cb9d6a22b4d4b146d70b6c2ec6a5125b/objects/extensions/hungarian.rb>)
- C# implementation (<http://noldorin.com/blog/2009/09/hungarian-algorithm-in-csharp/>)
- D implementation with unit tests (port of the Java $O(n^3)$ version) (<http://www.fantascienza.net/leonardo/so/hungarian.d>)
- Online interactive implementation (http://www.ifors.ms.unimelb.edu.au/tutorial/hungarian/welcome_frame.html) Please note that this implements a variant of the algorithm as described above.
- Graphical implementation with options (<http://web.axelero.hu/szilardandras/gaps.html>) (Java applet)
- Serial and parallel implementations. (<http://www.netlib.org/utk/lsi/pcwLSI/text/node220.html>)
- Implementation in Matlab and C (<http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=6543>)
- Perl implementation (<https://metacpan.org/module/Algorithm::Munkres>)
- Lisp implementation (<http://www.koders.com/lisp/fid7C3730AF4E356C65F93F20A6410814CBF5F40854.aspx?s=iso+3166>)
- C++ (STL) implementation (multi-functional bipartite graph version) (<http://students.cse.tamu.edu/lantao/codes/codes.php>)
- C++ implementation (<http://saebyn.info/2007/05/22/munkres-code-v2/>)
- C++ implementation of the $O(n^3)$ algorithm (http://dlib.net/optimization.html#max_cost_assignment) (BSD style open source licensed)
- Another C++ implementation with unit tests (<http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=hungarianAlgorithm>)
- Another Java implementation with JUnit tests (Apache 2.0) (<http://timefinder.svn.sourceforge.net/viewvc/timefinder/trunk/timefinder-algo/src/main/java/de/timefinder/algo/roomassignment/>)
- MATLAB implementation (<http://www.mathworks.com/matlabcentral/fileexchange/11609>)
- C implementation (<https://launchpad.net/lib-bipartite-match>)
- Javascript implementation (<http://twofourone.blogspot.com/2009/01/hungarian-algorithm-in-javascript.html>)
- The clue R package proposes an implementation, solve_LSAP (<http://cran.r-project.org/web/packages/clue/clue.pdf>)

FKT algorithm for counting matchings in planar graphs

The **FKT algorithm**, named after Fisher, Kasteleyn, and Temperley, counts the number of perfect matchings in a planar graph in polynomial time. This same task is #P-complete for general graphs. Counting the number of matchings, even for planar graphs, is also #P-complete. The key idea is to convert the problem into a Pfaffian computation of a skew-symmetric matrix derived from a planar embedding of the graph. The Pfaffian of this matrix is then computed efficiently using standard determinant algorithms.

History

The problem of counting planar perfect matchings has its roots in statistical mechanics and chemistry, where the original question was: If diatomic molecules are adsorbed on a surface, forming a single layer, how many ways can they be arranged? The partition function is an important quantity that encodes the statistical properties of a system at equilibrium and can be used to answer the previous question. However, trying to compute the partition function from its definition is not practical. Thus to exactly solve a physical system is to find an alternate form of the partition function for that particular physical system that is sufficiently simple to calculate exactly. In the early 1960s, the definition of *exactly solvable* was not rigorous. Computer science provided a rigorous definition with the introduction of polynomial time, which dates to 1965. Similarly, the notation of not *exactly solvable* should correspond to #P-hardness, which was defined in 1979.

Another type of physical system to consider is composed of dimers, which is a polymer with two atoms. The dimer model counts the number of dimer coverings of a graph. Another physical system to consider is the bonding of H₂O molecules in the form of ice. This can be modelled as a directed, 3-regular graph where the orientation of the edges at each vertex cannot all be the same. How many edge orientations does this model have?

Motivated by physical systems involving dimers, in 1961, Kasteleyn and Temperley and Fisher independently found the number of domino tilings for the m -by- n rectangle. This is equivalent to counting the number of perfect matchings for the m -by- n lattice graph. By 1967, Kasteleyn had generalized this result to all planar graphs.

Algorithm

Explanation

The main insight is that every non-zero term in the Pfaffian of the adjacency matrix of a graph G corresponds to a perfect matching. Thus, if one can find an orientation of G to align all signs of the terms in Pfaffian (no matter + or -), then the absolute value of the Pfaffian is just the number of perfect matchings in G . The FKT algorithm does such a task for a planar graph G .

Let $G = (V, E)$ be an undirected graph with adjacency matrix A . Define $PM(n)$ to be the set of partitions of n elements into pairs, then the number of perfect matchings in G is

$$\text{PerfMatch}(G) = \sum_{M \in PM(|V|)} \prod_{(i,j) \in M} A_{ij}.$$

Closely related to this is the Pfaffian for an n by n matrix A

$$\text{pf}(A) = \sum_{M \in PM(n)} \text{sgn}(M) \prod_{(i,j) \in M} A_{ij},$$

where $\text{sgn}(M)$ is the sign of the permutation M . A Pfaffian orientation of G is a directed graph H with (1, -1, 0)-adjacency matrix B such that $\text{pf}(B) = \text{PerfMatch}(G)$. In 1967, Kasteleyn proved that planar graphs have an

efficiently computable Pfaffian orientation. Specifically, for a planar graph G , let H be a directed version of G where an odd number of edges are oriented clockwise for every face in a planar embedding of G . Then H is a Pfaffian orientation of G .

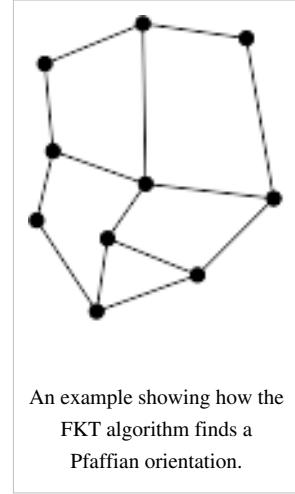
Finally, for any skew-symmetric matrix A ,

$$\text{pf}(A)^2 = \det(A),$$

where $\det(A)$ is the determinant of A . This result is due to Cayley. Since determinants are efficiently computable, so is $\text{PerfMatch}(G)$.

High-level description

1. Compute a planar embedding of G .
2. Compute a spanning tree T_1 of the input graph G .
3. Give an arbitrary orientation to each edge in G that is also in T_1 .
4. Use the planar embedding to create an (undirected) graph T_2 with the same vertex set as the dual graph of G .
5. Create an edge in T_2 between two vertices if their corresponding faces in G share an edge in G that is not in T_1 . (Note that T_2 is a tree.)
6. For each leaf v in T_2 (that is not also the root):
 1. Let e be the lone edge of G in the face corresponding to v that does not yet have an orientation.
 2. Give e an orientation such that the number of edges oriented clock-wise is odd.
3. Remove v from T_2 .
7. Return the absolute value of the Pfaffian of the $(1, -1, 0)$ -adjacency matrix of G , which is the absolute value of the square root of the determinant.



Generalizations

The sum of weighted perfect matchings can also be computed by using the Tutte matrix for the adjacency matrix in the last step.

Kuratowski's theorem states that

a finite graph is planar if and only if it contains no subgraph homeomorphic to K_5 (complete graph on five vertices) or $K_{3,3}$ (complete bipartite graph on two partitions of size three).

Vijay Vazirani generalized the FKT algorithm to graphs that do not contain a subgraph homeomorphic to $K_{3,3}$. Since counting the number of perfect matchings in a general graph is #P-complete, some restriction on the input graph is required unless FP, the function version of P, is equal to #P. Counting the number of matchings, which is known as the Hosoya index, is also #P-complete even for planar graphs.

Applications

The FKT algorithm has seen extensive use in holographic algorithms on planar graphs via matchgates. For example, consider the planar version of the ice model mentioned above, which has the technical name #PL-3-NAE-SAT (where NAE stands for "not all equal"). Valiant found a polynomial time algorithm for this problem which uses matchgates.

References

External links

- Presentation by Ashley Montanaro about the FKT algorithm (<http://www.damtp.cam.ac.uk/user/am994/presentations/matchings.pdf>)
- More history, information, and examples can be found in chapter 2 and section 5.3.2 of Dmitry Kamenetsky's PhD thesis (<https://digitalcollections.anu.edu.au/bitstream/1885/49338/2/02whole.pdf>)

Stable marriage problem

In mathematics, economics, and computer science, the **stable marriage problem (SMP)** is the problem of finding a **stable matching** between two sets of elements given a set of preferences for each element. A matching is a mapping from the elements of one set to the elements of the other set. A matching is stable whenever it is *not* the case that both:

- a. some given element A of the first matched set prefers some given element B of the second matched set over the element to which A is already matched, and
- b. B also prefers A over the element to which B is already matched

In other words, a matching is stable when there does not exist any alternative pairing (A, B) in which both A and B are individually better off than they would be with the element to which they are currently matched.

The stable marriage problem is commonly stated as:

Given n men and n women, where each person has ranked all members of the opposite sex with an unique number between 1 and n in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable".

Algorithms for finding solutions to the stable marriage problem have applications in a variety of real-world situations, perhaps the best known of these being in the assignment of graduating medical students to their first hospital appointments.^[1] In 2012, the Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel was awarded to Lloyd S. Shapley and Alvin E. Roth "for the theory of stable allocations and the practice of market design."

Solution

In 1962, David Gale and Lloyd Shapley proved that, for any equal number of men and women, it is always possible to solve the SMP and make all marriages stable. They presented an algorithm to do so.^{[2][3]}

The **Gale–Shapley algorithm** involves a number of "rounds" (or "iterations"). In the first round, first *a*) each unengaged man proposes to the woman he prefers most, and then *b*) each woman replies "maybe" to her suitor she most prefers and "no" to all other suitors. She is then provisionally "engaged" to the suitor she most prefers so far, and that suitor is likewise provisionally engaged to her. In each subsequent round, first *a*) each unengaged man proposes to the most-preferred woman to whom he has not yet proposed (regardless of whether the woman is already engaged), and then *b*) each woman replies "maybe" to her suitor she most prefers (whether her existing provisional partner or someone else) and rejects the rest (again, perhaps including her current provisional partner). The provisional nature of engagements preserves the right of an already-engaged woman to "trade up" (and, in the process, to "jilt" her until-then partner).

This algorithm guarantees that:

Everyone gets married

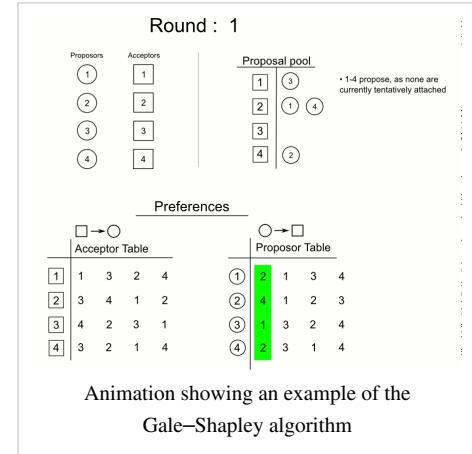
Once a woman becomes engaged, she is always engaged to someone. So, at the end, there cannot be a man and a woman both unengaged, as he must have proposed to her at some point (since a man will eventually propose to everyone, if necessary) and, being unengaged, she would have had to have said yes.

The marriages are stable

Let Alice be a woman and Bob be a man who are both engaged, but not to each other. Upon completion of the algorithm, it is not possible for both Alice and Bob to prefer each other over their current partners. If Bob prefers Alice to his current partner, he must have proposed to Alice before he proposed to his current partner. If Alice accepted his proposal, yet is not married to him at the end, she must have dumped him for someone she likes more, and therefore doesn't like Bob more than her current partner. If Alice rejected his proposal, she was already with someone she liked more than Bob.

Algorithm

```
function stableMatching {
    Initialize all  $m \in M$  and  $w \in W$  to free
    while  $\exists$  free man  $m$  who still has a woman  $w$  to propose to {
         $w = m$ 's highest ranked woman to whom he has not yet proposed
        if  $w$  is free
             $(m, w)$  become engaged
        else some pair  $(m', w)$  already exists
            if  $w$  prefers  $m$  to  $m'$ 
                 $(m, w)$  become engaged
                 $m'$  becomes free
            else
                 $(m', w)$  remain engaged
    }
}
```



Optimality of the solution

While the solution is stable, it is not necessarily optimal from all individuals' points of view. The traditional form of the algorithm is optimal for the initiator of the proposals and the stable, suitor-optimal solution may or may not be optimal for the reviewer of the proposals. An example is as follows:

There are three suitors (A,B,C) and three reviewers (X,Y,Z) which have preferences of:

A: YXZ B: ZYX C: XZY X: BAC Y: CBA Z: ACB

There are 3 stable solutions to this matching arrangement:

suitors get their first choice and reviewers their third (AY, BZ, CX)

all participants get their second choice (AX, BY, CZ)

reviewers get their first choice and suitors their third (AZ, BX, CY)

All three are stable because instability requires both participants to be happier with an alternative match. Giving one group their first choices ensures that the matches are stable because they would be unhappy with any other proposed match. Giving everyone their second choice ensures that any other match would be disliked by one of the parties. The algorithm converges in a single round on the suitor-optimal solution because each reviewer receives exactly one proposal, and therefore selects that proposal as its best choice, ensuring that each suitor has an accepted offer, ending the match. This asymmetry of optimality is driven by the fact that the suitors have the entire set to choose from, but reviewers choose between a limited subset of the suitors at any one time.

Similar problems

The **assignment problem** seeks to find a matching in a weighted bipartite graph that has maximum weight. Maximum weighted matchings do not have to be stable, but in some applications a maximum weighted matching is better than a stable one.

The **stable roommates problem** is similar to the stable marriage problem, but differs in that all participants belong to a single pool (instead of being divided into equal numbers of "men" and "women").

The **hospitals/residents problem** — also known as the **college admissions problem** — differs from the stable marriage problem in that the "women" can accept "proposals" from more than one "man" (e.g., a hospital can take multiple residents, or a college can take an incoming class of more than one student). Algorithms to solve the hospitals/residents problem can be *hospital-oriented* (female-optimal) or *resident-oriented* (male-optimal).

The **hospitals/residents problem with couples** allows the set of residents to include couples who must be assigned together, either to the same hospital or to a specific pair of hospitals chosen by the couple (e.g., a married couple want to ensure that they will stay together and not be stuck in programs that are far away from each other). The addition of couples to the hospitals/residents problem renders the problem NP-complete.^[4]

The matching with contracts problem is a generalization of matching problem, in which participants can be matched with different terms of contracts.^[5] An important special case of contracts is matching with flexible wages.^[6]

References

- [1] Stable Matching Algorithms (<http://www.dcs.gla.ac.uk/research/algorithms/stable/>)
- [2] D. Gale and L. S. Shapley: "College Admissions and the Stability of Marriage", *American Mathematical Monthly* 69, 9-14, 1962.
- [3] Harry Mairson: "The Stable Marriage Problem", *The Brandeis Review* 12, 1992 (online (<http://www1.cs.columbia.edu/~evs/intro/stable/writeup.html>)).
- [4] D. Gusfield and R. W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, p. 54; MIT Press, 1989.
- [5] John William Hatfield and Paul Milgrom, Matching with Contracts (with John Hatfield), *American Economic Review* 95(4), 2005, 913-935.
- [6] Vincent Crawford and Elsie Marie Knoer, Job matching with heterogeneous firms and workers, 1981, *Econometrica* 49 (2), 437-450.

Textbooks and other important references not cited in the text

- Dubins, L., and Freedman, D. (1981) "Machiavelli and the Gale–Shapley algorithm", *American Mathematical Monthly* 88(7), 485–494.
- Kleinberg, J., and Tardos, E. (2005) *Algorithm Design*, Chapter 1, pp 1–12. See companion website for the Text (<http://www.aw-bc.com/info/kleinberg/>).
- Knuth, D. E. (1976). *Marriages stables*. Montreal: Les Presses de l'Universite de Montreal.
- Roth, A. E. (1984). "The evolution of the labor market for medical interns and residents: A case study in game theory", *Journal of Political Economy* 92: 991–1016.
- Roth, A. E., and Sotomayor, M. A. O. (1990) *Two-sided matching: A study in game-theoretic modeling and analysis* Cambridge University Press.
- Shoham, Yoav; Leyton-Brown, Kevin (2009). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations* (<http://www.masfoundations.org>). New York: Cambridge University Press. ISBN 978-0-521-89943-7. See Section 10.6.4; downloadable free online (<http://www.masfoundations.org/download.html>).
- Schummer, J., and Vohra, R. V. (2007) "Mechanism design without money", in Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V. (Eds.). (2007) *Algorithmic game theory*, Cambridge, UK: Cambridge University Press, chapter 10, 243–265.

External links

- Interactive Flash Demonstration of SMP (<http://mathsite.math.berkeley.edu/smp/smp.html>)
- <http://kuznets.fas.harvard.edu/~aroth/alroth.html#NRMP>
- <http://www.dcs.gla.ac.uk/research/algorithms/stable/EGSapplet/EGS.html>
- Gale–Shapley JavaScript Demonstration (<http://www.sephlietz.com/gale-shapley/>)
- SMP Lecture Notes (<http://www.csee.wvu.edu/~ksmani/courses/fa01/random/lecnotes/lecture5.pdf>)

Stable roommates problem

In mathematics, economics and computer science, particularly in the fields of combinatorics, game theory and algorithms, the **stable-roommate problem (SRP)** is the problem of finding a **stable matching** — a matching in which there is no pair of elements, each from a different matched set, where each member of the pair prefers the other to their match. This is different from the stable-marriage problem in that the stable-roommates problem allows matches between any two elements, not just between classes of "men" and "women".

It is commonly stated as:

In a given instance of the stable-roommates problem (SRP), each of $2n$ participants ranks the others in strict order of preference. A matching is a set of n disjoint pairs of participants. A matching M in an instance of SRP is stable if there are no two participants x and y , each of whom prefers the other to his partner in M . Such a pair is said to block M , or to be a blocking pair with respect to M .

Solution

Unlike the stable marriage problem, a stable matching may fail to exist for certain sets of participants and their preferences. For a minimal counterexample, consider 4 people A, B, C, and D, whose rankings are:

A:(B,C,D), B:(C,A,D), C:(A,B,D), D:(A,B,C)

In this ranking, each of A, B, and C is the most favorite of someone. In any solution, one of A, B, or C must be paired with D and the other 2 with each other (for example AD and BC) yet D's partner and the one for whom D's partner is most favorite would each prefer to be with each other. In this example, AC is a more favorable pairing.

Algorithm

An efficient algorithm was given in (Irving 1985). The algorithm will determine, for any instance of the problem, whether a stable matching exists, and if so, will find such a matching.

Irving's algorithm has $O(n^2)$ complexity, provided suitable data structures are used to facilitate manipulation of the preference lists and identification of rotations.

The algorithm consists of two phases. In Phase 1, participants *propose* to each other, in a manner similar to that of the Gale-Shapley algorithm for the stable-marriage problem. Participants propose to each person on their preference list, in order, continuing to the next person if and when their current proposal is rejected. A participant rejects a proposal if he already holds, or subsequently receives, a proposal from someone he prefers. In this first phase, one participant might be rejected by all of the others, an indicator that no stable matching is possible. Otherwise, Phase 1 ends with each person holding a proposal from one of the others.

If q holds a proposal from p , then we remove from q 's list all the x after p , and symmetrically, for each of those x , we remove q from x 's list, so that q is first in p 's list; and p , last in q 's, since q and x cannot be partners in any stable matching. The resulting reduced set of preference lists together is called the Phase 1 table. In this table, if any reduced list is empty, then there is no stable matching. Otherwise, the Phase 1 table is a *stable table*. A stable table, by definition, is the set of preference lists together (the original table) with some entries removed, satisfying the following three conditions (where reduced list means a list in the stable table):

- (i) p is first on q 's reduced list if and only if q is last on p 's
- (ii) p is not on q 's reduced list if and only if q is not on p 's if and only if q prefers the last person on his list to p ; or p , the last person on his list to q
- (iii) no reduced list is empty

Stable tables have several important properties, which are used to justify the remainder of the procedure:

1. Any stable table must be a subtable of the Phase 1 table, where subtable is defined in the obvious sense, i.e. that the preference lists of the subtable are those of the supertable with some individuals removed from each other's lists.
2. In any stable table, if every reduced list contains *exactly* one individual, then pairing each individual with the single person on his list gives a stable matching.
3. If the stable roommates problem instance has a stable matching, then there is a stable matching contained in any one of the stable tables.
4. Any stable subtable of a stable table, and in particular any stable subtable that specifies a stable matching as in 2, can be obtained by a sequence of *rotation eliminations* on the stable table.

These rotation eliminations comprise Phase 2 of Irving's algorithm.

By 2, if each reduced list of the Phase 1 table contains exactly one individual, then this gives a matching.

Otherwise, the algorithm enters Phase 2. A *rotation* in a stable table T is defined as a sequence $(x_0, y_0), (x_1, y_1), \dots, (x_{k-1}, y_{k-1})$ such that the x_i are distinct, y_i is first on x_i 's reduced list (or x_i is last on y_i 's reduced list) and y_{i+1} is second on x_i 's reduced list, for $i = 0, \dots, k-1$ where the indices are taken modulo k . It follows that in any stable table with a reduced list containing at least two individuals, such a rotation always exists. To find it, start at such a p_0 containing at least two individuals in his reduced list, and define recursively q_{i+1} to be the second on p_i 's list and p_{i+1} to be the last on q_{i+1} 's list, until this sequence repeats some p_j , at which point a rotation is found: it is the sequence of pairs starting at the first occurrence of (p_j, q_j) and ending at the pair before the last occurrence. The sequence of p_i up until the p_j is called the *tail* of the rotation. The fact that it's a stable table in which this search occurs guarantees that each p_i has at least two individuals on his list.

To eliminate the rotation, y_i rejects x_i so that x_i proposes to y_{i+1} , for each i . To restore the stable table properties (i) and (ii), for each i , all successors of x_{i-1} are removed from y_i 's list, and y_i is removed from their lists. If a reduced list becomes empty during these removals, then there is no stable matching. Otherwise, the new table is again a stable table, and either already specifies a matching since each list contains exactly one individual or there remains another rotation to find and eliminate, so the step is repeated.

Phase 2 of the algorithm can now be summarized as follows:

```

T = Phase 1 table;
while (true) {
    identify a rotation r in T;
    eliminate r from T;
    if some list in T becomes empty,
        return null; (no stable matching can exist)
    else if (each reduced list in T has size 1)
        return the matching M = {{x, y} | x and y are on each
other's lists in T}; (this is a stable matching)
}

```

To achieve an $O(n^2)$ running time, a ranking matrix whose entry at row i and column j is the position of the j th individual in the i th's list; this takes $O(n^2)$ time. With the ranking matrix, checking whether an individual prefers one to another can be done in constant time by comparing their ranks in the matrix. Furthermore, instead of explicitly removing elements from the preference lists, the indices of the first, second and last on each individual's reduced list are maintained. The first individual that is *unmatched*, i.e. has at least two in his reduced lists, is also maintained. Then, in Phase 2, the sequence of p_i "traversed" to find a rotation is stored in a list, and an array is used to mark individuals as having been visited, as in a standard depth-first search graph traversal. After the elimination of a rotation, we continue to store only its tail, if any, in the list and as visited in the array, and start the search for the next rotation at the last individual on the tail, and otherwise at the next unmatched individual if there is no tail. This reduces repeated traversal of the tail, since it is largely unaffected by the elimination of the rotation.

Example

The following are the preference lists for a Stable Roommates instance involving 6 participants 1, 2, 3, 4, 5, 6.

1: 3 4 2 6 5
 2: 6 5 4 1 3
 3: 2 4 5 1 6
 4: 5 2 3 6 1
 5: 3 1 2 4 6
 6: 5 1 3 4 2

A possible execution of Phase 1 consists of the following sequence of proposals and rejections, where \rightarrow represents *proposes to* and \times represents *rejects*.

1 \rightarrow 3
 2 \rightarrow 6
 3 \rightarrow 2
 4 \rightarrow 5
 5 \rightarrow 3; 3 \times 1
 1 \rightarrow 4
 6 \rightarrow 5; 5 \times 6
 6 \rightarrow 1

So Phase 1 ends with the following reduced preference lists:

1: 3 4 2 6 5
 2: 6 5 4 1 3
 3: 2 4 5 1 6
 4: 5 2 3 6 1
 5: 3 1 2 4 6
 6: 5 1 3 4 2

In Phase 2, the rotation $r_1 = (1,4), (3,2)$ is first identified. This is because 2 is 1's second favorite, and 4 is the second favorite of 3. Eliminating r_1 gives:

1: 3 4 2 6 5
 2: 6 5 4 1 3
 3: 2 4 5 1 6
 4: 5 2 3 6 1
 5: 3 1 2 4 6
 6: 5 1 3 4 2

Next, the rotation $r_2 = (1,2), (2,6), (4,5)$ is identified, and its elimination yields:

1: 3 4 2 6 5
 2: 6 5 4 1 3
 3: 2 4 5 1 6
 4: 5 2 3 6 1
 5: 3 1 2 4 6
 6: 5 1 3 4 2

Hence 1 and 6 are matched. Finally, the rotation $r_3 = (2,5), (3,4)$ is identified, and its elimination gives:

1: 3 4 2 6 5
 2: 6 5 4 1 3
 3: 2 4 5 1 6
 4: 5 2 3 6 1

5 : 3 1 2 4 6
 6 : 5 1 3 4 2

Hence the matching $\{\{1, 6\}, \{2, 4\}, \{3, 5\}\}$ is stable.

References

- Irving, Robert W. (1985), "An efficient algorithm for the "stable roommates" problem", *Journal of Algorithms* **6** (4): 577–595, doi:10.1016/0196-6774(85)90033-1^[1]
- Irving, Robert W.; Manlove, David F. (2002), "The Stable Roommates Problem with Ties"^[2], *Journal of Algorithms* **43** (1): 85–105, doi:10.1006/jagm.2002.1219^[3]
- Gusfield, Daniel M.; Irving, Robert W. (1989), "The Stable Marriage Problem: Structure and Algorithms", *MIT Press*

Further reading

- Fleiner, Tamás; Irving, Robert W.; Manlove, David F. (2007), "An efficient algorithm for the "stable roommates" problem", *Theoretical Computer Science* **381** (1-3): 162–176, doi:10.1016/j.tcs.2007.04.029^[4]

References

- [1] [http://dx.doi.org/10.1016/0196-6774\(85\)90033-1](http://dx.doi.org/10.1016/0196-6774(85)90033-1)
 [2] <http://eprints.gla.ac.uk/11/01/SRT.pdf>
 [3] <http://dx.doi.org/10.1006/jagm.2002.1219>
 [4] <http://dx.doi.org/10.1016/j.tcs.2007.04.029>

Permanent

For other uses, see Permanent (disambiguation).

The **permanent** of a square matrix in linear algebra is a function of the matrix similar to the determinant. The permanent, as well as the determinant, is a polynomial in the entries of the matrix. Both permanent and determinant are special cases of a more general function of a matrix called the immanant.

Definition

The permanent of an n -by- n matrix $A = (a_{i,j})$ is defined as

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^n a_{i,\sigma(i)}.$$

The sum here extends over all elements σ of the symmetric group S_n ; i.e. over all permutations of the numbers 1, 2, ..., n .

For example,

$$\text{perm} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad + bc,$$

and

$$\text{perm} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = aei + bfg + cdh + ceg + bdi + afh.$$

The definition of the permanent of A differs from that of the determinant of A in that the signatures of the permutations are not taken into account.

The permanent of a matrix A is denoted $\text{perm } A$, $\text{perm} A$, or $\text{Per } A$, sometimes with parentheses around the argument. In his monograph, Minc (1984) uses $\text{Per}(A)$ for the permanent of rectangular matrices, and uses $\text{per}(A)$ when A is a square matrix. Muir (1882) uses the notation $\begin{array}{|c|c|} \hline + & + \\ \hline \end{array}$.

The word, *permanent*, originated with Cauchy in 1812 as "fonctions symétriques permanentes" for a related type of function, and was used by Muir (1882) in the modern, more specific, sense.

Properties and applications

If one views the permanent as a map that takes n vectors as arguments, then it is a multilinear map and it is symmetric (meaning that any order of the vectors results in the same permanent). Furthermore, given a square matrix $A = (a_{ij})$ of order n , we have:

- $\text{perm}(A)$ is invariant under arbitrary permutations of the rows and/or columns of A . This property may be written symbolically as $\text{perm}(A) = \text{perm}(PAQ)$ for any appropriately sized permutation matrices P and Q ,
- multiplying any single row or column of A by a scalar s changes $\text{perm}(A)$ to $s \cdot \text{perm}(A)$,
- $\text{perm}(A)$ is invariant under transposition, that is, $\text{perm}(A) = \text{perm}(A^\top)$.

If $A = (a_{ij})$ and $B = (b_{ij})$ are square matrices of order n then,

$$\text{perm}(A + B) = \sum_{s,t} \text{perm}(a_{ij})_{i \in s, j \in t} \text{perm}(b_{ij})_{i \in \bar{s}, j \in \bar{t}},$$

where s and t are subsets of the same size of $\{1, 2, \dots, n\}$ and \bar{s}, \bar{t} are their respective complements in that set.

On the other hand, the basic multiplicative property of determinants is not valid for permanents. A simple example shows that this is so.

$$4 = \text{perm} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \text{perm} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \neq \text{perm} \left(\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right) = \text{perm} \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} = 8.$$

A formula similar to Laplace's for the development of a determinant along a row, column or diagonal is also valid for the permanent; all signs have to be ignored for the permanent. For example, expanding along the first column,

$$\text{perm} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 \end{pmatrix} = 1 \cdot \text{perm} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + 2 \cdot \text{perm} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + 3 \cdot \text{perm} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} + 4 \cdot \text{perm} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 1(1) + 2(1) + 3(1) + 4(1) = 10,$$

while expanding along the last row gives,

$$\text{perm} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 \end{pmatrix} = 4 \cdot \text{perm} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} + 0 \cdot \text{perm} \begin{pmatrix} 1 & 1 & 1 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} + 0 \cdot \text{perm} \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 0 \\ 3 & 1 & 0 \end{pmatrix} + 1 \cdot \text{perm} \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 0 \\ 3 & 0 & 0 \end{pmatrix} = 4(1) + 0 + 0 + 1(6) = 10.$$

Unlike the determinant, the permanent has no easy geometrical interpretation; it is mainly used in combinatorics and in treating boson Green's functions in quantum field theory. However, it has two graph-theoretic interpretations: as the sum of weights of cycle covers of a directed graph, and as the sum of weights of perfect matchings in a bipartite graph.

Cycle covers

Any square matrix $A = (a_{ij})$ can be viewed as the adjacency matrix of a weighted directed graph, with a_{ij} representing the weight of the arc from vertex i to vertex j . A cycle cover of a weighted directed graph is a collection of vertex-disjoint directed cycles in the digraph that covers all vertices in the graph. Thus, each vertex i in the digraph has a unique "successor" $\sigma(i)$ in the cycle cover, and σ is a permutation on $\{1, 2, \dots, n\}$ where n is the number of vertices in the digraph. Conversely, any permutation σ on $\{1, 2, \dots, n\}$ corresponds to a cycle cover

in which there is an arc from vertex i to vertex $\sigma(i)$ for each i .

If the weight of a cycle-cover is defined to be the product of the weights of the arcs in each cycle, then

$$\text{Weight}(\sigma) = \prod_{i=1}^n a_{i,\sigma(i)}.$$

The permanent of an $n \times n$ matrix A is defined as

$$\text{perm}(A) = \sum_{\sigma} \prod_{i=1}^n a_{i,\sigma(i)}$$

where σ is a permutation over $\{1, 2, \dots, n\}$. Thus the permanent of A is equal to the sum of the weights of all cycle-covers of the digraph.

Perfect matchings

A square matrix $A = (a_{ij})$ can also be viewed as the adjacency matrix of a bipartite graph which has vertices x_1, x_2, \dots, x_n on one side and y_1, y_2, \dots, y_n on the other side, with a_{ij} representing the weight of the edge from vertex x_i to vertex y_j . If the weight of a perfect matching σ that matches x_i to $y_{\sigma(i)}$ is defined to be the product of the weights of the edges in the matching, then

$$\text{Weight}(\sigma) = \prod_{i=1}^n a_{i,\sigma(i)}.$$

Thus the permanent of A is equal to the sum of the weights of all perfect matchings of the graph.

Permanents of (0,1) matrices

The permanents of matrices that only have 0 and 1 as entries are often the answers to certain counting questions involving the structures that the matrices represent. This is particularly true of adjacency matrices in graph theory and incidence matrices of symmetric block designs.

In an unweighted, directed, simple graph (a *digraph*), if we set each a_{ij} to be 1 if there is an edge from vertex i to vertex j , then each nonzero cycle cover has weight 1, and the adjacency matrix has 0-1 entries. Thus the permanent of a (0,1)-matrix is equal to the *number* of vertex cycle covers of an unweighted directed graph.

For an unweighted bipartite graph, if we set $a_{ij} = 1$ if there is an edge between the vertices x_i and y_j and $a_{ij} = 0$ otherwise, then each perfect matching has weight 1. Thus the number of perfect matchings in G is equal to the permanent of matrix A .^[1]

Let $\Omega(n,k)$ be the class of all (0,1)-matrices of order n with each row and column sum equal to k . Every matrix A in this class has $\text{perm}(A) > 0$. The incidence matrices of projective planes are in the class $\Omega(n^2 + n + 1, n + 1)$ for n an integer > 1 . The permanents corresponding to the smallest projective planes have been calculated. For $n = 2, 3$, and 4 the values are 24, 3852 and 18,534,400 respectively. Let Z be the incidence matrix of the projective plane with $n = 2$, the Fano plane. Remarkably, $\text{perm}(Z) = 24 = |\det(Z)|$, the absolute value of the determinant of Z . This is a consequence of Z being a circulant matrix and the theorem:

If A is a circulant matrix in the class $\Omega(n,k)$ then if $k > 3$, $\text{perm}(A) > |\det(A)|$ and if $k = 3$, $\text{perm}(A) = |\det(A)|$.

Furthermore, when $k = 3$, by permuting rows and columns, A can be put into the form of a direct sum of e copies of the matrix Z and consequently, $n = 7e$ and $\text{perm}(A) = 24^e$.

Permanents can also be used to calculate the number of permutations with restricted (prohibited) positions. For the standard n -set, $\{1, 2, \dots, n\}$, let $A = (a_{ij})$ be the (0,1)-matrix where $a_{ij} = 1$ if $i \rightarrow j$ is allowed in a permutation and $a_{ij} = 0$ otherwise. Then $\text{perm}(A)$ counts the number of permutations of the n -set which satisfy all the restrictions. Two well known special cases of this are the solution of the derangement problem (the number of permutations with no fixed points) given by:

$$\text{perm}(J - I) = \text{perm} \begin{pmatrix} 0 & 1 & 1 & \dots & 1 \\ 1 & 0 & 1 & \dots & 1 \\ 1 & 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \dots & 0 \end{pmatrix} = n! \sum_{i=0}^n \frac{(-1)^i}{i!},$$

where J is the all 1's matrix and I is the identity matrix, each of order n , and the solution to the ménage problem given by:

$$\text{perm}(J - I - I') = \text{perm} \begin{pmatrix} 0 & 0 & 1 & \dots & 1 \\ 1 & 0 & 0 & \dots & 1 \\ 1 & 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & 1 & \dots & 0 \end{pmatrix} = 2 \cdot n! \sum_{k=0}^n (-1)^k \frac{2n}{2n-k} \binom{2n-k}{k} (n-k)!,$$

where I' is the $(0,1)$ -matrix whose only non-zero entries are on the first superdiagonal.

The following result was conjectured by H. Minc in 1967 and proved by L. M. Brégman in 1973.

Theorem: Let A be an $n \times n$ $(0,1)$ -matrix with r_i ones in row i , $1 \leq i \leq n$. Then

$$\text{perm} A \leq \prod_{i=1}^n (r_i)!^{1/r_i}.$$

Van der Waerden's conjecture

In 1926 Van der Waerden conjectured that the minimum permanent among all $n \times n$ doubly stochastic matrices is $n!/n^n$, achieved by the matrix for which all entries are equal to $1/n$. Proofs of this conjecture were published in 1980 by B. Gyires and in 1981 by G. P. Egorychev^[2] and D. I. Falikman; Egorychev's proof is an application of the Alexandrov–Fenchel inequality.^[3] For this work, Egorychev and Falikman won the Fulkerson Prize in 1982.^[4]

Computation

Main articles: Computing the permanent and Permanent is sharp-P-complete

The naïve approach, using the definition, of computing permanents is computationally infeasible even for relatively small matrices. One of the fastest known algorithms is due to H. J. Ryser (Ryser (1963, p. 27)). Ryser's method is based on an inclusion–exclusion formula that can be given^[5] as follows: Let A_k be obtained from A by deleting k columns, let $P(A_k)$ be the product of the row-sums of A_k , and let Σ_k be the sum of the values of $P(A_k)$ over all possible A_k . Then

$$\text{perm}(A) = \sum_{k=0}^{n-1} (-1)^k \Sigma_k.$$

It may be rewritten in terms of the matrix entries as follows:

$$\text{perm}(A) = (-1)^n \sum_{S \subseteq \{1, \dots, n\}} (-1)^{|S|} \prod_{i=1}^n \sum_{j \in S} a_{ij}.$$

The permanent is believed to be more difficult to compute than the determinant. While the determinant can be computed in polynomial time by Gaussian elimination, Gaussian elimination cannot be used to compute the permanent. Moreover, computing the permanent of a $(0,1)$ -matrix is #P-complete. Thus, if the permanent can be computed in polynomial time by any method, then $\mathbf{FP} = \#\mathbf{P}$, which is an even stronger statement than $\mathbf{P} = \mathbf{NP}$. When the entries of A are nonnegative, however, the permanent can be computed approximately in probabilistic polynomial time, up to an error of ϵM , where M is the value of the permanent and $\epsilon > 0$ is arbitrary.

MacMahon's Master Theorem

Main article: MacMahon Master theorem

Another way to view permanents is via multivariate generating functions. Let $A = (a_{ij})$ be a square matrix of order n . Consider the multivariate generating function:

$$F(x_1, x_2, \dots, x_n) = \prod_{i=1}^n \left(\sum_{j=1}^n a_{ij} x_j \right) = \left(\sum_{j=1}^n a_{1j} x_j \right) \left(\sum_{j=1}^n a_{2j} x_j \right) \cdots \left(\sum_{j=1}^n a_{nj} x_j \right).$$

The coefficient of $x_1 x_2 \dots x_n$ in $F(x_1, x_2, \dots, x_n)$ is $\text{perm}(A)$.

As a generalization, for any sequence of n non-negative integers, s_1, s_2, \dots, s_n define:

$$\text{perm}^{(s_1, s_2, \dots, s_n)}(A) := \text{coefficient of } x_1^{s_1} x_2^{s_2} \cdots x_n^{s_n} \text{ in } \left(\sum_{j=1}^n a_{1j} x_j \right)^{s_1} \left(\sum_{j=1}^n a_{2j} x_j \right)^{s_2} \cdots \left(\sum_{j=1}^n a_{nj} x_j \right)^{s_n}.$$

MacMahon's Master Theorem relating permanents and determinants is:

$$\text{perm}^{(s_1, s_2, \dots, s_n)}(A) = \text{coefficient of } x_1^{s_1} x_2^{s_2} \cdots x_n^{s_n} \text{ in } \frac{1}{\text{Det}(I - XA)},$$

where I is the order n identity matrix and X is the diagonal matrix with diagonal $[x_1, x_2, \dots, x_n]$.

Permanents of rectangular matrices

The permanent function can be generalized to apply to non-square matrices. Indeed, several authors make this the definition of a permanent and consider the restriction to square matrices a special case.^[6] Specifically, for an $m \times n$ matrix $A = (a_{ij})$ with $m \leq n$, define

$$\text{perm}(A) = \sum_{\sigma \in P(n,m)} a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{m\sigma(m)}$$

where $P(n,m)$ is the set of all m -permutations of the n -set $\{1, 2, \dots, n\}$.

Ryser's computational result for permanents also generalizes. If A is an $m \times n$ matrix with $m \leq n$, let A_k be obtained from A by deleting k columns, let $P(A_k)$ be the product of the row-sums of A_k , and let σ_k be the sum of the values of $P(A_k)$ over all possible A_k . Then

$$\text{perm}(A) = \sum_{k=0}^{m-1} (-1)^k \binom{n-m+k}{k} \sigma_{n-m+k}.$$

Systems of distinct representatives

The generalization of the definition of a permanent to non-square matrices allows the concept to be used in a more natural way in some applications. For instance:

Let S_1, S_2, \dots, S_m be subsets (not necessarily distinct) of an n -set with $m \leq n$. The incidence matrix of this collection of subsets is an $m \times n$ (0,1)-matrix A . The number of systems of distinct representatives (SDR's) of this collection is $\text{perm}(A)$.

Notes

- [1] Dexter Kozen. *The Design and Analysis of Algorithms*. (http://books.google.com/books?id=L_AMnf9UF9QC&pg=PA141&dq=%E2%80%9Cpermanent+of+a+matrix%E2%80%9D+valiant&as_brr=3&ei=h8BKScClJYOUNtTP6LEO#PPA142,M1) Springer-Verlag, New York, 1991.
ISBN 978-0-387-97687-7; pp. 141–142
- [2] ...
- [3] Brualdi (2006) p.487
- [4] Fulkerson Prize (<http://www.mathopt.org/?nav=fulkerson>), Mathematical Optimization Society, retrieved 2012-08-19.
- [5] p. 99 (<http://books.google.com/books?id=515ps2JkyT0C&pg=PA108&dq=permanent+ryser&lr=#PPA99,M1>)
- [6] In particular, and do this.

References

- Brualdi, Richard A. (2006). *Combinatorial matrix classes*. Encyclopedia of Mathematics and Its Applications **108**. Cambridge: Cambridge University Press. ISBN 0-521-86565-4. Zbl 1106.05001 (<http://www.zentralblatt-math.org/zmath/en/search/?format=complete&q=an:1106.05001>).
- Minc, Henryk (1978). *Permanents*. Encyclopedia of Mathematics and its Applications **6**. With a foreword by Marvin Marcus. Reading, MA: Addison-Wesley. ISSN 0953-4806 (<http://www.worldcat.org/issn/0953-4806>). OCLC 3980645 (<http://www.worldcat.org/oclc/3980645>). Zbl 0401.15005 (<http://www.zentralblatt-math.org/zmath/en/search/?format=complete&q=an:0401.15005>).
- Muir, Thomas; William H. Metzler. (1960) [1882]. *A Treatise on the Theory of Determinants*. New York: Dover. OCLC 535903 (<http://www.worldcat.org/oclc/535903>).
- Percus, J.K. (1971), *Combinatorial Methods*, Applied Mathematical Sciences #4, New York: Springer-Verlag, ISBN 0-387-90027-6
- Ryser, Herbert John (1963), *Combinatorial Mathematics*, The Carus Mathematical Monographs #14, The Mathematical Association of America
- van Lint, J.H.; Wilson, R.M. (2001), *A Course in Combinatorics*, Cambridge University Press, ISBN 0521422604

Further reading

- Hall, Jr., Marshall (1986), *Combinatorial Theory* (2nd ed.), New York: John Wiley & Sons, pp. 56–72, ISBN 0-471-09138-3 Contains a proof of the Van der Waerden conjecture.
- Marcus, M.; Minc, H. (1965), "Permanents", *The American Mathematical Monthly* **72**: 577–591, doi: [10.2307/2313846](https://doi.org/10.2307/2313846) (<http://dx.doi.org/10.2307/2313846>)

External links

- Permanent at PlanetMath (<http://planetmath.org/encyclopedia/Permanent.html>)
- Van der Waerden's permanent conjecture (<http://planetmath.org/?op=getobj&from=objects&id=6935>) at PlanetMath.org.

Computing the permanent

In mathematics, the **computation of the permanent of a matrix** is a problem that is known to be more difficult than the computation of the determinant of a matrix despite the apparent similarity of the definitions.

The permanent is defined similarly to the determinant, as a sum of products of sets of matrix entries that lie in distinct rows and columns. However, where the determinant weights each of these products with a ± 1 sign based on the parity of the set, the permanent weights them all with a $+1$ sign.

While the determinant can be computed in polynomial time by Gaussian elimination, the permanent cannot. In computational complexity theory, a theorem of Valiant states that computing permanents is $\#P$ -hard, and even $\#P$ -complete for matrices in which all entries are 0 or 1. Valiant (1979) This puts the computation of the permanent in a class of problems believed to be even more difficult to compute than NP. It is known that computing the permanent is impossible for logspace-uniform ACC^0 circuits. (Allender & Gore 1994)

The development of both exact and approximate algorithms for computing the permanent of a matrix is an active area of research.

Definition and naive algorithm

The permanent of an n -by- n matrix $A = (a_{i,j})$ is defined as

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^n a_{i,\sigma(i)}.$$

The sum here extends over all elements σ of the symmetric group S_n , i.e. over all permutations of the numbers 1, 2, ..., n . This formula differs from the corresponding formula for the determinant only in that, in the determinant, each product is multiplied by the sign of the permutation σ while in this formula each product is unsigned. The formula may be directly translated into an algorithm that naively expands the formula, summing over all permutations and within the sum multiplying out each matrix entry. This requires $n! n$ arithmetic operations.

Ryser formula

The best known^[1] general exact algorithm is due to H. J. Ryser (1963). Ryser's method is based on an inclusion–exclusion formula that can be given^[2] as follows: Let A_k be obtained from A by deleting k columns, let $P(A_k)$ be the product of the row-sums of A_k , and let Σ_k be the sum of the values of $P(A_k)$ over all possible A_k . Then

$$\text{perm}(A) = \sum_{k=0}^{n-1} (-1)^k \Sigma_k.$$

It may be rewritten in terms of the matrix entries as follows

$$\text{perm}(A) = (-1)^n \sum_{S \subseteq \{1, \dots, n\}} (-1)^{|S|} \prod_{i=1}^n \sum_{j \in S} a_{ij}.$$

Ryser's formula can be evaluated using $O(2^n n^2)$ arithmetic operations, or $O(2^n n)$ by processing the sets S in Gray code order.

Balasubramanian-Bax/Franklin-Glynn formula

Another formula that appears to be as fast as Ryser's (or perhaps even twice as fast) is to be found in the two Ph.D. theses; see (Balasubramanian 1980), (Bax 1998); also (Bax 1996). The methods to find the formula are quite different, being related to the combinatorics of the Muir algebra, and to finite difference theory respectively. Another way, connected with invariant theory is via the polarization identity for a symmetric tensor (Glynn 2010). The formula generalizes to infinitely many others, as found by all these authors, although it is not clear if they are any faster than the basic one. See (Glynn 2013).

The simplest known formula of this type (when the characteristic of the field is not two) is

$$\text{perm}(A) = \left[\sum_{\delta} \left(\prod_{k=1}^m \delta_k \right) \prod_{j=1}^m \sum_{i=1}^m \delta_i a_{ij} \right] / 2^{m-1},$$

where the outer sum is over all 2^{m-1} vectors $\delta = (\delta_1 = 1, \delta_2, \dots, \delta_m) \in \{\pm 1\}^m$.

Special cases

Planar and $K_{3,3}$ -free

The number of perfect matchings in a bipartite graph is counted by the permanent of the graph's biadjacency matrix, and the permanent of any 0-1 matrix can be interpreted in this way as the number of perfect matchings in a graph. For planar graphs (regardless of bipartiteness), the FKT algorithm computes the number of perfect matchings in polynomial time by changing the signs of a carefully chosen subset of the entries in the Tutte matrix of the graph, so that the Pfaffian of the resulting skew-symmetric matrix (the square root of its determinant) is the number of perfect matchings. This technique can be generalized to graphs that contain no subgraph homeomorphic to the complete bipartite graph $K_{3,3}$.

George Pólya had asked the question of when it is possible to change the signs of some of the entries of a 01 matrix A so that the determinant of the new matrix is the permanent of A . Not all 01 matrices are "convertible" in this manner; in fact it is known (Marcus & Minc (1961)) that there is no linear map T such that $\text{per } T(A) = \det A$ for all $n \times n$ matrices A . The characterization of "convertible" matrices was given by Little (1975) who showed that such matrices are precisely those that are the biadjacency matrix of bipartite graphs that have a Pfaffian orientation: an orientation of the edges such that for every even cycle C for which $G \setminus C$ has a perfect matching, there are an odd number of edges directed along C (and thus an odd number with the opposite orientation). It was also shown that these graphs are exactly those that do not contain a subgraph homeomorphic to $K_{3,3}$, as above.

Computation modulo a number

Modulo 2, the permanent is the same as the determinant, as $(-1) \equiv 1 \pmod{2}$. It can also be computed modulo 2^k in time $O(n^{4k-3})$ for $k \geq 2$. However, it is UP-hard to compute the permanent modulo any number that is not a power of 2. Valiant (1979)

There are various formulae given by Glynn (2010) for the computation modulo a prime p . Firstly there is one using symbolic calculations with partial derivatives.

Secondly for $p = 3$ there is the following formula (Grigoriy Kogan, 1996) using the determinants of the principal submatrices of the matrix:

$$\text{perm}(A) = (-1)^m \sum_{U \subseteq \{1, \dots, m\}} \det(A_U) \cdot \det(A_{\bar{U}}),$$

where A_U is the principal submatrix of A induced by the rows and columns of A indexed by U , and \bar{U} is the complement of U in $\{1, \dots, m\}$.

(The determinant of an empty submatrix is defined to be 1).

This formula implies the following identities over fields of Characteristic 3 (Grigoriy Kogan, 1996):

for any invertible A

$$\text{perm}(A^{-1}) \det(A)^2 = \text{perm}(A);$$

for any unitary U , i.e. a square matrix U such that $U^T U = I$,

$$\text{perm}(U)^2 = \det(U + V) \det(U)$$

where V is the matrix whose entries are the cubes of the corresponding entries of U .

Approximate computation

When the entries of A are nonnegative, the permanent can be computed approximately in probabilistic polynomial time, up to an error of εM , where M is the value of the permanent and $\varepsilon > 0$ is arbitrary. In other words, there exists a fully polynomial-time randomized approximation scheme (FPRAS) (Jerrum, Vigoda & Sinclair (2001)).

The most difficult step in the computation is the construction of an algorithm to sample almost uniformly from the set of all perfect matchings in a given bipartite graph: in other words, a fully polynomial almost uniform sampler (FPAUS). This can be done using a Markov chain Monte Carlo algorithm that uses a Metropolis rule to define and run a Markov chain whose distribution is close to uniform, and whose mixing time is polynomial.

It is possible to approximately count the number of perfect matchings in a graph via the self-reducibility of the permanent, by using the FPAUS in combination with a well-known reduction from sampling to counting due to Jerrum, Valiant & Vazirani (1986). Let $M(G)$ denote the number of perfect matchings in G . Roughly, for any particular edge e in G , by sampling many matchings in G and counting how many of them are matchings in $G \setminus e$, one can obtain an estimate of the ratio $\rho = \frac{M(G)}{M(G \setminus e)}$. The number $M(G)$ is then $\rho M(G \setminus e)$,

where $M(G \setminus e)$ can be approximated by applying the same method recursively.

Notes

[1] As of 2008, see

[2] p. 99 (<http://books.google.com/books?id=515ps2JkyT0C&pg=PA108&dq=permanent+ryser&lr=#PPA99,M1>)

References

- Allender, Eric; Gore, Vivec (1994), "A uniform circuit lower bound for the permanent", *SIAM Journal on Computing* **23** (5): 1026–1049, doi: 10.1137/s0097539792233907 (<http://dx.doi.org/10.1137/s0097539792233907>)
- Balasubramanian, K. (1980), *Combinatorics and Diagonals of Matrices*, Ph.D. Thesis, Department of Statistics, Loyola College, Madras, India **T073**, Indian Statistical Institute, Calcutta
- Bax, Eric (1998), *Finite-difference Algorithms for Counting Problems*, Ph.D. Dissertation **223**, California Institute of Technology
- Bax, Eric; Franklin, J. (1996), *A finite-difference sieve to compute the permanent*, CalTech-CS-TR-96-04, California Institute of Technology
- Glynn, David G. (2010), "The permanent of a square matrix", *European Journal of Combinatorics* **31** (7): 1887–1891, doi: 10.1016/j.ejc.2010.01.010 (<http://dx.doi.org/10.1016/j.ejc.2010.01.010>)
- Glynn, David G. (2013), "Permanent formulae from the Veronesean", *Designs Codes and Cryptography* **68** (1-3): 39–47, doi: 10.1007/s10623-012-9618-1 (<http://dx.doi.org/10.1007/s10623-012-9618-1>)
- Jerrum, M.; Sinclair, A.; Vigoda, E. (2001), "A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries", *Proc. 33rd Symposium on Theory of Computing*, pp. 712–721, doi: 10.1145/380752.380877 (<http://dx.doi.org/10.1145/380752.380877>), ECCC TR00-079 (<http://eccc.uni-trier.de/report/2000/079/>)

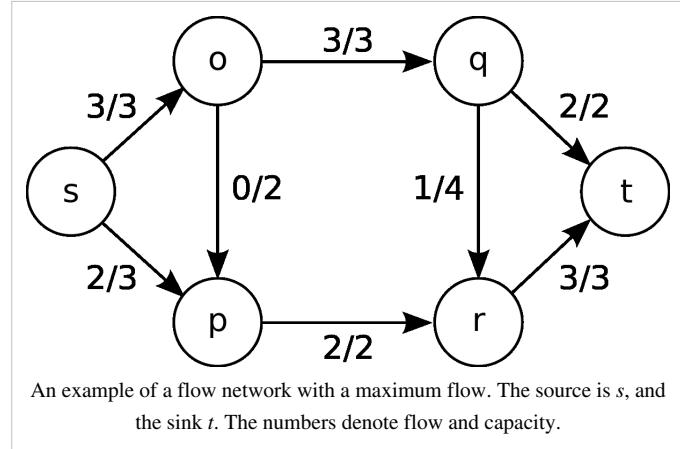
- Mark Jerrum; Leslie Valiant; Vijay Vazirani (1986), "Random generation of combinatorial structures from a uniform distribution", *Theoretical Computer Science* **43**: 169–188, doi: 10.1016/0304-3975(86)90174-X ([http://dx.doi.org/10.1016/0304-3975\(86\)90174-X](http://dx.doi.org/10.1016/0304-3975(86)90174-X))
- Kogan, Grigoriy (1996), "Computing permanents over fields of characteristic 3: where and why it becomes difficult", *37th Annual Symposium on Foundations of Computer Science (FOCS '96)*
- van Lint, Jacobus Hendricus; Wilson, Richard Michale (2001), *A Course in Combinatorics*, ISBN 0-521-00601-5
- Little, C. H. C. (1974), "An extension of Kasteleyn's method of enumerating the 1-factors of planar graphs", in Holton, D., *Proc. 2nd Australian Conf. Combinatorial Mathematics*, Lecture Notes in Mathematics **403**, Springer-Verlag, pp. 63–72
- Little, C. H. C. (1975), "A characterization of convertible (0, 1)-matrices" (<http://www.sciencedirect.com/science/article/B6WHT-4D7K7HW-H5/2/caa9448ac7c4e895fd7845515c7a68d1>), *Journal of Combinatorial Theory, Series B* **18** (3): 187–208, doi: 10.1016/0095-8956(75)90048-9 ([http://dx.doi.org/10.1016/0095-8956\(75\)90048-9](http://dx.doi.org/10.1016/0095-8956(75)90048-9))
- Marcus, M.; Minc, H. (1961), "On the relation between the determinant and the permanent", *Illinois Journal of Mathematics* **5**: 376–381
- Pólya, G. (1913), "Aufgabe 424", *Arch. Math. Phys.* **20** (3): 27
- Reich, Simeon (1971), "Another solution of an old problem of pólya", *American Mathematical Monthly* **78** (6): 649–650, doi: 10.2307/2316574 (<http://dx.doi.org/10.2307/2316574>), JSTOR 2316574 (<http://www.jstor.org/stable/2316574>)
- Rempała, Grzegorz A.; Wesolowski, Jacek (2008), *Symmetric Functionals on Random Matrices and Random Matchings Problems*, p. 4, ISBN 0-387-75145-9
- Ryser, Herbert John (1963), *Combinatorial Mathematics*, The Carus mathematical monographs, The Mathematical Association of America
- Vazirani, Vijay V. (1988), "NC algorithms for computing the number of perfect matchings in $K_{3,3}$ -free graphs and related problems", *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT '88)*, Lecture Notes in Computer Science **318**, Springer-Verlag, pp. 233–242, doi: 10.1007/3-540-19487-8_27 (http://dx.doi.org/10.1007/3-540-19487-8_27)
- Valiant, Leslie G. (1979), "The Complexity of Computing the Permanent", *Theoretical Computer Science* (Elsevier) **8** (2): 189–201, doi: 10.1016/0304-3975(79)90044-6 ([http://dx.doi.org/10.1016/0304-3975\(79\)90044-6](http://dx.doi.org/10.1016/0304-3975(79)90044-6))
- "Permanent", *CRC Concise Encyclopedia of Mathematics*, Chapman & Hall/CRC, 2002

Network flow

Maximum flow problem

In optimization theory, **maximum flow problems** involve finding a feasible flow through a single-source, single-sink flow network that is maximum.

The maximum flow problem can be seen as a special case of more complex network flow problems, such as the circulation problem. The maximum value of an s - t flow (i.e., flow from source s to sink t) is equal to the minimum capacity of an s - t cut (i.e., cut severing s from t) in the network, as stated in the max-flow min-cut theorem.



History

The maximum flow problem was first formulated in 1954 by T. E. Harris and F. S. Ross as a simplified model of Soviet railway traffic flow. In 1955, Lester R. Ford, Jr. and Delbert R. Fulkerson created the first known algorithm, the Ford–Fulkerson algorithm.^[1]

Over the years, various improved solutions to the maximum flow problem were discovered, notably the shortest augmenting path algorithm of Edmonds and Karp and independently Dinitz; the blocking flow algorithm of Dinitz; the push-relabel algorithm of Goldberg and Tarjan; and the binary blocking flow algorithm of Goldberg and Rao. The electrical flow algorithm of Christiano, Kelner, Madry, and Spielman finds an approximately optimal maximum flow but only works in undirected graphs.

Definition

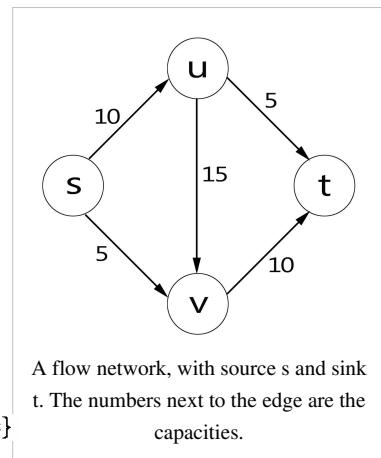
Let $N=(V,E)$ be a network with $s,t \in V$ being the source and the sink of N respectively.

The **capacity** of an edge is a mapping $c:E \rightarrow \mathbb{R}^+$, denoted by c_{uv} or $c(u,v)$. It represents the maximum amount of flow that can pass through an edge.

A **flow** is a mapping $f:E \rightarrow \mathbb{R}^+$, denoted by f_{uv} or $f(u,v)$, subject to the following two constraints:

1. $f_{uv} \leq c_{uv}$, for each $(u,v) \in E$ (capacity constraint: the flow of an edge cannot exceed its capacity)
2. $\sum_{u:(u,v) \in E} f_{uv} = \sum_{u:(v,u) \in E} f_{vu}$, for each $v \in V \setminus \{s,t\}$

(conservation of flows: the sum of the flows entering a node must equal the sum of the flows exiting a node, except for the source and the sink nodes)



The **value of flow** is defined by $|f| = \sum_{v:(s,v) \in E} f_{sv}$, where s is the source of N . It represents the amount of flow passing from the source to the sink.

The **maximum flow problem** is to maximize $|f|$, that is, to route as much flow as possible from s to t .

Solutions

We can define the **Residual Graph**, which provides a systematic way to search for forward-backward operations in order to find the maximum flow.

Given a flow network G , and a flow f on G , we define the residual graph G_f of G with respect to f as follows.

1. The node set of G_f is the same as that of G .
2. Each edge $e=(u,v)$ of G_f is with a capacity of $c_e - f(e)$.
3. Each edge $e'=(v,u)$ of G_f is with a capacity of $f(e)$.

The following table lists algorithms for solving the maximum flow problem.

Method	Complexity	Description
Linear programming		Constraints given by the definition of a legal flow. See the linear program here.
Ford–Fulkerson algorithm	$O(E \max\{f\})$	As long as there is an open path through the residual graph, send the minimum of the residual capacities on the path. The algorithm works only if all weights are integers. Otherwise it is possible that the Ford–Fulkerson algorithm will not converge to the maximum value.
Edmonds–Karp algorithm	$O(VE^2)$	A specialization of Ford–Fulkerson, finding augmenting paths with breadth-first search.
Dinitz blocking flow algorithm	$O(V^2E)$	In each phase the algorithm builds a layered graph with breadth-first search on the residual graph. The maximum flow in a layered graph can be calculated in $O(VE)$ time, and the maximum number of the phases is $n-1$. In networks with unit capacities, Dinic's algorithm terminates in $O(E\sqrt{V})$ time.
General push-relabel maximum flow algorithm	$O(V^2E)$	The push relabel algorithm maintains a preflow, i.e. a flow function with the possibility of excess in the vertices. The algorithm runs while there is a vertex with positive excess, i.e. an active vertex in the graph. The push operation increases the flow on a residual edge, and a height function on the vertices controls which residual edges can be pushed. The height function is changed with a relabel operation. The proper definitions of these operations guarantee that the resulting flow function is a maximum flow.
Push-relabel algorithm with FIFO vertex selection rule	$O(V^3)$	Push-relabel algorithm variant which always selects the most recently active vertex, and performs push operations until the excess is positive or there are admissible residual edges from this vertex.
Dinitz blocking flow algorithm with dynamic trees	$O(VE \log(V))$	The dynamic trees data structure speeds up the maximum flow computation in the layered graph to $O(E \log(V))$.
Push-relabel algorithm with dynamic trees	$O(VE \log(V^2/E))$	The algorithm builds limited size trees on the residual graph regarding to height function. These trees provide multilevel push operations.
Binary blocking flow algorithm	$O(E \min(V^{2/3}, \sqrt{E}) \log(V^2/E) \log U)$	The value U corresponds to the maximum capacity of the network.
MPM (Malhotra, Pramodh-Kumar and Maheshwari) algorithm	$O(V^3)$	Refer to the Original Paper [2].
Jim Orlin's + KRT (King, Rao, Tarjan)'s algorithm	$O(VE)$	Orlin's algorithm [3] solves max-flow in $O(VE)$ time for $m \leq O(n^{(16/15)-\epsilon})$ while KRT solves it in $O(VE)$ for $m > n^{1+\epsilon}$

For a more extensive list, see

Integral flow theorem

The integral flow theorem states that

If each edge in a flow network has integral capacity, then there exists an integral maximal flow.

Application

Multi-source multi-sink maximum flow problem

Given a network $N = (V, E)$ with a set of sources $S = \{s_1, \dots, s_n\}$ and a set of sinks $T = \{t_1, \dots, t_m\}$ instead of only one source and one sink, we are to find the maximum flow across N . We can transform the multi-source multi-sink problem into a maximum flow problem by adding a *consolidated source* connecting to each vertex in S and a *consolidated sink* connected by each vertex in T (also known as *supersource* and *supersink*) with infinite capacity on each edge (See Fig. 4.1.1.).

Minimum path cover in directed acyclic graph

Given a directed acyclic graph $G = (V, E)$, we are to find the minimum number of paths to cover each vertex in V . We can construct a bipartite graph $G' = (V_{out} \cup V_{in}, E')$ from G , where

1. $V_{out} = \{v \in V : v \text{ has positive out-degree}\}$.
2. $V_{in} = \{v \in V : v \text{ has positive in-degree}\}$.
3. $E' = \{(u, v) \in (V_{out} \cup V_{in})^2 : (u, v) \in E\}$.

Then it can be shown, via König's theorem, that G' has a matching of size m if and only if there exists $n-m$ paths that cover each vertex in G , where n is the number of vertices in G . Therefore, the problem can be solved by finding the maximum cardinality matching in G' instead.

Maximum cardinality bipartite matching

Given a bipartite graph $G = (X \cup Y, E)$, we are to find a maximum cardinality matching in G , that is a matching that contains the largest possible number of edges. This problem can be transformed into a maximum flow problem by constructing a network $N = (X \cup Y \cup \{s, t\}, E')$, where

1. E' contains the edges in G directed from X to Y .
2. $(s, x) \in E'$ for each $x \in X$ and $(y, t) \in E'$ for each $y \in Y$.
3. $c(e) = 1$ for each $e \in E'$ (See Fig. 4.3.1).

Then the value of the maximum flow in N is equal to the size of the maximum matching in G .

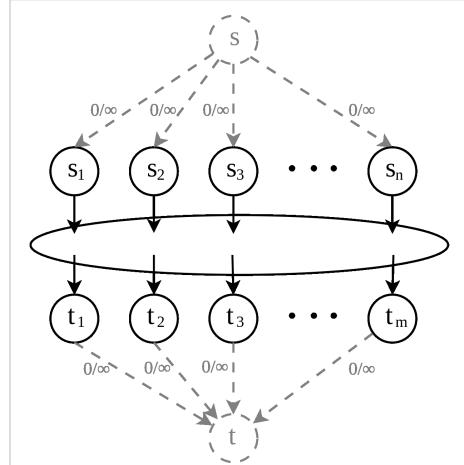


Fig. 4.1.1. Transformation of a multi-source multi-sink maximum flow problem into a single-source single-sink maximum flow problem

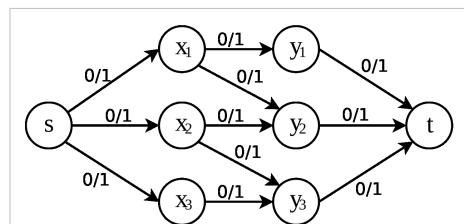


Fig. 4.3.1. Transformation of a maximum bipartite matching problem into a maximum flow problem

Maximum flow problem with vertex capacities

Given a network $N = (V, E)$, in which there is capacity at each node in addition to edge capacity, that is, a mapping $c : V \mapsto \mathbb{R}^+$, denoted by $c(v)$, such that the flow f has to satisfy not only the capacity constraint and the conservation of flows, but also the vertex capacity constraint

$$\sum_{i \in V} f_{iv} \leq c(v) \quad \forall v \in V \setminus \{s, t\}.$$

In other words, the amount of flow passing through a vertex cannot exceed its capacity. To find the maximum flow across N , we can transform the problem into the maximum flow problem in the original sense by expanding N . First, each $v \in V$ is replaced by v_{in} and v_{out} , where v_{in} is connected by edges going into v and v_{out} is connected to edges coming out from v , then assign capacity $c(v)$ to the edge connecting v_{in} and v_{out} (see Fig. 4.4.1, but note that it has incorrectly swapped v_{in} and v_{out}). In this expanded network, the vertex capacity constraint is removed and therefore the problem can be treated as the original maximum flow problem.

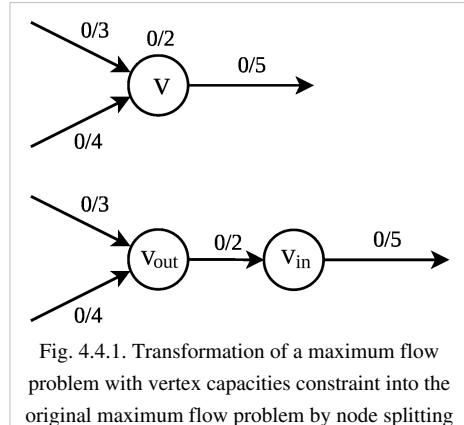


Fig. 4.4.1. Transformation of a maximum flow problem with vertex capacities constraint into the original maximum flow problem by node splitting

Maximum independent path

Given a directed graph $G = (V, E)$ and two vertices s and t , we are to find the maximum number of independent paths from s to t . Two paths are said to be independent if they do not have a vertex in common apart from s and t . We can construct a network $N = (V, E)$ from G with vertex capacities, where

1. s and t are the source and the sink of N respectively.
2. $c(v) = 1$ for each $v \in V$.
3. $c(e) = 1$ for each $e \in E$.

Then the value of the maximum flow is equal to the maximum number of independent paths from s to t .

Maximum edge-disjoint path

Given a directed graph $G = (V, E)$ and two vertices s and t , we are to find the maximum number of edge-disjoint paths from s to t . This problem can be transformed to a maximum flow problem by constructing a network $N = (V, E)$ from G with s and t being the source and the sink of N respectively and assign each edge with unit capacity.

Maximum Vertex Disjoint Path

Given a directed graph $G = (V, E)$ and two vertices s and t , we have to find the maximum number of vertex-disjoint paths from s to t . This problem can be transformed to a maximum flow problem by constructing a network $N = (V, E)$ as follows

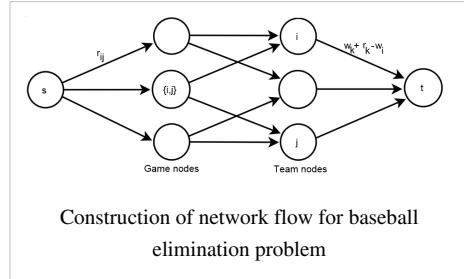
1. Split each node v in the graph into two nodes: v_{in} and v_{out} .
2. For each node v , add an edge of capacity one from v_{in} to v_{out} .
3. Replace each other edge (u, v) in the graph with an edge from u_{out} to v_{in} of capacity 1.
4. Add in a new dedicated destination node t .
5. For each of the target nodes v , add an edge from v_{in} to t with capacity 1.
6. Find a max-flow from s_{out} to t . The value of the flow is the number of vertex-disjoint paths

Real world applications

Baseball Elimination

In the Baseball Elimination Problem there are n teams competing in a league. At a specific stage of the league season, w_i is the number of wins and r_i is the number of games left to play for team i and r_{ij} is the number of games left against team j . A team is eliminated if it has no chance to finish the season in the first place. The task of Baseball Elimination Problem is to determine which teams are eliminated at each point during the season. Schwartz proposed a method which reduces this problem to maximum network flow. In this method a network is created to determine whether team k is eliminated.

Let $G = (V, E)$ be a network with $s, t \in V$ being the source and the sink respectively. We add a game node $\{i,j\}$ with $i < j$ to V and connect each of them to s by an edge and set its capacity to r_{ij} , which represents the number of plays between these two teams. We also add a node for each team i and connect each node $\{i,j\}$ with team nodes i and j to ensure one of them wins. We do not need to restrict the flow value on these edges. Finally, we make edges from team node i to the sink node t and set the capacity of $w_k + r_k - w_i$ to prevent team i from winning more than $w_k + r_k$. Let S be the set of all team participating in the league and let $r(S - \{k\}) = \sum_{i,j \in \{S - \{k\}\}, i < j} r_{ij}$. In this method it is claimed team k is not eliminated if and only if a flow value of size $r(S - \{k\})$ exists in network G . In the mentioned article it is proved that this flow value is the maximum flow value from s to t .



Airline scheduling

In the airline industry a major problem is the scheduling of the flight crews. Airline scheduling problem could be considered as an application of extended maximum network flow. The input of this problem is a set of flights F which contains the information about where and when each flight departs and arrives. In one version of Airline Scheduling the goal is to produce a feasible schedules with at most k crews.

In order to solve this problem we use a variation of circulation problem called bounded circulation which is the generalization of network flow problems, with the added constraint of a lower bound on edge flows.

Let $G = (V, E)$ be a network with $s, t \in V$ as the source and the sink nodes. For the source and destination of every flight i we add two nodes to V , node s_i as the source and node d_i as the destination node of flight i . We also add the following edges to E :

1. An edge with capacity $[0, 1]$ between s and each s_i .
2. An edge with capacity $[0, 1]$ between each d_i and t .
3. An edge with capacity $[1, 1]$ between each pair of s_i and d_i .
4. An edge with capacity $[0, 1]$ between each d_i and s_j , if source s_j is reachable with a reasonable amount of time and cost from the destination of flight i .
5. An edge with capacity $[0, \infty]$ between s and t .

In the mentioned method, it is claimed and proved that finding a flow value of k in G between s and t is equal to finding a feasible schedule for flight set F with at most k crews.

Another version of Airline Scheduling is finding the minimum needed crews to perform all the flights. In order to find an answer to this problem we create a bipartite graph $G' = (A \cup B, E)$ where each flight has a copy in set A and set B . If the same plane can perform flight j after flight i , connect $i \in A$ to $j \in B$. A matching in G' induces a schedule for F and obviously maximum bipartite matching in this graph produces the an airline schedule with minimum number of crews. As it is mentioned in the Application part of this article, the maximum cardinality bipartite matching is an application of maximum flow problem.

Circulation-demand problem

There are some factories that produce goods and some villages where the goods have to be delivered. They are connected by a networks of roads with each road having a capacity c for maximum goods that can flow through it. The problem is to find if there is a circulation that satisfies the demand. This problem can be transformed into a max-flow problem.

1. Add a source node s and add edges from it to every factory node f_i with capacity p_i where p_i is the production rate of factory f_i .
2. Add a sink node t and add edges from all villages v_i to t with capacity d_i where d_i is the demand rate of village v_i .

Let $G = (V, E)$ be this new network. There exists a circulation that satisfies the demand if and only if :

$$\text{Maximumflowvalue}(G) = \sum_{i \in v} d_i$$

If there exists a circulation, looking at the max-flow solution would give us the answer as to how much goods have to be send on a particular road for satisfying the demands.

Fairness in car sharing (carpool)

The problem exactly is that N people are pooling a car for k days. Each participant can choose if he participates on each day. We aim to fairly decide who will be driving on a given day.

The solution is the following:

We can decide this on the basis of the number of people using the car i.e.; if m people use the car on a day, each person has a responsibility of $1/m$. Thus, for every person i , his driving obligation D_i as shown. Then person i is required to drive only $[D_i]$ times in k days.

Our aim is to find if such a setting is possible. For this we try to make the problem as a network, as we can see in the figure.

Now, it can be proved that if a flow k exists then such a fair setting exists and such a flow of value k always exists.

References

- [1] Ford, L.R., Jr.; Fulkerson, D.R., *Flows in Networks*, Princeton University Press (1962).
- [2] [http://dx.doi.org/10.1016/0020-0190\(78\)90016-9](http://dx.doi.org/10.1016/0020-0190(78)90016-9)
- [3] [http://jorlin.scripts.mit.edu/Max_flows_in_O\(nm\)_time.html](http://jorlin.scripts.mit.edu/Max_flows_in_O(nm)_time.html)

Further reading

- Joseph Cheriyan and Kurt Mehlhorn (1999). "An analysis of the highest-level selection rule in the preflow-push max-flow algorithm". *Information Processing Letters* **69** (5): 239–242. doi: 10.1016/S0020-0190(99)00019-8 ([http://dx.doi.org/10.1016/S0020-0190\(99\)00019-8](http://dx.doi.org/10.1016/S0020-0190(99)00019-8)).
- Daniel D. Sleator and Robert E. Tarjan (1983). "A data structure for dynamic trees" (<http://www.cs.cmu.edu/~sleator/papers/dynamic-trees.pdf>). *Journal of Computer and System Sciences* **26** (3): 362–391. doi: 10.1016/0022-0000(83)90006-5 ([http://dx.doi.org/10.1016/0022-0000\(83\)90006-5](http://dx.doi.org/10.1016/0022-0000(83)90006-5)). ISSN 0022-0000 (<http://www.worldcat.org/issn/0022-0000>).
- Eugene Lawler (2001). "4. Network Flows". *Combinatorial Optimization: Networks and Matroids*. Dover. pp. 109–177. ISBN 0-486-41453-1.

Max-flow min-cut theorem

In optimization theory, the **max-flow min-cut theorem** states that in a flow network, the maximum amount of flow passing from the *source* to the *sink* is equal to the minimum capacity that, when removed in a specific way from the network, causes the situation that no flow can pass from the source to the sink.

The **max-flow min-cut theorem** is a special case of the duality theorem for linear programs and can be used to derive Menger's theorem and the König-Egerváry Theorem.

Definition

Let $N = (V, E)$ be a network (directed graph) with s and t being the source and the sink of N respectively.

The **capacity** of an edge is a mapping $c: E \rightarrow \mathbb{R}^+$, denoted by c_{uv} or $c(u,v)$. It represents the maximum amount of flow that can pass through an edge.

A **flow** is a mapping $f: E \rightarrow \mathbb{R}^+$, denoted by f_{uv} or $f(u,v)$, subject to the following two constraints:

1. $f_{uv} \leq c_{uv}$ for each $(u, v) \in E$ (capacity constraint)
2. $\sum_{u: (u,v) \in E} f_{uv} = \sum_{u: (v,u) \in E} f_{vu}$ for each $v \in V \setminus \{s, t\}$ (conservation of flows).

The **value of flow** is defined by $|f| = \sum_{v \in V} f_{sv}$, where s is the source of N . It represents the amount of flow passing from the source to the sink.

The *maximum flow problem* is to maximize $|f|$, that is, to route as much flow as possible from s to t .

An **s-t cut** $C = (S, T)$ is a partition of V such that $s \in S$ and $t \in T$. The **cut-set** of C is the set $\{(u,v) \in E \mid u \in S, v \in T\}$.

Note that if the edges in the cut-set of C are removed, $|f| = 0$.

The **capacity** of an *s-t cut* is defined by $c(S, T) = \sum_{(u,v) \in S \times T} c_{uv}$.

The *minimum s-t cut problem* is minimizing $c(S, T)$, that is, to determine S and T such that the capacity of the *S-T cut* is minimal.

Statement

The max-flow min-cut theorem states

The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts.

Linear program formulation

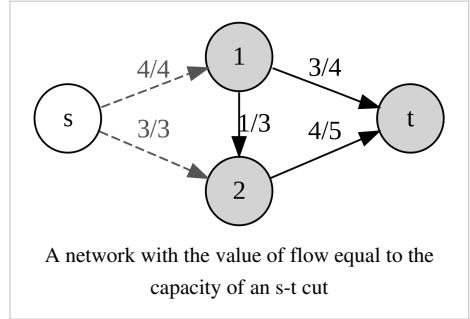
The max-flow problem and min-cut problem can be formulated as two primal-dual linear programs.

Max-flow (Primal)	Min-cut (Dual)
maximize $ f = \nabla_s$	minimize $\sum_{(i,j) \in E} c_{ij} d_{ij}$
subject to	subject to
$\begin{array}{lcl} f_{ij} & \leq & c_{ij} & (i, j) \in E \\ \sum_{j:(j,i) \in E} f_{ji} - \sum_{j:(i,j) \in E} f_{ij} & \leq & 0 & i \in V, i \neq s, t \\ \nabla_s + \sum_{j:(j,s) \in E} f_{js} - \sum_{j:(s,j) \in E} f_{sj} & \leq & 0 \\ -\nabla_s + \sum_{j:(j,t) \in E} f_{jt} - \sum_{j:(t,j) \in E} f_{tj} & \leq & 0 \\ f_{ij} & \geq & 0 & (i, j) \in E \end{array}$	$\begin{array}{lcl} d_{ij} - p_i + p_j & \geq & 0 & (i, j) \in E \\ p_s - p_t & \geq & 1 \\ p_i & \geq & 0 & i \in V \\ d_{ij} & \geq & 0 & (i, j) \in E \end{array}$

The equality in the **max-flow min-cut theorem** follows from the strong duality theorem in linear programming, which states that if the primal program has an optimal solution, x^* , then the dual program also has an optimal solution, y^* , such that the optimal values formed by the two solutions are equal.

Example

The figure on the right is a network having a value of flow of 7. The vertex in white and the vertices in grey form the subsets S and T of an s - t cut, whose cut-set contains the dashed edges. Since the capacity of the s - t cut is 7, which equals to the value of flow, the max-flow min-cut theorem tells us that the value of flow and the capacity of the s - t cut are both optimal in this network.



Application

Generalized max-flow min-cut theorem

In addition to edge capacity, consider there is capacity at each vertex, that is, a mapping $c: V \rightarrow R^+$, denoted by $c(v)$, such that the flow f has to satisfy not only the capacity constraint and the conservation of flows, but also the vertex capacity constraint

$$\sum_{i \in V} f_{iv} \leq c(v) \text{ for each } v \in V \setminus \{s, t\}.$$

In other words, the amount of flow passing through a vertex cannot exceed its capacity. Define an s - t cut to be the set of vertices and edges such that for any path from s to t , the path contains a member of the cut. In this case, the *capacity of the cut* is the sum the capacity of each edge and vertex in it.

In this new definition, the **generalized max-flow min-cut theorem** states that the maximum value of an s - t flow is equal to the minimum capacity of an s - t cut in the new sense.

Menger's theorem

See also: Menger's Theorem

In the undirected edge-disjoint paths problem, we are given an undirected graph $G = (V, E)$ and two vertices s and t , and we have to find the maximum number of edge-disjoint s - t paths in G .

The **Menger's theorem** states that the maximum number of edge-disjoint s - t paths in an undirected graph is equal to the minimum number of edges in an s - t cut-set.

Project selection problem

See also: Maximum flow problem

In the project selection problem, there are n projects and m equipments. Each project p_i yields revenue $r(p_i)$ and each equipment q_j costs $c(q_j)$ to purchase. Each project requires a number of equipments and each equipment can be shared by several projects. The problem is to determine which projects and equipments should be selected and purchased respectively, so that the profit is maximized.

Let P be the set of projects *not* selected and Q be the set of equipments purchased, then the problem can be formulated as,

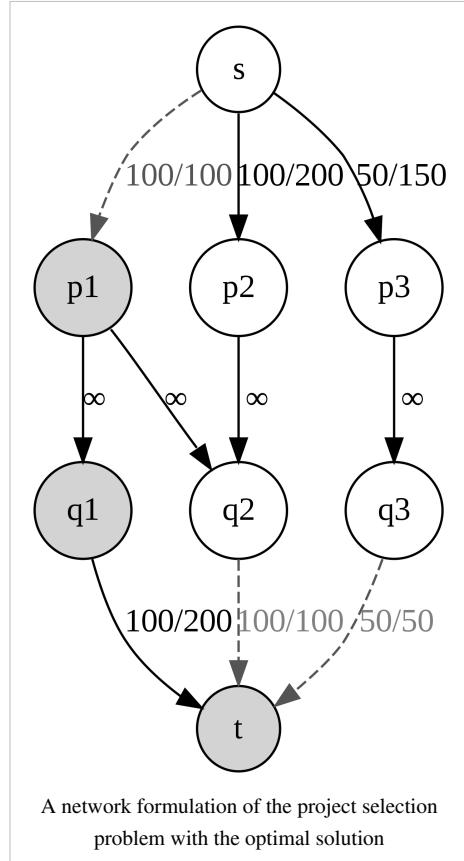
$$\max\{g\} = \sum_i r(p_i) - \sum_{p_i \in P} r(p_i) - \sum_{q_j \in Q} c(q_j).$$

Since the first term does not depend on the choice of P and Q , this maximization problem can be formulated as a minimization problem instead, that is,

$$\min\{g'\} = \sum_{p_i \in P} r(p_i) + \sum_{q_j \in Q} c(q_j).$$

The above minimization problem can then be formulated as a minimum-cut problem by constructing a network, where the source is connected to the projects with capacity $r(p_i)$, and the sink is connected by the equipments with capacity $c(q_j)$. An edge (p_i, q_j) with *infinite* capacity is added if project p_i requires equipment q_j . The s - t cut-set represents the projects and equipments in P and Q respectively. By the max-flow min-cut theorem, one can solve the problem as a maximum flow problem.

The figure on the right gives a network formulation of the following project selection problem:



	Project $r(p_i)$	Equipment $c(q_j)$	
1	100	200	Project 1 requires equipments 1 and 2.
2	200	100	Project 2 requires equipment 2.
3	150	50	Project 3 requires equipment 3.

The minimum capacity of a s-t cut is 250 and the sum of the revenue of each project is 450; therefore the maximum profit g is $450 - 250 = 200$, by selecting projects p_2 and p_3 .

The idea here is to 'flow' the project profits through the 'pipes' of the equipment. If we cannot fill the pipe, the equipment's return is less than its cost, and the min cut algorithm will find it cheaper to cut the project's profit edge instead of the equipment's cost edge.

Image Segmentation problem

See also: Maximum flow problem

In the image segmentation problem, there are n pixels. Each pixel i can be assigned a foreground value f_i or a background value b_i . There is a penalty of p_{ij} if pixels i, j are adjacent and have different assignments. The problem is to assign pixels to foreground or background such that the sum of their values minus the penalties is maximum.

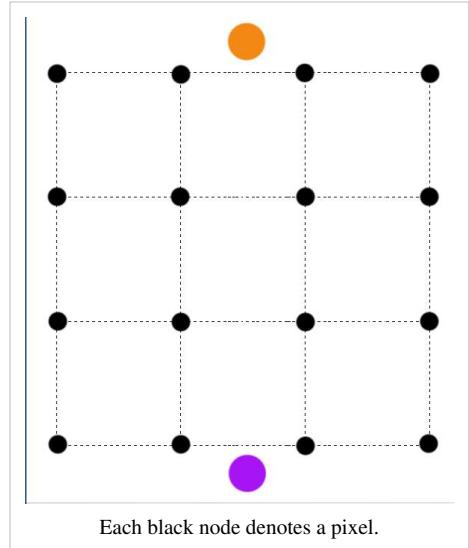
Let P be the set of pixels assigned to foreground and Q be the set of points assigned to background, then the problem can be formulated as,

$$\max\{g\} = \sum_{i \in P} f_i + \sum_{i \in Q} b_i - \sum_{i \in P, j \in Q \vee j \in P, i \in Q} p_{ij}.$$

This maximization problem can be formulated as a minimization problem instead, that is,

$$\min\{g'\} = \sum_{i \in Q} f_i + \sum_{i \in P} b_i + \sum_{i \in P, j \in Q \vee j \in P, i \in Q} p_{ij}.$$

The above minimization problem can be formulated as a minimum-cut problem by constructing a network where the source (orange node) is connected to all the pixels with capacity f_i , and the sink (purple node) is connected by all the pixels with capacity b_i . Two edges (i, j) and (j, i) with p_{ij} capacity are added between two adjacent pixels. The s-t cut-set then represents the pixels assigned to the foreground in P and pixels assigned to background in Q .



History

The **max-flow min-cut theorem** was proven by P. Elias, A. Feinstein, and C.E. Shannon in 1956[1], and independently also by L.R. Ford, Jr. and D.R. Fulkerson in the same year[1].

Proof

Let $N = (V, E)$ be a network (directed graph) with s and t being the source and the sink of N respectively.

Consider the flow f computed for G by Ford-Fulkerson algorithm. In the residual graph (G_f) obtained for G (after the final flow assignment by Ford-Fulkerson algorithm), define two subsets of vertices as follows:

1. A : the set of vertices reachable from s in G_f
2. A^c : the set of remaining vertices i.e. $V - A$

Claim: value (f) = $c(A, A^c)$

where the **capacity** of an $s-t$ cut is defined by $c(S, T) = \sum_{(u,v) \in S \times T} c_{uv}$.

Now, we know, $\text{value}(f) = f_{out}(A) - f_{in}(A^c)$ for any subset of vertices, A .

For **value** (f) to be equal to $c(A, A^c)$,

- All *outgoing edges* from the cut must be **fully saturated**.
- All *incoming edges* to the cut must have **zero flow**.

To prove the above claim, let us assume that in G , there exists an *outgoing edge* (x, y) , $x \in A$, $y \in A^c$ such that it is not saturated.

i.e. $f(x, y) < c_{xy}$

This implies, that in G_f , there exists a **forward edge** from x to y , which further implies that there exists a path from s to t in G_f , which is a **contradiction**. Hence, any outgoing edge (x, y) is fully saturated.

Now, let us assume that in G , there exists an *incoming edge* (y, x) , $x \in A$, $y \in A^c$ such that it carries some non-zero flow.

i.e. $f(x, y) > 0$

This implies, that in G_f , there exists a **backward edge** from x to y , which further implies that there exists a path from s to t in G_f , which is again a **contradiction**. Hence, any incoming edge (x, y) must have zero flow.

Both of the above statements prove that the capacity of cut obtained in the above described manner is equal to the flow obtained in the network. Also, the flow was obtained by Ford-Fulkerson algorithm, so it is the max-flow of the network as well.

Also, since *any flow in the network is always less than or equal to capacity of every cut possible in a network*, hence, the above described cut is also the min-cut which obtains the max-flow.

References

- [1] http://en.wikipedia.org/wiki/Max-flow_min-cut_theorem#endnote_P._Elias.2C_A._Feinstein.2C_and_C._E._Shannon.2C_A_note_on_the_maximum_flow_through_a_network.2C_IRE._Transactions_on_Information_Theory.2C_2.2C_4_.281956.29.2C_117.E2.80.93119
1. Eugene Lawler (2001). "4.5. Combinatorial Implications of Max-Flow Min-Cut Theorem, 4.6. Linear Programming Interpretation of Max-Flow Min-Cut Theorem". *Combinatorial Optimization: Networks and Matroids*. Dover. pp. 117–120. ISBN 0-486-41453-1.
2. Christos H. Papadimitriou, Kenneth Steiglitz (1998). "6.1 The Max-Flow, Min-Cut Theorem". *Combinatorial Optimization: Algorithms and Complexity*. Dover. pp. 120–128. ISBN 0-486-40258-4.
3. Vijay V. Vazirani (2004). "12. Introduction to LP-Duality". *Approximation Algorithms*. Springer. pp. 93–100. ISBN 3-540-65367-8.

Ford–Fulkerson algorithm for maximum flows

The **Ford–Fulkerson method** is an algorithm which computes the maximum flow in a flow network. It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson.

The idea behind the algorithm is as follows: As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

Algorithm

Let $G(V, E)$ be a graph, and for each edge from u to v , let $c(u, v)$ be the capacity and $f(u, v)$ be the flow.

We want to find the maximum flow from the source s to the sink t . After every step in the algorithm the following is maintained:

Capacity constraints: $\forall(u, v) \in E f(u, v) \leq c(u, v)$ The flow along an edge can not exceed its capacity.

Skew symmetry: $\forall(u, v) \in E f(u, v) = -f(v, u)$ The net flow from u to v must be the opposite of the net flow from v to u (see example).

Flow conservation: $\forall u \in V : u \neq s \text{ and } u \neq t \Rightarrow \sum_{w \in V} f(u, w) = 0$ That is, unless u is s or t . The net flow to a node is zero, except for the source, which "produces" flow, and the sink, which "consumes" flow.

Value(f): $\sum_{(s,u) \in E} f(s, u) = \sum_{(v,t) \in E} f(v, t)$ That is, the flow leaving from s or arriving at t .

This means that the flow through the network is a *legal flow* after each round in the algorithm. We define the **residual network** $G_f(V, E_f)$ to be the network with capacity $c_f(u, v) = c(u, v) - f(u, v)$ and no flow.

Notice that it can happen that a flow from v to u is allowed in the residual network, though disallowed in the original network: if $f(u, v) > 0$ and $c(v, u) = 0$ then $c_f(v, u) = c(v, u) - f(v, u) = f(u, v) > 0$.

Algorithm Ford–Fulkerson

Inputs Given a Network $G = (V, E)$ with flow capacity c , a source node s , and a sink node t

Output Compute a flow f from s to t of maximum value

1. $f(u, v) \leftarrow 0$ for all edges (u, v)
2. While there is a path p from s to t in G_f , such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$:
 1. Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
 2. For each edge $(u, v) \in p$
 1. $f(u, v) \leftarrow f(u, v) + c_f(p)$ (*Send flow along the path*)
 2. $f(v, u) \leftarrow f(v, u) - c_f(p)$ (*The flow might be "returned" later*)

The path in step 2 can be found with for example a breadth-first search or a depth-first search in $G_f(V, E_f)$. If you use the former, the algorithm is called Edmonds–Karp.

When no more paths in step 2 can be found, s will not be able to reach t in the residual network. If S is the set of nodes reachable by s in the residual network, then the total capacity in the original network of edges from S to the remainder of V is on the one hand equal to the total flow we found from s to t , and on the other hand serves as an upper bound for all such flows. This proves that the flow we found is maximal. See also Max-flow Min-cut theorem.

If the graph $G(V, E)$ has multi Sources and Sinks, we act as follows. Suppose that $T = \{t | t \text{ is a sink}\}$ and $S = \{s | s \text{ is a source}\}$. Add a new source s^* with an edge (s^*, s) from s^* to every node $s \in S$, with

capacity $f(s^*, s) = d_s$ ($d_s = \sum_{(s,u) \in E} f(s, u)$). And add a new sink t^* with an edge (t^*, t) from t^* to every node $t \in T$, with algorithm.

Also if every nodes u has constraint d_u , we replace this node with two nodes u_{in}, u_{out} , and an edge (u_{in}, u_{out}) , with capacity $f(u_{in}, u_{out}) = d_u$. and then applying the Ford–Fulkerson algorithm.

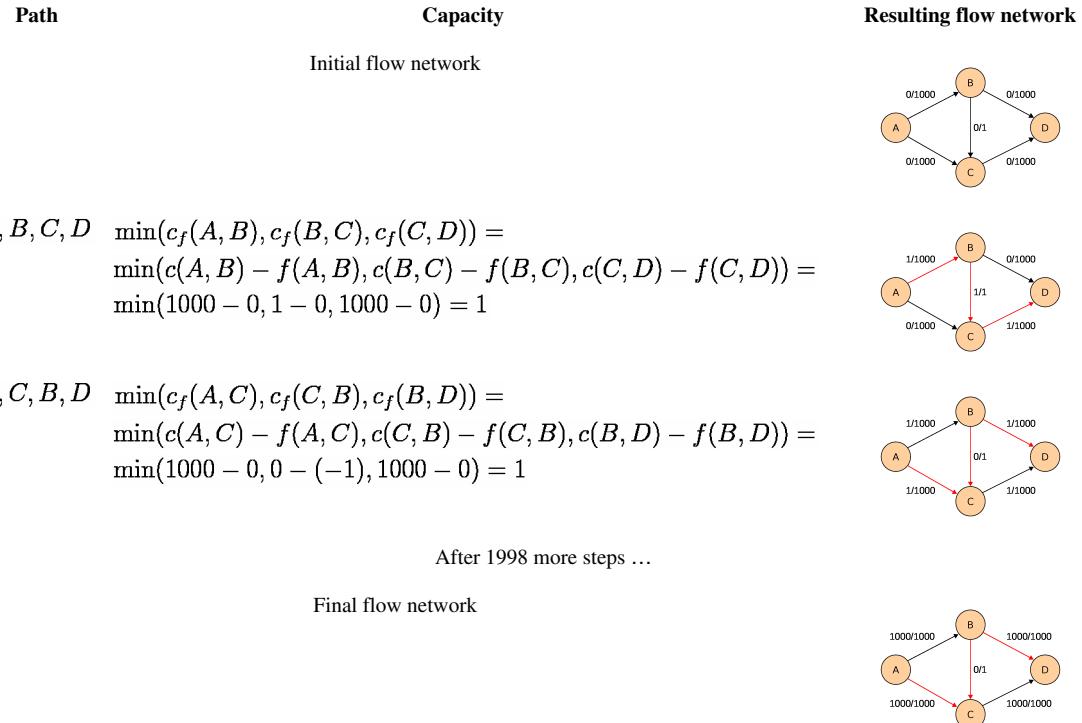
Complexity

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values. When the capacities are integers, the runtime of Ford–Fulkerson is bounded by $O(Ef)$ (see big O notation), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in $O(E)$ time and increases the flow by an integer amount which is at least 1.

A variation of the Ford–Fulkerson algorithm with guaranteed termination and a runtime independent of the maximum flow value is the Edmonds–Karp algorithm, which runs in $O(VE^2)$ time.

Integral example

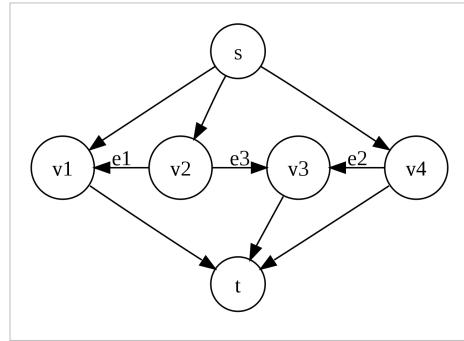
The following example shows the first steps of Ford–Fulkerson in a flow network with 4 nodes, source A and sink D . This example shows the worst-case behaviour of the algorithm. In each step, only a flow of 1 is sent across the network. If breadth-first-search were used instead, only two steps would be needed.



Notice how flow is "pushed back" from C to B when finding the path A, C, B, D .

Non-terminating example

Consider the flow network shown on the right, with source s , sink t , capacities of edges e_1 , e_2 and e_3 respectively 1, $r = (\sqrt{5} - 1)/2$ and 1 and the capacity of all other edges some integer $M \geq 2$. The constant r was chosen so, that $r^2 = 1 - r$. We use augmenting paths according to the following table, where $p_1 = \{s, v_4, v_3, v_2, v_1, t\}$, $p_2 = \{s, v_2, v_3, v_4, t\}$ and $p_3 = \{s, v_1, v_2, v_3, t\}$.



Step	Augmenting path	Sent flow	Residual capacities		
			e_1	e_2	e_3
0			$r^0 = 1$	r	1
1	$\{s, v_2, v_3, t\}$	1	r^0	r^1	0
2	p_1	r^1	r^2	0	r^1
3	p_2	r^1	r^2	r^1	0
4	p_1	r^2	0	r^3	r^2
5	p_3	r^2	r^2	r^3	0

Note that after step 1 as well as after step 5, the residual capacities of edges e_1 , e_2 and e_3 are in the form r^n , r^{n+1} and 0, respectively, for some $n \in \mathbb{N}$. This means that we can use augmenting paths p_1 , p_2 , p_1 and p_3 infinitely many times and residual capacities of these edges will always be in the same form. Total flow in the network after step 5 is $1 + 2(r^1 + r^2)$. If we continue to use augmenting paths as above, the total flow converges to $1 + 2\sum_{i=1}^{\infty} r^i = 3 + 2r$, while the maximum flow is $2M + 1$. In this case, the algorithm never terminates and the flow doesn't even converge to the maximum flow.

Python implementation

```

class Edge(object):
    def __init__(self, u, v, w):
        self.source = u
        self.sink = v
        self.capacity = w
    def __repr__(self):
        return "%s->%s:%s" % (self.source, self.sink, self.capacity)

class FlowNetwork(object):
    def __init__(self):
        self.adj = {}
        self.flow = {}

    def add_vertex(self, vertex):
        self.adj[vertex] = []

    def get_edges(self, v):
        adj = self.adj[v]
        edges = []
        for w in adj:
            edges.append((v, w, self.flow[v][w], self.adj[v][w].capacity))
        return edges

```

```

    return self.adj[v]

def add_edge(self, u, v, w=0):
    if u == v:
        raise ValueError("u == v")
    edge = Edge(u,v,w)
    redge = Edge(v,u,0)
    edge.redge = redge
    redge.redge = edge
    self.adj[u].append(edge)
    self.adj[v].append(redge)
    self.flow[edge] = 0
    self.flow[redge] = 0

def find_path(self, source, sink, path):
    if source == sink:
        return path
    for edge in self.get_edges(source):
        residual = edge.capacity - self.flow[edge]
        if residual > 0 and edge not in path:
            result = self.find_path( edge.sink, sink, path +
[edge])
            if result != None:
                return result

def max_flow(self, source, sink):
    path = self.find_path(source, sink, [])
    while path != None:
        residuals = [edge.capacity - self.flow[edge] for edge in
path]
        flow = min(residuals)
        for edge in path:
            self.flow[edge] += flow
            self.flow[edge.redge] -= flow
        path = self.find_path(source, sink, [])
    return sum(self.flow[edge] for edge in self.get_edges(source))

```

Usage example

For the example flow network in maximum flow problem we do the following:

```

>>> g = FlowNetwork()
>>> for v in "sopqrt":
>>>     g.add_vertex(v)
>>> g.add_edge('s','o',3)
>>> g.add_edge('s','p',3)
>>> g.add_edge('o','p',2)
>>> g.add_edge('o','q',3)
>>> g.add_edge('p','r',2)

```

```
>>> g.add_edge('r', 't', 3)
>>> g.add_edge('q', 'r', 4)
>>> g.add_edge('q', 't', 2)
>>> print g.max_flow('s', 't')
5
```

Notes

References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 26.2: The Ford–Fulkerson method". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw–Hill. pp. 651–664. ISBN 0-262-03293-7.
- George T. Heineman, Gary Pollice, and Stanley Selkow (2008). "Chapter 8:Network Flow Algorithms". *Algorithms in a Nutshell*. O'reilly Media. pp. 226–250. ISBN 978-0-596-51624-6.
- Jon Kleinberg and Éva Tardos (2006). "Chapter 7:Extensions to the Maximum-Flow Problem". *Algorithm Design*. Pearson Education. pp. 378–384. ISBN 0-321-29535-8.

External links

- Another Java animation (<http://www.cs.pitt.edu/~kirk/cs1501/animations/Network.html>)
- Java Web Start application (<http://rrusin.blogspot.com/2011/03/implementing-graph-editor-in-javafx.html>)

Media related to Ford–Fulkerson algorithm at Wikimedia Commons

Edmonds–Karp algorithm for maximum flows

In computer science, the **Edmonds–Karp algorithm** is an implementation of the Ford–Fulkerson method for computing the maximum flow in a flow network in $O(V E^2)$ time. The algorithm was first published by Yefim (Chaim) Dinic in 1970 and independently published by Jack Edmonds and Richard Karp in 1972. Dinic's algorithm includes additional techniques that reduce the running time to $O(V^2E)$.

Algorithm

The algorithm is identical to the Ford–Fulkerson algorithm, except that the search order when finding the augmenting path is defined. The path found must be a shortest path that has available capacity. This can be found by a breadth-first search, as we let edges have unit length. The running time of $O(V E^2)$ is found by showing that each augmenting path can be found in $O(E)$ time, that every time at least one of the E edges becomes saturated (an edge which has the maximum possible flow), that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the length is at most V . Another property of this algorithm is that the length of the shortest augmenting path increases monotonically. There is an accessible proof in *Introduction to Algorithms*.

Pseudocode



The Wikibook *Algorithm implementation* has a page on the topic of: [Edmonds-Karp](#)

For a more high level description, see [Ford–Fulkerson algorithm](#).

```
algorithm EdmondsKarp
    input:
        C[1..n, 1..n] (Capacity matrix)
        E[1..n, 1..?] (Neighbour lists)
        s                (Source)
        t                (Sink)
    output:
        f                (Value of maximum flow)
        F                (A matrix giving a legal flow with the maximum value)
    f := 0 (Initial flow is zero)
    F := array(1..n, 1..n) (Residual capacity from u to v is C[u,v] - F[u,v])
    forever
        m, P := BreadthFirstSearch(C, E, s, t, F)
        if m = 0
            break
        f := f + m
        (Backtrack search, and write flow)
        v := t
        while v ≠ s
            u := P[v]
            F[u,v] := F[u,v] + m
            F[v,u] := F[v,u] - m
            v := u
    return (f, F)
```

```
algorithm BreadthFirstSearch
    input:
        C, E, s, t, F
    output:
        M[t]           (Capacity of path found)
        P              (Parent table)
    P := array(1..n)
    for u in 1..n
        P[u] := -1
    P[s] := -2 (make sure source is not rediscovered)
    M := array(1..n) (Capacity of found path to node)
    M[s] := ∞
    Q := queue()
    Q.push(s)
    while Q.size() > 0
        u := Q.pop()
        for v in E[u]
```

```

(If there is available capacity, and v is not seen before in search)
if C[u,v] - F[u,v] > 0 and P[v] = -1
    P[v] := u
    M[v] := min(M[u], C[u,v] - F[u,v])
    if v ≠ t
        Q.push(v)
    else
        return M[t], P
return 0, P

```

EdmondsKarp pseudo code using Adjacency nodes.

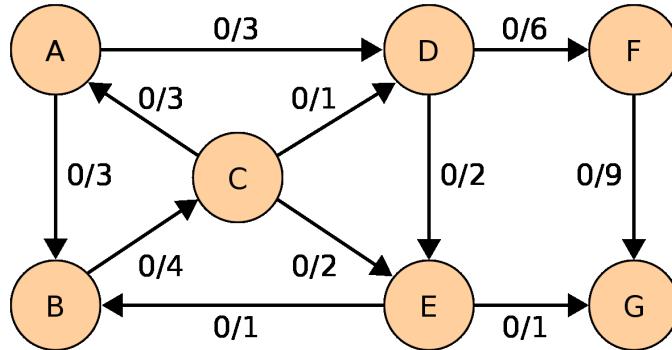
```

algorithm EdmondsKarp
    input:
        graph (Graph with list of Adjacency nodes with capacities, flow, reverse and destinations)
        s           (Source)
        t           (Sink)
    output:
        flow          (Value of maximum flow)
        flow := 0 (Initial flow to zero)
        q := array(1..n) (Initialize q to graph length)
        while true
            qt := 0           (Variable to iterate over all the corresponding edges for a source)
            q[qt+1] := s     (initialize source array)
            pred := array(q.length) (Initialize predecessor List with the graph length)
            for qh=0; qh < qt && pred[t] := null
                cur := q[qh]
                for (graph[cur]) (Iterate over list of Edges)
                    Edge[] e := graph[cur] (Each edge should be associated with Capacity)
                    if pred[e.t] == null && e.cap > e.f
                        pred[e.t] := e
                        q[qt++] := e.t
                if pred[t] = null
                    break
                int df := MAX VALUE (Initialize to max integer value)
                for u = t; u != s; u = pred[u].s
                    df := min(df, pred[u].cap - pred[u].f)
                for u = t; u != s; u = pred[u].s
                    pred[u].f := pred[u].f + df
                    pEdge := array(PredEdge)
                    pEdge := graph[pred[u].t]
                    pEdge[pred[u].rev].f := pEdge[pred[u].rev].f - df;
                flow := flow + df
    return flow

```

Example

Given a network of seven nodes, source A, sink G, and capacities as shown below:



In the pairs f/c written on the edges, f is the current flow, and c is the capacity. The residual capacity from u to v is $c_f(u, v) = c(u, v) - f(u, v)$, the total capacity, minus the flow that is already used. If the net flow from u to v is negative, it *contributes* to the residual capacity.

Capacity	Path
Resulting network	
$\min(c_f(A, D), c_f(D, E), c_f(E, G)) =$ $\min(3 - 0, 2 - 0, 1 - 0) =$ $\min(3, 2, 1) = 1$	A, D, E, G
$\min(c_f(A, D), c_f(D, F), c_f(F, G)) =$ $\min(3 - 1, 6 - 0, 9 - 0) =$ $\min(2, 6, 9) = 2$	A, D, F, G
$\min(c_f(A, B), c_f(B, C), c_f(C, D), c_f(D, F), c_f(F, G)) =$ $\min(3 - 0, 4 - 0, 1 - 0, 6 - 2, 9 - 2) =$ $\min(3, 4, 1, 4, 7) = 1$	A, B, C, D, F, G
$\min(c_f(A, B), c_f(B, C), c_f(C, E), c_f(E, D), c_f(D, F), c_f(F, G)) =$ $\min(3 - 1, 4 - 1, 2 - 0, 0 - (-1), 6 - 3, 9 - 3) =$ $\min(2, 3, 2, 1, 3, 6) = 1$	A, B, C, E, D, F, G

Notice how the length of the augmenting path found by the algorithm (in red) never decreases. The paths found are the shortest possible. The flow found is equal to the capacity across the minimum cut in the graph separating the source and the sink. There is only one minimal cut in this graph, partitioning the nodes into the sets $\{A, B, C, E\}$ and $\{D, F, G\}$, with the capacity

$$c(A, D) + c(C, D) + c(E, G) = 3 + 1 + 1 = 5.$$

Notes

References

- Algorithms and Complexity (see pages 63–69). <http://www.cis.upenn.edu/~wilf/AlgComp3.html>

Dinic's algorithm for maximum flows

Dinitz's algorithm is a strongly polynomial algorithm for computing the maximum flow in a flow network, conceived in 1970 by Israeli (formerly Soviet) computer scientist Yefim Dinitz. The algorithm runs in $O(V^2E)$ time and is similar to the Edmonds–Karp algorithm, which runs in $O(VE^2)$ time, in that it uses shortest augmenting paths. The introduction of the concepts of the *level graph* and *blocking flow* enable Dinic's algorithm to achieve its performance.

Definition

Let $G = ((V, E), c, s, t)$ be a network with $c(u, v)$ and $f(u, v)$ the capacity and the flow of the edge (u, v) respectively.

The **residual capacity** is a mapping $c_f: V \times V \rightarrow R^+$ defined as,

- if $(u, v) \in E$,

$$c_f(u, v) = c(u, v) - f(u, v)$$

$$c_f(v, u) = f(u, v)$$

- $c_f(u, v) = 0$ otherwise.

The **residual graph** is the graph $G_f = ((V, E_f), c_f|_{E_f}, s, t)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

An **augmenting path** is an $s - t$ path in the residual graph G_f .

Define $\text{dist}(v)$ to be the length of the shortest path from s to v in G_f . Then the **level graph** of G_f is the graph $G_L = (V, E_L, c_f|_{E_L}, s, t)$, where

$$E_L = \{(u, v) \in E_f : \text{dist}(v) = \text{dist}(u) + 1\}.$$

A **blocking flow** is an $s - t$ flow f such that the graph $G' = (V, E'_L, s, t)$ with $E'_L = \{(u, v) : f(u, v) < c_f|_{E_L}(u, v)\}$ contains no $s - t$ path.

Algorithm

Dinic's Algorithm

Input: A network $G = ((V, E), c, s, t)$.

Output: An $s - t$ flow f of maximum value.

- Set $f(e) = 0$ for each $e \in E$.
- Construct G_L from G_f of G . If $\text{dist}(t) = \infty$, stop and output f .
- Find a blocking flow f' in G_L .
- Augment flow f by f' and go back to step 2.

Analysis

It can be shown that the number of edges in each blocking flow increases by at least 1 each time and thus there are at most $n - 1$ blocking flows in the algorithm, where n is the number of vertices in the network. The level graph G_L can be constructed by Breadth-first search in $O(E)$ time and a blocking flow in each level graph can be found in $O(VE)$ time. Hence, the running time of Dinic's algorithm is $O(V^2E)$.

Using a data structure called dynamic trees, the running time of finding a blocking flow in each phase can be reduced to $O(E \log V)$ and therefore the running time of Dinic's algorithm can be improved to $O(VE \log V)$.

Special cases

In networks with unit capacities, a much stronger time bound holds. Each blocking flow can be found in $O(E)$ time, and it can be shown that the number of phases does not exceed $O(\sqrt{E})$ and $O(V^{2/3})$. Thus the algorithm runs in $O(\min\{V^{2/3}, E^{1/2}\}E)$ time.

In networks arising during the solution of bipartite matching problem, the number of phases is bounded by $O(\sqrt{V})$, therefore leading to the $O(\sqrt{V}E)$ time bound. The resulting algorithm is also known as Hopcroft–Karp algorithm. More generally, this bound holds for any *unit network* — a network in which each vertex, except for source and sink, either has a single entering edge of capacity one, or a single outgoing edge of capacity one, and all other capacities are arbitrary integers.^[1]

Example

The following is a simulation of the Dinic's algorithm. In the level graph G_L , the vertices with labels in red are the values $\text{dist}(v)$. The paths in blue form a blocking flow.

	G	G_f	G_L
1.			
	<p>The blocking flow consists of</p> <ol style="list-style-type: none"> $\{s, 1, 3, t\}$ with 4 units of flow, $\{s, 1, 4, t\}$ with 6 units of flow, and $\{s, 2, 4, t\}$ with 4 units of flow. <p>Therefore the blocking flow is of 14 units and the value of flow f is 14. Note that each augmenting path in the blocking flow has 3 edges.</p>		
2.			
	<p>The blocking flow consists of</p> <ol style="list-style-type: none"> $\{s, 2, 4, 3, t\}$ with 5 units of flow. <p>Therefore the blocking flow is of 5 units and the value of flow f is $14 + 5 = 19$. Note that each augmenting path has 4 edges.</p>		
3.			
	<p>Since t cannot be reached in G_f. The algorithm terminates and returns a flow with maximum value of 19. Note that in each blocking flow, the number of edges in the augmenting path increases by at least 1.</p>		

History

Dinic's algorithm was published in 1970 by former Russian Computer Scientist Yefim (Chaim) A. Dinitz, who is today a member of the Computer Science department at Ben-Gurion University of the Negev (Israel), earlier than the Edmonds–Karp algorithm, which was published in 1972 but was discovered earlier. They independently showed that in the Ford–Fulkerson algorithm, if each augmenting path is the shortest one, the length of the augmenting paths is non-decreasing.

Notes

[1] Tarjan 1983, p. 102.

References

- Yefim Dinitz (2006). "Dinitz' Algorithm: The Original Version and Even's Version" (http://www.cs.bgu.ac.il/~dinitz/Papers/Dinitz_alg.pdf). In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman. *Theoretical Computer Science: Essays in Memory of Shimon Even*. Springer. pp. 218–240. ISBN 978-3-540-32880-3.
- Tarjan, R. E. (1983). *Data structures and network algorithms*.
- B. H. Korte, Jens Vygen (2008). "8.4 Blocking Flows and Fujishige's Algorithm". *Combinatorial Optimization: Theory and Algorithms (Algorithms and Combinatorics, 21)*. Springer Berlin Heidelberg. pp. 174–176. ISBN 978-3-540-71844-4.

Push–relabel maximum flow algorithm

In mathematical optimization, the **push–relabel algorithm** (alternatively, **preflow–push algorithm**) is an algorithm for computing maximum flows. The name "push–relabel" comes from the two basic operations used in the algorithm. Throughout its execution, the algorithm maintains a "preflow" and gradually converts it into a maximum flow by moving flow locally between neighboring vertices using *push* operations under the guidance of an admissible network maintained by *relabel* operations. In comparison, the Ford–Fulkerson algorithm performs global augmentations that send flow following paths from the source all the way to the sink.

The push–relabel algorithm is considered one of the most efficient maximum flow algorithms. The generic algorithm has a strongly polynomial $O(V^2E)$ time complexity, which is asymptotically more efficient than the $O(VE^2)$ Edmonds–Karp algorithm. Specific variants of the algorithms achieve even lower time complexities. The variant based on the highest label vertex selection rule has $O(V^2\sqrt{E})$ time complexity and is generally regarded as the benchmark for maximum flow algorithms. Subcubic $O(VE \log (V^2/E))$ time complexity can be achieved using dynamic trees, although in practice it is less efficient.

The push–relabel algorithm has been extended to compute minimum cost flows. The idea of distance labels has led to a more efficient augmenting path algorithm, which in turn can be incorporated back into the push–relabel algorithm to create a variant with even higher empirical performance.

Concepts

Definitions and notations

Main article: Flow network

Consider a flow network $G(V, E)$ with a pair of distinct vertices s and t designated as the source and the sink, respectively. For each edge $(u, v) \in E$, $c(u, v) \geq 0$ denotes its capacity; if $(u, v) \notin E$, we assume that $c(u, v) = 0$. A flow on G is a function $f: V \times V \rightarrow \mathbf{R}$ satisfying the following conditions:

Capacity constraints

$$f(u, v) \leq c(u, v) \quad \forall u, v \in V$$

Skew symmetry

$$f(u, v) = -f(v, u) \quad \forall u, v \in V$$

Flow conservation

$$\sum_{v \in V} f(v, u) = 0 \quad \forall u \in V \setminus \{s, t\}$$

The push–relabel algorithm introduces the concept of *preflows*. A preflow is a function with a definition almost identical to that of a flow except that it relaxes the flow conservation condition. Instead of requiring strict flow balance at vertices other than s and t , it allows them to carry positive excesses. Put symbolically:

Nonnegative excesses

$$e(u) = \sum_{v \in V} f(v, u) \geq 0 \quad \forall u \in V \setminus \{s\}$$

$e(s)$ is assumed to be infinite. A vertex u is called *active* if $e(u) > 0$.

For each $(u, v) \in V \times V$, denote its *residual capacity* by $c_f(u, v) = c(u, v) - f(u, v)$. The residual network of G with respect to a preflow f is defined as $G_f(V, E_f)$ where $E_f = \{(u, v) \mid u, v \in V \wedge c_f(u, v) > 0\}$. If there is no path from any active vertex to t in G_f , the preflow is called *maximum*. In a maximum preflow, $e(t)$ is equal to the value of a maximum flow; if T is the set of vertices from which t is reachable in G_f and $S = V \setminus T$, then (S, T) is a minimum s - t cut.

The push–relabel algorithm makes use of distance labels, or *heights*, of the vertices denoted by $h(u)$. For each vertex $u \in V \setminus \{s, t\}$, $h(u)$ is a nonnegative integer satisfying

Valid labeling

$$h(u) \leq h(v) + 1 \quad \forall (u, v) \in E_f$$

The heights of s and t are fixed at $|V|$ and 0, respectively. $h(u)$ is a lower bound of the unweighted distance from u to t in G_f if t is reachable from u . If u has been disconnected from t , then $(h(u) - |V|)$ is a lower bound of the unweighted distance from u to s . As a result, if a valid height function exists, there are no s - t paths in G_f because no such paths can be longer than $(|V| - 1)$.

An edge $(u, v) \in E_f$ is called *admissible* if $h(u) = h(v) + 1$. The network $G_f(V, \tilde{E}_f)$ where $\tilde{E}_f = \{(u, v) \mid (u, v) \in E_f \wedge h(u) = h(v) + 1\}$ is called the *admissible network*. The admissible network is acyclic.

Operations

Push

The push operation applies on an admissible out-edge (u, v) of an active vertex u in G_f . It moves $\min\{e(u), c_f(u, v)\}$ units of flow from u to v .

```
push(u, v) :
    assert e[u] > 0 and h[u] == h[v] + 1
    Δ = min(e[u], c[u][v] - f[u][v])
    f[u][v] += Δ
    f[v][u] -= Δ
    e[u] -= Δ
    e[v] += Δ
```

A push operation that causes $f(u, v)$ to reach $c(u, v)$ is called a *saturating* push; otherwise, it is called an *unsaturating* push. After an unsaturating push, $e(u) = 0$.

Relabel

The relabel operation applies on an active vertex u without any admissible out-edges in G_f . It modifies $h(u)$ to the minimum value such that an admissible out-edge is created. Note that this always increases $h(u)$ and never creates a steep edge (an edge (u, v) such that $c_f(u, v) > 0$, and $h(u) > h(v) + 1$).

```
relabel(u) :
    assert e[u] > 0 and h[u] <= h[v] ∀v such that f[u][v] < c[u][v]
    h[u] = min(h[v] ∀v such that f[u][v] < c[u][v]) + 1
```

Effects of push and relabel

After a push or relabel operation, h remains a valid height function with respect to f .

For a push operation on an admissible edge (u, v) , it may add an edge (v, u) to E_f , where $h(v) = h(u) - 1 \leq h(u) + 1$; it may also remove the edge (u, v) from E_f , where it effectively removes the constraint $h(u) \leq h(v) + 1$.

To see that a relabel operation on vertex u preserves the validity of $h(u)$, notice that this is trivially guaranteed by definition for the out-edges of u in G_f . For the in-edges of u in the G_f , the increased $h(u)$ can only satisfy the constraints less tightly, not violate them.

The generic push–relabel algorithm

Description

Since $h(s) = |V|$, $h(t) = 0$, and there are no paths longer than $(|V| - 1)$ in G_f in order for $h(s)$ to satisfy the valid labeling condition, s must be disconnected from t . At initialization, the algorithm fulfills this requirement by creating a preflow f that saturates all out-edges of s , after which $h(u) = 0$ is trivially valid for all $v \in V \setminus \{s, t\}$.

After initialization, the algorithm repeatedly executes an applicable push or relabel operation until no such operations apply, at which point the preflow has been converted into a maximum flow.

```
generic-push-relabel(G(V, E), s, t) :
    create a preflow f that saturates all out-edges of s
    let h[u] = 0 ∀v ∈ V
    while there is an applicable push or relabel operation
        execute the operation
```

Correctness

Because push and relabel operations preserve the validity of preflow f and height function h , when the algorithm terminates, there are no active vertices other than s and t , and thus the preflow becomes a valid flow. Since it is also guaranteed throughout that there are no s - t paths in residual network G_f , the algorithm terminates with a maximum flow.

Time complexity

We bound the numbers of relabels, saturating pushes and nonsaturating pushes separately. Because $0 \leq h(u) \leq 2n - 1$ for every vertex u , and each relabel increases $h(u)$ by at least one, the total number of relabels on all vertices is $O(V^2)$. Each saturating push on an admissible edge (u, v) removes the edge from G_f . For the edge to be reinserted into G_f for another saturating push, v must be first relabeled, followed by a push on edge (v, u) , then u must be relabeled. In the process, $h(u)$ increases by at least two. Therefore, there are $O(V)$ saturating pushes on (u, v) , and the total number of saturating pushes on all edges is $O(VE)$.

Bounding the number of nonsaturating pushes can be achieved via a potential argument. We use the potential function $\Phi = \sum_{[u \in V \wedge e(u) > 0]} h(u)$, i.e., Φ is the sum of the heights of all active vertices. It is obvious that Φ is $|V|$ initially and stays nonnegative throughout the execution of the algorithm. Both relabels and saturating pushes can increase Φ . Relabels increase $h(u)$ by $O(1)$ for every vertex u and thus contribute $O(V^2)$ increase to Φ . A saturating push on (u, v) activates v if it was inactive before the push, increasing Φ by $O(1)$. Hence, the total contribution of all saturating pushes is $O(V^2E)$. An unsaturating push on (u, v) always deactivates u , but it can also activate v as in a saturating push. As a result, it decreases Φ by at least $h(u) - h(v) = 1$. Since relabels and saturating pushes increase Φ by $O(V^2 + V^2E) = O(V^2E)$, the total number of unsaturating pushes is $O(V^2E)$.

In sum, the algorithm executes $O(V^2)$ relabels, $O(VE)$ saturating pushes and $O(V^2E)$ nonsaturating pushes. Data structures can be designed to pick and execute an applicable operation in $O(1)$ time. Therefore, the time complexity of the algorithm is $O(V^2E)$.

Practical implementations

While the generic push–relabel algorithm has $O(V^2E)$ time complexity, efficient implementations achieve $O(V^3)$ or lower time complexity by enforcing appropriate rules in selecting applicable push and relabel operations. The empirical performance can be further improved by heuristics.

"Current-edge" data structure and discharge operation

The "current-edge" data structure is a mechanism for visiting the in- and out-neighbors of a vertex in the flow network in a static circular order. If a singly linked list of neighbors is created for a vertex, the data structure can be as simple as a pointer into the list that steps through the list and rewinds to the head when it runs off the end.

Based on the "current-edge" data structure, the discharge operation can be defined. A discharge operation applies on an active node and repeatedly pushes flow from the node until it becomes inactive, relabeling it as necessary to create admissible edges in the process.

```
discharge(u) :
    while e[u] > 0
        if current-edge[u] has run off the end of neighbors[u]
            relabel(u)
            rewind current-edge[u]
        else
            let (u, v) = current-edge[u]
            if (u, v) is admissible
```

```

    push(u, v)
else
    let current-edge[u] point to the next neighbor of u

```

Active vertex selection rules

Definition of the discharge operation reduces the push–relabel algorithm to repeatedly selecting an active node to discharge. Depending on the selection rule, the algorithm exhibits different time complexities. For the sake of brevity, we ignore s and t when referring to the vertices in the following discussion.

FIFO selection rule

The FIFO push–relabel algorithm organizes the active vertices into a queue. The initial active nodes can be inserted in arbitrary order. The algorithm always removes the vertex at the front of the queue for discharging. Whenever an inactive vertex becomes active, it is appended to the back of the queue.

The algorithm has $O(V^3)$ time complexity.

Relabel-to-front selection rule

The relabel-to-front push–relabel algorithm organizes all vertices into a linked list and maintains the invariant that the list is topologically sorted with respect to the admissible network. The algorithm scans the list from front to back and performs a discharge operation on the current vertex if it is active. If the node is relabeled, it is moved to the front of the list, and the scan is restarted from the front.

The algorithm also has $O(V^3)$ time complexity.

Highest label selection rule

The highest-label push–relabel algorithm organizes all vertices into buckets indexed by their heights. The algorithm always selects an active vertex with the largest height to discharge.

The algorithm has $O(V^2\sqrt{E})$ time complexity. If the lowest-label selection rule is used instead, the time complexity becomes $O(V^2E)$.

Implementation techniques

Although in the description of the generic push–relabel algorithm above, $h(u)$ is set to zero for each vertex u other than s and t at the beginning, it is preferable to perform a backward breadth-first search from t to compute the exact heights.

The algorithm is typically separated into two phases. Phase one computes a maximum preflow by discharging only active vertices whose heights are below n . Phase two converts the maximum preflow into a maximum flow by returning excess flow that cannot reach t to s . It can be shown that phase two has $O(VE)$ time complexity regardless of the order of push and relabel operations and is therefore dominated by phase one. Alternatively, it can be implemented using flow decomposition.

Heuristics are crucial to improving the empirical performance of the algorithm. Two commonly used heuristics are the gap heuristic and the global relabeling heuristic. The gap heuristic detects gaps in the height function. If there is a height $0 < \tilde{h} < |V|$ for which there is no vertex u such that $h(u) = \tilde{h}$, then any vertex u with $\tilde{h} < h(u) < |V|$ has been disconnected from t and can be relabeled to $(|V| + 1)$ immediately. The global relabeling heuristic periodically performs backward breadth-first search from t in G_f to compute the exact heights of the vertices. Both heuristics skip unhelpful relabel operations, which are a bottleneck of the algorithm and contribute to the ineffectiveness of dynamic trees.

References

Closure problem

In graph theory and combinatorial optimization, a **closure** of a directed graph is a set of vertices with no outgoing edges. The **closure problem** is the task of finding the maximum-weight or minimum-weight closure in a vertex-weighted directed graph. It may be solved in polynomial time using a reduction to the maximum flow problem. It may be used to model various application problems of choosing an optimal subset of tasks to perform, with dependencies between pairs of tasks, one example being in open pit mining.

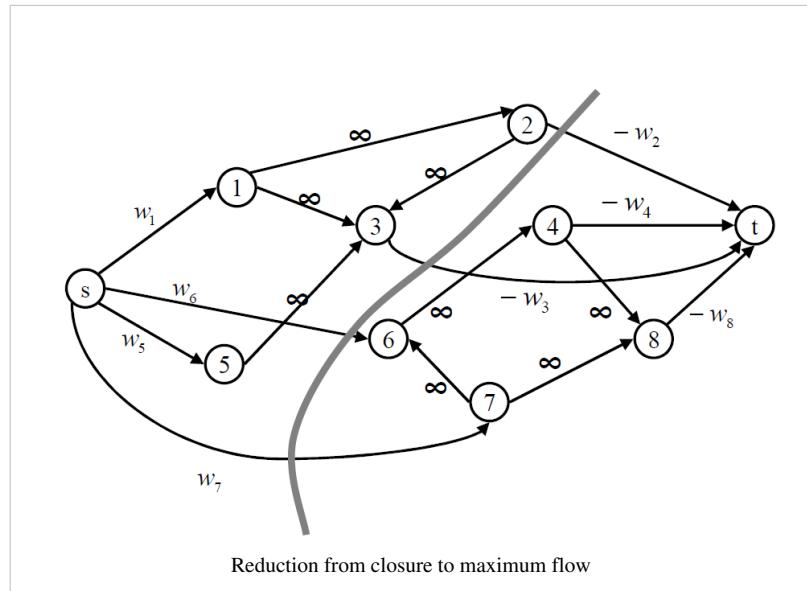
Algorithms

Condensation

The maximum-weight closure of a given graph G is the same as the complement of the minimum-weight closure on the transpose graph of G , so the two problems are equivalent in computational complexity. If two vertices of the graph belong to the same strongly connected component, they must behave the same as each other with respect to all closures: it is not possible for a closure to contain one vertex without containing the other. For this reason, the input graph to a closure problem may be replaced by its condensation, in which every strongly connected component is replaced by a single vertex. The condensation is always a directed acyclic graph.

Reduction to maximum flow

As Picard (1976) showed, a maximum-weight closure may be obtained from G by solving a maximum flow problem on a graph H constructed from G by adding to it two additional vertices s and t . For each vertex v with positive weight in G , the augmented graph H contains an edge from s to v with capacity equal to the weight of v , and for each vertex v with negative weight in G , the augmented graph H contains an edge from v to t whose capacity is the negation of the weight of v . All of the edges in G are given infinite capacity in H .



A minimum cut separating s from t in this graph cannot have any edges of G passing in the forward direction across the cut: a cut with such an edge would have infinite capacity and would not be minimum. Therefore, the set of vertices on the same side of the cut as s automatically forms a closure C . The capacity of the cut equals the weight of all non-negative vertices minus the weight of the vertices in C , which is minimized when the weight of C is maximized. By the max-flow min-cut theorem, a minimum cut, and the optimal closure derived from it, can be found by solving a maximum flow problem.

Alternative algorithms

Alternative algorithms for the maximum closure problem that do not compute flows have also been studied.^[1] Their running time is similar to that of the fastest known flow algorithms.

Applications

Open pit mining

An open pit mine may be modeled as a set of blocks of material which may be removed by mining it once all the blocks directly above it have been removed. A block has a total value, equal to the value of the minerals that can be extracted from it minus the cost of removal and extraction; in some cases, a block has no extraction value but must still be removed to reach other blocks, giving it a negative value. One may define an acyclic network that has as its vertices the blocks of a mine, with an edge from each block to the blocks above it that must be removed earlier than it. The weight of each vertex in this network is the total value of its block, and the most profitable plan for mining can be determined by finding a maximum weight closure, and then forming a topological ordering of the blocks in this closure.^[2]

Military targeting

In military operations, high-value targets such as command centers are frequently protected by layers of defense systems, which may in turn be protected by other systems. In order to reach a target, all of its defenses must be taken down, making it into a secondary target. Each target needs a certain amount of resources to be allocated to it in order to perform a successful attack. The optimal set of targets to attack, to obtain the most value for the resources expended, can be modeled as a closure problem.^[3]

Transportation network design

The problem of planning a freight delivery system may be modeled by a network in which the vertices represent cities and the (undirected) edges represent potential freight delivery routes between pairs of cities. Each route can achieve a certain profit, but can only be used if freight depots are constructed at both its ends, with a certain cost. The problem of designing a network that maximizes the difference between the profits and the costs can be solved as a closure problem, by subdividing each undirected edge into two directed edges, both directed outwards from the subdivision point. The weight of each subdivision point is a positive number, the profit of the corresponding route, and the weight of each original graph vertex is a negative number, the cost of building a depot in that city. Together with open pit mining, this was one of the original motivating applications for studying the closure problem; it was originally studied in 1970, in two independent papers published in the same issue of the same journal by J. M. W. Rhys and Michel Balinski.

References

[1] . As cited by .

[2] . As cited by .

[3] . As cited by .

Minimum-cost flow problem

The **minimum-cost flow problem** is to find the cheapest possible way of sending a certain amount of flow through a flow network. Solving this problem is useful for real-life situations involving networks with costs associated (e.g. telecommunications networks), as well as in other situations where the analogy is not so obvious, such as where to locate warehouses.

Definition

Given a flow network, that is, a directed graph $G = (V, E)$ with source $s \in V$ and sink $t \in V$, where edge $(u, v) \in E$ has capacity $c(u, v) > 0$, flow $f(u, v) \geq 0$ and cost $a(u, v)$ (most minimum-cost flow algorithms support edges with negative costs). The cost of sending this flow is $f(u, v) \cdot a(u, v)$. You are required to send an amount of flow d from s to t .

The definition of the problem is to minimize the **total cost** of the flow:

$$\sum_{(u,v) \in E} a(u, v) \cdot f(u, v)$$

with the constraints

Capacity constraints: $f(u, v) \leq c(u, v)$

Skew symmetry: $f(u, v) = -f(v, u)$

Flow conservation: $\sum_{w \in V} f(u, w) = 0$ for all $u \neq s, t$

Required flow: $\sum_{w \in V} f(s, w) = d$ and $\sum_{w \in V} f(w, t) = d$

Relation to other problems

A variation of this problem is to find a flow which is maximum, but has the lowest cost among the maximums. This could be called a minimum-cost maximum-flow problem. This is useful for finding minimum cost maximum matchings.

With some solutions, finding the minimum cost maximum flow instead is straightforward. If not, you can do a binary search on d .

A related problem is the minimum cost circulation problem, which can be used for solving minimum cost flow. You do this by setting the lower bound on all edges to zero, and then make an extra edge from the sink t to the source s , with capacity $c(t, s) = d$ and lower bound $l(t, s) = d$, forcing the total flow from s to t to also be d .

The problem can be specialized into two other problems:

- if the capacity constraint is removed, the problem is reduced to the shortest path problem,
- if the costs are all set equal to zero, the problem is reduced to the maximum flow problem.

Solutions

The minimum cost flow problem can be solved by linear programming, since we optimize a linear function, and all constraints are linear.

Apart from that, many combinatorial algorithms exist, for a comprehensive survey, see [1]. Some of them are generalizations of maximum flow algorithms, others use entirely different approaches.

Well-known fundamental algorithms (they have many variations):

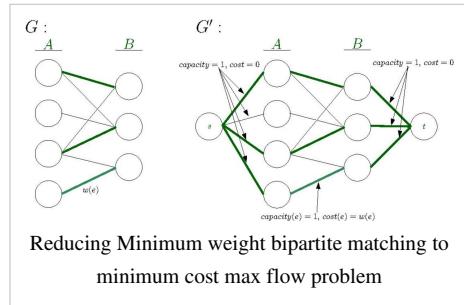
- *Cycle canceling*: a general primal method.[2]
- *Minimum mean cycle canceling*: a simple strongly polynomial algorithm.[3]
- *Successive shortest path* and *capacity scaling*: dual methods, which can be viewed as the generalizations of the Ford–Fulkerson algorithm.[4]
- *Cost scaling*: a primal-dual approach, which can be viewed as the generalization of the push-relabel algorithm.[5]
- *Network simplex*: a specialized version of the linear programming simplex method, which runs in polynomial time.[6]
- *Out-of-kilter algorithm* by D. R. Fulkerson

Application

Minimum weight bipartite matching

Given an bipartite graph $G = (A \cup B, E)$, we like to find the maximum cardinality matching in G that has minimum cost. Let $w: E \rightarrow R$ be a weight function on the edges of E . The minimum weight bipartite matching problem or assignment problem is to find a perfect matching $M \subseteq E$ whose total weight is minimized. The idea is to reduce this problem to a network flow problem.

Let $G' = (V' = A \cup B, E' = E)$. Assign the capacity of all the edges in E' to 1. Add a source vertex s and connect it to all the vertices in A' and add a sink vertex t and connect all vertices inside group B' to this vertex. The capacity of all the new edges is 1 and their costs is 0. It is proved that there is minimum weight perfect bipartite matching in G if and only if there a minimum cost flow in G' . [1]



References

1. ^ Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc. ISBN 0-13-617549-X.
2. ^ Morton Klein (1967). "A primal method for minimal cost flows with applications to the assignment and transportation problems". *Management Science* **14**: 205–220. doi:10.1287/mnsc.14.3.205 [7].
3. ^ Andrew V. Goldberg and Robert E. Tarjan (1989). "Finding minimum-cost circulations by canceling negative cycles". *Journal of the ACM* **36** (4): 873–886. doi:10.1145/76359.76368 [8].
4. ^ Jack Edmonds and Richard M. Karp (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". *Journal of the ACM* **19** (2): 248–264. doi:10.1145/321694.321699 [9].
5. ^ Andrew V. Goldberg and Robert E. Tarjan (1990). "Finding minimum-cost circulations by successive approximation". *Math. Oper. Res.* **15** (3): 430–466. doi:10.1287/moor.15.3.430 [10].
6. ^ James B. Orlin (1997). "A polynomial time primal network simplex algorithm for minimum cost flows". *Mathematical Programming* **78**: 109–129. doi:10.1007/bf02614365 [11].

External links

- LEMON C++ library with several maximum flow and minimum cost circulation algorithms [12]

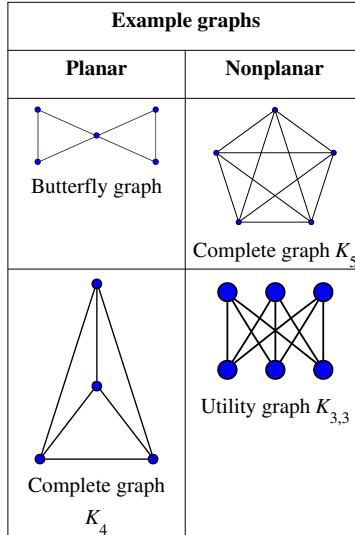
References

- [1] http://en.wikipedia.org/wiki/Minimum-cost_flow_problem#endnote_AMO93
- [2] http://en.wikipedia.org/wiki/Minimum-cost_flow_problem#endnote_K67
- [3] http://en.wikipedia.org/wiki/Minimum-cost_flow_problem#endnote_GT89
- [4] http://en.wikipedia.org/wiki/Minimum-cost_flow_problem#endnote_EK72
- [5] http://en.wikipedia.org/wiki/Minimum-cost_flow_problem#endnote_GT90
- [6] http://en.wikipedia.org/wiki/Minimum-cost_flow_problem#endnote_O97
- [7] <http://dx.doi.org/10.1287%2Fmnsc.14.3.205>
- [8] <http://dx.doi.org/10.1145%2F76359.76368>
- [9] <http://dx.doi.org/10.1145%2F321694.321699>
- [10] <http://dx.doi.org/10.1287%2Fmoor.15.3.430>
- [11] <http://dx.doi.org/10.1007%2Fbf02614365>
- [12] <http://lemon.cs.elte.hu/>

Graph drawing and planar graphs

Planar graph

"Triangular graph" redirects here. For the alternative name of chordal graphs, see Triangulated graph. For data graphs plotted across three variables, see Ternary plot.



In graph theory, a **planar graph** is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other. Such a drawing is called a **plane graph** or **planar embedding of the graph**. A plane graph can be defined as a planar graph with a mapping from every node to a point on a plane, and from every edge to a plane curve on that plane, such that the extreme points of each curve are the points mapped from its end nodes, and all curves are disjoint except on their extreme points.

Every graph that can be drawn on a plane can be drawn on the sphere as well, and vice versa.

Plane graphs can be encoded by combinatorial maps.

The equivalence class of topologically equivalent drawings on the sphere is called a **planar map**. Although a plane graph has an **external** or **unbounded** face, none of the faces of a planar map have a particular status.

A generalization of planar graphs are graphs which can be drawn on a surface of a given genus. In this terminology, planar graphs have graph genus 0, since the plane (and the sphere) are surfaces of genus 0. See "graph embedding" for other related topics.

Kuratowski's and Wagner's theorems

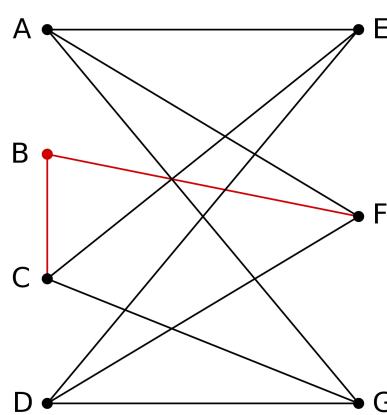
The Polish mathematician Kazimierz Kuratowski provided a characterization of planar graphs in terms of forbidden graphs, now known as Kuratowski's theorem:

A finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 (the complete graph on five vertices) or $K_{3,3}$ (complete bipartite graph on six vertices, three of which connect to each of the other three, also known as the utility graph).

A subdivision of a graph results from inserting vertices into edges (for example, changing an edge $\bullet-\bullet$ to $\bullet-\bullet-\bullet$) zero or more times.

Instead of considering subdivisions, Wagner's theorem deals with minors:

A finite graph is planar if and only if it does not have K_5 or $K_{3,3}$ as a minor.



An example of a graph which doesn't have K_5 or $K_{3,3}$ as its subgraph. However, it has a subgraph that is homeomorphic to $K_{3,3}$ and is therefore not planar.

Klaus Wagner asked more generally whether any minor-closed class of graphs is determined by a finite set of "forbidden minors". This is now the Robertson–Seymour theorem, proved in a long series of papers. In the language of this theorem, K_5 and $K_{3,3}$ are the forbidden minors for the class of finite planar graphs.

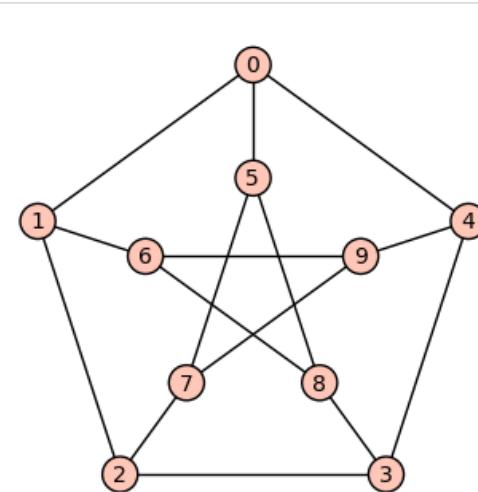
Other planarity criteria

In practice, it is difficult to use Kuratowski's criterion to quickly decide whether a given graph is planar. However, there exist fast algorithms for this problem: for a graph with n vertices, it is possible to determine in time $O(n)$ (linear time) whether the graph may be planar or not (see planarity testing).

For a simple, connected, planar graph with v vertices and e edges, the following simple conditions hold:

Theorem 1. If $v \geq 3$ then $e \leq 3v - 6$;

Theorem 2. If $v \geq 3$ and there are no cycles of length 3, then $e \leq 2v - 4$.



An animation showing that the Petersen graph contains a minor isomorphic to the $K_{3,3}$ graph

In this sense, planar graphs are sparse graphs, in that they have only $O(v)$ edges, asymptotically smaller than the maximum $O(v^2)$. The graph $K_{3,3}$, for example, has 6 vertices, 9 edges, and no cycles of length 3. Therefore, by Theorem 2, it cannot be planar. Note that these theorems provide necessary conditions for planarity that are not sufficient conditions, and therefore can only be used to prove a graph is not planar, not that it is planar. If both theorem 1 and 2 fail, other methods may be used.

- Whitney's planarity criterion gives a characterization based on the existence of an algebraic dual;
- MacLane's planarity criterion gives an algebraic characterization of finite planar graphs, via their cycle spaces;

- The Fraysseix–Rosenstiehl planarity criterion gives a characterization based on the existence of a bipartition of the cotree edges of a depth-first search tree. It is central to the **left-right planarity testing algorithm**;
- Schnyder's theorem gives a characterization of planarity in terms of partial order dimension;
- Colin de Verdière's planarity criterion gives a characterization based on the maximum multiplicity of the second eigenvalue of certain Schrödinger operators defined by the graph.

Euler's formula

Main article: Euler characteristic § Planar graphs

Euler's formula states that if a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely large region), then

$$v - e + f = 2.$$

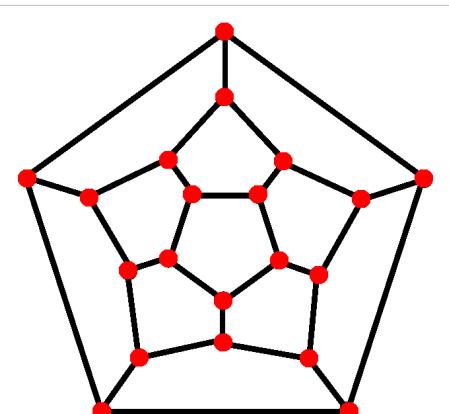
As an illustration, in the butterfly graph given above, $v = 5$, $e = 6$ and $f = 3$. If the second graph is redrawn without edge intersections, it has $v = 4$, $e = 6$ and $f = 4$. In general, if the property holds for all planar graphs of f faces, any change to the graph that creates an additional face while keeping the graph planar would keep $v - e + f$ an invariant. Since the property holds for all graphs with $f = 2$, by mathematical induction it holds for all cases. Euler's formula can also be proved as follows: if the graph isn't a tree, then remove an edge which completes a cycle. This lowers both e and f by one, leaving $v - e + f$ constant. Repeat until the remaining graph is a tree; trees have $v = e + 1$ and $f = 1$, yielding $v - e + f = 2$. i.e. the Euler characteristic is 2.

In a finite, connected, *simple*, planar graph, any face (except possibly the outer one) is bounded by at least three edges and every edge touches at most two faces; using Euler's formula, one can then show that these graphs are *sparse* in the sense that $e \leq 3v - 6$ if $v \geq 3$.

Euler's formula is also valid for convex polyhedra. This is no coincidence: every convex polyhedron can be turned into a connected, simple, planar graph by using the Schlegel diagram of the polyhedron, a perspective projection of the polyhedron onto a plane with the center of perspective chosen near the center of one of the polyhedron's faces. Not every planar graph corresponds to a convex polyhedron in this way: the trees do not, for example. Steinitz's theorem says that the polyhedral graphs formed from convex polyhedra are precisely the finite 3-connected simple planar graphs. More generally, Euler's formula applies to any polyhedron whose faces are simple polygons that form a surface topologically equivalent to a sphere, regardless of its convexity.

Average degree

From $v - e + f = 2$ and $2e \geq 3f$ (one face has minimum 3 edges and each edge has maximum two faces) it follows via algebraic transformations that the average degree is strictly less than 6. Otherwise the given graph can't be planar.



A Schlegel diagram of a regular dodecahedron, forming a planar graph from a convex polyhedron.

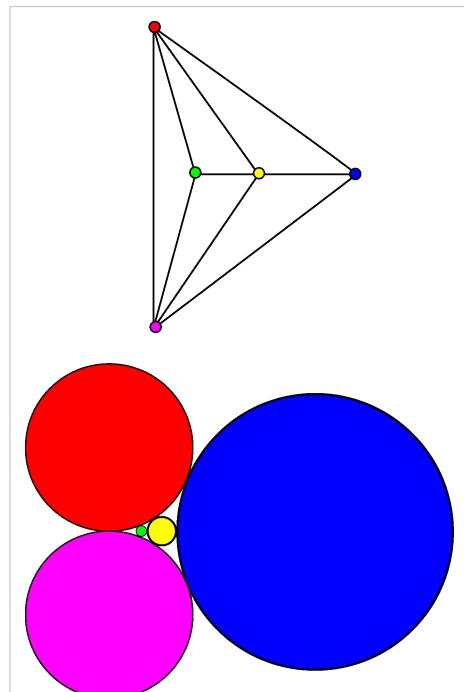
Coin graphs

Main article: Circle packing theorem

We say that two circles drawn in a plane *kiss* (or *osculate*) whenever they intersect in exactly one point. A "coin graph" is a graph formed by a set of circles, no two of which have overlapping interiors, by making a vertex for each circle and an edge for each pair of circles that kiss. The circle packing theorem, first proved by Paul Koebe in 1936, states that a graph is planar if and only if it is a coin graph.

This result provides an easy proof of Fáry's theorem, that every planar graph can be embedded in the plane in such a way that its edges are straight line segments that do not cross each other. If one places each vertex of the graph at the center of the corresponding circle in a coin graph representation, then the line segments between centers of kissing circles do not cross any of the other edges.

Related families of graphs

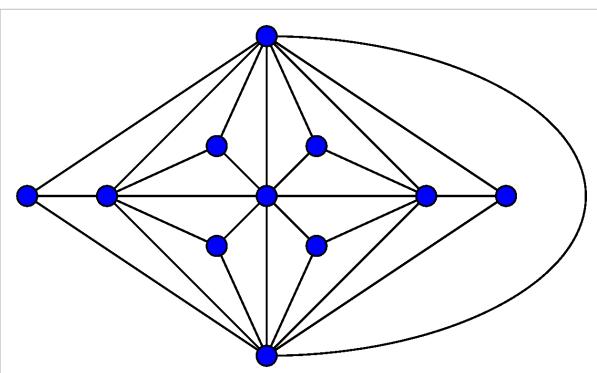


Example of the circle packing theorem on K_5 , the complete graph on five vertices, minus one edge.

Maximal planar graphs

A simple graph is called **maximal planar** if it is planar but adding any edge (on the given vertex set) would destroy that property. All faces (including the outer one) are then bounded by three edges, explaining the alternative term **plane triangulation**. The alternative names "triangular graph" or "triangulated graph" have also been used, but are ambiguous, as they more commonly refer to the line graph of a complete graph and to the chordal graphs respectively.

If a maximal planar graph has v vertices with $v > 2$, then it has precisely $3v - 6$ edges and $2v - 4$ faces.



The Goldner–Harary graph is maximal planar. All its faces are bounded by three edges.

Apollonian networks are the maximal planar graphs formed by repeatedly splitting triangular faces into triples of smaller triangles. Equivalently, they are the planar 3-trees.

Strangulated graphs are the graphs in which every peripheral cycle is a triangle. In a maximal planar graph (or more generally a polyhedral graph) the peripheral cycles are the faces, so maximal planar graphs are strangulated. The strangulated graphs include also the chordal graphs, and are exactly the graphs that can be formed by clique-sums (without deleting edges) of complete graphs and maximal planar graphs.

Outerplanar graphs

Outerplanar graphs are graphs with an embedding in the plane such that all vertices belong to the unbounded face of the embedding. Every outerplanar graph is planar, but the converse is not true: K_4 is planar but not outerplanar. A theorem similar to Kuratowski's states that a finite graph is outerplanar if and only if it does not contain a subdivision of K_4 or of $K_{2,3}$.

A 1-outerplanar embedding of a graph is the same as an outerplanar embedding. For $k > 1$ a planar embedding is k -outerplanar if removing the vertices on the outer face results in a $(k - 1)$ -outerplanar embedding. A graph is k -outerplanar if it has a k -outerplanar embedding.

Halin graphs

A Halin graph is a graph formed from an undirected plane tree (with no degree-two nodes) by connecting its leaves into a cycle, in the order given by the plane embedding of the tree. Equivalently, it is a polyhedral graph in which one face is adjacent to all the others. Every Halin graph is planar. Like outerplanar graphs, Halin graphs have low treewidth, making many algorithmic problems on them more easily solved than in unrestricted planar graphs.

Other related families

An apex graph is a graph that may be made planar by the removal of one vertex, and a k -apex graph is a graph that may be made planar by the removal of at most k vertices.

A 1-planar graph is a graph that may be drawn in the plane with at most one simple crossing per edge, and a k -planar graph is a graph that may be drawn with at most k simple crossings per edge.

A toroidal graph is a graph that can be embedded without crossings on the torus. More generally, the genus of a graph is the minimum genus of a two-dimensional surface into which the graph may be embedded; planar graphs have genus zero and nonplanar toroidal graphs have genus one.

Any graph may be embedded into three-dimensional space without crossings. However, a three-dimensional analogue of the planar graphs is provided by the linklessly embeddable graphs, graphs that can be embedded into three-dimensional space in such a way that no two cycles are topologically linked with each other. In analogy to Kuratowski's and Wagner's characterizations of the planar graphs as being the graphs that do not contain K_5 or $K_{3,3}$ as a minor, the linklessly embeddable graphs may be characterized as the graphs that do not contain as a minor any of the seven graphs in the Petersen family. In analogy to the characterizations of the outerplanar and planar graphs as being the graphs with Colin de Verdière graph invariant at most two or three, the linklessly embeddable graphs are the graphs that have Colin de Verdière invariant at most four.

An upward planar graph is a directed acyclic graph that can be drawn in the plane with its edges as non-crossing curves that are consistently oriented in an upward direction. Not every planar directed acyclic graph is upward planar, and it is NP-complete to test whether a given graph is upward planar.

Enumeration of planar graphs

The asymptotic for the number of (labeled) planar graphs on n vertices is $g \cdot n^{-7/2} \cdot \gamma^n \cdot n!$, where $\gamma \approx 27.22687$ and $g \approx 0.43 \times 10^{-5}$.^[1]

Almost all planar graphs have an exponential number of automorphisms.^[2]

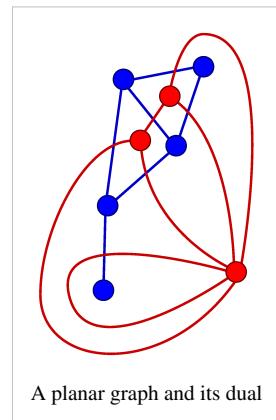
The number of unlabeled (non-isomorphic) planar graphs on n vertices is between 27.2^n and 30.06^n .^[3]

Other facts and definitions

Every planar graph is 4-partite, or 4-colorable; this is the graph-theoretical formulation of the four color theorem.

Fáry's theorem states that every simple planar graph admits an embedding in the plane such that all edges are straight line segments which don't intersect. A universal point set is a set of points such that every planar graph with n vertices has such an embedding with all vertices in the point set; there exist universal point sets of quadratic size, formed by taking a rectangular subset of the integer lattice. Every simple outerplanar graph admits an embedding in the plane such that all vertices lie on a fixed circle and all edges are straight line segments that lie inside the disk and don't intersect, so n -vertex regular polygons are universal for outerplanar graphs.

Given an embedding G of a (not necessarily simple) connected graph in the plane without edge intersections, we construct the **dual graph** G^* as follows: we choose one vertex in each face of G (including the outer face) and for each edge e in G we introduce a new edge in G^* connecting the two vertices in G^* corresponding to the two faces in G that meet at e . Furthermore, this edge is drawn so that it crosses e exactly once and that no other edge of G or G^* is intersected. Then G^* is again the embedding of a (not necessarily simple) planar graph; it has as many edges as G , as many vertices as G has faces and as many faces as G has vertices. The term "dual" is justified by the fact that $G^{**} = G$; here the equality is the equivalence of embeddings on the sphere. If G is the planar graph corresponding to a convex polyhedron, then G^* is the planar graph corresponding to the dual polyhedron.



A planar graph and its dual

Duals are useful because many properties of the dual graph are related in simple ways to properties of the original graph, enabling results to be proven about graphs by examining their dual graphs.

While the dual constructed for a particular embedding is unique (up to isomorphism), graphs may have different (i.e. non-isomorphic) duals, obtained from different (i.e. non-homeomorphic) embeddings.

A *Euclidean graph* is a graph in which the vertices represent points in the plane, and the edges are assigned lengths equal to the Euclidean distance between those points; see Geometric graph theory.

A plane graph is said to be *convex* if all of its faces (including the outer face) are convex polygons. A planar graph may be drawn convexly if and only if it is a subdivision of a 3-vertex-connected planar graph.

Scheinerman's conjecture (now a theorem) states that every planar graph can be represented as an intersection graph of line segments in the plane.

The planar separator theorem states that every n -vertex planar graph can be partitioned into two subgraphs of size at most $2n/3$ by the removal of $O(\sqrt{n})$ vertices. As a consequence, planar graphs also have treewidth and branch-width $O(\sqrt{n})$.

For two planar graphs with v vertices, it is possible to determine in time $O(v)$ whether they are isomorphic or not (see also graph isomorphism problem).^[4]

Notes

- [1] Omer Giménez and Marc Noy. Asymptotic enumeration and limit laws of planar graphs. *J. Amer. Math. Soc.* 22 (2009), 309–329.
- [2] Colin McDiarmid, Angelika Steger, Dominic J.A. Welsh, Random planar graphs, *Journal of Combinatorial Theory, Series B*, Volume 93, Issue 2, March 2005, Pages 187–205, ISSN 0095-8956, <http://dx.doi.org/10.1016/j.jctb.2004.09.007>.
- [3] N. Bonichon, C. Gavoille, N. Hanusse, D. Poulalhon, G. Schaeffer, Planar Graphs, via Well-Orderly Maps and Trees, *Graphs and Combinatorics* 22 (2006), 185–202.
- [4] I. S. Filotti, Jack N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, p.236–243. 1980.

References

- Kuratowski, Kazimierz (1930), "Sur le problème des courbes gauches en topologie" (<http://matwbn.icm.edu.pl/ksiazki/fm/fm15/fm15126.pdf>), *Fund. Math.* (in French) **15**: 271–283.
- Wagner, K. (1937), "Über eine Eigenschaft der ebenen Komplexe", *Math. Ann.* **114**: 570–590, doi: [10.1007/BF01594196](https://doi.org/10.1007/BF01594196) (<http://dx.doi.org/10.1007/BF01594196>).
- Boyer, John M.; Myrvold, Wendy J. (2005), "On the cutting edge: Simplified O(n) planarity by edge addition" (<http://jgaa.info/accepted/2004/BoyerMyrvold2004.8.3.pdf>), *Journal of Graph Algorithms and Applications* **8** (3): 241–273, doi: [10.7155/jgaa.00091](https://doi.org/10.7155/jgaa.00091) (<http://dx.doi.org/10.7155/jgaa.00091>).
- McKay, Brendan; Brinkmann, Gunnar, *A useful planar graph generator* (<http://cs.anu.edu.au/~bdm/plantri/>).
- de Fraysseix, H.; Ossona de Mendez, P.; Rosenstiehl, P. (2006), "Trémaux trees and planarity", *International Journal of Foundations of Computer Science* **17** (5): 1017–1029, doi: [10.1142/S0129054106004248](https://doi.org/10.1142/S0129054106004248) (<http://dx.doi.org/10.1142/S0129054106004248>). Special Issue on Graph Drawing.
- D.A. Bader and S. Sreshta, A New Parallel Algorithm for Planarity Testing (<http://www.cc.gatech.edu/~bader/papers/planarity2003.html>), UNM-ECE Technical Report 03-002, October 1, 2003.
- Fisk, Steve (1978), "A short proof of Chvátal's watchman theorem", *J. Comb. Theory, Ser. B* **24** (3): 374, doi: [10.1016/0095-8956\(78\)90059-X](https://doi.org/10.1016/0095-8956(78)90059-X) ([http://dx.doi.org/10.1016/0095-8956\(78\)90059-X](http://dx.doi.org/10.1016/0095-8956(78)90059-X)).

External links



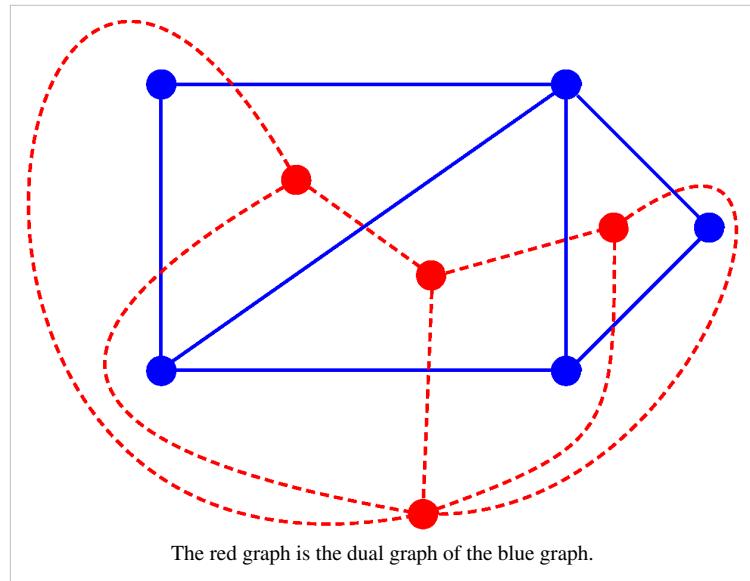
Wikimedia Commons has media related to *Planar graphs*.

- Edge Addition Planarity Algorithm Source Code, version 1.0 (<http://jgaa.info/accepted/2004/BoyerMyrvold2004.8.3/planarity.zip>) — Free C source code for reference implementation of Boyer–Myrvold planarity algorithm, which provides both a combinatorial planar embedder and Kuratowski subgraph isolator. An open source project with free licensing provides the Edge Addition Planarity Algorithms, current version (<http://code.google.com/p/planarity/>).
- Public Implementation of a Graph Algorithm Library and Editor (<http://pigale.sourceforge.net>) — GPL graph algorithm library including planarity testing, planarity embedder and Kuratowski subgraph exhibition in linear time.
- Boost Graph Library tools for planar graphs (http://www.boost.org/doc/libs/1_40_0/libs/graph/doc/planar_graphs.html), including linear time planarity testing, embedding, Kuratowski subgraph isolation, and straight-line drawing.
- 3 Utilities Puzzle and Planar Graphs (http://www.cut-the-knot.org/do_you_know/3Utilities.shtml)
- NetLogo Planarity model (<http://ccl.northwestern.edu/netlogo/models/Planarity>) — NetLogo version of John Tantalo's game

Dual graph

In the mathematical discipline of graph theory, the **dual graph** of a plane graph G is a graph that has a vertex corresponding to each face of G , and an edge joining two neighboring faces for each edge in G . The term "dual" is used because this property is symmetric, meaning that if H is a dual of G , then G is a dual of H (if G is connected). The same notion of duality may also be used for more general embeddings of graphs in manifolds.

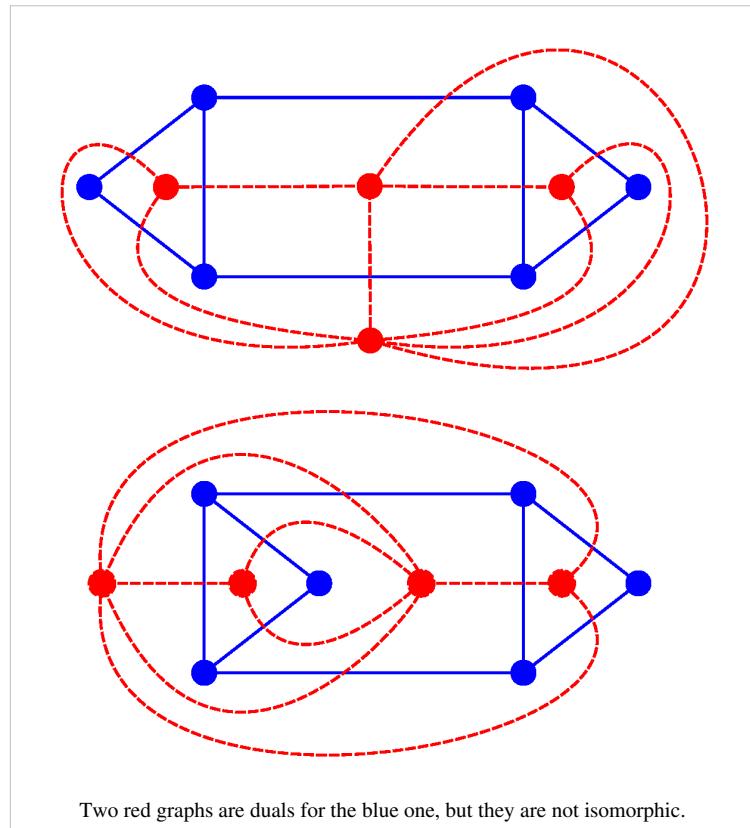
The notion described in this page is different from the edge-to-vertex dual (Line graph) of a graph and should not be confused with it.



Properties

- The dual of a plane graph is a plane multigraph - multiple edges.^[1]
- If G is a connected plane graph and if G' is the dual of G then G is isomorphic to the dual of G' .
- Since the dual graph depends on a particular embedding, the dual graph of a planar graph is not unique in the sense that the same planar graph can have non-isomorphic dual graphs. In the picture, the red graphs are not isomorphic because the upper one has a vertex with degree 6 (the outer face). However, if the graph is 3-connected, then Whitney showed that the embedding, and thus the dual graph, is unique.

Because of the duality, any result involving counting faces and vertices can be dualized by exchanging them.



Algebraic dual

Let G be a connected graph. An **algebraic dual** of G is a graph G^* such that G and G^* have the same set of edges, any cycle of G is a cut of G^* , and any cut of G is a cycle of G^* . Every planar graph has an algebraic dual, which is in general not unique (any dual defined by a plane embedding will do). The converse is actually true, as settled by Hassler Whitney in the Whitney's planarity criterion:

A connected graph G is planar if and only if it has an algebraic dual.

The same fact can be expressed in the theory of matroids: if M is the graphic matroid of a graph G , then the dual matroid of M is a graphic matroid if and only if G is planar. If G is planar, the dual matroid is the graphic matroid of the dual graph of G .

Weak dual

The **weak dual** of a plane graph is the subgraph of the dual graph whose vertices correspond to the bounded faces of the primal graph. A plane graph is outerplanar if and only if its weak dual is a forest, and a plane graph is a Halin graph if and only if its weak dual is biconnected and outerplanar. For any plane graph G , let G^+ be the plane multigraph formed by adding a single new vertex v in the unbounded face of G , and connecting v to each vertex of the outer face (multiple times, if a vertex appears multiple times on the boundary of the outer face); then, G is the weak dual of the (plane) dual of G^+ .

Notes

[1] Here we consider that graphs may have loops and multiple edges to avoid uncommon considerations.

External links

- Weisstein, Eric W., "Dual graph" (<http://mathworld.wolfram.com/DualGraph.html>), *MathWorld*.
- Weisstein, Eric W., "Self-dual graph" (<http://mathworld.wolfram.com/Self-DualGraph.html>), *MathWorld*.

Fáry's theorem

For other uses, see Fary–Milnor theorem.

In mathematics, **Fáry's theorem** states that any simple planar graph can be drawn without crossings so that its edges are straight line segments. That is, the ability to draw graph edges as curves instead of as straight line segments does not allow a larger class of graphs to be drawn. The theorem is named after István Fáry, although it was proved independently by Klaus Wagner (1936), Fáry (1948), and S. K. Stein (1951).

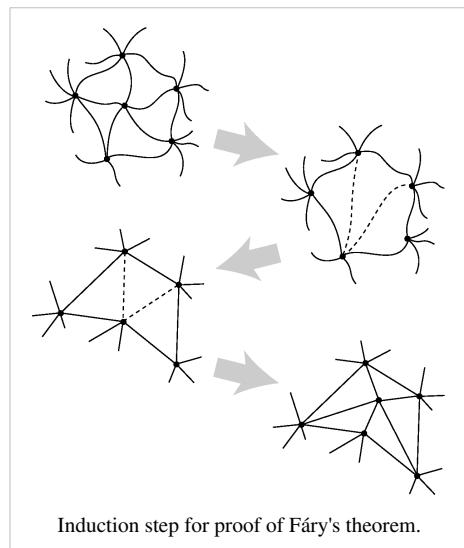
Proof

One way of proving Fáry's theorem is to use mathematical induction.^[1]

Let G be a simple planar graph with n vertices; we may add edges if necessary so that G is maximal planar. All faces of G will be triangles, as we could add an edge into any face with more sides while preserving planarity, contradicting the assumption of maximal planarity. Choose some three vertices a, b, c forming a triangular face of G . We prove by induction on n that there exists a straight-line embedding of G in which triangle abc is the outer face of the embedding. As a base case, the result is trivial when $n = 3$ and a, b and c are the only vertices in G . Otherwise, all vertices in G have at least three neighbors.

By Euler's formula for planar graphs, G has $3n - 6$ edges; equivalently, if one defines the *deficiency* of a vertex v in G to be $6 - \deg(v)$, the sum of the deficiencies is 12. Each vertex in G can have deficiency at most three, so there are at least four vertices with positive deficiency. In particular we can choose a vertex v with at most five neighbors that is different from a, b and c . Let G' be formed by removing v from G and retriangulating the face formed by removing v . By induction, G' has a straight line embedding in which abc is the outer face. Remove the added edges in G' , forming a polygon P with at most five sides into which v should be placed to complete the drawing. By the Art gallery theorem, there exists a point interior to P at which v can be placed so that the edges from v to the vertices of P do not cross any other edges, completing the proof.

The induction step of this proof is illustrated at right.



Induction step for proof of Fáry's theorem.

Related results

De Fraysseix, Pach and Pollack showed how to find in linear time a straight-line drawing in a grid with dimensions linear in the size of the graph, giving a universal point set with quadratic size. A similar method has been followed by Schnyder to prove enhanced bounds and a characterization of planarity based on the incidence partial order. His work stressed the existence of a particular partition of the edges of a maximal planar graph into three trees known as a Schnyder wood.

Tutte's spring theorem states that every 3-connected planar graph can be drawn on a plane without crossings so that its edges are straight line segments and an outside face is a convex polygon (Tutte 1963). It is so called because such an embedding can be found as the equilibrium position for a system of springs representing the edges of the graph.

Steinitz's theorem states that every 3-connected planar graph can be represented as the edges of a convex polyhedron in three-dimensional space. A straight-line embedding of G , of the type described by Tutte's theorem, may be formed by projecting such a polyhedral representation onto the plane.

The Circle packing theorem states that every planar graph may be represented as the intersection graph of a collection of non-crossing circles in the plane. Placing each vertex of the graph at the center of the corresponding circle leads to a straight line representation.

List of unsolved problems in mathematics

Does every planar graph have a straight line representation in which all edge lengths are integers?

Heiko Harborth raised the question of whether every planar graph has a straight line representation in which all edge lengths are integers.^[2] The answer remains unknown as of 2009[3]. However, integer-distance straight line embeddings are known to exist for cubic graphs.

Sachs (1983) raised the question of whether every graph with a linkless embedding in three-dimensional Euclidean space has a linkless embedding in which all edges are represented by straight line segments, analogously to Fáry's theorem for two-dimensional embeddings.

Notes

[1] The proof that follows can be found in .

[2] ; ; .

[3] http://en.wikipedia.org/w/index.php?title=F%C3%A1ry%27s_theorem&action=edit

References

- Fáry, István (1948), "On straight-line representation of planar graphs", *Acta Sci. Math. (Szeged)* **11**: 229–233, MR 0026311 (<http://www.ams.org/mathscinet-getitem?mr=0026311>).
- de Fraysseix, Hubert; Pach, János; Pollack, Richard (1988), "Small sets supporting Fáry embeddings of planar graphs", *Twentieth Annual ACM Symposium on Theory of Computing*, pp. 426–433, doi: 10.1145/62212.62254 (<http://dx.doi.org/10.1145/62212.62254>), ISBN 0-89791-264-0.
- de Fraysseix, Hubert; Pach, János; Pollack, Richard (1990), "How to draw a planar graph on a grid", *Combinatorica* **10**: 41–51, doi: 10.1007/BF02122694 (<http://dx.doi.org/10.1007/BF02122694>), MR 1075065 (<http://www.ams.org/mathscinet-getitem?mr=1075065>).
- Geelen, Jim; Guo, Anjie; McKinnon, David (2008), "Straight line embeddings of cubic planar graphs with integer edge lengths" (<http://www.math.uwaterloo.ca/~dmckinnon/Papers/Planar.pdf>), *J. Graph Theory* **58** (3): 270–274, doi: 10.1002/jgt.20304 (<http://dx.doi.org/10.1002/jgt.20304>).
- Harborth, H.; Kemnitz, A.; Moller, M.; Sussenbach, A. (1987), "Ganzzahlige planare Darstellungen der platonischen Körper", *Elem. Math.* **42**: 118–122.
- Kemnitz, A.; Harborth, H. (2001), "Plane integral drawings of planar graphs", *Discrete Math.* **236**: 191–195, doi: 10.1016/S0012-365X(00)00442-8 ([http://dx.doi.org/10.1016/S0012-365X\(00\)00442-8](http://dx.doi.org/10.1016/S0012-365X(00)00442-8)).
- Mohar, Bojan (2003), *Problems from the book Graphs on Surfaces* (<http://www.fmf.uni-lj.si/~mohar/Book/BookProblems.html>).
- Mohar, Bojan; Thomassen, Carsten (2001), *Graphs on Surfaces*, Johns Hopkins University Press, pp. roblem 2.8.15, ISBN 0-8018-6689-8.
- Sachs, Horst (1983), "On a spatial analogue of Kuratowski's theorem on planar graphs — An open problem", in Horowiecki, M.; Kennedy, J. W.; Sysło, M. M., *Graph Theory: Proceedings of a Conference held in Łagów, Poland, February 10–13, 1981*, Lecture Notes in Mathematics **1018**, Springer-Verlag, pp. 230–241, doi: 10.1007/BFb0071633 (<http://dx.doi.org/10.1007/BFb0071633>), ISBN 978-3-540-12687-4.
- Schnyder, Walter (1990), "Embedding planar graphs on the grid" (<http://portal.acm.org/citation.cfm?id=320176.320191>), *Proc. 1st ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pp. 138–148.

- Stein, S. K. (1951), "Convex maps", *Proceedings of the American Mathematical Society* **2** (3): 464–466, doi: 10.2307/2031777 (<http://dx.doi.org/10.2307/2031777>), JSTOR 2031777 (<http://www.jstor.org/stable/2031777>), MR 0041425 (<http://www.ams.org/mathscinet-getitem?mr=0041425>).
- Tutte, W. T. (1963), "How to draw a graph", *Proceedings of the London Mathematical Society* **13**: 743–767, doi: 10.1112/plms/s3-13.1.743 (<http://dx.doi.org/10.1112/plms/s3-13.1.743>), MR 0158387 (<http://www.ams.org/mathscinet-getitem?mr=0158387>).
- Wagner, Klaus (1936), "Bemerkungen zum Vierfarbenproblem" (<http://www.digizeitschriften.de/index.php?id=resolveppn&PPN=GDZPPN002131633>), *Jahresbericht der Deutschen Mathematiker-Vereinigung* **46**: 26–32.

Steinitz's theorem

In polyhedral combinatorics, a branch of mathematics, **Steinitz's theorem** is a characterization of the undirected graphs formed by the edges and vertices of three-dimensional convex polyhedra: they are exactly the 3-vertex-connected planar graphs.^{[1][2]} That is, every convex polyhedron forms a 3-connected planar graph, and every 3-connected planar graph can be represented as the graph of a convex polyhedron. For this reason, the 3-connected planar graphs are also known as polyhedral graphs. Steinitz's theorem is named after Ernst Steinitz, who proved it in 1922. Branko Grünbaum has called this theorem "the most important and deepest known result on 3-polytopes."

The name "Steinitz's theorem" has also been applied to other results of Steinitz:

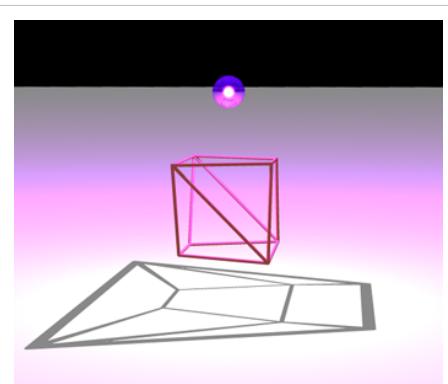
- the Steinitz exchange lemma implying that each basis of a vector space has the same number of vectors,
- the theorem that if the convex hull of a point set contains a unit sphere, then the convex hull of a finite subset of the point contains a smaller concentric sphere, and
- Steinitz's vectorial generalization of the Riemann series theorem on the rearrangements of conditionally convergent series.

Definitions and statement of the theorem

An undirected graph is a system of vertices and edges, each edge connecting two of the vertices. From any polyhedron one can form a graph, by letting the vertices of the graph correspond to the vertices of the polyhedron and by connecting any two graph vertices by an edge whenever the corresponding two polyhedron vertices are the endpoints of an edge of the polyhedron. This graph is known as the skeleton of the polyhedron.

A graph is planar if it can be drawn with its vertices as points in the Euclidean plane, and its edges as curves that connect these points, such that no two edge curves cross each other and such that the point representing a vertex lies on the curve representing an edge only when the vertex is an endpoint of the edge. By Fáry's theorem, it is sufficient to consider only planar drawings in which the curves representing the edges are line segments. A graph is 3-connected if, after the removal of any two of its vertices, any other pair of vertices remain connected by a path.

One direction of Steinitz's theorem (the easier direction to prove) states that the graph of every convex polyhedron is planar and 3-connected. As shown in the illustration, planarity can be shown by using a Schlegel diagram: if one



Illuminating the skeleton of a convex polyhedron from a light source close to one of its faces causes its shadows to form a planar Schlegel diagram.

places a light source near one face of the polyhedron, and a plane on the other side, the shadows of the polyhedron edges will form a planar graph, embedded in such a way that the edges are straight line segments. The 3-connectivity of a polyhedral graph is a special case of Balinski's theorem that the graph of any k -dimensional convex polytope is k -connected.

The other, more difficult, direction of Steinitz's theorem states that every planar 3-connected graph is the graph of a convex polyhedron.

Strengthenings and related results

It is possible to prove a stronger form of Steinitz's theorem, that any polyhedral graph can be realized by a convex polyhedron for which all of the vertex coordinates are integers. The integers resulting from Steinitz' original proof are doubly exponential in the number of vertices of the given polyhedral graph; that is, writing them down would require an exponential number of bits. However, subsequent researchers have found graph drawing algorithms that use only $O(n)$ bits per vertex. It is also possible to relax the requirement that the coordinates be integers, and assign coordinates in such a way that the x -coordinates of the vertices are distinct integers in the range $[0, 2n - 4]$ and the other two coordinates are real numbers in the range $[0, 1]$, so that each edge has length at least one while the overall polyhedron has volume $O(n)$.

In any polyhedron that represents a given polyhedral graph G , the faces of G are exactly the cycles in G that do not separate G into two components: that is, removing a facial cycle from G leaves the rest of G as a connected subgraph. It is possible to specify the shape of any one face of the polyhedron: if any non-separating cycle C is chosen, and its vertices are placed in correspondence with the vertices of a two-dimensional convex polygon P , then there exists a polyhedral representation of G in which the face corresponding to C is congruent to P . It is also always possible, given a polyhedral graph G and an arbitrary cycle C , to find a realization such that C forms the silhouette of the realization under parallel projection.

The Koebe–Andreev–Thurston circle packing theorem can be interpreted as providing another strengthening of Steinitz's theorem, that every 3-connected planar graph may be represented as a convex polyhedron in such a way that all of its edges are tangent to the same unit sphere. More generally, if G is a polyhedral graph and K is any smooth three-dimensional convex body, it is possible to find a polyhedral representation of G in which all edges are tangent to K .

In dimensions higher than three, the algorithmic Steinitz problem (given a lattice, determine whether it is the face lattice of a convex polytope) is complete for the existential theory of the reals by Richter-Gebert's universality theorem.

References

- [1] *Lectures on Polytopes*, by Günter M. Ziegler (1995) ISBN 0-387-94365-X , Chapter 4 "Steinitz' Theorem for 3-Polytopes", p.103.
- [2] Branko Grünbaum, *Convex Polytopes*, 2nd edition, prepared by Volker Kaibel, Victor Klee, and Günter M. Ziegler, 2003, ISBN 0-387-40409-0, ISBN 978-0-387-40409-7, 466pp.

Planarity testing

In graph theory, the **planarity testing** problem is the algorithmic problem of testing whether a given graph is a planar graph (that is, whether it can be drawn in the plane without edge intersections). This is a well-studied problem in computer science for which many practical algorithms have emerged, many taking advantage of novel data structures. Most of these methods operate in $O(n)$ time (linear time), where n is the number of edges (or vertices) in the graph, which is asymptotically optimal. Rather than just being a single Boolean value, the output of a planarity testing algorithm may be a planar graph embedding, if the graph is planar, or an obstacle to planarity such as a Kuratowski subgraph if it is not.

Planarity criteria

Planarity testing algorithms typically take advantage of theorems in graph theory that characterize the set of planar graphs in terms that are independent of graph drawings. These include

- Kuratowski's theorem that a graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 (the complete graph on five vertices) or $K_{3,3}$ (the utility graph, a complete bipartite graph on six vertices, three of which connect to each of the other three).
- Wagner's theorem that a graph is planar if and only if it does not contain a minor (subgraph of a contraction) that is isomorphic to K_5 or $K_{3,3}$.
- The Fraysseix–Rosenstiehl planarity criterion, characterizing planar graphs in terms of a left-right ordering of the edges in a depth-first search tree.

The Fraysseix–Rosenstiehl planarity criterion can be used directly as part of algorithms for planarity testing, while Kuratowski's and Wagner's theorems have indirect applications: if an algorithm can find a copy of K_5 or $K_{3,3}$ within a given graph, it can be sure that the input graph is not planar and return without additional computation.

Other planarity criteria, that characterize planar graphs mathematically but are less central to planarity testing algorithms, include Whitney's planarity criterion that a graph is planar if and only if its graphic matroid is also cographic, MacLane's planarity criterion characterizing planar graphs by the bases of their cycle spaces, Schnyder's theorem characterizing planar graphs by the order dimension of an associated partial order, and Colin de Verdière's planarity criterion using spectral graph theory.

Algorithms

Path addition method

The classic *path addition* method of Hopcroft and Tarjan was the first published linear-time planarity testing algorithm in 1974.

Vertex addition method

Vertex addition methods work by maintaining a data structure representing the possible embeddings of an induced subgraph of the given graph, and adding vertices one at a time to this data structure. These methods began with an inefficient $O(n^2)$ method conceived by Lempel, Even and Cederbaum in 1967. It was improved by Even and Tarjan, who found a linear-time solution for the s,t -numbering step, and by Booth and Lueker, who developed the PQ tree data structure. With these improvements it is linear-time and outperforms the path addition method in practice.^[1] This method was also extended to allow a planar embedding (drawing) to be efficiently computed for a planar graph. In 1999, Shih and Hsu simplified these methods using the PC tree (an unrooted variant of the PQ tree) and a postorder traversal of the depth-first search tree of the vertices.

Edge addition method

In 2004, Boyer and Myrvold developed a simplified $O(n)$ algorithm, originally inspired by the PQ tree method, which gets rid of the PQ tree and uses edge additions to compute a planar embedding, if possible. Otherwise, a Kuratowski subdivision (of either K_5 or $K_{3,3}$) is computed. This is one of the two current state-of-the-art algorithms today (the other one is the planarity testing algorithm of de Fraysseix, de Mendez and Rosenstiehl). See for an experimental comparison with a preliminary version of the Boyer and Myrvold planarity test. Furthermore, the Boyer–Myrvold test was extended to extract multiple Kuratowski subdivisions of a non-planar input graph in a running time linearly dependent on the output size. The source code for the planarity test^{[2][3]} and the extraction of multiple Kuratowski subdivisions is publicly available. Algorithms that locate a Kuratowski subgraph in linear time in vertices were developed by Williamson in the 1980s.

Construction sequence method

A different method uses an inductive construction of 3-connected graphs to incrementally build planar embeddings of every 3-connected component of G (and hence a planar embedding of G itself). The construction starts with K_4 and is defined in such a way that every intermediate graph on the way to the full component is again 3-connected. Since such graphs have a unique embedding (up to flipping and the choice of the external face), the next bigger graph, if still planar, must be a refinement of the former graph. This allows to reduce the planarity test to just testing for each step whether the next added edge has both ends in the external face of the current embedding. While this is conceptually very simple (and gives linear running time), the method itself suffers from the complexity of finding the construction sequence.

References

- [1] , p. 243: "Its implementation in LEDA is slower than LEDA implementations of many other $O(n)$ -time planarity algorithms."
- [2] <http://www.ogdf.net>
- [3] http://www.boost.org/doc/libs/1_40_0/libs/graph/doc/boyer_myrvold.html

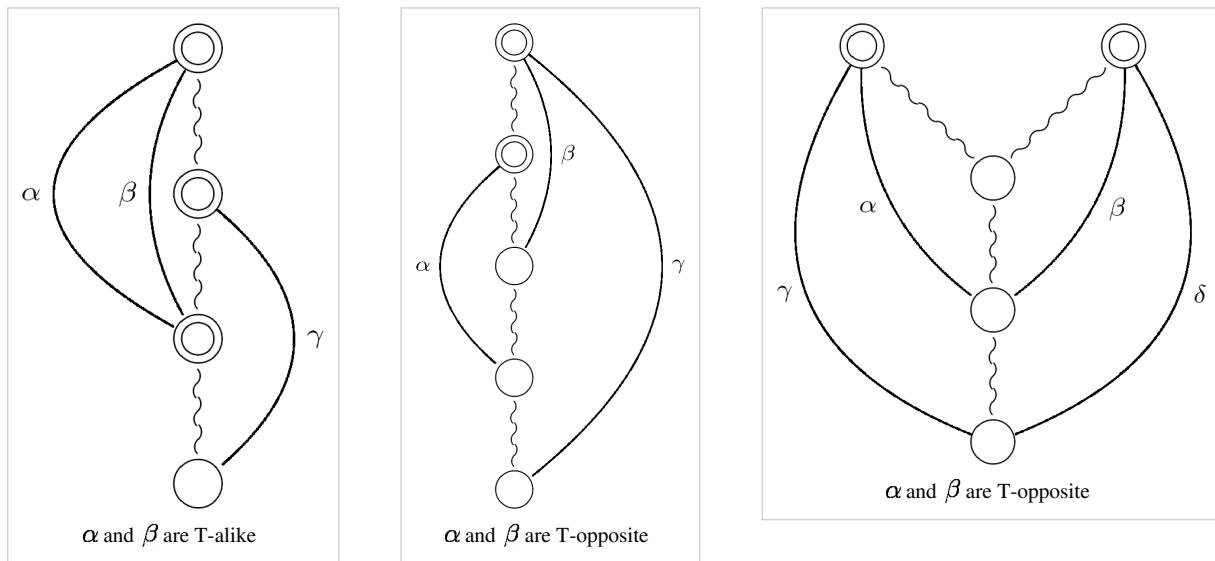
Fraysseix–Rosenstiehl planarity criterion

In graph theory, a branch of mathematics, the **left-right planarity test** or **de Fraysseix–Rosenstiehl planarity criterion** is a characterization of planar graphs based on the properties of the depth-first search trees, published by de Fraysseix and Rosenstiehl (1982, 1985) and used by them with Patrice Ossona Mendez to develop a linear time planarity testing algorithm. In a 2003 experimental comparison of six planarity testing algorithms, this was one of the fastest algorithms tested.

T-alike and T-opposite edges

For any depth-first search of a graph G , the edges encountered when discovering a vertex for the first time define a depth-first search tree T of G . This is a Trémaux tree, meaning that the remaining edges (the **cotree**) each connect a pair of vertices that are related to each other as an ancestor and descendant in T . Three types of patterns can be used to define two relations between pairs of cotree edges, named the **T -alike** and **T -opposite** relations.

In the following figures, simple circle nodes represent vertices, double circle nodes represent subtrees, twisted segments represent tree paths, and curved arcs represent cotree edges. The root of each tree is shown at the bottom of the figure. In the first figure, the edges labeled α and β are T -alike, meaning that at the endpoints nearest the root of the tree, they will both be on the same side of the tree in every planar drawing. In the next two figures, the edges with the same labels are T -opposite, meaning that they will be on different sides of the tree in every planar drawing.



The characterization

Let G be a graph and let T be a Trémaux tree of G . The graph G is planar if and only if there exists a partition of the cotree edges of G into two classes so that any two edges belong to a same class if they are T -alike and any two edges belong to different classes if they are T -opposite.

This characterization immediately leads to an (inefficient) planarity test: determine for all pairs of edges whether they are T -alike or T -opposite, form an auxiliary graph that has a vertex for each connected component of T -alike edges and an edge for each pair of T -opposite edges, and check whether this auxiliary graph is bipartite. Making this algorithm efficient involves finding a subset of the T -alike and T -opposite pairs that is sufficient to carry out this method without determining the relation between all edge pairs in the input graph.

References

Additional reading

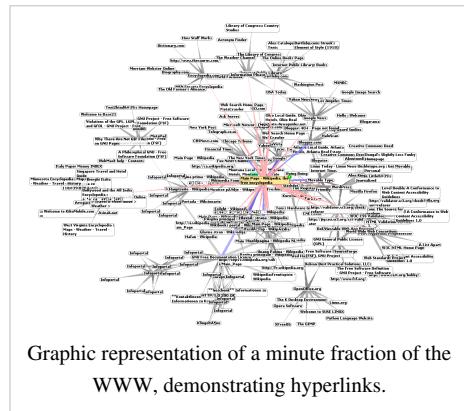
- Kaiser, Daniel (2009), *Implementation und Animation des Links-Rechts-Planaritätstests* (http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-187358/bachelor_kaiser-d.pdf), Bachelorarbeit (in German), University of Konstanz, FB Informatik und Informationswissenschaft

Graph drawing

This article is about the general subject of graph drawing. For the annual research symposium, see International Symposium on Graph Drawing.

Graph drawing is an area of mathematics and computer science combining methods from geometric graph theory and information visualization to derive two-dimensional depictions of graphs arising from applications such as social network analysis, cartography, and bioinformatics.^[1]

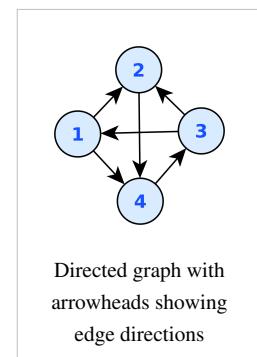
A drawing of a graph or **network diagram** is a pictorial representation of the vertices and edges of a graph. This drawing should not be confused with the graph itself: very different layouts can correspond to the same graph.^[2] In the abstract, all that matters is which pairs vertices are connected by edges. In the concrete, however, the arrangement of these vertices and edges within a drawing affects its understandability, usability, fabrication cost, and aesthetics. The problem gets worse, if the graph changes over time by adding and deleting edges (dynamic graph drawing) and the goal is to preserve the user's mental map.



Graphical conventions

Graphs are frequently drawn as **node-link diagrams** in which the vertices are represented as disks, boxes, or textual labels and the edges are represented as line segments, polylines, or curves in the Euclidean plane.^[1] Node-link diagrams can be traced back to the 13th century work of Ramon Llull, who drew diagrams of this type for complete graphs in order to analyze all pairwise combinations among sets of metaphysical concepts.

In the case of directed graphs, arrowheads form a commonly used graphical convention to show their orientation; however, user studies have shown that other conventions such as tapering provide this information more effectively.^[3] Upward planar drawing uses the convention that every edge is oriented from a lower vertex to a higher vertex, making arrowheads unnecessary.

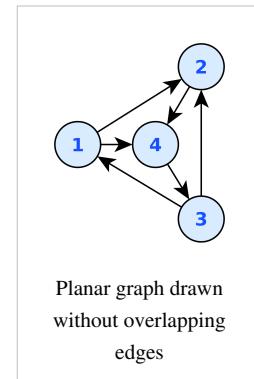


Alternative conventions to node-link diagrams include adjacency representations such as circle packings, in which vertices are represented by disjoint regions in the plane and edges are represented by adjacencies between regions; intersection representations in which vertices are represented by non-disjoint geometric objects and edges are represented by their intersections; visibility representations in which vertices are represented by regions in the plane and edges are represented by regions that have an unobstructed line of sight to each other; confluent drawings, in which edges are represented as smooth curves within mathematical train tracks; and visualizations of the adjacency matrix of the graph.

Quality measures

Many different quality measures have been defined for graph drawings, in an attempt to find objective means of evaluating their aesthetics and usability.^[4] In addition to guiding the choice between different layout methods for the same graph, some layout methods attempt to directly optimize these measures.

- The crossing number of a drawing is the number of pairs of edges that cross each other. If the graph is planar, then it is often convenient to draw it without any edge intersections; that is, in this case, a graph drawing represents a graph embedding. However, nonplanar graphs frequently arise in applications, so graph drawing algorithms must generally allow for edge crossings.^[5]
- The area of a drawing is the size of its smallest bounding box, relative to the closest distance between any two vertices. Drawings with smaller area are generally preferable to those with larger area, because they allow the features of the drawing to be shown at greater size and therefore more legibly. The aspect ratio of the bounding box may also be important.
- Symmetry display is the problem of finding symmetry groups within a given graph, and finding a drawing that displays as much of the symmetry as possible. Some layout methods automatically lead to symmetric drawings; alternatively, some drawing methods start by finding symmetries in the input graph and using them to construct a drawing.^[6]
- It is important that edges have shapes that are as simple as possible, to make it easier for the eye to follow them. In polyline drawings, the complexity of an edge may be measured by its number of bends, and many methods aim to provide drawings with few total bends or few bends per edge. Similarly for spline curves the complexity of an edge may be measured by the number of control points on the edge.
- Several commonly used quality measures concern lengths of edges: it is generally desirable to minimize the total length of the edges as well as the maximum length of any edge. Additionally, it may be preferable for the lengths of edges to be uniform rather than highly varied.
- Angular resolution is a measure of the sharpest angles in a graph drawing. If a graph has vertices with high degree then it necessarily will have small angular resolution, but the angular resolution can be bounded below by a function of the degree.
- The slope number of a graph is the minimum number of distinct edge slopes needed in a drawing with straight line segment edges (allowing crossings). Cubic graphs have slope number at most four, but graphs of degree five may have unbounded slope number; it remains open whether the slope number of degree-4 graphs is bounded.



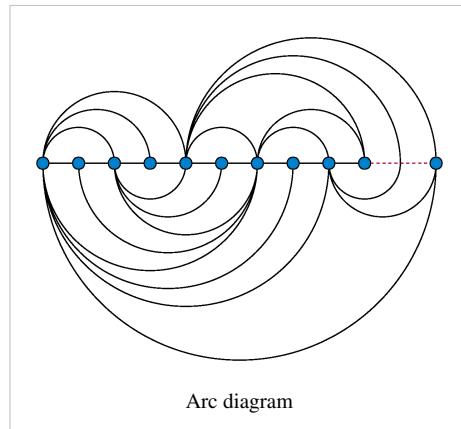
Layout methods

There are many different graph layout strategies:

- In force-based layout systems, the graph drawing software modifies an initial vertex placement by continuously moving the vertices according to a system of forces based on physical metaphors related to systems of springs or molecular mechanics. Typically, these systems combine attractive forces between adjacent vertices with repulsive forces between all pairs of vertices, in order to seek a layout in which edge lengths are small while vertices are well-separated. These systems may perform gradient descent based minimization of an energy function, or they may translate the forces directly into velocities or accelerations for the moving vertices.^[7]
- Spectral layout methods use as coordinates the eigenvectors of a matrix such as the Laplacian derived from the adjacency matrix of the graph.^[8]
- Orthogonal layout methods, which allow the edges of the graph to run horizontally or vertically, parallel to the coordinate axes of the layout. These methods were originally designed for VLSI and PCB layout problems but they have also been adapted for graph drawing. They typically involve a multiphase approach in which an input graph is planarized by replacing crossing points by vertices, a topological embedding of the planarized graph is

found, edge orientations are chosen to minimize bends, vertices are placed consistently with these orientations, and finally a layout compaction stage reduces the area of the drawing.^[9]

- Tree layout algorithms these show a rooted tree-like formation, suitable for trees. Often, in a technique called "balloon layout", the children of each node in the tree are drawn on a circle surrounding the node, with the radii of these circles diminishing at lower levels in the tree so that these circles do not overlap.^[10]
- Layered graph drawing methods (often called Sugiyama-style drawing) are best suited for directed acyclic graphs or graphs that are nearly acyclic, such as the graphs of dependencies between modules or functions in a software system. In these methods, the nodes of the graph are arranged into horizontal layers using methods such as the Coffman–Graham algorithm, in such a way that most edges go downwards from one layer to the next; after this step, the nodes within each layer are arranged in order to minimize crossings.^[11]
- Arc diagrams, a layout style dating back to the 1960s, place vertices on a line; edges may be drawn as semicircles above or below the line, or as smooth curves linked together from multiple semicircles.
- Circular layout methods place the vertices of the graph on a circle, choosing carefully the ordering of the vertices around the circle to reduce crossings and place adjacent vertices close to each other. Edges may be drawn either as chords of the circle or as arcs inside or outside of the circle. In some cases, multiple circles may be used.
- Dominance drawing places vertices in such a way that one vertex is upwards, rightwards, or both of another if and only if it is reachable from the other vertex. In this way, the layout style makes the reachability relation of the graph visually apparent.^[12]



Application-specific graph drawings

Graphs and graph drawings arising in other areas of application include

- Sociograms, drawings of a social network, as often offered by social network analysis software^[13]
- Hasse diagrams, a type of graph drawing specialized to partial orders^[14]
- Dessin d'enfants, a type of graph drawing used in algebraic geometry^[15]
- State diagrams, graphical representations of finite state machines^[16]
- Computer network diagrams, depictions of the nodes and connections in a computer network^[17]
- Flow charts, drawings in which the nodes represent the steps of an algorithm and the edges represent control flow between steps.
- Data flow diagrams, drawings in which the nodes represent the components of an information system and the edges represent the movement of information from one component to another.
- Bioinformatics including phylogenetic trees, protein-protein interaction networks, and metabolic pathways.^[18]

In addition, the placement and routing steps of electronic design automation are similar in many ways to graph drawing, as is the problem of greedy embedding in distributed computing, and the graph drawing literature includes several results borrowed from the EDA literature. However, these problems also differ in several important ways: for instance, in EDA, area minimization and signal length are more important than aesthetics, and the routing problem in EDA may have more than two terminals per net while the analogous problem in graph drawing generally only involves pairs of vertices for each edge.

Software

Software, systems, and providers of systems for drawing graphs include:

- Cytoscape, open-source software for visualizing molecular interaction networks
- Gephi, open-source network analysis and visualization software
- Graphviz, an open-source graph drawing system from AT&T Corporation^[19]
- Mathematica, a general purpose computation tool that includes 2D and 3D graph visualization and graph analysis tools.^{[20][21]}
- Microsoft Automatic Graph Layout, a .NET library (formerly called GLEE) for laying out graphs
- Tom Sawyer Software Tom Sawyer Perspectives is a graphics-based software for building enterprise-class data visualization and social network analysis applications. It is a Software Development Kit (SDK) with a graphics-based design and preview environment.
- Tulip (software)^[22]
- yEd, a graph editor with graph layout functionality^[23]
- PGF/TikZ 3.0 with the `graphdrawing` package (requires LuaTeX).^[24]

References

Footnotes

- [1] , pp. vii–viii; , Section 1.1, "Typical Application Areas".
- [2] , p. 6.
- [3] ; .
- [4] , Section 2.1.2, Aesthetics, pp. 14–16; .
- [5] , p 14.
- [6] , p. 16.
- [7] , Section 2.7, "The Force-Directed Approach", pp. 29–30, and Chapter 10, "Force-Directed Methods", pp. 303–326.
- [8] ; .
- [9] , Chapter 5, "Flow and Orthogonal Drawings", pp. 137–170; .
- [10] , Section 2.2, "Traditional Layout – An Overview".
- [11] ; ; , Chapter 9, "Layered Drawings of Digraphs", pp. 265–302.
- [12] , Section 4.7, "Dominance Drawings", pp. 112–127.
- [13] ; .
- [14] , pp. 15–16, and Chapter 6, "Flow and Upward Planarity", pp. 171–214; .
- [15] Zapponi (2003).
- [16] Anderson & Head (2006).
- [17] Di Battista & Rimondini (2014).
- [18] Bachmaier, Brandes & Schreiber (2014).
- [19] "Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools", by John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull, in .
- [20] GraphPlot (<http://reference.wolfram.com/mathematica/ref/GraphPlot.html>) Mathematica documentation
- [21] Graph drawing tutorial (<http://reference.wolfram.com/mathematica/tutorial/GraphDrawingIntroduction.html>)
- [22] "Tulip – A Huge Graph Visualization Framework", by David Auber, in .
- [23] "yFiles – Visualization and Automatic Layout of Graphs", by Roland Wiese, Markus Eiglsperger, and Michael Kaufmann, in .
- [24] ; see also the older GD 2012 presentation (<http://www.tcs.uni-luebeck.de/downloads/mitarbeiter/tantau/2012-gd-presentation.pdf>)

General references

- Di Battista, Giuseppe; Eades, Peter; Tamassia, Roberto; Tollis, Ioannis G. (1994), "Algorithms for Drawing Graphs: an Annotated Bibliography" (<http://www.cs.brown.edu/people/rt/gd.html>), *Computational Geometry: Theory and Applications* 4: 235–282, doi: 10.1016/0925-7721(94)00014-x ([http://dx.doi.org/10.1016/0925-7721\(94\)00014-x](http://dx.doi.org/10.1016/0925-7721(94)00014-x)).
- Di Battista, Giuseppe; Eades, Peter; Tamassia, Roberto; Tollis, Ioannis G. (1998), *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, ISBN 978-0-13-301615-4.

- Herman, Ivan; Melançon, Guy; Marshall, M. Scott (2000), "Graph Visualization and Navigation in Information Visualization: A Survey" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.8892>), *IEEE Transactions on Visualization and Computer Graphics* **6** (1): 24–43, doi: 10.1109/2945.841119 (<http://dx.doi.org/10.1109/2945.841119>).
- Jünger, Michael; Mutzel, Petra (2004), *Graph Drawing Software*, Springer-Verlag, ISBN 978-3-540-00881-1.
- Kaufmann, Michael; Wagner, Dorothea, eds. (2001), *Drawing Graphs: Methods and Models*, Lecture Notes in Computer Science **2025**, Springer-Verlag, doi: 10.1007/3-540-44969-8 (<http://dx.doi.org/10.1007/3-540-44969-8>).
- Tamassia, Roberto, ed. (2014), *Handbook of Graph Drawing and Visualization* (<http://cs.brown.edu/~rt/gdhandbook/>), CRC Press.

Specialized subtopics

- Anderson, James Andrew; Head, Thomas J. (2006), *Automata Theory with Modern Applications* (<http://books.google.com/books?id=ikS8BLdLDxIC&pg=PA38>), Cambridge University Press, pp. 38–41, ISBN 978-0-521-84887-9.
- Bachmaier, Christian; Brandes, Ulrik; Schreiber, Falk (2014), "Biological Networks", in Tamassia, Roberto, *Handbook of Graph Drawing and Visualization*, CRC Press, pp. 621–651.
- Bastert, Oliver; Matuszewski, Christian (2001), "Layered drawings of digraphs", in Kaufmann, Michael; Wagner, Dorothea, *Drawing Graphs: Methods and Models*, Lecture Notes in Computer Science **2025**, Springer-Verlag, pp. 87–120, doi: 10.1007/3-540-44969-8_5 (http://dx.doi.org/10.1007/3-540-44969-8_5).
- Beckman, Brian (1994), *Theory of Spectral Graph Layout* (<http://research.microsoft.com/apps/pubs/default.aspx?id=69611>), Tech. Report MSR-TR-94-04, Microsoft Research.
- Brandes, Ulrik; Freeman, Linton C.; Wagner, Dorothea (2014), "Social Networks", in Tamassia, Roberto, *Handbook of Graph Drawing and Visualization*, CRC Press, pp. 805–839.
- Di Battista, Giuseppe; RIMONDINI, Massimo (2014), "Computer Networks", in Tamassia, Roberto, *Handbook of Graph Drawing and Visualization*, CRC Press, pp. 763–803.
- Doğrusöz, Uğur; Madden, Brendan; Madden, Patrick (1997), "Circular layout in the Graph Layout toolkit", in North, Stephen, *Symposium on Graph Drawing, GD '96 Berkeley, California, USA, September 18–20, 1996, Proceedings*, Lecture Notes in Computer Science **1190**, Springer-Verlag, pp. 92–100, doi: 10.1007/3-540-62495-3_40 (http://dx.doi.org/10.1007/3-540-62495-3_40).
- Eiglsperger, Markus; Fekete, Sándor; Klau, Gunnar (2001), "Orthogonal graph drawing", in Kaufmann, Michael; Wagner, Dorothea, *Drawing Graphs*, Lecture Notes in Computer Science **2025**, Springer Berlin / Heidelberg, pp. 121–171, doi: 10.1007/3-540-44969-8_6 (http://dx.doi.org/10.1007/3-540-44969-8_6).
- Freese, Ralph (2004), "Automated lattice drawing" (<http://www.math.hawaii.edu/~ralph/Preprints/latdrawing.pdf>), in Eklund, Peter, *Concept Lattices: Second International Conference on Formal Concept Analysis, ICFCA 2004, Sydney, Australia, February 23–26, 2004, Proceedings*, Lecture Notes in Computer Science **2961**, Springer-Verlag, pp. 589–590, doi: 10.1007/978-3-540-24651-0_12 (http://dx.doi.org/10.1007/978-3-540-24651-0_12).
- Garg, Ashim; Tamassia, Roberto (1995), "Upward planarity testing", *Order* **12** (2): 109–133, doi: 10.1007/BF01108622 (<http://dx.doi.org/10.1007/BF01108622>), MR 1354797 (<http://www.ams.org/mathscinet-getitem?mr=1354797>).
- Holten, Danny; Isenberg, Petra; van Wijk, Jarke J.; Fekete, Jean-Daniel (2011), "An extended evaluation of the readability of tapered, animated, and textured directed-edge representations in node-link graphs" (http://www.lri.fr/~isenberg/publications/papers/Holten_2011_AEP.pdf), *IEEE Pacific Visualization Symposium (PacificVis 2011)*, pp. 195–202, doi: 10.1109/PACIFICVIS.2011.5742390 (<http://dx.doi.org/10.1109/PACIFICVIS.2011.5742390>).
- Holten, Danny; van Wijk, Jarke J. (2009), "A user study on visualizing directed edges in graphs" (http://www.win.tue.nl/~dholten/papers/directed_edges_chi.pdf), *Proceedings of the 27th International Conference on*

- Human Factors in Computing Systems (CHI '09)*, pp. 2299–2308, doi: 10.1145/1518701.1519054 (<http://dx.doi.org/10.1145/1518701.1519054>).
- Koren, Yehuda (2005), "Drawing graphs by eigenvectors: theory and practice" (https://akpublic.research.att.com/areas/visualization/papers_videos/pdf/DBLP-journals-camwa-Koren05.pdf), *Computers & Mathematics with Applications* **49** (11-12): 1867–1888, doi: 10.1016/j.camwa.2004.08.015 (<http://dx.doi.org/10.1016/j.camwa.2004.08.015>), MR 2154691 (<http://www.ams.org/mathscinet-getitem?mr=2154691>).
 - Madden, Brendan; Madden, Patrick; Powers, Steve; Himsolt, Michael (1996), "Portable graph layout and editing", in Brandenburg, Franz J., *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995, Proceedings*, Lecture Notes in Computer Science **1027**, Springer-Verlag, pp. 385–395, doi: 10.1007/BFb0021822 (<http://dx.doi.org/10.1007/BFb0021822>).
 - Misue, K.; Eades, P.; Lai, W.; Sugiyama, K. (1995), "Layout Adjustment and the Mental Map", *Journal of Visual Languages and Computing* **6** (2): 183–210, doi: 10.1006/jvlc.1995.1010 (<http://dx.doi.org/10.1006/jvlc.1995.1010>).
 - Nachmanson, Lev; Robertson, George; Lee, Bongshin (2008), "Drawing Graphs with GLEE" (<ftp://ftp.research.microsoft.com/pub/TR/TR-2007-72.pdf>), in Hong, Seok-Hee; Nishizeki, Takao; Quan, Wu, *Graph Drawing, 15th International Symposium, GD 2007, Sydney, Australia, September 24–26, 2007, Revised Papers*, Lecture Notes in Computer Science **4875**, Springer-Verlag, pp. 389–394, doi: 10.1007/978-3-540-77537-9_38 (http://dx.doi.org/10.1007/978-3-540-77537-9_38).
 - Pach, János; Sharir, Micha (2009), "5.5 Angular resolution and slopes", *Combinatorial Geometry and Its Algorithmic Applications: The Alcalá Lectures*, Mathematical Surveys and Monographs **152**, American Mathematical Society, pp. 126–127.
 - Purchase, H. C.; Cohen, R. F.; James, M. I. (1997), "An experimental study of the basis for graph drawing algorithms" (<https://secure.cs.uvic.ca/twiki/pub/Research/Chisel/ComputationalAestheticsProject/Vol2Nbr4.pdf>), *Journal of Experimental Algorithms* **2**, Article 4, doi: 10.1145/264216.264222 (<http://dx.doi.org/10.1145/264216.264222>).
 - Saaty, Thomas L. (1964), "The minimum number of intersections in complete graphs", *Proc. Natl. Acad. Sci. U.S.A.* **52**: 688–690, doi: 10.1073/pnas.52.3.688 (<http://dx.doi.org/10.1073/pnas.52.3.688>).
 - Scott, John (2000), "Sociograms and Graph Theory" (http://books.google.com/books?id=Ww3_bKcz6kgC&pg=PA), *Social network analysis: a handbook* (2nd ed.), Sage, pp. 64–69, ISBN 978-0-7619-6339-4.
 - Sugiyama, Kozo; Tagawa, Shôjirô; Toda, Mitsuhiko (1981), "Methods for visual understanding of hierarchical system structures", *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11 (2): 109–125, doi: 10.1109/TSMC.1981.4308636 (<http://dx.doi.org/10.1109/TSMC.1981.4308636>), MR 0611436 (<http://www.ams.org/mathscinet-getitem?mr=0611436>).
 - Tantau, Till (2013), "Graph Drawing in TikZ", *Journal of Graph Algorithms and Applications* **17** (4): 495–513, doi: 10.7155/jgaa.00301 (<http://dx.doi.org/10.7155/jgaa.00301>).
 - Zapponi, Leonardo (August 2003), "What is a Dessin d'Enfant" (<http://www.ams.org/notices/200307/what-is-pdf>), *Notices of the American Mathematical Society* **50**: 788–789.

External links

- GraphX library for .NET (<http://graphx.codeplex.com>): open-source WPF library for graph calculation and visualization. Supports many layout and edge routing algorithms.
- Graph drawing e-print archive (<http://gdea.informatik.uni-koeln.de/>): including information on papers from all Graph Drawing symposia.
- Graph drawing (http://www.dmoz.org/Science/Math/Combinatorics/Software/Graph_Drawing) at DMOZ for many additional links related to graph drawing.

Force-directed graph drawing

Force-directed graph drawing algorithms are a class of algorithms for drawing graphs in an aesthetically pleasing way. Their purpose is to position the nodes of a graph in two-dimensional or three-dimensional space so that all the edges are of more or less equal length and there are as few crossing edges as possible, by assigning forces among the set of edges and the set of nodes, based on their relative positions, and then using these forces either to simulate the motion of the edges and nodes or to minimize their energy.

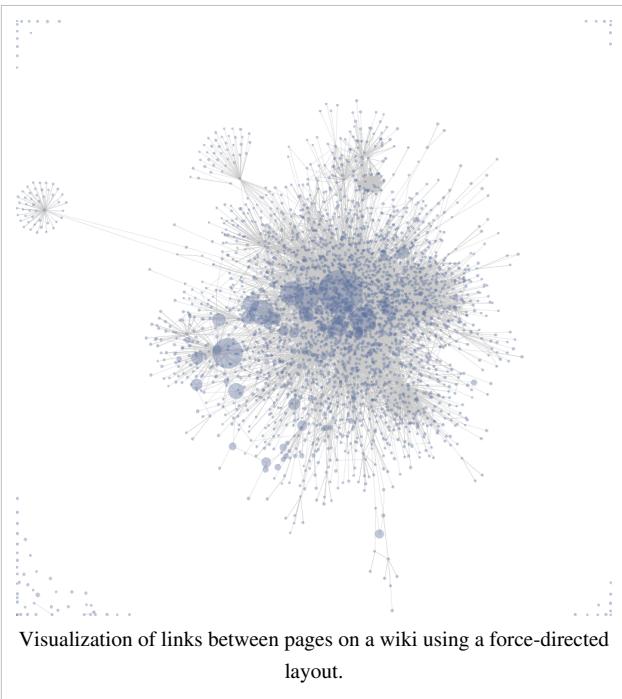
While graph drawing can be a difficult problem, force-directed algorithms, being physical simulations, usually require no special knowledge about graph theory such as planarity.

Forces

Force-directed graph drawing algorithms assign forces among the set of edges and the set of nodes of a graph drawing. Typically, spring-like attractive forces based on Hooke's law are used to attract pairs of endpoints of the graph's edges towards each other, while simultaneously repulsive forces like those of electrically charged particles based on Coulomb's law are used to separate all pairs of nodes. In equilibrium states for this system of forces, the edges tend to have uniform length (because of the spring forces), and nodes that are not connected by an edge tend to be drawn further apart (because of the electrical repulsion). Edge attraction and vertex repulsion forces may be defined using functions that are not based on the physical behavior of springs and particles; for instance, some force-directed systems use springs whose attractive force is logarithmic rather than linear.

An alternative model considers a spring-like force for every pair of nodes (i, j) where the ideal length δ_{ij} of each spring is proportional to the graph-theoretic distance between nodes i and j , without using a separate repulsive force. Minimizing the difference (usually the squared difference) between Euclidean and ideal distances between nodes is then equivalent to a metric multidimensional scaling problem.

A force-directed graph can involve forces other than mechanical springs and electrical repulsion. A force analogous to gravity may be used to pull vertices towards a fixed point of the drawing space; this may be used to pull together different connected components of a disconnected graph, which would otherwise tend to fly apart from each other because of the repulsive forces, and to draw nodes with greater centrality to more central positions in the drawing; it may also affect the vertex spacing within a single component. Analogues of magnetic fields may be used for directed



Visualization of links between pages on a wiki using a force-directed layout.

graphs. Repulsive forces may be placed on edges as well as on nodes in order to avoid overlap or near-overlap in the final drawing. In drawings with curved edges such as circular arcs or spline curves, forces may also be placed on the control points of these curves, for instance to improve their angular resolution.

Methods

Once the forces on the nodes and edges of a graph have been defined, the behavior of the entire graph under these sources may then be simulated as if it were a physical system. In such a simulation, the forces are applied to the nodes, pulling them closer together or pushing them further apart. This is repeated iteratively until the system comes to a mechanical equilibrium state; i.e., their relative positions do not change anymore from one iteration to the next. The positions of the nodes in this equilibrium are used to generate a drawing of the graph.

For forces defined from springs whose ideal length is proportional to the graph-theoretic distance, stress majorization gives a very well-behaved (i.e., monotonically convergent) and mathematically elegant way to minimise these differences and, hence, find a good layout for the graph.

It is also possible to employ mechanisms that search more directly for energy minima, either instead of or in conjunction with physical simulation. Such mechanisms, which are examples of general global optimization methods, include simulated annealing and genetic algorithms.

Advantages

The following are among the most important advantages of force-directed algorithms:

Good-quality results

At least for graphs of medium size (up to 50–100 vertices), the results obtained have usually very good results based on the following criteria: uniform edge length, uniform vertex distribution and showing symmetry. This last criterion is among the most important ones and is hard to achieve with any other type of algorithm.

Flexibility

Force-directed algorithms can be easily adapted and extended to fulfill additional aesthetic criteria. This makes them the most versatile class of graph drawing algorithms. Examples of existing extensions include the ones for directed graphs, 3D graph drawing, cluster graph drawing, constrained graph drawing, and dynamic graph drawing.

Intuitive

Since they are based on physical analogies of common objects, like springs, the behavior of the algorithms is relatively easy to predict and understand. This is not the case with other types of graph-drawing algorithms.

Simplicity

Typical force-directed algorithms are simple and can be implemented in a few lines of code. Other classes of graph-drawing algorithms, like the ones for orthogonal layouts, are usually much more involved.

Interactivity

Another advantage of this class of algorithm is the interactive aspect. By drawing the intermediate stages of the graph, the user can follow how the graph evolves, seeing it unfold from a tangled mess into a good-looking configuration. In some interactive graph drawing tools, the user can pull one or more nodes out of their equilibrium state and watch them migrate back into position. This makes them a preferred choice for dynamic and online graph-drawing systems.

Strong theoretical foundations

While simple *ad-hoc* force-directed algorithms often appear in the literature and in practice (because they are relatively easy to understand), more reasoned approaches are starting to gain traction. Statisticians have been

solving similar problems in multidimensional scaling (MDS) since the 1930s, and physicists also have a long history of working with related n-body problems - so extremely mature approaches exist. As an example, the stress majorization approach to metric MDS can be applied to graph drawing as described above. This has been proven to converge monotonically. Monotonic convergence, the property that the algorithm will at each iteration decrease the stress or cost of the layout, is important because it guarantees that the layout will eventually reach a local minimum and stop. Damping schedules cause the algorithm to stop, but cannot guarantee that a true local minimum is reached.

Disadvantages

The main disadvantages of force-directed algorithms include the following:

High running time

The typical force-directed algorithms are in general *considered* to have a running time equivalent to $O(n^3)$, where n is the number of nodes of the input graph. This is because the number of iterations is estimated to be $O(n)$, and in every iteration, all pairs of nodes need to be visited and their mutual repulsive forces computed. This is related to the N-body problem in physics. However, since repulsive forces are local in nature the graph can be partitioned such that only neighboring vertices are considered. Common techniques used by algorithms for determining the layout of large graphs include high-dimensional embedding, multi-layer drawing and other methods related to N-body simulation. For example, the Barnes–Hut simulation-based method FADE can improve running time to $n \cdot \log(n)$ per iteration. As a rough guide, in a few seconds one can expect to draw at most 1,000 nodes with a standard n^2 per iteration technique, and 100,000 with a $n \cdot \log(n)$ per iteration technique. Force-directed algorithm, when combined with a multilevel approach, can draw graphs of millions of nodes.

Poor local minima

It is easy to see that force-directed algorithms produce a graph with minimal energy, in particular one whose total energy is only a local minimum. The local minimum found can be, in many cases, considerably worse than a global minimum, which translates into a low-quality drawing. For many algorithms, especially the ones that allow only *down-hill* moves of the vertices, the final result can be strongly influenced by the initial layout, that in most cases is randomly generated. The problem of poor local minima becomes more important as the number of vertices of the graph increases. A combined application of different algorithms is helpful to solve this problem. For example, using the Kamada–Kawai algorithm to quickly generate a reasonable initial layout and then the Fruchterman–Reingold algorithm to improve the placement of neighbouring nodes. Another technique to achieve a global minimum is to use a multilevel approach.

History

Force-directed methods in graph drawing date back to the work of Tutte (1963), who showed that polyhedral graphs may be drawn in the plane with all faces convex by fixing the vertices of the outer face of a planar embedding of the graph into convex position, placing a spring-like attractive force on each edge, and letting the system settle into an equilibrium. Because of the simple nature of the forces in this case, the system cannot get stuck in local minima, but rather converges to a unique global optimum configuration. Because of this work, embeddings of planar graphs with convex faces are sometimes called Tutte embeddings.

The combination of attractive forces on adjacent vertices, and repulsive forces on all vertices, was first used by Eades (1984); additional pioneering work on this type of force-directed layout was done by Fruchterman & Reingold (1991). The idea of using only spring forces between all pairs of vertices, with ideal spring lengths equal to the vertices' graph-theoretic distance, is from Kamada & Kawai (1989).

References

Further reading

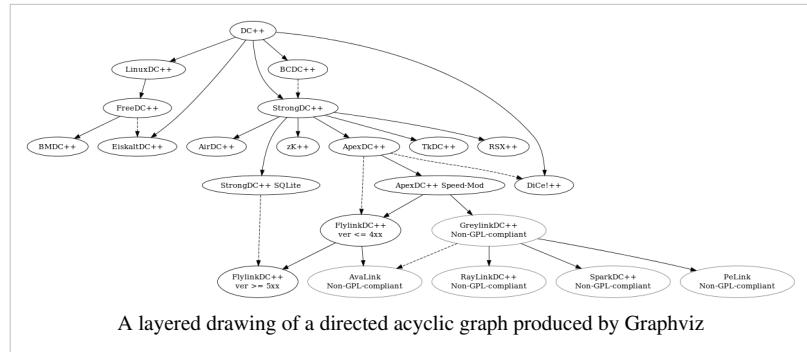
- di Battista, Giuseppe; Peter Eades, Roberto Tamassia, Ioannis G. Tollis (1999), *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, ISBN 978-0-13-301615-4
- Kaufmann, Michael; Wagner, Dorothea, eds. (2001), *Drawing graphs: methods and models*, Lecture Notes in Computer Science 2025, Springer, doi: 10.1007/3-540-44969-8 (<http://dx.doi.org/10.1007/3-540-44969-8>), ISBN 978-3-540-42062-0

External links

- Video of Spring Algorithm (<http://www.cs.usyd.edu.au/~aquigley/avi/spring.avi>)
- Live visualisation in flash + source code and description (<http://blog.ivank.net/force-based-graph-drawing-in-as3.html>)
- Daniel Tunkelang's dissertation (<http://reports-archive.adm.cs.cmu.edu/anon/1998/abstracts/98-189.html>) (with source code (<http://www.cs.cmu.edu/~quixote/JiggleSource.zip>) and demonstration applet (<http://www.cs.cmu.edu/~quixote/gd.html>)) on force-directed graph layout
- Hyperassociative Map Algorithm (http://wiki.syncleus.com/index.php/DANN:Hyperassociative_Map)
- Interactive and real-time force-directed graphing algorithms used in an online database modeling tool (<http://www.anchormodeling.com/modeler>)

Layered graph drawing

Layered graph drawing or **hierarchical graph drawing** is a type of graph drawing in which the vertices of a directed graph are drawn in horizontal rows or layers with the edges generally directed downwards. It is also known as **Sugiyama-style graph drawing** after Kozo Sugiyama, who first developed this drawing style.



The ideal form for a layered drawing would be an upward planar drawing, in which all edges are oriented in a consistent direction and no pairs of edges cross. However, graphs often contain cycles, minimizing the number of inconsistently-oriented edges is NP-hard, and minimizing the number of crossings is also NP-hard, so layered graph drawing systems typically apply a sequence of heuristics that reduce these types of flaws in the drawing without guaranteeing to find a drawing with the minimum number of flaws.

Layout algorithm

The construction of a layered graph drawing proceeds in a sequences of steps:

- If the input graph is not already a directed acyclic graph, a set of edges is identified the reversal of which will make it acyclic. Finding the smallest possible set of edges is the NP-complete feedback arc set problem, so often greedy heuristics are used here in place of exact optimization algorithms. The exact solution to this problem can be formulated using integer programming. Alternatively, if the number of reversed edges is very small, these edges can be found by a fixed-parameter-tractable algorithm.

- The vertices of the directed acyclic graph resulting from the first step are assigned to layers, such that each edge goes from a higher layer to a lower layer. The goals of this stage are to simultaneously produce a small number of layers, few edges that span large numbers of layers, and a balanced assignment of vertices to layers. For instance, by Mirsky's theorem, assigning vertices by layers according to the length of the longest path starting from each vertex produces an assignment with the minimum possible number of layers. The Coffman–Graham algorithm may be used to find a layering with a predetermined limit on the number of vertices per layer and approximately minimizing the number of layers subject to that constraint. Minimizing the width of the widest layer is NP-hard but may be solved by branch-and-cut or approximated heuristically. Alternatively, the problem of minimizing the total number of layers spanned by the edges (without any limits on the number of vertices per layer) may be solved using linear programming. Integer programming procedures, although more time-consuming, may be used to combine edge length minimization with limits on the number of vertices per level.
- Edges that span multiple layers are replaced by paths of dummy vertices so that, after this step, each edge in the expanded graph connects two vertices on adjacent layers of the drawing.
- As an optional step, a layer of edge concentrator vertices (or confluent junctions) may be imposed between two existing vertex layers, reducing the edge density by replacing complete bipartite subgraphs by stars through these edge concentrators.
- The vertices within each layer are permuted in an attempt to reduce the number of crossings among the edges connecting it to the previous layer. Finding the minimum number of crossings or finding a maximum crossing-free set of edges is NP-complete, even when ordering a single layer at a time in this way, so again it is typical to resort to heuristics, such as placing each vertex at a position determined by finding the average or median of the positions of its neighbors on the previous level and then swapping adjacent pairs as long as that improves the number of crossings. Alternatively, the ordering of the vertices in one layer at a time may be chosen using an algorithm that is fixed-parameter tractable in the number of crossings between it and the previous layer.
- Each vertex is assigned a coordinate within its layer, consistent with the permutation calculated in the previous step. Considerations in this step include placing dummy nodes on a line between their two neighbors to prevent unnecessary bends, and placing each vertex in a centered position with respect to its neighbors. Sugiyama's original work proposed a quadratic programming formulation of this step; a later method of Brandes and Köpf takes linear time and guarantees at most two bends per edge.
- The edges reversed in the first step of the algorithm are returned to their original orientations, the dummy vertices are removed from the graph and the vertices and edges are drawn. To avoid intersections between vertices and edges, edges that span multiple layers of the drawing may be drawn as polygonal chains or spline curves passing through each of the positions assigned to the dummy vertices along the edge.

Implementations

In its simplest form, layered graph drawing algorithms may require $O(mn)$ time in graphs with n vertices and m edges, because of the large number of dummy vertices that may be created. However, for some variants of the algorithm, it is possible to simulate the effect of the dummy vertices without actually constructing them explicitly, leading to a near-linear time implementation.

The "dot" tool in Graphviz produces layered drawings. A layered graph drawing algorithm is also included in Microsoft Automatic Graph Layout and in Tulip.

Variations

Although typically drawn with vertices in rows and edges proceeding from top to bottom, layered graph drawing algorithms may instead be drawn with vertices in columns and edges proceeding from left to right. The same algorithmic framework has also been applied to radial layouts in which the graphs are arranged in concentric circles around some starting node and to three-dimensional layered drawings of graphs.

In layered graph drawings with many long edges, edge clutter may be reduced by grouping sets of edges into bundles and routing them together through the same set of dummy vertices. Similarly, for drawings with many edges crossing between pairs of consecutive layers, the edges in maximal bipartite subgraphs may be grouped into confluent bundles.

Drawings in which the vertices are arranged in layers may be constructed by algorithms that do not follow Sugiyama's framework. For instance, it is possible to tell whether an undirected graph has a drawing with at most k crossings, using h layers, in an amount of time that is polynomial for any fixed choice of k and h , using the fact that the graphs that have drawings of this type have bounded pathwidth.

For layered drawings of concept lattices, a hybrid approach combining Sugiyama's framework with additive methods (in which each vertex represents a set and the position of the vertex is a sum of vectors representing elements in the set) may be used. In this hybrid approach, the vertex permutation and coordinate assignment phases of the algorithm are replaced by a single phase in which the horizontal position of each vertex is chosen as a sum of scalars representing the elements for that vertex. Layered graph drawing methods have also been used to provide an initial placement for force-directed graph drawing algorithms.

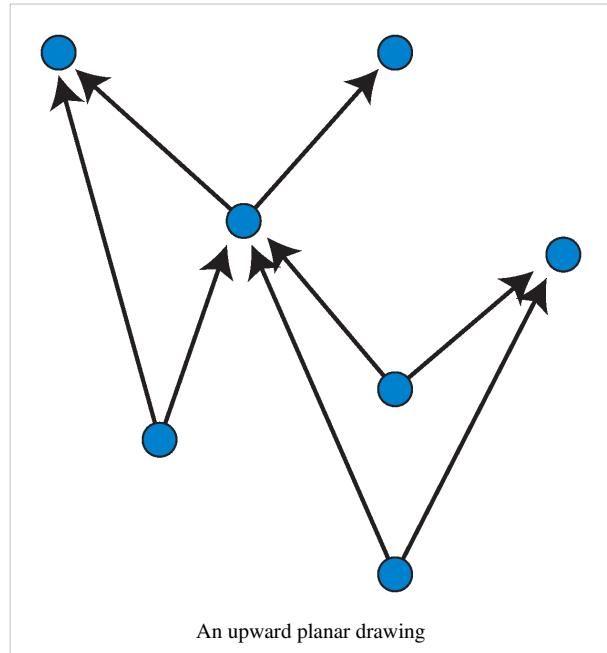
References

Upward planar drawing

In graph drawing, an **upward planar drawing** of a directed acyclic graph is an embedding of the graph into the Euclidean plane, in which the edges are represented as non-crossing monotonic upwards curves. That is, the curve representing each edge should have the property that every horizontal line intersects it in at most one point, and no two edges may intersect except at a shared endpoint.^[1] In this sense, it is the ideal case for layered graph drawing, a style of graph drawing in which edges are monotonic curves that may cross, but in which crossings are to be minimized.

Characterizations

A directed acyclic graph must be planar in order to have an upward planar drawing, but not every planar acyclic graph has such a drawing. Among the planar directed acyclic graphs with a single



source (vertex with no

incoming edges) and sink (vertex with no outgoing edges), the graphs with upward planar drawings are the *st*-planar graphs, planar graphs in which the source and sink both belong to the same face of at least one of the planar embeddings of the graph. More generally, a graph G has an upward planar drawing if and only if it is directed and acyclic, and is a subgraph of an *st*-planar graph on the same vertex set.^[2]

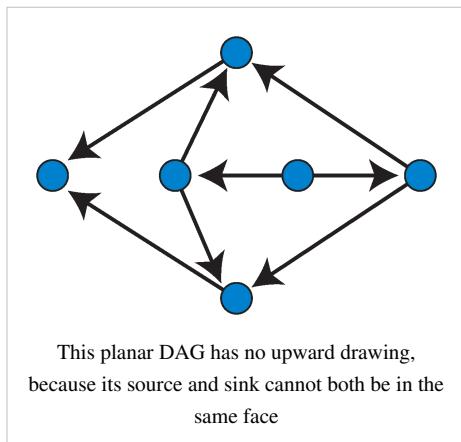
In an upward embedding, the sets of incoming and outgoing edges incident to each vertex are contiguous in the cyclic ordering of the edges at the vertex. A planar embedding of a given directed acyclic graph is said to be *bimodal* when it has this property. Additionally, the angle between two consecutive edges with the same orientation at a given vertex may be labeled as *small* if it is less than π , or *large* if it is greater than π . Each source or sink must have exactly one large angle, and each vertex that is neither a source nor a sink must have none. Additionally, each internal face of the drawing must have two more small angles than large ones, and the external face must have two more large angles than small ones. A *consistent assignment* is a labeling of the angles that satisfies these properties; every upward embedding has a consistent assignment. Conversely, every directed acyclic graph that has a bimodal planar embedding with a consistent assignment has an upward planar drawing, that can be constructed from it in linear time.^[3]

Another characterization is possible for graphs with a single source. In this case an upward planar embedding must have the source on the outer face, and every undirected cycle of the graph must have at least one vertex at which both cycle edges are incoming (for instance, the vertex with the highest placement in the drawing). Conversely, if an embedding has both of these properties, then it is equivalent to an upward embedding.^[4]

Computational complexity

Several special cases of upward planarity testing are known to be possible in polynomial time:

- Testing whether a graph is *st*-planar may be accomplished in linear time by adding an edge from s to t and testing whether the remaining graph is planar. Along the same lines, it is possible to construct an upward planar drawing (when it exists) of a directed acyclic graph with a single source and sink, in linear time.^[5]
- Testing whether a directed graph with a fixed planar embedding can be drawn upward planar, with an embedding consistent with the given one, can be accomplished by checking that the embedding is bipolar and modeling the consistent assignment problem as a network flow problem. The running time is linear in the size of the input graph, and polynomial in its number of sources and sinks.^[6]
- Because oriented polyhedral graphs have a unique planar embedding, the existence of an upward planar drawing for these graphs may be tested in polynomial time.^[7]
- Testing whether an outerplanar directed acyclic graph has an upward planar drawing is also polynomial.^[8]
- Every series-parallel graph, oriented consistently with the series-parallel structure, is upward planar. An upward planar drawing can be constructed directly from the series-parallel decomposition of the graph.^[9] More generally, arbitrary orientations of undirected series-parallel graphs may be tested for upward planarity in polynomial time.
- Every oriented tree is upward planar.
- Every bipartite planar graph, with its edges oriented consistently from one side of the bipartition to the other, is upward planar
- A more complicated polynomial time algorithm is known for testing upward planarity of graphs that have a single source, but multiple sinks, or vice versa.^[10]
- Testing upward planarity can be performed in polynomial time when there are a constant number of triconnected components and cut vertices, and is fixed-parameter tractable in these two numbers.^[11] It is also fixed-parameter



tractable in the cyclomatic number of the input graph.

- If the y -coordinates of all vertices are fixed, then a choice of x -coordinates that makes the drawing upward planar can be found in polynomial time.

However, it is NP-complete to determine whether a planar directed acyclic graph with multiple sources and sinks has an upward planar drawing.^[12]

Straight-line drawing and area requirements

Fáry's theorem states that every planar graph has a drawing in which its edges are represented by straight line segments, and the same is true of upward planar drawing: every upward planar graph has a straight upward planar drawing.^[13] A straight-line upward drawing of a transitively reduced st -planar graph may be obtained by the technique of dominance drawing, with all vertices having integer coordinates within an $n \times n$ grid.^[14] However, certain other upward planar graphs may require exponential area in all of their straight-line upward planar drawings. If a choice of embedding is fixed, even oriented series parallel graphs and oriented trees may require exponential area.^[15]

Hasse diagrams

Upward planar drawings are particularly important for Hasse diagrams of partially ordered sets, as these diagrams are typically required to be drawn upwardly. In graph-theoretic terms, these correspond to the transitively reduced directed acyclic graphs; such a graph can be formed from the covering relation of a partial order, and the partial order itself forms the reachability relation in the graph. If a partially ordered set has one minimal element, has one maximal element, and has an upward planar drawing, then it must necessarily form a lattice, a set in which every pair of elements has a unique greatest lower bound and a unique least upper bound.^[16] The Hasse diagram of a lattice is planar if and only if its order dimension is at most two.^[17] However, some partial orders of dimension two and with one minimal and maximal element do not have an upward planar drawing (take the order defined by the transitive closure of $a < b, a < c, b < d, b < e, c < d, c < e, d < f, e < f$).

References

Footnotes

- [1] ; .
- [2] , pp. 111–112; , 6.1 "Inclusion in a Planar st -Graph", pp. 172–179; ; .
- [3] , pp. 112–115; , 6.2 "Angles in Upward Drawings", pp. 180–188; ; .
- [4] , p. 115; , 6.7.2 "Forbidden Cycles for Single-Source Digraphs", pp. 209–210; .
- [5] , p. 119; , p. 179.
- [6] , pp. 119–121; , 6.3 "Upward Planarity Testing of Embedded Digraphs", pp. 188–192; ; .
- [7] , pp. 191–192; ; .
- [8] , pp. 125–126; , 6.7.1 "Outerplanar Digraph", p. 209; .
- [9] , 6.7.4 "Some Classes of Upward Planar Digraphs", p. 212.
- [10] , pp. 122–125; , 6.5 "Optimal Upward Planarity Testing of Single-Source Digraphs", pp. 195–200; ; .
- [11] ; .
- [12] , pp. 126–132; , 6.6 "Upward Planarity Testing is NP-complete", pp. 201–209; .
- [13] ; .
- [14] , 4.7 "Dominance Drawings", pp. 112–127; .
- [15] ; ; .
- [16] , 6.7.3 "Forbidden Structures for Lattices", pp. 210–212; .
- [17] , pp. 118; .

Surveys and textbooks

- Di Battista, Giuseppe; Eades, Peter; Tamassia, Roberto; Tollis, Ioannis G. (1998), "Flow and Upward Planarity", *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, pp. 171–213,

ISBN 978-0-13-301615-4.

- Di Battista, Giuseppe; Frati, Fabrizio (2012), "Drawing trees, outerplanar graphs, series-parallel graphs, and planar graphs in small area", *Thirty Essays on Geometric Graph Theory*, Algorithms and combinatorics **29**, Springer, pp. 121–165, doi: 10.1007/978-1-4614-0110-0_9 (http://dx.doi.org/10.1007/978-1-4614-0110-0_9), ISBN 9781461401100. Section 5, "Upward Drawings", pp. 149–151.
- Garg, Ashim; Tamassia, Roberto (1995), "Upward planarity testing", *Order* **12** (2): 109–133, doi: 10.1007/BF01108622 (<http://dx.doi.org/10.1007/BF01108622>), MR 1354797 (<http://www.ams.org/mathscinet-getitem?mr=1354797>).

Research articles

- Abbasi, Sarmad; Healy, Patrick; Rextin, Aimal (2010), "Improving the running time of embedded upward planarity testing", *Information Processing Letters* **110** (7): 274–278, doi: 10.1016/j.ipl.2010.02.004 (<http://dx.doi.org/10.1016/j.ipl.2010.02.004>), MR 2642837 (<http://www.ams.org/mathscinet-getitem?mr=2642837>).
- Baker, K. A.; Fishburn, P. C.; Roberts, F. S. (1972), "Partial orders of dimension 2", *Networks* **2** (1): 11–28, doi: 10.1002/net.3230020103 (<http://dx.doi.org/10.1002/net.3230020103>).
- Bertolazzi, Paola; Cohen, Robert F.; Di Battista, Giuseppe; Tamassia, Roberto; Tollis, Ioannis G. (1994), "How to draw a series-parallel digraph", *International Journal of Computational Geometry & Applications* **4** (4): 385–402, doi: 10.1142/S0218195994000215 (<http://dx.doi.org/10.1142/S0218195994000215>), MR 1310911 (<http://www.ams.org/mathscinet-getitem?mr=1310911>).
- Bertolazzi, Paola; Di Battista, Giuseppe (1991), "On upward drawing testing of triconnected digraphs", *Proceedings of the Seventh Annual Symposium on Computational Geometry (SCG '91, North Conway, New Hampshire, USA)*, New York, NY, USA: ACM, pp. 272–280, doi: 10.1145/109648.109679 (<http://dx.doi.org/10.1145/109648.109679>), ISBN 0-89791-426-0.
- Bertolazzi, P.; Di Battista, G.; Liotta, G.; Mannino, C. (1994), "Upward drawings of triconnected digraphs", *Algorithmica* **12** (6): 476–497, doi: 10.1007/BF01188716 (<http://dx.doi.org/10.1007/BF01188716>), MR 1297810 (<http://www.ams.org/mathscinet-getitem?mr=1297810>).
- Bertolazzi, Paola; Di Battista, Giuseppe; Mannino, Carlo; Tamassia, Roberto (1998), "Optimal upward planarity testing of single-source digraphs", *SIAM Journal on Computing* **27** (1): 132–169, doi: 10.1137/S0097539794279626 (<http://dx.doi.org/10.1137/S0097539794279626>), MR 1614821 (<http://www.ams.org/mathscinet-getitem?mr=1614821>).
- Chan, Hubert (2004), "A parameterized algorithm for upward planarity testing", *Proc. 12th European Symposium on Algorithms (ESA '04)*, Lecture Notes in Computer Science **3221**, Springer-Verlag, pp. 157–168, doi: 10.1007/978-3-540-30140-0_16 (http://dx.doi.org/10.1007/978-3-540-30140-0_16).
- Di Battista, Giuseppe; Liu, Wei-Ping; Rival, Ivan (1990), "Bipartite graphs, upward drawings, and planarity", *Information Processing Letters* **36** (6): 317–322, doi: 10.1016/0020-0190(90)90045-Y ([http://dx.doi.org/10.1016/0020-0190\(90\)90045-Y](http://dx.doi.org/10.1016/0020-0190(90)90045-Y)), MR 1084490 (<http://www.ams.org/mathscinet-getitem?mr=1084490>).
- Di Battista, Giuseppe; Tamassia, Roberto (1988), "Algorithms for plane representations of acyclic digraphs", *Theoretical Computer Science* **61** (2-3): 175–198, doi: 10.1016/0304-3975(88)90123-5 ([http://dx.doi.org/10.1016/0304-3975\(88\)90123-5](http://dx.doi.org/10.1016/0304-3975(88)90123-5)), MR 980241 (<http://www.ams.org/mathscinet-getitem?mr=980241>).
- Di Battista, Giuseppe; Tamassia, Roberto; Tollis, Ioannis G. (1992), "Area requirement and symmetry display of planar upward drawings", *Discrete and Computational Geometry* **7** (4): 381–401, doi: 10.1007/BF02187850 (<http://dx.doi.org/10.1007/BF02187850>), MR 1148953 (<http://www.ams.org/mathscinet-getitem?mr=1148953>).
- Didimo, Walter; Giordano, Francesco; Liotta, Giuseppe (2009), "Upward spirality and upward planarity testing", *SIAM Journal on Discrete Mathematics* **23** (4): 1842–1899, doi: 10.1137/070696854 (<http://dx.doi.org/10.1137/070696854>), MR 2594962 (<http://www.ams.org/mathscinet-getitem?mr=2594962>).

- Frati, Fabrizio (2008), "On minimum area planar upward drawings of directed trees and other families of directed acyclic graphs", *International Journal of Computational Geometry & Applications* **18** (3): 251–271, doi: 10.1142/S021819590800260X (<http://dx.doi.org/10.1142/S021819590800260X>), MR 2424444 (<http://www.ams.org/mathscinet-getitem?mr=2424444>).
- Garg, Ashim; Tamassia, Roberto (2001), "On the computational complexity of upward and rectilinear planarity testing", *SIAM Journal on Computing* **31** (2): 601–625, doi: 10.1137/S0097539794277123 (<http://dx.doi.org/10.1137/S0097539794277123>), MR 1861292 (<http://www.ams.org/mathscinet-getitem?mr=1861292>).
- Healy, Patrick; Lynch, Karol (2006), "Two fixed-parameter tractable algorithms for testing upward planarity", *International Journal of Foundations of Computer Science* **17** (5): 1095–1114, doi: 10.1142/S0129054106004285 (<http://dx.doi.org/10.1142/S0129054106004285>).
- Hutton, Michael D.; Lubiw, Anna (1996), "Upward planar drawing of single-source acyclic digraphs", *SIAM Journal on Computing* **25** (2): 291–311, doi: 10.1137/S0097539792235906 (<http://dx.doi.org/10.1137/S0097539792235906>), MR 1379303 (<http://www.ams.org/mathscinet-getitem?mr=1379303>). First presented at the 2nd ACM-SIAM Symposium on Discrete Algorithms, 1991.
- Jünger, Michael; Leipert, Sebastian (1999), "Level planar embedding in linear time", *Graph Drawing (Proc. GD '99)*, Lecture Notes in Computer Science **1731**, pp. 72–81, doi: 10.1007/3-540-46648-7_7 (http://dx.doi.org/10.1007/3-540-46648-7_7), ISBN 978-3-540-66904-3.
- Kelly, David (1987), "Fundamentals of planar ordered sets", *Discrete Mathematics* **63** (2-3): 197–216, doi: 10.1016/0012-365X(87)90008-2 ([http://dx.doi.org/10.1016/0012-365X\(87\)90008-2](http://dx.doi.org/10.1016/0012-365X(87)90008-2)), MR 885497 (<http://www.ams.org/mathscinet-getitem?mr=885497>).
- Papakostas, Achilleas (1995), "Upward planarity testing of outerplanar dags (extended abstract)", *Graph Drawing: DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994, Proceedings*, Lecture Notes in Computer Science **894**, Berlin: Springer, pp. 298–306, doi: 10.1007/3-540-58950-3_385 (http://dx.doi.org/10.1007/3-540-58950-3_385), MR 1337518 (<http://www.ams.org/mathscinet-getitem?mr=1337518>).
- Platt, C. R. (1976), "Planar lattices and planar graphs", *Journal of Combinatorial Theory, Ser. B* **21** (1): 30—39, doi: 10.1016/0095-8956(76)90024-1 ([http://dx.doi.org/10.1016/0095-8956\(76\)90024-1](http://dx.doi.org/10.1016/0095-8956(76)90024-1)).
- Thomassen, Carsten (1989), "Planar acyclic oriented graphs", *Order* **5** (4): 349–361, doi: 10.1007/BF00353654 (<http://dx.doi.org/10.1007/BF00353654>), MR 1010384 (<http://www.ams.org/mathscinet-getitem?mr=1010384>).

Graph embedding

In topological graph theory, an **embedding** (also spelled **imbedding**) of a graph G on a surface Σ is a representation of G on Σ in which points of Σ are associated to vertices and simple arcs (homeomorphic images of $[0,1]$) are associated to edges in such a way that:

- the endpoints of the arc associated to an edge e are the points associated to the end vertices of e ,
- no arcs include points associated with other vertices,
- two arcs never intersect at a point which is interior to either of the arcs.

Here a surface is a compact, connected 2-manifold.

Informally, an embedding of a graph into a surface is a drawing of the graph on the surface in such a way that its edges may intersect only at their endpoints. It is well known that any graph can be embedded in 3-dimensional Euclidean space \mathbb{R}^3 and planar graphs can be embedded in 2-dimensional Euclidean space \mathbb{R}^2 .

Often, an **embedding** is regarded as an equivalence class (under homeomorphisms of Σ) of representations of the kind just described.

Some authors define a weaker version of the definition of "graph embedding" by omitting the non-intersection condition for edges. In such contexts the stricter definition is described as "non-crossing graph embedding".

This article deals only with the strict definition of graph embedding. The weaker definition is discussed in the articles "graph drawing" and "crossing number".

Terminology

If a graph G is embedded on a closed surface Σ , the complement of the union of the points and arcs associated to the vertices and edges of G is a family of **regions** (or **faces**). A **2-cell embedding** or **map** is an embedding in which every face is homeomorphic to an open disk. A **closed 2-cell embedding** is an embedding in which the closure of every face is homeomorphic to a closed disk.

The **genus** of a graph is the minimal integer n such that the graph can be embedded in a surface of genus n . In particular, a planar graph has genus 0, because it can be drawn on a sphere without self-crossing. The **non-orientable genus** of a graph is the minimal integer n such that the graph can be embedded in a non-orientable surface of (non-orientable) genus n .

The **Euler genus** of a graph is the minimal integer n such that the graph can be embedded in an orientable surface of (orientable) genus $n/2$ or in a non-orientable surface of (non-orientable) genus n . A graph is **orientably simple** if its Euler genus is smaller than its non-orientable genus.

The **maximum genus** of a graph is the maximal integer n such that the graph can be 2-cell embedded in an orientable surface of genus n .

Combinatorial embedding

Main article: Rotation system

An embedded graph uniquely defines cyclic orders of edges incident to the same vertex. The set of all these cyclic orders is called a rotation system. Embeddings with the same rotation system are considered to be equivalent and the corresponding equivalence class of embeddings is called **combinatorial embedding** (as opposed to the term **topological embedding**, which refers to the previous definition in terms of points and curves). Sometimes, the rotation system itself is called a "combinatorial embedding".

An embedded graph also defines natural cyclic orders of edges which constitutes the boundaries of the faces of the embedding. However handling these face-based orders is less straightforward, since in some cases some edges may

be traversed twice along a face boundary. For example this is always the case for embeddings of trees, which have a single face. To overcome this combinatorial nuisance, one may consider that every edge is "split" lengthwise in two "half-edges", or "sides". Under this convention in all face boundary traversals each half-edge is traversed only once and the two half-edges of the same edge are always traversed in opposite directions.

Computational complexity

The problem of finding the graph genus is NP-hard (the problem of determining whether an n -vertex graph has genus g is NP-complete).

At the same time, the graph genus problem is fixed-parameter tractable, i.e., polynomial time algorithms are known to check whether a graph can be embedded into a surface of a given fixed genus as well as to find the embedding.

The first breakthrough in this respect happened in 1979, when algorithms of time complexity $O(n^{O(g)})$ were independently submitted to the Annual ACM Symposium on Theory of Computing: one by I. Filotti and G.L. Miller and another one by John Reif. Their approaches were quite different, but upon the suggestion of the program committee they presented a joint paper.

In 1999 it was reported that the fixed-genus case can be solved in time linear in the graph size and doubly exponential in the genus.

Embeddings of graphs into higher-dimensional spaces

It is known that any graph can be embedded into a three-dimensional space.

One method for doing this is to place the points on any line in space and to draw the m edges as curves each of which lies in one of m distinct halfplanes having that line as their common boundary. An embedding like this in which the edges are drawn on halfplanes is called a book embedding of the graph. This metaphor comes from imagining that each of the planes where an edge is drawn is like a page of a book. It was observed that in fact several edges may be drawn in the same "page"; the *book thickness* of the graph is the minimum number of halfplanes needed for such a drawing.

Alternatively, any graph can be drawn with straight-line edges in three dimensions without crossings by placing its vertices in general position so that no four are coplanar. For instance, this may be achieved by placing the i th vertex at the point (i, i^2, i^3) of the moment curve.

An embedding of a graph into three-dimensional space in which no two of the cycles are topologically linked is called a linkless embedding. A graph has a linkless embedding if and only if it does not have one of the seven graphs of the Petersen family as a minor.

References

Application: Sociograms

A **sociogram** is a graphic representation of social links that a person has. It is a graph drawing that plots the structure of interpersonal relations in a group situation.^[1]

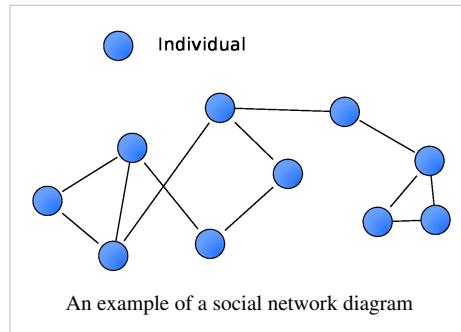
Overview

Sociograms were developed by Jacob L. Moreno to analyze choices or preferences within a group.^[2] They can diagram the structure and patterns of group interactions. A sociogram can be drawn on the basis of many different criteria: Social relations, channels of influence, lines of communication etc.

Those points on a sociogram who have many choices are called Stars. Those with few or no choices are called isolates. Individuals who choose each other are known to have made a Mutual Choice. One-Way Choice refers to individuals who choose someone but the choice is not reciprocated. Cliques are groups of three or more people within a larger group who all choose each other (Mutual Choice).

Sociograms are the charts or tools used to find the Sociometry of a social space.

Under the Social Discipline Model, sociograms are sometimes used to reduce misbehavior in a classroom environment.^[3] A sociogram is constructed after students answer a series of questions probing for affiliations with other classmates. The diagram can then be used to identify pathways for social acceptance for misbehaving students. In this context, the resulting sociograms are known as a friendship chart. Often, the most important person/thing is in a bigger bubble in relation to everyone else. The size of the bubble represents the importance, with the biggest bubble meaning most important and the smallest representing the least important.

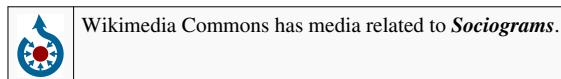


An example of a social network diagram

References

- [1] Sociogram (<http://www.merriam-webster.com/dictionary/sociogram>) at merriam-webster.com.
- [2] "An Experiential Approach to Organization Development 7th ed." Brown, Donald R. and Harvey, Don. Page 134
- [3] Wolfgang, Charles H., *Solving Discipline And Classroom Management Problems: Methods and Models for Today's Teachers*; U.S.A, John Wiley and Sons, 2001.; p. 116

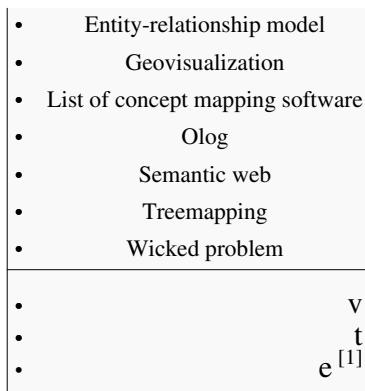
External links



Application: Concept maps

For concept maps in generic programming, see [Concept \(generic programming\)](#).

	Information mapping
	Topics & fields
	<ul style="list-style-type: none">• Business decision mapping• Cognitive map• Conceptual graph• Data visualization• Decision tree• Educational psychology• Educational technology• Graphic communication• Information design• Information graphics• Interactive visualization• Knowledge visualization• Mental model• Morphological analysis• Visual analytics• Visual language
	Tree-like approaches
	<ul style="list-style-type: none">• Cladistics• Argument map• Cognitive map• Concept mapping• Conceptual graphs• Dendrogram• Graph drawing• Hyperbolic tree• Layered graph drawing• Mental model• Mind mapping• Object-role modeling• Organizational chart• Radial tree• Refined concept map• Semantic network• Sociogram• Timeline• Topic Maps• Tree structure
	See also
	<ul style="list-style-type: none">• Diagrammatic reasoning



A **concept map** is a diagram that depicts suggested relationships between concepts. It is a graphical tool that designers, engineers, technical writers, and others use to organize and structure knowledge.

A concept map typically represents ideas and information as boxes or circles, which it connects with labeled arrows in a downward-branching hierarchical structure. The relationship between concepts can be articulated in linking phrases such as *causes*, *requires*, or *contributes to*.^[2]

The technique for visualizing these relationships among different concepts is called *concept mapping*. Concept maps define the ontology of computer systems, for example with the object-role modeling or Unified Modeling Language formalism.

Overview

A concept map is a way of representing relationships between ideas, images, or words in the same way that a sentence diagram represents the grammar of a sentence, a road map represents the locations of highways and towns, and a circuit diagram represents the workings of an electrical appliance. In a concept map, each word or phrase connects to another, and links back to the original idea, word, or phrase. Concept maps are a way to develop logical thinking and study skills by revealing connections and helping students see how individual ideas form a larger whole. An example of the use of concept maps is provided in the context of learning about types of fuel.^[3]

Concept maps were developed to enhance meaningful learning in the sciences. A well-made concept map grows within a *context frame* defined by an explicit "focus question", while a mind map often has only branches radiating out from a central picture. Some research evidence suggests that the brain stores knowledge as productions (situation-response conditionals) that act on declarative memory content, which is also referred to as chunks or propositions.^{[4][5]} Because concept maps are constructed to reflect organization of the declarative memory system, they facilitate sense-making and meaningful learning on the part of individuals who make concept maps and those who use them.

Concept mapping versus topic maps and mind mapping

Concept maps are rather similar to topic maps (in that both allow to connect concepts or topics via graphs), while both can be contrasted with the similar idea of mind mapping, which is often restricted to radial hierarchies and tree structures. Among the various schema and techniques for visualizing ideas, processes, organizations, concept mapping, as developed by Joseph Novak is unique in its philosophical basis, which "makes concepts, and propositions composed of concepts, the central elements in the structure of knowledge and construction of meaning."^[6] Another contrast between Concept mapping and Mind mapping is the speed and spontaneity when a Mind map is created. A Mind map reflects what you think about a single topic, which can focus group brainstorming. A Concept map can be a map, a system view, of a real (abstract) system or set of concepts. Concept maps are more free form, as multiple hubs and clusters can be created, unlike mind maps, which fix on a single two centered approach.

History

The technique of concept mapping was developed by Joseph D. Novak and his research team at Cornell University in the 1970s as a means of representing the emerging science knowledge of students. It has subsequently been used as a tool to increase meaningful learning in the sciences and other subjects as well as to represent the expert knowledge of individuals and teams in education, government and business. Concept maps have their origin in the learning movement called constructivism. In particular, constructivists hold that learners actively construct knowledge.

Novak's work is based on the cognitive theories of David Ausubel (assimilation theory), who stressed the importance of prior knowledge in being able to learn new concepts: "The most important single factor influencing learning is what the learner already knows. Ascertain this and teach accordingly."^[7] Novak taught students as young as six years old to make concept maps to represent their response to focus questions such as "What is water?" "What causes the seasons?" In his book *Learning How to Learn*, Novak states that a "meaningful learning involves the assimilation of new concepts and propositions into existing cognitive structures."

Various attempts have been made to conceptualize the process of creating concept maps. Ray McAleese, in a series of articles, has suggested that mapping is a process of *off-loading*. In this 1998 paper, McAleese draws on the work of Sowa^[8] and a paper by Sweller & Chandler.^[9] In essence, McAleese suggests that the process of making knowledge explicit, using *nodes* and *relationships*, allows the individual to become aware of what they know and as a result to be able to modify what they know.^[10] Maria Birbili applies that same idea to helping young children learn to think about what they know.^[11] The concept of the **Knowledge Arena** is suggestive of a virtual space where learners may explore what they know and what they do not know.

Use

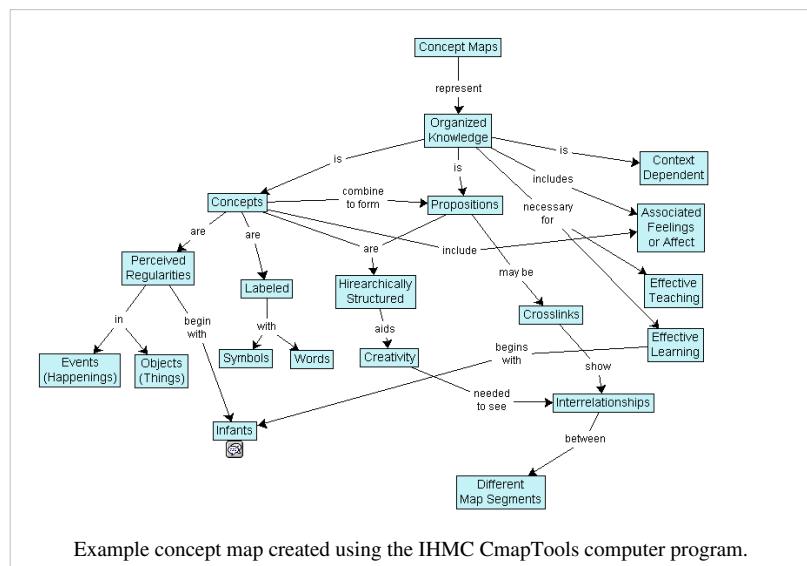
Concept maps are used to stimulate the generation of ideas, and are believed to aid creativity. Wikipedia:Manual of Style/Words to watch#Unsupported attributions Concept mapping is also sometimes used for brain-storming. Although they are often personalized and idiosyncratic, concept maps can be used to communicate complex ideas.

Formalized concept maps are used in software design, where a common usage is Unified Modeling Language diagramming amongst similar conventions and development methodologies.

Concept mapping can also be seen as a first step in ontology-building, and can also be used flexibly to represent formal argument.

Concept maps are widely used in education and business.^[12] Uses include:

- Note taking and summarizing gleaned key concepts, their relationships and hierarchy from documents and source materials
- New knowledge creation: e.g., transforming tacit knowledge into an organizational resource, mapping team knowledge



- Institutional knowledge preservation (retention), e.g., eliciting and mapping expert knowledge of employees prior to retirement
- Collaborative knowledge modeling and the transfer of expert knowledge
- Facilitating the creation of shared vision and shared understanding within a team or organization
- Instructional design: concept maps used as Ausubelian "advance organizers" that provide an initial conceptual frame for subsequent information and learning.
- Training: concept maps used as Ausubelian "advanced organizers" to represent the training context and its relationship to their jobs, to the organization's strategic objectives, to training goals.
- Business Concept Mapping used as part of business analysis activities.^[13]
- Increasing meaningful learning for example through writing activities where concept maps automatically generated from an essay are shown to the writer.
- Communicating complex ideas and arguments
- Examining the symmetry of complex ideas and arguments and associated terminology
- Detailing the entire structure of an idea, train of thought, or line of argument (with the specific goal of exposing faults, errors, or gaps in one's own reasoning) for the scrutiny of others.
- Enhancing metacognition (learning to learn, and thinking about knowledge)
- Improving language ability
- Knowledge Elicitation
- Assessing learner understanding of learning objectives, concepts, and the relationship among those concepts
- Lexicon development

Concept maps have also been used to explore the complex roles and definitions of internet use and social media:

- Defining problematic internet usage. Concept mapping identified 7 concept clusters that take into account the physical, emotional, social and functional impairments of this phenomenon.
- Identifying Facebook's influence on users' attitudes, intentions and behaviors.

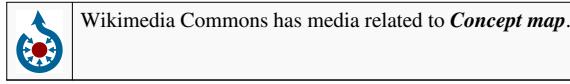
References

- [1] <http://en.wikipedia.org/w/index.php?title=Template:InfoMaps&action=edit>
- [2] Joseph D. Novak & Alberto J. Cañas (2006). "The Theory Underlying Concept Maps and How To Construct and Use Them" (<http://cmap.ihmc.us/Publications/ResearchPapers/TheoryCmaps/TheoryUnderlyingConceptMaps.htm>), Institute for Human and Machine Cognition. Accessed 24 Nov 2008.
- [3] CONCEPT MAPPING FUELS (http://www.energyeducation.tx.gov/pdf/223_inv.pdf). Accessed 24 Nov 2008.
- [4] Anderson, J. R., & Lebiere, C. (1998). The atomic components of thought. Mahwah, NJ: Erlbaum.
- [5] Anderson, J. R., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An Integrated Theory of the Mind. *Psychological Review*, 111(4), 1036–1050.
- [6] Novak, J.D. & Gowin, D.B. (1996). Learning How To Learn, Cambridge University Press: New York, p. 7.
- [7] Ausubel, D. (1968) Educational Psychology: A Cognitive View. Holt, Rinehart & Winston, New York.
- [8] Sowa, J.F., 1983. *Conceptual structures: information processing in mind and machine*, Addison-Wesley.
- [9] Sweller, J. & Chandler, P., 1991. Evidence for Cognitive Load Theory. *Cognition and Instruction*, 8(4), p.351-362.
- [10] McAleese,R (1998) **The Knowledge Arena** as an Extension to the Concept Map: Reflection in Action, *Interactive Learning Environments*, 6,3,p.251-272.
- [11] Birbili, M. (2006) "Mapping Knowledge: Concept Maps in Early Childhood Education" (<http://ecrp.uiuc.edu/v8n2/birbili.html>), *Early Childhood Research & Practice*, 8(2), Fall 2006
- [12] Moon, B.M., Hoffman, R.R., Novak, J.D., & Cañas, A.J. (2011). Applied Concept Mapping: Capturing, Analyzing and Organizing Knowledge. (<http://www.appliedconceptmapping.info>) CRC Press: New York.
- [13] Frisendal, T. (2012). Design Thinking Business Analysis - Business Concept Mapping Applied. (<http://www.businessconceptmapping.com>) Springer-Verlag: Berlin Heidelberg.

Further reading

- Novak, J.D., Learning, Creating, and Using Knowledge: Concept Maps as Facilitative Tools in Schools and Corporations, Lawrence Erlbaum Associates, (Mahwah), 1998.
- Novak, J.D. & Gowin, D.B., Learning How to Learn, Cambridge University Press, (Cambridge), 1984.

External links



- Concept Mapping: A Graphical System for Understanding the Relationship between Concepts (<http://www.ericdigests.org/1998-1/concept.htm>) - From the ERIC Clearinghouse on Information and Technology.
- A large catalog of papers on cognitive maps and learning (<http://cmap.ihmc.us/Publications/>) by Novak, Cañas, and others.
- Example of a concept map from 1957 (<http://www.mind-mapping.org/images/walt-disney-business-map.png>) by Walt Disney.

Special classes of graphs

Interval graph

In graph theory, an **interval graph** is the intersection graph of a family of intervals on the real line. It has one vertex for each interval in the family, and an edge between every pair of vertices corresponding to intervals that intersect.

Definition

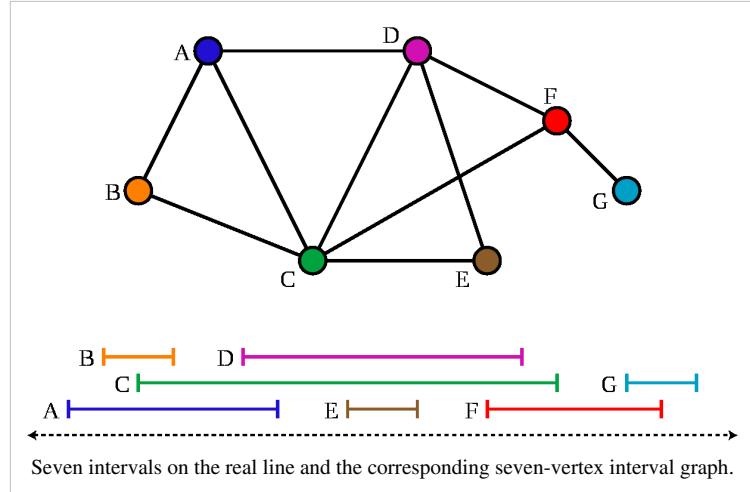
Formally, an interval graph is an undirected graph formed from a family of intervals

$$S_i, i = 0, 1, 2, \dots$$

by creating one vertex v_i for each interval S_i , and connecting two vertices v_i and v_j by an edge whenever the corresponding two sets have a nonempty intersection, that is,

$$E(G) = \{ \{v_i, v_j\} \mid S_i \cap S_j \neq \emptyset \}.$$

From this construction one can verify a common property held by all interval graphs. That is, graph G is an interval graph if and only if the maximal cliques of G can be ordered M_1, M_2, \dots, M_k such that for any $v \in M_i \cap M_k$, where $i < k$, it is also the case that $v \in M_j$ for any M_j , $i \leq j \leq k$.



Efficient recognition algorithms

Determining whether a given graph $G = (V, E)$ is an interval graph can be done in $O(|V|+|E|)$ time by seeking an ordering of the maximal cliques of G that is consecutive with respect to vertex inclusion.

The original linear time recognition algorithm of Booth & Lueker (1976) is based on their complex PQ tree data structure, but Habib et al. (2000) showed how to solve the problem more simply using lexicographic breadth-first search, based on the fact that a graph is an interval graph if and only if it is chordal and its complement is a comparability graph.

Related families of graphs

Interval graphs are chordal graphs and hence perfect graphs. Their complements belong to the class of comparability graphs, and the comparability relations are precisely the interval orders.

The interval graphs that have an interval representation in which every two intervals are either disjoint or nested are the trivially perfect graphs.

A graph has *boxicity* at most one if and only if it is an interval graph; the *boxicity* of an arbitrary graph G is the minimum number of interval graphs on the same set of vertices such that the intersection of the edges sets of the interval graphs is G .

Proper interval graphs are interval graphs that have an interval representation in which no interval properly contains any other interval; unit interval graphs are the interval graphs that have an interval representation in which each interval has unit length. A unit interval representation without repeated intervals is necessarily a proper interval representation. Not every proper interval representation is a unit interval representation, but every proper interval graph is a unit interval graph, and vice versa. Every proper interval graph is a claw-free graph; conversely, the proper interval graphs are exactly the claw-free interval graphs. However, there exist claw-free graphs that are not interval graphs.^[1]

The intersection graphs of arcs of a circle form circular-arc graphs, a class of graphs that contains the interval graphs. The trapezoid graphs, intersections of trapezoids whose parallel sides all lie on the same two parallel lines, are also a generalization of the interval graphs.

The pathwidth of an interval graph is one less than the size of its maximum clique (or equivalently, one less than its chromatic number), and the pathwidth of any graph G is the same as the smallest pathwidth of an interval graph that contains G as a subgraph.

The connected triangle-free interval graphs are exactly the caterpillar trees.

Applications

The mathematical theory of interval graphs was developed with a view towards applications by researchers at the RAND Corporation's mathematics department, which included young researchers—such as Peter C. Fishburn and students like Alan C. Tucker and Joel E. Cohen—besides leaders—such as Delbert Fulkerson and (recurring visitor) Victor Klee. Cohen applied interval graphs to mathematical models of population biology, specifically food webs.

Interval graphs are used to represent resource allocation problems in operations research and scheduling theory. In these applications, each interval represents a request for a resource (such as a processing unit of a distributed computing system or a room for a class) for a specific period of time. The maximum weight independent set problem for the graph represents the problem of finding the best subset of requests that can be satisfied without conflicts. An optimal graph coloring of the interval graph represents an assignment of resources that covers all of the requests with as few resources as possible; it can be found in polynomial time by a greedy coloring algorithm that colors the intervals in sorted order by their left endpoints.

Other applications include genetics, bioinformatics, and computer science. Finding a set of intervals that represent an interval graph can also be used as a way of assembling contiguous subsequences in DNA mapping. Interval graphs also play an important role in temporal reasoning.

Notes

[1] , p. 89.

References

- Bar-Noy, Amotz; Bar-Yehuda, Reuven; Freund, Ari; Naor, Joseph (Seffi); Schieber, Baruch (2001), "A unified approach to approximating resource allocation and scheduling" (<http://portal.acm.org/citation.cfm?id=335410&coll=portal&dl=ACM>), *Journal of the ACM* **48** (5): 1069–1090, doi: 10.1145/502102.502107 (<http://dx.doi.org/10.1145/502102.502107>).
- Bodlaender, Hans L. (1998), "A partial k -arboretum of graphs with bounded treewidth", *Theoretical Computer Science* **209** (1–2): 1–45, doi: 10.1016/S0304-3975(97)00228-4 ([http://dx.doi.org/10.1016/S0304-3975\(97\)00228-4](http://dx.doi.org/10.1016/S0304-3975(97)00228-4)).
- Booth, K. S.; Lueker, G. S. (1976), "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms", *J. Comput. System Sci.* **13** (3): 335–379, doi: 10.1016/S0022-0000(76)80045-1 ([http://dx.doi.org/10.1016/S0022-0000\(76\)80045-1](http://dx.doi.org/10.1016/S0022-0000(76)80045-1)).

- Cohen, Joel E. (1978). *Food webs and niche space*. Monographs in Population Biology **11**. Princeton, NJ: Princeton University Press (<http://press.princeton.edu/titles/324.html>). pp. xv+1–190. ISBN 978-0-691-08202-8.
- Eckhoff, Jürgen (1993), "Extremal interval graphs", *Journal of Graph Theory* **17** (1): 117–127, doi: [10.1002/jgt.3190170112](https://doi.org/10.1002/jgt.3190170112) (<http://dx.doi.org/10.1002/jgt.3190170112>).
- Faudree, Ralph; Flandrin, Evelyne; Ryjáček, Zdeněk (1997), "Claw-free graphs — A survey", *Discrete Mathematics* **164** (1–3): 87–147, doi: [10.1016/S0012-365X\(96\)00045-3](https://doi.org/10.1016/S0012-365X(96)00045-3) ([http://dx.doi.org/10.1016/S0012-365X\(96\)00045-3](http://dx.doi.org/10.1016/S0012-365X(96)00045-3)), MR 1432221 (<http://www.ams.org/mathscinet-getitem?mr=1432221>).
- Fishburn, Peter C. (1985). *Interval orders and interval graphs: A study of partially ordered sets*. Wiley-Interscience Series in Discrete Mathematics. New York: John Wiley & Sons.
- Fulkerson, D. R.; Gross, O. A. (1965), "Incidence matrices and interval graphs", *Pacific Journal of Mathematics* **15**: 835–855, doi: [10.2140/pjm.1965.15.835](https://doi.org/10.2140/pjm.1965.15.835) (<http://dx.doi.org/10.2140/pjm.1965.15.835>).
- Gardi, Frédéric (October 28, 2007). "The Roberts characterization of proper and unit interval graphs" (<http://www.sciencedirect.com/science/article/pii/S0012365X07000696>). *Discrete Mathematics* **307** (22): 2906–2908. doi: [10.1016/j.disc.2006.04.043](https://doi.org/10.1016/j.disc.2006.04.043) (<http://dx.doi.org/10.1016/j.disc.2006.04.043>). ISSN 0012-365X (<http://www.worldcat.org/issn/0012-365X>). Retrieved March 30, 2014..
- Gilmore, P. C.; Hoffman, A. J. (1964), "A characterization of comparability graphs and of interval graphs", *Can. J. Math.* **16**: 539–548, doi: [10.4153/CJM-1964-055-5](https://doi.org/10.4153/CJM-1964-055-5) (<http://dx.doi.org/10.4153/CJM-1964-055-5>).
- Golumbic, Martin Charles (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, ISBN 0-12-289260-7.
- Golumbic, Martin Charles; Shamir, Ron (1993), "Complexity and algorithms for reasoning about time: a graph-theoretic approach", *J. Assoc. Comput. Mach.* **40**: 1108–1133, doi: [10.1145/174147.169675](https://doi.org/10.1145/174147.169675) (<http://dx.doi.org/10.1145/174147.169675>).
- Habib, Michel; McConnell, Ross; Paul, Christophe; Viennot, Laurent (2000), "Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition, and consecutive ones testing" (<http://www.cs.colostate.edu/~rmm/lexbfs.ps>), *Theor. Comput. Sci.* **234**: 59–84, doi: [10.1016/S0304-3975\(97\)00241-7](https://doi.org/10.1016/S0304-3975(97)00241-7) ([http://dx.doi.org/10.1016/S0304-3975\(97\)00241-7](http://dx.doi.org/10.1016/S0304-3975(97)00241-7)).
- Roberts, F. S. (1969), "Indifference graphs", in Harary, Frank, *Proof Techniques in Graph Theory*, New York, NY: Academic Press, pp. 139–146, ISBN 978-0123242600, OCLC 30287853 (<http://www.worldcat.org/oclc/30287853>).
- Zhang, Peisen; Schon, Eric A.; Fischer, Stuart G.; Cayanis, Eftihia; Weiss, Janie; Kistler, Susan; Bourne, Philip E. (1994), "An algorithm based on graph theory for the assembly of contigs in physical mapping of DNA", *Bioinformatics* **10** (3): 309–317, doi: [10.1093/bioinformatics/10.3.309](https://doi.org/10.1093/bioinformatics/10.3.309) (<http://dx.doi.org/10.1093/bioinformatics/10.3.309>).

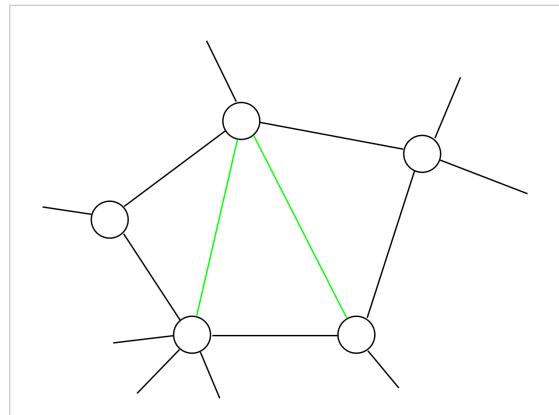
External links

- "interval graph" (http://www.graphclasses.org/classes/gc_234.html). *Information System on Graph Classes and their Inclusions* (<http://www.graphclasses.org/index.html>).
- Weisstein, Eric W., "Interval graph" (<http://mathworld.wolfram.com/IntervalGraph.html>), *MathWorld*.

Chordal graph

In the mathematical area of graph theory, a graph is **chordal** if each of its cycles of four or more vertices has a *chord*, which is an edge that is not part of the cycle but connects two vertices of the cycle. Equivalently, every induced cycle in the graph should have at most three nodes. The chordal graphs may also be characterized as the graphs that have perfect elimination orderings, as the graphs in which each minimal separator is a clique, and as the intersection graphs of subtrees of a tree. They are sometimes also called **rigid circuit graphs** or **triangulated graphs**.^[1]

Chordal graphs are a subset of the perfect graphs. They may be recognized in polynomial time, and several problems that are hard on other classes of graphs such as graph coloring may be solved in polynomial time when the input is chordal. The treewidth of an arbitrary graph may be characterized by the size of the cliques in the chordal graphs that contain it.



A cycle (black) with two chords (green). As for this part, the graph is chordal. However, removing one green edge would result in a non-chordal graph. Indeed, the other green edge with three black edges would form a cycle of length four with no chords.

Perfect elimination and efficient recognition

A *perfect elimination ordering* in a graph is an ordering of the vertices of the graph such that, for each vertex v , v and the neighbors of v that occur after v in the order form a clique. A graph is chordal if and only if it has a perfect elimination ordering.^[2]

Rose, Lueker & Tarjan (1976) (see also Habib et al. 2000) show that a perfect elimination ordering of a chordal graph may be found efficiently using an algorithm known as lexicographic breadth-first search. This algorithm maintains a partition of the vertices of the graph into a sequence of sets; initially this sequence consists of a single set with all vertices. The algorithm repeatedly chooses a vertex v from the earliest set in the sequence that contains previously unchosen vertices, and splits each set S of the sequence into two smaller subsets, the first consisting of the neighbors of v in S and the second consisting of the non-neighbors. When this splitting process has been performed for all vertices, the sequence of sets has one vertex per set, in the reverse of a perfect elimination ordering.

Since both this lexicographic breadth first search process and the process of testing whether an ordering is a perfect elimination ordering can be performed in linear time, it is possible to recognize chordal graphs in linear time. The graph sandwich problem on chordal graphs is NP-complete^[3] whereas the probe graph problem on chordal graphs has polynomial-time complexity.^[4]

The set of all perfect elimination orderings of a chordal graph can be modeled as the *basic words* of an antimatroid; Chandran et al. (2003) use this connection to antimatroids as part of an algorithm for efficiently listing all perfect elimination orderings of a given chordal graph.

Maximal cliques and graph coloring

Another application of perfect elimination orderings is finding a maximum clique of a chordal graph in polynomial-time, while the same problem for general graphs is NP-complete. More generally, a chordal graph can have only linearly many maximal cliques, while non-chordal graphs may have exponentially many. To list all maximal cliques of a chordal graph, simply find a perfect elimination ordering, form a clique for each vertex v together with the neighbors of v that are later than v in the perfect elimination ordering, and test whether each of the

resulting cliques is maximal.

The largest maximal clique is a maximum clique, and, as chordal graphs are perfect, the size of this clique equals the chromatic number of the chordal graph. Chordal graphs are perfectly orderable: an optimal coloring may be obtained by applying a greedy coloring algorithm to the vertices in the reverse of a perfect elimination ordering.^[5]

Minimal separators

In any graph, a vertex separator is a set of vertices the removal of which leaves the remaining graph disconnected; a separator is minimal if it has no proper subset that is also a separator. According to a theorem of Dirac (1961), chordal graphs are graphs in which each minimal separator is a clique; Dirac used this characterization to prove that chordal graphs are perfect.

The family of chordal graphs may be defined inductively as the graphs whose vertices can be divided into three nonempty subsets A , S , and B , such that $A \cup S$ and $S \cup B$ both form chordal induced subgraphs, S is a clique, and there are no edges from A to B . That is, they are the graphs that have a recursive decomposition by clique separators into smaller subgraphs. For this reason, chordal graphs have also sometimes been called **decomposable graphs**.

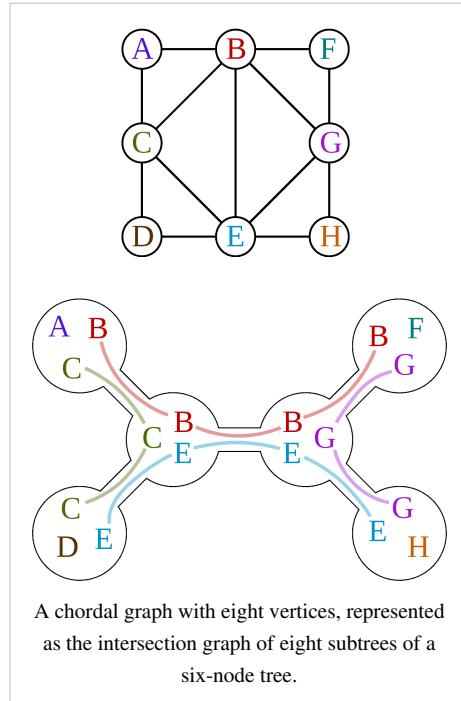
Intersection graphs of subtrees

An alternative characterization of chordal graphs, due to Gavril (1974), involves trees and their subtrees.

From a collection of subtrees of a tree, one can define a **subtree graph**, which is an intersection graph that has one vertex per subtree and an edge connecting any two subtrees that overlap in one or more nodes of the tree. Gavril showed that the subtree graphs are exactly the chordal graphs.

A representation of a chordal graph as an intersection of subtrees forms a tree decomposition of the graph, with treewidth equal to one less than the size of the largest clique in the graph; the tree decomposition of any graph G can be viewed in this way as a representation of G as a subgraph of a chordal graph. The tree decomposition of a graph is also the junction tree of the junction tree algorithm.

Relation to other graph classes



A chordal graph with eight vertices, represented as the intersection graph of eight subtrees of a six-node tree.

Subclasses

Interval graphs are the intersection graphs of subtrees of path graphs, a special case of trees. Therefore, they are a subfamily of chordal graphs.

Split graphs are graphs that are both chordal and the complements of chordal graphs. Bender, Richmond & Wormald (1985) showed that, in the limit as n goes to infinity, the fraction of n -vertex chordal graphs that are split approaches one.

Ptolemaic graphs are graphs that are both chordal and distance hereditary. Quasi-threshold graphs are a subclass of Ptolemaic graphs that are both chordal and cographs. Block graphs are another subclass of Ptolemaic graphs in which every two maximal cliques have at most one vertex in common. A special type is windmill graphs, where the

common vertex is the same for every pair of cliques.

Strongly chordal graphs are graphs that are chordal and contain no n -sun ($n \geq 3$) as induced subgraph.

K -trees are chordal graphs in which all maximal cliques and all maximal clique separators have the same size. Apollonian networks are chordal maximal planar graphs, or equivalently planar 3-trees. Maximal outerplanar graphs are a subclass of 2-trees, and therefore are also chordal.

Superclasses

Chordal graphs are a subclass of the well known perfect graphs. Other superclasses of chordal graphs include weakly chordal graphs, odd-hole-free graphs, and even-hole-free graphs. In fact, chordal graphs are precisely the graphs that are both odd-hole-free and even-hole-free (see holes in graph theory).

Every chordal graph is a strangulated graph, a graph in which every peripheral cycle is a triangle, because peripheral cycles are a special case of induced cycles. Strangulated graphs are graphs that can be formed by clique-sums of chordal graphs and maximal planar graphs. Therefore strangulated graphs include maximal planar graphs.^[6]

Chordal completions and treewidth

If G is an arbitrary graph, a **chordal completion** of G is a chordal graph that contains G as a subgraph. The treewidth of G is one less than the number of vertices in a maximum clique of a chordal completion chosen to minimize this clique size. The k -trees are the graphs to which no additional edges can be added without increasing their treewidth to a number larger than k . Therefore, the k -trees are their own chordal completions, and form a subclass of the chordal graphs. Chordal completions can also be used to characterize several other related classes of graphs.^[7]

Notes

[1] . Note however that "triangulated graphs" also sometimes refers to maximal planar graphs.

[2] Fulkerson & Gross (1965).

[3] Bodlaender, Fellows & Warnow (1992).

[4] Berry, Golumbic & Lipshteyn (2007).

[5] Maffray (2003).

[6] Seymour & Weaver (1984).

[7] Parra & Scheffler (1997).

References

- Bender, E. A.; Richmond, L. B.; Wormald, N. C. (1985), "Almost all chordal graphs split", *J. Austral. Math. Soc.*, A **38** (2): 214–221, doi: 10.1017/S1446788700023077 (<http://dx.doi.org/10.1017/S1446788700023077>), MR 0770128 (<http://www.ams.org/mathscinet-getitem?mr=0770128>).
- Berry, Anne; Golumbic, Martin Charles; Lipshteyn, Marina (2007), "Recognizing chordal probe graphs and cycle-bicolorable graphs", *SIAM Journal on Discrete Mathematics* **21** (3): 573–591, doi: 10.1137/050637091 (<http://dx.doi.org/10.1137/050637091>).
- Bodlaender, H. L.; Fellows, M. R.; Warnow, T. J. (1992), "Two strikes against perfect phylogeny", *Proc. of 19th International Colloquium on Automata Languages and Programming*, Lecture Notes in Computer Science **623**, pp. 273–283, doi: 10.1007/3-540-55719-9_80 (http://dx.doi.org/10.1007/3-540-55719-9_80).
- Chandran, L. S.; Ibarra, L.; Ruskey, F.; Sawada, J. (2003), "Enumerating and characterizing the perfect elimination orderings of a chordal graph" (<http://www.cis.uoguelph.ca/~sawada/papers/chordal.pdf>), *Theoretical Computer Science* **307** (2): 303–317, doi: 10.1016/S0304-3975(03)00221-4 ([http://dx.doi.org/10.1016/S0304-3975\(03\)00221-4](http://dx.doi.org/10.1016/S0304-3975(03)00221-4)).
- Dirac, G. A. (1961), "On rigid circuit graphs", *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* **25**: 71–76, doi: 10.1007/BF02992776 (<http://dx.doi.org/10.1007/BF02992776>), MR 0130190

- (<http://www.ams.org/mathscinet-getitem?mr=0130190>).
- Fulkerson, D. R.; Gross, O. A. (1965), "Incidence matrices and interval graphs" (<http://projecteuclid.org/Dienst/UI/1.0/Summarize/euclid.pjm/1102995572>), *Pacific J. Math* **15**: 835–855.
 - Gavril, Fănică (1974), "The intersection graphs of subtrees in trees are exactly the chordal graphs", *Journal of Combinatorial Theory, Series B* **16**: 47–56, doi: 10.1016/0095-8956(74)90094-X ([http://dx.doi.org/10.1016/0095-8956\(74\)90094-X](http://dx.doi.org/10.1016/0095-8956(74)90094-X)).
 - Golumbic, Martin Charles (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press.
 - Habib, Michel; McConnell, Ross; Paul, Christophe; Viennot, Laurent (2000), "Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition, and consecutive ones testing" (<http://www.cs.colostate.edu/~rmm/lexbfs.ps>), *Theoretical Computer Science* **234**: 59–84, doi: 10.1016/S0304-3975(97)00241-7 ([http://dx.doi.org/10.1016/S0304-3975\(97\)00241-7](http://dx.doi.org/10.1016/S0304-3975(97)00241-7)).
 - Maffray, Frédéric (2003), "On the coloration of perfect graphs", in Reed, Bruce A.; Sales, Cláudia L., *Recent Advances in Algorithms and Combinatorics*, CMS Books in Mathematics **11**, Springer-Verlag, pp. 65–84, doi: 10.1007/0-387-22444-0_3 (http://dx.doi.org/10.1007/0-387-22444-0_3), ISBN 0-387-95434-1.
 - Parra, Andreas; Scheffler, Petra (1997), "Characterizations and algorithmic applications of chordal graph embeddings", *Discrete Applied Mathematics* **79** (1-3): 171–188, doi: 10.1016/S0166-218X(97)00041-3 ([http://dx.doi.org/10.1016/S0166-218X\(97\)00041-3](http://dx.doi.org/10.1016/S0166-218X(97)00041-3)), MR 1478250 (<http://www.ams.org/mathscinet-getitem?mr=1478250>).
 - Patil, H. P. (1986), "On the structure of k -trees", *Journal of Combinatorics, Information and System Sciences* **11** (2–4): 57–64, MR 966069 (<http://www.ams.org/mathscinet-getitem?mr=966069>).
 - Rose, D.; Lueker, George; Tarjan, Robert E. (1976), "Algorithmic aspects of vertex elimination on graphs", *SIAM Journal on Computing* **5** (2): 266–283, doi: 10.1137/0205021 (<http://dx.doi.org/10.1137/0205021>).
 - Seymour, P. D.; Weaver, R. W. (1984), "A generalization of chordal graphs", *Journal of Graph Theory* **8** (2): 241–251, doi: 10.1002/jgt.3190080206 (<http://dx.doi.org/10.1002/jgt.3190080206>), MR 742878 (<http://www.ams.org/mathscinet-getitem?mr=742878>).

External links

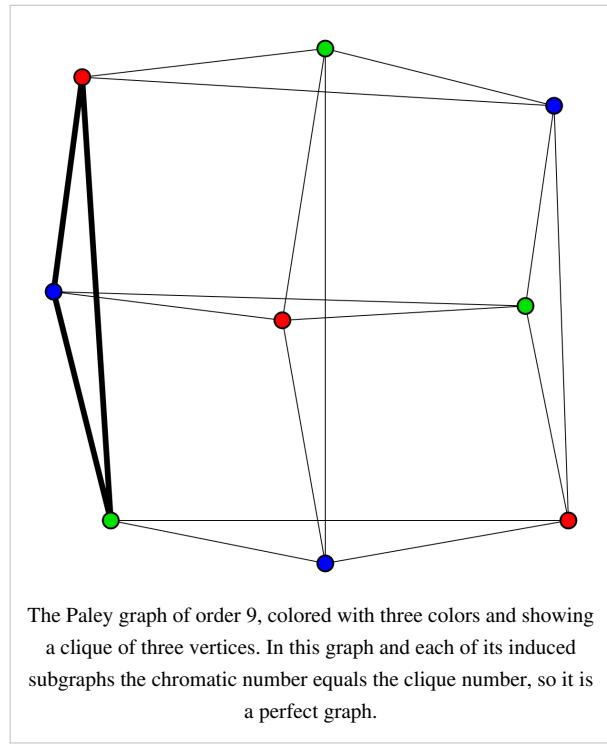
- Information System on Graph Class Inclusions (<http://www.graphclasses.org/index.html>): chordal graph (http://www.graphclasses.org/classes/gc_32.html)
- Weisstein, Eric W., "Chordal Graph" (<http://mathworld.wolfram.com/ChordalGraph.html>), *MathWorld*.

Perfect graph

In graph theory, a **perfect graph** is a graph in which the chromatic number of every induced subgraph equals the size of the largest clique of that subgraph. Due to the strong perfect graph theorem, perfect graphs are the same as **Berge graphs**. A graph G is a Berge graph if neither G nor its complement has an odd-length induced cycle of length 5 or more.

The perfect graphs include many important families of graphs, and serve to unify results relating colorings and cliques in those families. For instance, in all perfect graphs, the graph coloring problem, maximum clique problem, and maximum independent set problem can all be solved in polynomial time. In addition, several important min-max theorems in combinatorics, such as Dilworth's theorem, can be expressed in terms of the perfection of certain associated graphs.

The theory of perfect graphs developed from a 1958 result of Tibor Gallai that in modern language can be interpreted as stating that the complement of a bipartite graph is perfect; this result can also be viewed as a simple equivalent of König's theorem, a much earlier result relating matchings and vertex covers in bipartite graphs. The first use of the phrase "perfect graph" appears to be in a 1963 paper of Claude Berge, after whom Berge graphs are named. In this paper he unified Gallai's result with several similar results by defining perfect graphs, and he conjectured the equivalence of the perfect graph and Berge graph definitions; Berge's conjecture was proved in 2002 as the strong perfect graph theorem.



Families of graphs that are perfect

Some of the more well-known perfect graphs are:

- Bipartite graphs, the graphs that can be colored with two colors, including the forests, graphs with no cycles.
- The line graphs of bipartite graphs (see König's theorem). The rook's graphs (line graphs of complete bipartite graphs) are a special case.
- Chordal graphs, the graphs in which every cycle of four or more vertices has a *chord*, an edge between two vertices that are not consecutive in the cycle. These include the forests, k -trees (maximal graphs with a given treewidth), split graphs (graphs that can be partitioned into a clique and an independent set), block graphs (graphs in which every biconnected component is a clique), interval graphs (graphs in which each vertex represents an interval of a line and each edge represents a nonempty intersection between two intervals), trivially perfect graphs (interval graphs for nested intervals), threshold graphs (graphs in which two vertices are adjacent when their total weight exceeds a numerical threshold), windmill graphs (formed by joining equal cliques at a common vertex), and strongly chordal graphs (chordal graphs in which every even cycle of length six or more has an odd chord).
- Comparability graphs formed from partially ordered sets by connecting pairs of elements by an edge whenever they are related in the partial order. These include the bipartite graphs, the complements of interval graphs, the trivially perfect graphs, the threshold graphs, the windmill graphs, the permutation graphs (graphs in which the edges represent pairs of elements that are reversed by a permutation), and the cographs (graphs formed by recursive operations of disjoint union and complementation).

- Perfectly orderable graphs, the graphs that can be ordered in such a way that a greedy coloring algorithm is optimal on every induced subgraph. These include the bipartite graphs, the chordal graphs, the comparability graphs, the distance-hereditary graphs (in which shortest path distances in connected induced subgraphs equal those in the whole graph), and the wheel graphs that have an odd number of vertices.
- Trapezoid graphs, the intersection graphs of trapezoids whose parallel pairs of edges lie on two parallel lines. These include the interval graphs, trivially perfect graphs, threshold graphs, windmill graphs, and permutation graphs; their complements are a subset of the comparability graphs.

Relation to min-max theorems

In all graphs, the clique number provides a lower bound for the chromatic number, as all vertices in a clique must be assigned distinct colors in any proper coloring. The perfect graphs are those for which this lower bound is tight, not just in the graph itself but in all of its induced subgraphs. For graphs that are not perfect, the chromatic number and clique number can differ; for instance, a cycle of length five requires three colors in any proper coloring but its largest clique has size two.

A proof that a class of graphs is perfect can be seen as a min-max theorem: the minimum number of colors needed for these graphs equals the maximum size of a clique. Many important min-max theorems in combinatorics can be expressed in these terms. For instance, Dilworth's theorem states that the minimum number of chains in a partition of a partially ordered set into chains equals the maximum size of an antichain, and can be rephrased as stating that the complements of comparability graphs are perfect. Mirsky's theorem states that the minimum number of antichains into a partition into antichains equals the maximum size of a chain, and corresponds in the same way to the perfection of comparability graphs.

The perfection of permutation graphs is equivalent to the statement that, in every sequence of ordered elements, the length of the longest decreasing subsequence equals the minimum number of sequences in a partition into increasing subsequences. The Erdős–Szekeres theorem is an easy consequence of this statement.

König's theorem in graph theory states that a minimum vertex cover in a bipartite graph corresponds to a maximum matching, and vice versa; it can be interpreted as the perfection of the complements of bipartite graphs. Another theorem about bipartite graphs, that their chromatic index equals their maximum degree, is equivalent to the perfection of the line graphs of bipartite graphs.

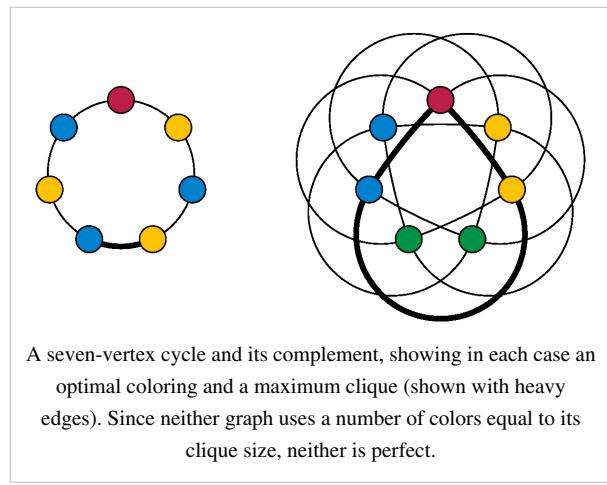
Characterizations and the perfect graph theorems

In his initial work on perfect graphs, Berge made two important conjectures on their structure that were only proved later.

The first of these two theorems was the perfect graph theorem of Lovász (1972), stating that a graph is perfect if and only if its complement is perfect. Thus, perfection (defined as the equality of maximum clique size and chromatic number in every induced subgraph) is equivalent to the equality of maximum independent set size and clique cover number.

The second theorem, conjectured by Berge, provided a forbidden graph characterization of the perfect graphs. An induced cycle of odd length at least 5 is called an **odd hole**. An induced subgraph that is the complement of an odd hole is called an **odd antihole**. An odd cycle of length greater than 3 cannot be perfect, because its chromatic number is three and its clique number is two. Similarly, the complement of an odd cycle of length $2k + 1$ cannot be perfect, because its chromatic number is $k + 1$ and its clique number is k . (Alternatively, the imperfection of this graph follows from the perfect graph theorem and the imperfection of the complementary odd cycle). Because these graphs are not perfect, every perfect graph must be a **Berge graph**, a graph with no odd holes and no odd antiholes. Berge conjectured the converse, that every Berge graph is perfect. This was finally proven as the strong perfect graph theorem of Chudnovsky, Robertson, Seymour, and Thomas (2006).

The perfect graph theorem has a short proof, but the proof of the strong perfect graph theorem is long and technical, based on a deep structural decomposition of Berge graphs. Related decomposition techniques have also borne fruit in the study of other graph classes, and in particular for the claw-free graphs.



Algorithms on perfect graphs

In all perfect graphs, the graph coloring problem, maximum clique problem, and maximum independent set problem can all be solved in polynomial time (Grötschel, Lovász & Schrijver 1988). The algorithm for the general case involves the use of the ellipsoid method for linear programming, but more efficient combinatorial algorithms are known for many special cases.

For many years the complexity of recognizing Berge graphs and perfect graphs remained open. From the definition of Berge graphs, it follows immediately that their recognition is in co-NP (Lovász 1983). Finally, subsequent to the proof of the strong perfect graph theorem, a polynomial time algorithm was discovered by Chudnovsky, Cornuéjols, Liu, Seymour, and Vušković.

References

- Berge, Claude (1961). "Färbung von Graphen, deren sämtliche bzw. deren ungerade Kreise starr sind". *Wiss. Z. Martin-Luther-Univ. Halle-Wittenberg Math.-Natur. Reihe* **10**: 114.
- Berge, Claude (1963). "Perfect graphs". *Six Papers on Graph Theory*. Calcutta: Indian Statistical Institute. pp. 1–21.
- Chudnovsky, Maria; Cornuéjols, Gérard; Liu, Xinming; Seymour, Paul; Vušković, Kristina (2005). "Recognizing Berge graphs". *Combinatorica* **25** (2): 143–186. doi:10.1007/s00493-005-0012-8^[1].
- Chudnovsky, Maria; Robertson, Neil; Seymour, Paul; Thomas, Robin (2006). "The strong perfect graph theorem"^[2]. *Annals of Mathematics* **164** (1): 51–229. doi:10.4007/annals.2006.164.51^[3].
- Gallai, Tibor (1958). "Maximum-minimum Sätze über Graphen". *Acta Math. Acad. Sci. Hungar.* **9** (3-4): 395–434. doi:10.1007/BF02020271^[4].
- Golumbic, Martin Charles (1980). *Algorithmic Graph Theory and Perfect Graphs*^[5]. Academic Press. ISBN 0-444-51530-5. Second edition, Annals of Discrete Mathematics 57, Elsevier, 2004.
- Grötschel, Martin; Lovász, László; Schrijver, Alexander (1988). *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag.. See especially chapter 9, "Stable Sets in Graphs", pp. 273–303.

- Lovász, László (1972). "Normal hypergraphs and the perfect graph conjecture". *Discrete Mathematics* **2** (3): 253–267. doi:10.1016/0012-365X(72)90006-4^[6].
- Lovász, László (1972). "A characterization of perfect graphs". *Journal of Combinatorial Theory, Series B* **13** (2): 95–98. doi:10.1016/0095-8956(72)90045-7^[7].
- Lovász, László (1983). "Perfect graphs". In Beineke, Lowell W.; Wilson, Robin J. (Eds.). *Selected Topics in Graph Theory, Vol. 2*. Academic Press. pp. 55–87. ISBN 0-12-086202-6.

External links

- *The Strong Perfect Graph Theorem*^[8] by Václav Chvátal.
- Open problems on perfect graphs^[9], maintained by the American Institute of Mathematics.
- *Perfect Problems*^[10], maintained by Václav Chvátal.
- Information System on Graph Class Inclusions^[11]: perfect graph^[12]

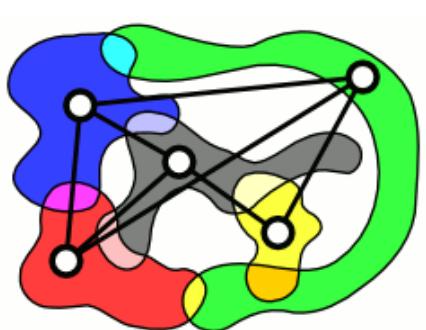
References

- [1] <http://dx.doi.org/10.1007%2Fs00493-005-0012-8>
- [2] <http://annals.princeton.edu/annals/2006/164-1/p02.xhtml>
- [3] <http://dx.doi.org/10.4007%2Fannals.2006.164.51>
- [4] <http://dx.doi.org/10.1007%2FBF02020271>
- [5] http://www.elsevier.com/wps/find/bookdescription.cws_home/699916/description#description
- [6] <http://dx.doi.org/10.1016%2F0012-365X%2872%2990006-4>
- [7] <http://dx.doi.org/10.1016%2F0095-8956%2872%2990045-7>
- [8] <http://www.cs.concordia.ca/~chvatal/perfect/splt.html>
- [9] <http://www.aimath.org/WWN/perfectgraph/>
- [10] <http://www.cs.concordia.ca/~chvatal/perfect/problems.html>
- [11] <http://www.graphclasses.org/index.html>
- [12] http://www.graphclasses.org/classes/gc_56.html

Intersection graph

In the mathematical area of graph theory, an **intersection graph** is a graph that represents the pattern of intersections of a family of sets. Any graph can be represented as an intersection graph, but some important special classes of graphs can be defined by the types of sets that are used to form an intersection representation of them.

For an overview of both the theory of intersection graphs and important special classes of intersection graphs, see McKee & McMorris (1999).



An example of how intersecting sets defines a graph.

Formal definition

Formally, an intersection graph is an undirected graph formed from a family of sets

$$S_i, i = 0, 1, 2, \dots$$

by creating one vertex v_i for each set S_i , and connecting two vertices v_i and v_j by an edge whenever the corresponding two sets have a nonempty intersection, that is,

$$E(G) = \{ \{v_i, v_j\} \mid S_i \cap S_j \neq \emptyset \}.$$

All graphs are intersection graphs

Any undirected graph G may be represented as an intersection graph: for each vertex v_i of G , form a set S_i consisting of the edges incident to v_i ; then two such sets have a nonempty intersection if and only if the corresponding vertices share an edge. Erdős, Goodman & Pósa (1966) provide a construction that is more efficient (which is to say requires a smaller total number of elements in all of the sets S_i combined) in which the total number of set elements is at most $n^2/4$ where n is the number of vertices in the graph. They credit the observation that all graphs are intersection graphs to Szpilrajn-Marczewski (1945), but say to see also Čulík (1964). The intersection number of a graph is the minimum total number of elements in any intersection representation of the graph.

Classes of intersection graphs

Many important graph families can be described as intersection graphs of more restricted types of set families, for instance sets derived from some kind of geometric configuration:

- An interval graph is defined as the intersection graph of intervals on the real line, or of connected subgraphs of a path graph.
- An indifference graph may be defined as the intersection graph of unit intervals on the real line
- A circular arc graph is defined as the intersection graph of arcs on a circle.
- A polygon-circle graph is defined as the intersection of polygons with corners on a circle.
- One characterization of a chordal graph is as the intersection graph of connected subgraphs of a tree.
- A trapezoid graph is defined as the intersection graph of trapezoids formed from two parallel lines. They are a generalization of the notion of permutation graph, in turn they are a special case of the family of the complements of comparability graphs known as cocomparability graphs.
- A unit disk graph is defined as the intersection graph of unit disks in the plane.

- A circle graph is the intersection graph of a set of chords of a circle.
- The circle packing theorem states that planar graphs are exactly the intersection graphs of families of closed disks in the plane bounded by non-crossing circles.
- Scheinerman's conjecture (now a theorem) states that every planar graph can also be represented as an intersection graph of line segments in the plane. However, intersection graphs of line segments may be nonplanar as well, and recognizing intersection graphs of line segments is complete for the existential theory of the reals (Schaefer 2010).
- The line graph of a graph G is defined as the intersection graph of the edges of G , where we represent each edge as the set of its two endpoints.
- A string graph is the intersection graph of curves on a plane.
- A graph has boxicity k if it is the intersection graph of multidimensional boxes of dimension k , but not of any smaller dimension.
- A clique graph is the intersection graph of maximal cliques of another graph
- A block graph of clique tree is the intersection graph of biconnected components of another graph

Scheinerman (1985) characterized the **intersection classes of graphs**, families of finite graphs that can be described as the intersection graphs of sets drawn from a given family of sets. It is necessary and sufficient that the family have the following properties:

- Every induced subgraph of a graph in the family must also be in the family.
- Every graph formed from a graph in the family by replacing a vertex by a clique must also belong to the family.
- There exists an infinite sequence of graphs in the family, each of which is an induced subgraph of the next graph in the sequence, with the property that every graph in the family is an induced subgraph of a graph in the sequence.

If the intersection graph representations have the additional requirement that different vertices must be represented by different sets, then the clique expansion property can be omitted.

Related concepts

An order-theoretic analog to the intersection graphs are the containment orders. In the same way that an intersection representation of a graph labels every vertex with a set so that vertices are adjacent if and only if their sets have nonempty intersection, so a containment representation f of a poset labels every element with a set so that for any x and y in the poset, $x \leq y$ if and only if $f(x) \subseteq f(y)$.

References

- Čulík, K. (1964), "Applications of graph theory to mathematical logic and linguistics", *Theory of Graphs and its Applications (Proc. Sympos. Smolenice, 1963)*, Prague: Publ. House Czechoslovak Acad. Sci., pp. 13–20, MR 0176940^[1].
- Erdős, Paul; Goodman, A. W.; Pósa, Louis (1966), "The representation of a graph by set intersections"^[2], *Canadian Journal of Mathematics* **18** (1): 106–112, doi:10.4153/CJM-1966-014-3^[3], MR 0186575^[4].
- Golumbic, Martin Charles (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, ISBN 0-12-289260-7.
- McKee, Terry A.; McMorris, F. R. (1999), *Topics in Intersection Graph Theory*, SIAM Monographs on Discrete Mathematics and Applications **2**, Philadelphia: Society for Industrial and Applied Mathematics, ISBN 0-89871-430-3, MR 1672910^[5].
- Szpilrajn-Marczewski, E. (1945), "Sur deux propriétés des classes d'ensembles", *Fund. Math.* **33**: 303–307, MR 0015448^[6].
- Schaefer, Marcus (2010), "Complexity of some geometric and topological problems"^[7], *Graph Drawing, 17th International Symposium, GS 2009, Chicago, IL, USA, September 2009, Revised Papers*, Lecture Notes in

- Computer Science **5849**, Springer-Verlag, pp. 334–344, doi:10.1007/978-3-642-11805-0_32^[8], ISBN 978-3-642-11804-3.
- Scheinerman, Edward R. (1985), "Characterizing intersection classes of graphs", *Discrete Mathematics* **55** (2): 185–193, doi:10.1016/0012-365X(85)90047-0^[9], MR 798535^[10].

External links

- Jan Kratochvíl, A video lecture on intersection graphs (June 2007)^[11]
- E. Prisner, A Journey through Intersection Graph County^[12]

References

- [1] <http://www.ams.org/mathscinet-getitem?mr=0176940>
- [2] http://www.renyi.hu/~p_erdos/1966-21.pdf
- [3] <http://dx.doi.org/10.4153%2FCJM-1966-014-3>
- [4] <http://www.ams.org/mathscinet-getitem?mr=0186575>
- [5] <http://www.ams.org/mathscinet-getitem?mr=1672910>
- [6] <http://www.ams.org/mathscinet-getitem?mr=0015448>
- [7] <http://ovid.cs.depaul.edu/documents/convex.pdf>
- [8] http://dx.doi.org/10.1007%2F978-3-642-11805-0_32
- [9] <http://dx.doi.org/10.1016%2F0012-365X%2885%2990047-0>
- [10] <http://www.ams.org/mathscinet-getitem?mr=798535>
- [11] http://videolectures.net/sicgt07_kratochvil_gig/
- [12] <http://www.eprisner.de/Journey/Rahmen.html>

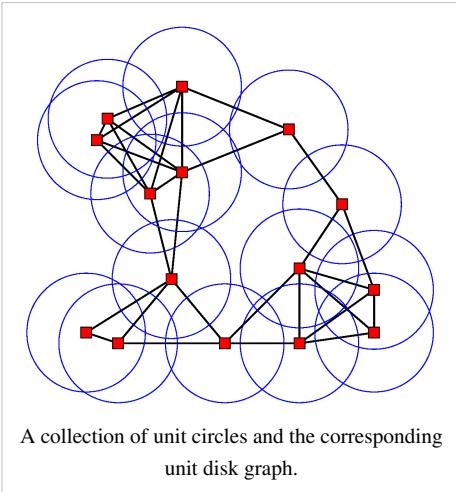
Unit disk graph

In geometric graph theory, a **unit disk graph** is the intersection graph of a family of unit disks in the Euclidean plane. That is, it is a graph with one vertex for each disk, and with an edge between two vertices whenever the corresponding disks have non-empty intersection.

Characterizations

There are several possible definitions of the unit disk graph, equivalent to each other up to a choice of scale factor:

- An intersection graph of equal-radius circles, or of equal-radius disks
- A graph formed from a collection of equal-radius circles, in which two circles are connected by an edge if one circle contains the center of the other circle
- A graph formed from a collection of points in the Euclidean plane, in which two points are connected if their distance is below a fixed threshold



Properties

Every induced subgraph of a unit disk graph is also a unit disk graph. An example of a graph that is not a unit disk graph is the star $K_{1,7}$ with one central node connected to seven leaves: if each of seven unit disks touches a common unit disk, some two of the seven disks must touch each other (as the kissing number in the plane is 6). Therefore, unit disk graphs cannot contain an induced $K_{1,7}$ subgraph.

Applications

Beginning with the work of Huson & Sen (1995), unit disk graphs have been used in computer science to model the topology of ad hoc wireless communication networks. In this application, nodes are connected through a direct wireless connection without a base station. It is assumed that all nodes are homogeneous and equipped with omnidirectional antennas. Node locations are modeled as Euclidean points, and the area within which a signal from one node can be received by another node is modeled as a circle. If all nodes have transmitters of equal power, these circles are all equal. Random geometric graphs, formed as unit disk graphs with randomly generated disk centers, have also been used as a model of percolation and various other phenomena.^[1]

Computational complexity

If one is given a collection of unit disks (or their centers) in a space of any fixed dimension, it is possible to construct the corresponding unit disk graph in linear time, by rounding the centers to nearby integer grid points, using a hash table to find all pairs of centers within constant distance of each other, and filtering the resulting list of pairs for the ones whose circles intersect. The ratio of the number of pairs considered by this algorithm to the number of edges in the eventual graph is a constant, giving the linear time bound. However, this constant grows exponentially as a function of the dimension (Bentley, Stanat & Williams 1977).

It is NP-hard (more specifically, complete for the existential theory of the reals) to determine whether a graph, given without geometry, can be represented as a unit disk graph.^[2] Additionally, it is provably impossible in polynomial time to output explicit coordinates of a unit disk graph representation: there exist unit disk graphs that require exponentially many bits of precision in any such representation.

However, many important and difficult graph optimization problems such as maximum independent set, graph coloring, and minimum dominating set can be approximated efficiently by using the geometric structure of these graphs,^[3] and the maximum clique problem can be solved exactly for these graphs in polynomial time, given a disk representation. More strongly, if a graph is given as input, it is possible in polynomial time to produce either a maximum clique or a proof that the graph is not a unit disk graph.

When a given vertex set forms a subset of a triangular lattice, a necessary and sufficient condition for the perfectness of a unit graph is known. For the perfect graphs, a number of NP-complete optimization problems (graph coloring problem, maximum clique problem, and maximum independent set problem) are polynomially solvable.

Notes

[1] See, e.g., .

[2] ; .

[3] ; .

References

- Bentley, Jon L.; Stanat, Donald F.; Williams, E. Hollins, Jr. (1977), "The complexity of finding fixed-radius near neighbors", *Information Processing Letters* **6** (6): 209–212, doi: 10.1016/0020-0190(77)90070-9 ([http://dx.doi.org/10.1016/0020-0190\(77\)90070-9](http://dx.doi.org/10.1016/0020-0190(77)90070-9)), MR 0489084 (<http://www.ams.org/mathscinet-getitem?mr=0489084>).

- Breu, Heinz; Kirkpatrick, David G. (1998), "Unit disk graph recognition is NP-hard", *Computational Geometry Theory and Applications* **9** (1–2): 3–24, doi: 10.1016/s0925-7721(97)00014-x ([http://dx.doi.org/10.1016/s0925-7721\(97\)00014-x](http://dx.doi.org/10.1016/s0925-7721(97)00014-x)).
- Clark, Brent N.; Colbourn, Charles J.; Johnson, David S. (1990), "Unit disk graphs", *Discrete Mathematics* **86** (1–3): 165–177, doi: 10.1016/0012-365X(90)90358-O ([http://dx.doi.org/10.1016/0012-365X\(90\)90358-O](http://dx.doi.org/10.1016/0012-365X(90)90358-O)).
- Dall, Jesper; Christensen, Michael (2002), "Random geometric graphs", *Phys. Rev. E* **66**: 016121, arXiv: cond-mat/0203026 (<http://arxiv.org/abs/cond-mat/0203026>), doi: 10.1103/PhysRevE.66.016121 (<http://dx.doi.org/10.1103/PhysRevE.66.016121>).
- Huson, Mark L.; Sen, Arunabha (1995), "Broadcast scheduling algorithms for radio networks", *Military Communications Conference, IEEE MILCOM '95* **2**, pp. 647–651, doi: 10.1109/MILCOM.1995.483546 (<http://dx.doi.org/10.1109/MILCOM.1995.483546>), ISBN 0-7803-2489-7.
- Kang, Ross J.; Müller, Tobias (2011), "Sphere and dot product representations of graphs", *Proceedings of the Twenty-Seventh Annual Symposium on Computational Geometry (SCG'11), June 13–15, 2011, Paris, France*, pp. 308–314.
- Marathe, Madhav V.; Breu, Heinz; Hunt, III, Harry B.; Ravi, S. S.; Rosenkrantz, Daniel J. (1994), *Geometry based heuristics for unit disk graphs*, arXiv: math.CO/9409226 (<http://arxiv.org/abs/math.CO/9409226>).
- Matsui, Tomomi (2000), "Approximation Algorithms for Maximum Independent Set Problems and Fractional Coloring Problems on Unit Disk Graphs", *Lecture Notes in Computer Science*, Lecture Notes in Computer Science **1763**: 194–200, doi: 10.1007/978-3-540-46515-7_16 (http://dx.doi.org/10.1007/978-3-540-46515-7_16), ISBN 978-3-540-67181-7.
- McDiarmid, Colin; Mueller, Tobias (2011), *Integer realizations of disk and segment graphs*, arXiv: 1111.2931 (<http://arxiv.org/abs/1111.2931>)
- Miyamoto, Yuichiro; Matsui, Tomomi (2005), "Perfectness and Imperfectness of the kth Power of Lattice Graphs", *Lecture Notes in Computer Science*, Lecture Notes in Computer Science **3521**: 233–242, doi: 10.1007/11496199_26 (http://dx.doi.org/10.1007/11496199_26), ISBN 978-3-540-26224-4.
- Raghavan, Vijay; Spinrad, Jeremy (2003), "Robust algorithms for restricted domains", *Journal of Algorithms* **48** (1): 160–172, doi: 10.1016/S0196-6774(03)00048-8 ([http://dx.doi.org/10.1016/S0196-6774\(03\)00048-8](http://dx.doi.org/10.1016/S0196-6774(03)00048-8)), MR 2006100 (<http://www.ams.org/mathscinet-getitem?mr=2006100>).

Line graph

This article is about the mathematical concept. For statistical presentation method, see line chart.

Not to be confused with linear graph.

In the mathematical discipline of graph theory, the **line graph** of an undirected graph G is another graph $L(G)$ that represents the adjacencies between edges of G . The name line graph comes from a paper by Harary & Norman (1960) although both Whitney (1932) and Krausz (1943) used the construction before this.^[1] Other terms used for the line graph include the **theta-obrazom**, the **covering graph**, the **derivative**, the **edge-to-vertex dual**, the **conjugate**, and the **representative graph**,^[1] as well as the **edge graph**, the **interchange graph**, the **adjoint graph**, and the **derived graph**.^[2]

Hassler Whitney (1932) proved that with one exceptional case the structure of a connected graph G can be recovered completely from its line graph. Many other properties of line graphs follow by translating the properties of the underlying graph from vertices into edges, and by Whitney's theorem the same translation can also be done in the other direction. Line graphs are claw-free, and the line graphs of bipartite graphs are perfect. Line graphs can be characterized by nine forbidden subgraphs, and can be recognized in linear time.

Various generalizations of line graphs have also been studied, including the line graphs of line graphs, line graphs of multigraphs, line graphs of hypergraphs, and line graphs of weighted graphs.

Formal definition

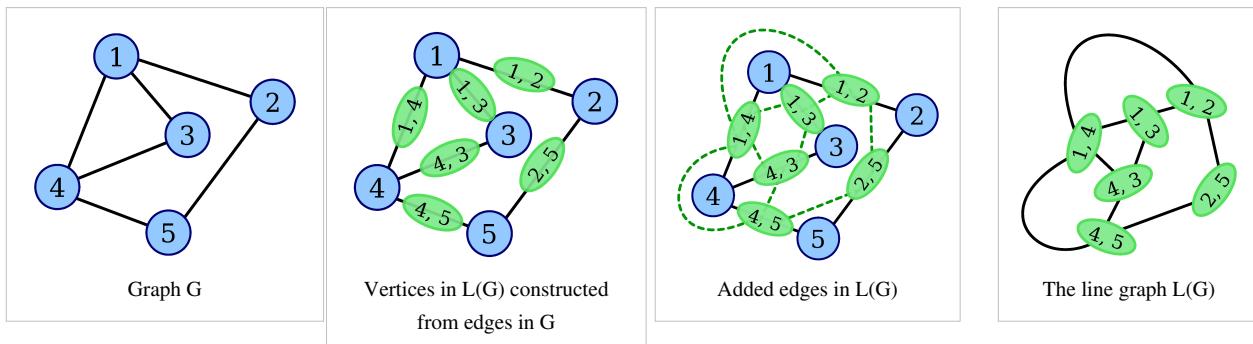
Given a graph G , its line graph $L(G)$ is a graph such that

- each vertex of $L(G)$ represents an edge of G ; and
- two vertices of $L(G)$ are adjacent if and only if their corresponding edges share a common endpoint ("are incident") in G .

That is, it is the intersection graph of the edges of G , representing each edge by the set of its two endpoints.

Example

The following figures show a graph (left, with blue vertices) and its line graph (right, with green vertices). Each vertex of the line graph is shown labeled with the pair of endpoints of the corresponding edge in the original graph. For instance, the green vertex on the right labeled 1,3 corresponds to the edge on the left between the blue vertices 1 and 3. Green vertex 1,3 is adjacent to three other green vertices: 1,4 and 1,2 (corresponding to edges sharing the endpoint 1 in the blue graph) and 4,3 (corresponding to an edge sharing the endpoint 3 in the blue graph).



Properties

Translated properties of the underlying graph

Properties of a graph G that depend only on adjacency between edges may be translated into equivalent properties in $L(G)$ that depend on adjacency between vertices. For instance, a matching in G is a set of edges no two of which are adjacent, and corresponds to a set of vertices in $L(G)$ no two of which are adjacent, that is, an independent set.

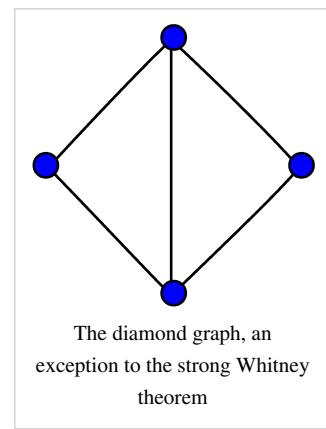
Thus,

- The line graph of a connected graph is connected. If G is connected, it contains a path connecting any two of its edges, which translates into a path in $L(G)$ containing any two of the vertices of $L(G)$. However, a graph G that has some isolated vertices, and is therefore disconnected, may nevertheless have a connected line graph.^[3]
- A line graph has an articulation point if and only if the underlying graph has a bridge for which neither endpoint has degree one.
- For a graph G with n vertices and m edges, the number of vertices of the line graph $L(G)$ is m , and the number of edges of $L(G)$ is half the sum of the squares of the degrees of the vertices in G , minus m .^[4]
- A maximum independent set in a line graph corresponds to maximum matching in the original graph. Since maximum matchings may be found in polynomial time, so may the maximum independent sets of line graphs, despite the hardness of the maximum independent set problem for more general families of graphs.
- The edge chromatic number of a graph G is equal to the vertex chromatic number of its line graph $L(G)$.^[5]
- The line graph of an edge-transitive graph is vertex-transitive. This property can be used to generate families of graphs that (like the Petersen graph) are vertex-transitive but are not Cayley graphs: if G is an edge-transitive graph that has at least five vertices, is not bipartite, and has odd vertex degrees, then $L(G)$ is a vertex-transitive non-Cayley graph.^[6]
- If a graph G has an Euler cycle, that is, if G is connected and has an even number of edges at each vertex, then the line graph of G is Hamiltonian. However, not all Hamiltonian cycles in line graphs come from Euler cycles in this way; for instance, the line graph of a Hamiltonian graph G is itself Hamiltonian, regardless of whether G is also Eulerian.^[7]
- If two simple graphs are isomorphic then their line graphs are also isomorphic. The Whitney graph isomorphism theorem provides a converse to this.
- In the context of complex network theory, the line graph of a random network preserves many of the properties of the network such as the small-world property (the existence of short paths between all pairs of vertices) and the shape of its degree distribution.^[8] Evans & Lambiotte (2009) observe that any method for finding vertex clusters in a complex network can be applied to the line graph and used to cluster its edges instead.

Whitney isomorphism theorem

If the line graphs of two connected graphs are isomorphic, then the underlying graphs are isomorphic, except in the case of the triangle graph K_3 and the claw $K_{1,3}$, which have isomorphic line graphs but are not themselves isomorphic.^[1]

As well as K_3 and $K_{1,3}$, there are some other exceptional small graphs with the property that their line graph has a higher degree of symmetry than the graph itself. For instance, the diamond graph $K_{1,1,2}$ (two triangles sharing an edge) has four graph automorphisms but its line graph $K_{1,2,2}$ has eight. In the illustration of the diamond graph shown, rotating the graph by 90 degrees is not a symmetry of the graph, but is a symmetry of its line graph. However, all such exceptional cases have at most four vertices. A strengthened version of the Whitney isomorphism theorem states that, for connected graphs with more than four vertices, there is a one-to-one correspondence between isomorphisms of the graphs and isomorphisms of their line graphs.^[9]

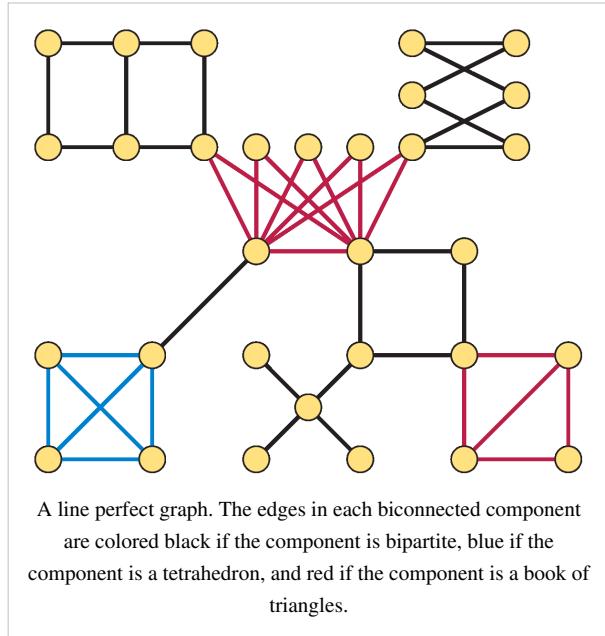


Analogues of the Whitney isomorphism theorem have been proven for the line graphs of multigraphs, but are more complicated in this case.

Strongly regular and perfect line graphs

The line graph of the complete graph K_n is also known as the **triangular graph**, the Johnson graph $J(n,2)$, or the complement of the Kneser graph $KG_{n,2}$. Triangular graphs are characterized by their spectra, except for $n = 8$.^[10] They may also be characterized (again with the exception of K_8) as the strongly regular graphs with parameters $srg(n(n-1)/2, 2(n-2), n-2, 4)$.^[11] The three strongly regular graphs with the same parameters and spectrum as $L(K_8)$ are the Chang graphs, which may be obtained by graph switching from $L(K_8)$.

The line graph of a bipartite graph is perfect (see König's theorem). The line graphs of bipartite graphs form one of the key building blocks of perfect graphs, used in the proof of the strong perfect graph theorem.^[12] A special case of these graphs are the rook's graphs, line graphs of complete bipartite graphs. Like the line graphs of complete graphs, they can be characterized with one exception by their numbers of vertices, numbers of edges, and number of shared neighbors for adjacent and non-adjacent points. The one exceptional case is $L(K_{4,4})$, which shares its parameters with the Shrikhande graph. When both sides of the bipartition have the same number of vertices, these graphs are again strongly regular.^[13]



More generally, a graph G is said to be **line perfect** if $L(G)$ is a perfect graph. The line perfect graphs are exactly the graphs that do not contain a simple cycle of odd length greater than three.^[14] Equivalently, a graph is line perfect if and only if each of its biconnected components is either bipartite or of the form K_4 (the tetrahedron) or $K_{1,1,n}$ (a book of one or more triangles all sharing a common edge).^[15] Every line perfect graph is itself perfect.^[16]

Other related graph families

All line graphs are claw-free graphs, graphs without an induced subgraph in the form of a three-leaf tree. As with claw-free graphs more generally, every connected line graph $L(G)$ with an even number of edges has a perfect matching;^[17] equivalently, this means that if the underlying graph G has an even number of edges, its edges can be partitioned into two-edge paths.

The line graphs of trees are exactly the claw-free block graphs.^[18] These graphs have been used to solve a problem in extremal graph theory, of constructing a graph with a given number of edges and vertices whose largest tree induced as a subgraph is as small as possible.

All eigenvalues of the adjacency matrix of a line graph are at least -2 . For this reason, the graphs whose eigenvalues have this property have been called generalized line graphs.^[19]

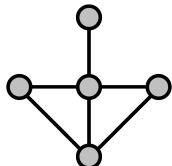
Characterization and recognition

Clique partition

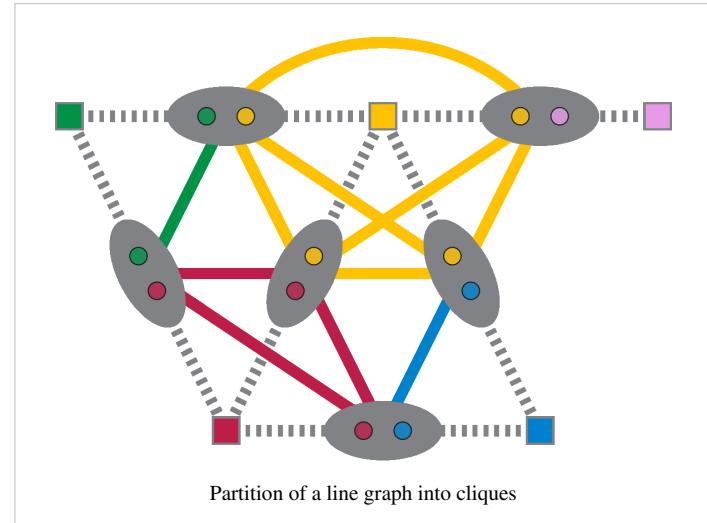
For an arbitrary graph G , and an arbitrary vertex v in G , the set of edges incident to v corresponds to a clique in the line graph $L(G)$. The cliques formed in this way partition the edges of $L(G)$. Each vertex of $L(G)$ belongs to exactly two of them (the two cliques corresponding to the two endpoints of the corresponding edge in G).

The existence of such a partition into cliques can be used to characterize the line graphs: A graph L is the line graph of some other graph or multigraph if and only if it is possible to find a collection of cliques in L (allowing some of the cliques to be single vertices) that partition the edges of L , such that each vertex of L belongs to exactly two of the cliques.^[1] It is the line graph of a graph (rather than a multigraph) if this set of cliques satisfies the additional condition that no two vertices of L are both in the same two cliques. Given such a family of cliques, the underlying graph G for which L is the line graph can be recovered by making one vertex in G for each clique, and an edge in G for each vertex in L with its endpoints being the two cliques containing the vertex in L . By the strong version of Whitney's isomorphism theorem, if the underlying graph G has more than four vertices, there can be only one partition of this type.

For example, this characterization can be used to show that the following graph is not a line graph:

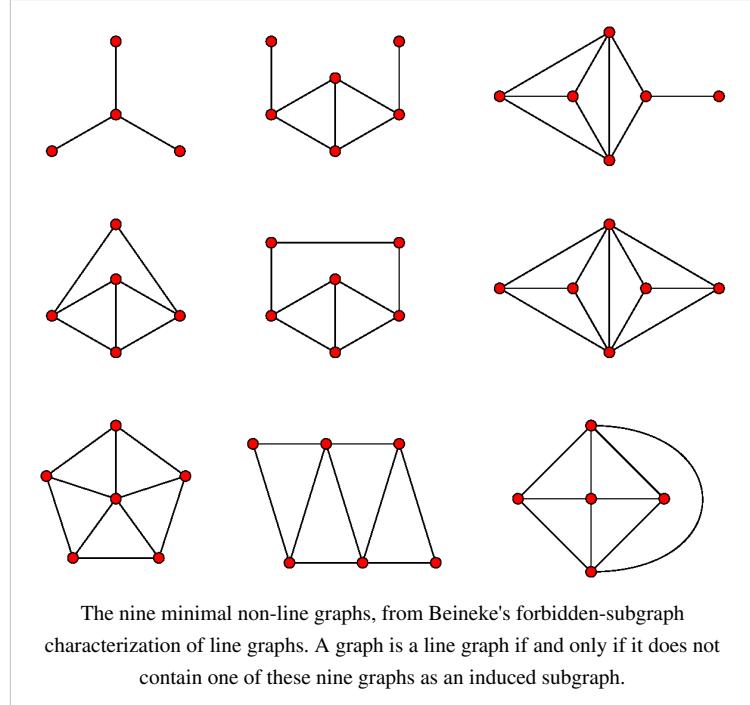


In this example, the edges going upward, to the left, and to the right from the central degree-four vertex do not have any cliques in common. Therefore, any partition of the graph's edges into cliques would have to have at least one clique for each of these three edges, and these three cliques would all intersect in that central vertex, violating the requirement that each vertex appear in exactly two cliques. Thus, the graph shown is not a line graph.



Forbidden subgraphs

An alternative characterization of line graphs was proven by Beineke (1970) (and reported earlier without proof by Beineke (1968)). He showed that there are nine minimal graphs that are not line graphs, such that any graph that is not a line graph has one of these nine graphs as an induced subgraph. That is, a graph is a line graph if and only if no subset of its vertices induces one of these nine graphs. In the example above, the four topmost vertices induce a claw (that is, a complete bipartite graph $K_{1,3}$), shown on the top left of the illustration of forbidden subgraphs. Therefore, by Beineke's characterization, this example cannot be a line graph. For graphs with minimum degree at least 5, only the six subgraphs in the left and right columns of the figure are needed in the characterization. Line graphs of multigraphs may be similarly characterized by three of Beineke's nine forbidden subgraphs.^[20]



Algorithms

Roussopoulos (1973) and Lehot (1974) described linear time algorithms for recognizing line graphs and reconstructing their original graphs. Sysło (1982) generalized these methods to directed graphs. Degiorgi & Simon (1995) described an efficient data structure for maintaining a dynamic graph, subject to vertex insertions and deletions, and maintaining a representation of the input as a line graph (when it exists) in time proportional to the number of changed edges at each step.

The algorithms of Roussopoulos (1973) and Lehot (1974) are based on characterizations of line graphs involving odd triangles (triangles in the line graph with the property that there exists another vertex adjacent to an odd number of triangle vertices). However, the algorithm of Degiorgi & Simon (1995) uses only Whitney's isomorphism theorem. It is complicated by the need to recognize deletions that cause the remaining graph to become a line graph, but when specialized to the static recognition problem only insertions need to be performed, and the algorithm performs the following steps:

- Construct the input graph L by adding vertices one at a time, at each step choosing a vertex to add that is adjacent to at least one previously-added vertex. While adding vertices to L , maintain a graph G for which $L = L(G)$; if the algorithm ever fails to find an appropriate graph G , then the input is not a line graph and the algorithm terminates.
- When adding a vertex v to a graph $L(G)$ for which G has four or fewer vertices, it might be the case that the line graph representation is not unique. But in this case, the augmented graph is small enough that a representation of it as a line graph can be found by a brute force search in constant time.
- When adding a vertex v to a larger graph L that equals the line graph of another graph G , let S be the subgraph of G formed by the edges that correspond to the neighbors of v in L . Check that S has a vertex cover consisting of one vertex or two non-adjacent vertices. If there are two vertices in the cover, augment G by adding an edge (corresponding to v) that connects these two vertices. If there is only one vertex in the cover, then add a new vertex to G , adjacent to this vertex.

Each step either takes constant time, or involves finding a vertex cover of constant size within a graph S whose size is proportional to the number of neighbors of v . Thus, the total time for the whole algorithm is proportional to the sum of the numbers of neighbors of all vertices, which (by the handshaking lemma) is proportional to the number of input edges.

Iterating the line graph operator

van Rooij & Wilf (1965) consider the sequence of graphs

$$G, L(G), L(L(G)), L(L(L(G))), \dots$$

They show that, when G is a finite connected graph, only four possible behaviors are possible for this sequence:

- If G is a cycle graph then $L(G)$ and each subsequent graph in this sequence is isomorphic to G itself. These are the only connected graphs for which $L(G)$ is isomorphic to G .^[21]
- If G is a claw $K_{1,3}$, then $L(G)$ and all subsequent graphs in the sequence are triangles.
- If G is a path graph then each subsequent graph in the sequence is a shorter path until eventually the sequence terminates with an empty graph.
- In all remaining cases, the sizes of the graphs in this sequence eventually increase without bound.

If G is not connected, this classification applies separately to each component of G .

For connected graphs that are not paths, all sufficiently high numbers of iteration of the line graph operation produce graphs that are Hamiltonian.^[22]

Generalizations

Medial graphs and convex polyhedra

Main article: Medial graph

When a planar graph G has maximum vertex degree three, its line graph is planar, and every planar embedding of G can be extended to an embedding of $L(G)$. However, there exist planar graphs with higher degree whose line graphs are nonplanar. These include, for example, the 5-star $K_{1,5}$, the gem graph formed by adding two non-crossing diagonals within a regular pentagon, and all convex polyhedra with a vertex of degree four or more.^[23]

An alternative construction, the medial graph, coincides with the line graph for planar graphs with maximum degree three, but is always planar. It has the same vertices as the line graph, but potentially fewer edges: two vertices of the medial graph are adjacent if and only if the corresponding two edges are consecutive on some face of the planar embedding. The medial graph of the dual graph of a plane graph is the same as the medial graph of the original plane graph.

For regular polyhedra or simple polyhedra, the medial graph operation can be represented geometrically by the operation of cutting off each vertex of the polyhedron by a plane through the midpoints of all its incident edges. This operation is known variously as the second truncation, degenerate truncation, or rectification.

Total graphs

The total graph $T(G)$ of a graph G has as its vertices the elements (vertices or edges) of G , and has an edge between two elements whenever they are either incident or adjacent. The total graph may also be obtained by subdividing each edge of G and then taking the square of the subdivided graph.^[24]

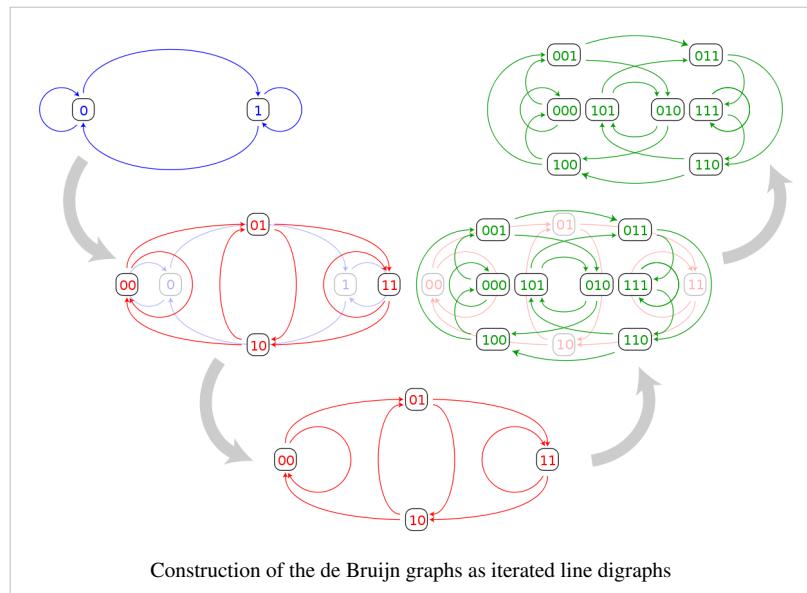
Multigraphs

The concept of the line graph of G may naturally be extended to the case where G is a multigraph. In this case, the characterizations of these graphs can be simplified: the characterization in terms of clique partitions no longer needs to prevent two vertices from belonging to the same two cliques, and the characterization by forbidden graphs has fewer forbidden graphs.

However, for multigraphs, there are larger numbers of pairs of non-isomorphic graphs that have the same line graphs. For instance a complete bipartite graph $K_{1,n}$ has the same line graph as the dipole graph and Shannon multigraph with the same number of edges. Nevertheless, analogues to Whitney's isomorphism theorem can still be derived in this case.

Line digraphs

It is also possible to generalize line graphs to directed graphs.^[25] If G is a directed graph, its **directed line graph** or **line digraph** has one vertex for each edge of G . Two vertices representing directed edges from u to v and from w to x in G are connected by an edge from uv to wx in the line digraph when $v = w$. That is, each edge in the line digraph of G represents a length-two directed path in G . The de Bruijn graphs may be formed by repeating this process of forming directed line graphs, starting from a complete directed graph.^[26]



Weighted line graphs

In a line graph $L(G)$, each vertex of degree k in the original graph G creates $k(k-1)/2$ edges in the line graph. For many types of analysis this means high degree nodes in G are over represented in the line graph $L(G)$. For instance consider a random walk on the vertices of the original graph G . This will pass along some edge e with some frequency f . On the other hand this edge e is mapped to a unique vertex, say v , in the line graph $L(G)$. If we now perform the same type of random walk on the vertices of the line graph, the frequency with which v is visited can be completely different from f . If our edge e in G was connected to nodes of degree $O(k)$, it will be traversed $O(k^2)$ more frequently in the line graph $L(G)$. Put another way, the Whitney graph isomorphism theorem guarantees that the line graph almost always encodes the topology of the original graph G faithfully but it does not guarantee that dynamics on these two graphs have a simple relationship. One solution is to construct a weighted line graph, that is, a line graph with weighted edges. There are several natural ways to do this.^[27] For instance if edges d and e in the graph G are incident at a vertex v with degree k , then in the line graph $L(G)$ the edge connecting the two vertices d and e can be given weight $1/(k-1)$. In this way every edge in G (provided neither end is connected to a vertex of

degree '1') will have strength 2 in the line graph $L(G)$ corresponding to the two ends that the edge has in G . It is straightforward to extend this definition of a weighted line graph to cases where the original graph G was directed or even weighted.^[28] The principle in all cases is to ensure the line graph $L(G)$ reflects the dynamics as well as the topology of the original graph G .

Line graphs of hypergraphs

Main article: Line graph of a hypergraph

The edges of a hypergraph may form an arbitrary family of sets, so the line graph of a hypergraph is the same as the intersection graph of the sets from the family.

Notes

- [1] Hemminger & Beineke (1978), p. 273.
- [2] , p. 71.
- [3] The need to consider isolated vertices when considering the connectivity of line graphs is pointed out by , p. 32 (<http://books.google.com/books?id=FA8SOBZcbs4C&pg=PA32>).
- [4] , Theorem 8.1, p. 72.
- [5] . Also in free online edition (<http://diestel-graph-theory.com/basic.html>), Chapter 5 ("Colouring"), p. 118.
- [6] . Lauri and Scapellato credit this result to Mark Watkins.
- [7] , Theorem 8.8, p. 80.
- [8] Ramezanpour, Karimipour & Mashaghi (2003).
- [9] ;.
- [10] . See in particular Proposition 8, p. 262.
- [11] , Theorem 8.6, p. 79. Harary credits this result to independent papers by L. C. Chang (1959) and A. J. Hoffman (1960).
- [12] . See also .
- [13] , Theorem 8.7, p. 79. Harary credits this characterization of line graphs of complete bipartite graphs to Moon and Hoffman. The case of equal numbers of vertices on both sides had previously been proven by Shrikhande.
- [14] ;.
- [15] Maffray (1992).
- [16] Trotter (1977).
- [17] ..
- [18] , Theorem 8.5, p. 78. Harary credits the result to Gary Chartrand.
- [19] Cvetković, Rowlinson & Simić (2004).
- [20] , Exercise 8.14, p. 83.
- [21] This result is also Theorem 8.2 of .
- [22] , Theorem 8.11, p. 81. Harary credits this result to Gary Chartrand.
- [23] ;.
- [24] , p. 82.
- [25] Harary & Norman (1960).
- [26] Zhang & Lin (1987).
- [27] Evans & Lambiotte (2009).
- [28] Evans & Lambiotte (2010).

References

- Beineke, L. W. (1968), "Derived graphs of digraphs", in Sachs, H.; Voss, H.-J.; Walter, H.-J., *Beiträge zur Graphentheorie*, Leipzig: Teubner, pp. 17–33.
- Beineke, L. W. (1970), "Characterizations of derived graphs", *Journal of Combinatorial Theory* **9** (2): 129–135, doi: [10.1016/S0021-9800\(70\)80019-9](https://dx.doi.org/10.1016/S0021-9800(70)80019-9) ([http://dx.doi.org/10.1016/S0021-9800\(70\)80019-9](http://dx.doi.org/10.1016/S0021-9800(70)80019-9)), MR 0262097 (<http://www.ams.org/mathscinet-getitem?mr=0262097>).
- Cvetković, Dragoš; Rowlinson, Peter; Simić, Slobodan (2004), *Spectral generalizations of line graphs*, London Mathematical Society Lecture Note Series **314**, Cambridge: Cambridge University Press, doi: [10.1017/CBO9780511751752](https://dx.doi.org/10.1017/CBO9780511751752) (<http://dx.doi.org/10.1017/CBO9780511751752>), ISBN 0-521-83663-8, MR 2120511 (<http://www.ams.org/mathscinet-getitem?mr=2120511>).

- Degiorgi, Daniele Giorgio; Simon, Klaus (1995), "A dynamic algorithm for line graph recognition", *Graph-theoretic concepts in computer science (Aachen, 1995)*, Lecture Notes in Computer Science **1017**, Berlin: Springer, pp. 37–48, doi: 10.1007/3-540-60618-1_64 (http://dx.doi.org/10.1007/3-540-60618-1_64), MR 1400011 (<http://www.ams.org/mathscinet-getitem?mr=1400011>).
- Evans, T.S.; Lambiotte, R. (2009), "Line graphs, link partitions and overlapping communities", *Physical Review E* **80**: 016105, arXiv: 0903.2181 (<http://arxiv.org/abs/0903.2181>), doi: 10.1103/PhysRevE.80.016105 (<http://dx.doi.org/10.1103/PhysRevE.80.016105>).
- Evans, T.S.; Lambiotte, R. (2010), "Line Graphs of Weighted Networks for Overlapping Communities", *European Physical Journal B* **77**: 265–272, arXiv: 0912.4389 (<http://arxiv.org/abs/0912.4389>), doi: 10.1140/epjb/e2010-00261-8 (<http://dx.doi.org/10.1140/epjb/e2010-00261-8>).
- Greenwell, D. L.; Hemminger, Robert L. (1972), "Forbidden subgraphs for graphs with planar line graphs", *Discrete Mathematics* **2**: 31–34, doi: 10.1016/0012-365X(72)90058-1 ([http://dx.doi.org/10.1016/0012-365X\(72\)90058-1](http://dx.doi.org/10.1016/0012-365X(72)90058-1)), MR 0297604 (<http://www.ams.org/mathscinet-getitem?mr=0297604>).
- Harary, F.; Norman, R. Z. (1960), "Some properties of line digraphs", *Rendiconti del Circolo Matematico di Palermo* **9** (2): 161–169, doi: 10.1007/BF02854581 (<http://dx.doi.org/10.1007/BF02854581>).
- Harary, F. (1972), "8. Line Graphs" (<http://www.dtic.mil/dtic/tr/fulltext/u2/705364.pdf>), *Graph Theory*, Massachusetts: Addison-Wesley, pp. 71–83.
- Hemminger, R. L.; Beineke, L. W. (1978), "Line graphs and line digraphs", in Beineke, L. W.; Wilson, R. J., *Selected Topics in Graph Theory*, Academic Press Inc., pp. 271–305.
- Jung, H. A. (1966), "Zu einem Isomorphiesatz von H. Whitney für Graphen", *Mathematische Annalen* (in German) **164**: 270–271, doi: 10.1007/BF01360250 (<http://dx.doi.org/10.1007/BF01360250>), MR 0197353 (<http://www.ams.org/mathscinet-getitem?mr=0197353>).
- Krausz, J. (1943), "Démonstration nouvelle d'un théorème de Whitney sur les réseaux", *Mat. Fiz. Lapok* **50**: 75–85, MR 0018403 (<http://www.ams.org/mathscinet-getitem?mr=0018403>).
- Lehot, Philippe G. H. (1974), "An optimal algorithm to detect a line graph and output its root graph", *Journal of the ACM* **21**: 569–575, doi: 10.1145/321850.321853 (<http://dx.doi.org/10.1145/321850.321853>), MR 0347690 (<http://www.ams.org/mathscinet-getitem?mr=0347690>).
- Maffray, Frédéric (1992), "Kernels in perfect line-graphs", *Journal of Combinatorial Theory, Series B* **55** (1): 1–8, doi: 10.1016/0095-8956(92)90028-V ([http://dx.doi.org/10.1016/0095-8956\(92\)90028-V](http://dx.doi.org/10.1016/0095-8956(92)90028-V)), MR 1159851 (<http://www.ams.org/mathscinet-getitem?mr=1159851>).
- Metelsky, Yury; Tyshkevich, Regina (1997), "On line graphs of linear 3-uniform hypergraphs", *Journal of Graph Theory* **25** (4): 243–251, doi: 10.1002/(SICI)1097-0118(199708)25:4<243::AID-JGT1>3.0.CO;2-K ([http://dx.doi.org/10.1002/\(SICI\)1097-0118\(199708\)25:4<243::AID-JGT1>3.0.CO;2-K](http://dx.doi.org/10.1002/(SICI)1097-0118(199708)25:4<243::AID-JGT1>3.0.CO;2-K)).
- Ramezanpour, A.; Karimipour, V.; Mashaghi, A. (2003), "Generating correlated networks from uncorrelated ones" (<http://pre.aps.org/abstract/PRE/v67/i4/e046107>), *Phys. Rev. E* **67**: 046107, doi: 10.1103/physreve.67.046107 (<http://dx.doi.org/10.1103/physreve.67.046107>).
- van Rooij, A. C. M.; Wilf, H. S. (1965), "The interchange graph of a finite graph", *Acta Mathematica Hungarica* **16** (3–4): 263–269, doi: 10.1007/BF01904834 (<http://dx.doi.org/10.1007/BF01904834>).
- Roussopoulos, N. D. (1973), "A max { m,n } algorithm for determining the graph H from its line graph G ", *Information Processing Letters* **2** (4): 108–112, doi: 10.1016/0020-0190(73)90029-X ([http://dx.doi.org/10.1016/0020-0190\(73\)90029-X](http://dx.doi.org/10.1016/0020-0190(73)90029-X)), MR 0424435 (<http://www.ams.org/mathscinet-getitem?mr=0424435>).
- Sedláček, J. (1964), "Some properties of interchange graphs", *Theory of Graphs and its Applications (Proc. Sympos. Smolenice, 1963)*, Publ. House Czechoslovak Acad. Sci., Prague, pp. 145–150, MR 0173255 (<http://www.ams.org/mathscinet-getitem?mr=0173255>).
- Sysło, Maciej M. (1982), "A labeling algorithm to recognize a line digraph and output its root graph", *Information Processing Letters* **15** (1): 28–30, doi: 10.1016/0020-0190(82)90080-1 ([http://dx.doi.org/10.1016/0020-0190\(82\)90080-1](http://dx.doi.org/10.1016/0020-0190(82)90080-1)), MR 678028 (<http://www.ams.org/mathscinet-getitem?mr=678028>).

- Trotter, L. E., Jr. (1977), "Line perfect graphs", *Mathematical Programming* **12** (2): 255–259, doi: 10.1007/BF01593791 (<http://dx.doi.org/10.1007/BF01593791>), MR 0457293 (<http://www.ams.org/mathscinet-getitem?mr=0457293>).
- de Werra, D. (1978), "On line perfect graphs", *Mathematical Programming* **15** (2): 236–238, doi: 10.1007/BF01609025 (<http://dx.doi.org/10.1007/BF01609025>), MR 509968 (<http://www.ams.org/mathscinet-getitem?mr=509968>).
- Whitney, H. (1932), "Congruent graphs and the connectivity of graphs", *American Journal of Mathematics* **54** (1): 150–168, doi: 10.2307/2371086 (<http://dx.doi.org/10.2307/2371086>), JSTOR 2371086 (<http://www.jstor.org/stable/2371086>).
- Zhang, Fu Ji; Lin, Guo Ning (1987), "On the de Bruijn–Good graphs", *Acta Math. Sinica* **30** (2): 195–205, MR 0891925 (<http://www.ams.org/mathscinet-getitem?mr=0891925>).
- Зверович, И. Э. (1997), "Аналог теоремы Уитни для реберных графов мультиграфов и реберные мультиграфы", *Diskretnaya Matematika* (in Russian) **9** (2): 98–105, doi: 10.4213/dm478 (<http://dx.doi.org/10.4213/dm478>), MR 1468075 (<http://www.ams.org/mathscinet-getitem?mr=1468075>). Translated into English as Zverovich, I. È. (1997), "An analogue of the Whitney theorem for edge graphs of multigraphs, and edge multigraphs", *Discrete Mathematics and Applications* **7** (3): 287–294, doi: 10.1515/dma.1997.7.3.287 (<http://dx.doi.org/10.1515/dma.1997.7.3.287>).

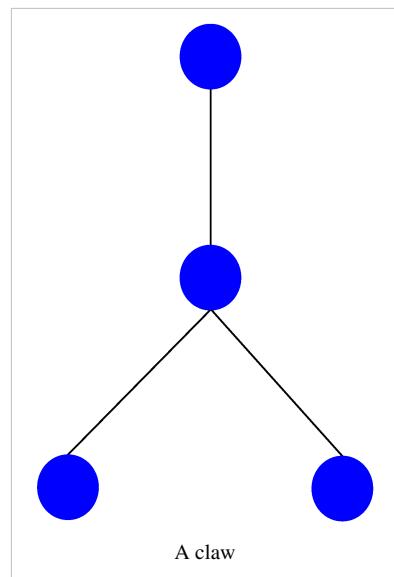
External links

- *line graphs* (http://graphclasses.org/classes/gc_249.html), Information System on Graph Class Inclusions (<http://graphclasses.org/index.html>) (last visited Sep 23 2013)
- Weisstein, Eric W., "Line Graph" (<http://mathworld.wolfram.com/LineGraph.html>), *MathWorld*.

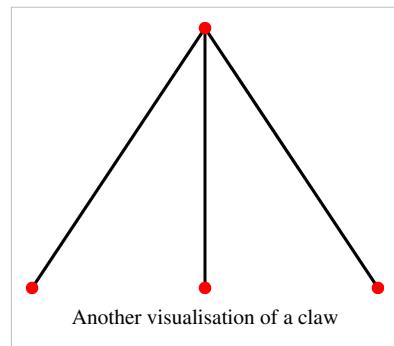
Claw-free graph

In graph theory, an area of mathematics, a **claw-free graph** is a graph that does not have a claw as an induced subgraph.

A claw is another name for the complete bipartite graph $K_{1,3}$ (that is, a star graph with three edges, three leaves, and one central vertex). A claw-free graph is a graph in which no induced subgraph is a claw; i.e., any subset of four vertices has other than only three edges connecting them in this pattern. Equivalently, a claw-free graph is a graph in which the neighborhood of any vertex is the complement of a triangle-free graph.

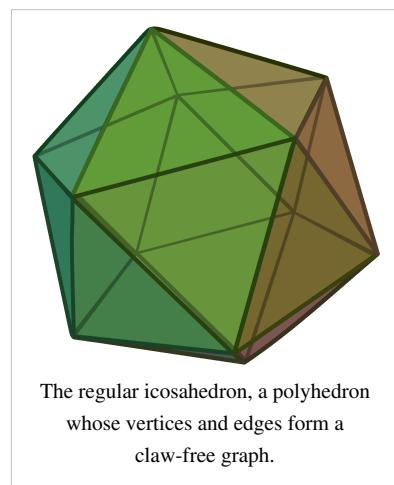


Claw-free graphs were initially studied as a generalization of line graphs, and gained additional motivation through three key discoveries about them: the fact that all claw-free connected graphs of even order have perfect matchings, the discovery of polynomial time algorithms for finding maximum independent sets in claw-free graphs, and the characterization of claw-free perfect graphs.^[1] They are the subject of hundreds of mathematical research papers and several surveys.



Examples

- The line graph $L(G)$ of any graph G is claw-free; $L(G)$ has a vertex for every edge of G , and vertices are adjacent in $L(G)$ whenever the corresponding edges share an endpoint in G . A line graph $L(G)$ cannot contain a claw, because if three edges e_1, e_2 , and e_3 in G all share endpoints with another edge e_4 then by the pigeonhole principle at least two of e_1, e_2 , and e_3 must share one of those endpoints with each other. Line graphs may be characterized in terms of nine forbidden subgraphs; the claw is the simplest of these nine graphs. This characterization provided the initial motivation for studying claw-free graphs.
- The de Bruijn graphs (graphs whose vertices represent n -bit binary strings for some n , and whose edges represent $(n - 1)$ -bit overlaps between two strings) are claw-free. One way to show this is via the construction of the de Bruijn graph for n -bit strings as the line graph of the de Bruijn graph for $(n - 1)$ -bit strings.
- The complement of any triangle-free graph is claw-free.^[2] These graphs include as a special case any complete graph.
- Proper interval graphs, the interval graphs formed as intersection graphs of families of intervals in which no interval contains another interval, are claw-free, because four properly intersecting intervals cannot intersect in the pattern of a claw.
- The Moser spindle, a seven-vertex graph used to provide a lower bound for the chromatic number of the plane, is claw-free.
- The graphs of several polyhedra and polytopes are claw-free, including the graph of the tetrahedron and more generally of any simplex (a complete graph), the graph of the octahedron and more generally of any cross polytope (isomorphic to the cocktail party graph formed by removing a perfect matching from a complete graph), the graph of the regular icosahedron, and the graph of the 16-cell.
- The Schläfli graph, a strongly regular graph with 27 vertices, is claw-free.



Recognition

It is straightforward to verify that a given graph with n vertices and m edges is claw-free in time $O(n^4)$, by testing each 4-tuple of vertices to determine whether they induce a claw.^[3] Somewhat more efficiently, but more complicatedly, one can test whether a graph is claw-free by checking, for each vertex of the graph, that the complement graph of its neighbors does not contain a triangle. A graph contains a triangle if and only if the cube of its adjacency matrix contains a nonzero diagonal element, so finding a triangle may be performed in the same asymptotic time bound as $n \times n$ matrix multiplication. Therefore, using the Coppersmith–Winograd algorithm, the

total time for this claw-free recognition algorithm would be $O(n^{3.376})$.

Kloks, Kratsch & Müller (2000) observe that in any claw-free graph, each vertex has at most $2\sqrt{m}$ neighbors; for otherwise by Turán's theorem the neighbors of the vertex would not have enough remaining edges to form the complement of a triangle-free graph. This observation allows the check of each neighborhood in the fast matrix multiplication based algorithm outlined above to be performed in the same asymptotic time bound as $2\sqrt{m} \times 2\sqrt{m}$ matrix multiplication, or faster for vertices with even lower degrees. The worst case for this algorithm occurs when $\Omega(\sqrt{m})$ vertices have $\Omega(\sqrt{m})$ neighbors each, and the remaining vertices have few neighbors, so its total time is $O(m^{3.376/2}) = O(m^{1.688})$.

Enumeration

Because claw-free graphs include complements of triangle-free graphs, the number of claw-free graphs on n vertices grows at least as quickly as the number of triangle-free graphs, exponentially in the square of n . The numbers of connected claw-free graphs on n nodes, for $n = 1, 2, \dots$ are

1, 1, 2, 5, 14, 50, 191, 881, 4494, 26389, 184749, ... (sequence A022562 in OEIS).

If the graphs are allowed to be disconnected, the numbers of graphs are even larger: they are

1, 2, 4, 10, 26, 85, 302, 1285, 6170, ... (sequence A086991 in OEIS).

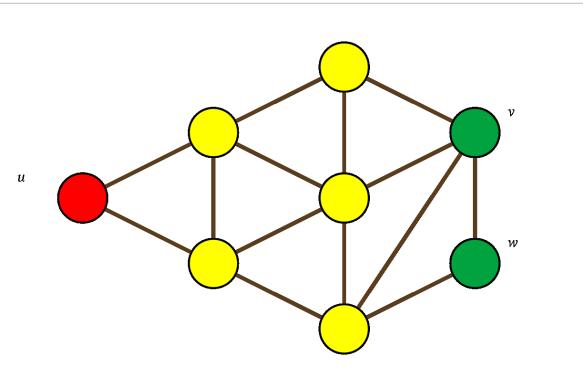
A technique of Palmer, Read & Robinson (2002) allows the number of claw-free cubic graphs to be counted very efficiently, unusually for graph enumeration problems.

Matchings

Sumner (1974) and, independently, Las Vergnas (1975) proved that every claw-free connected graph with an even number of vertices has a perfect matching.^[4] That is, there exists a set of edges in the graph such that each vertex is an endpoint of exactly one of the matched edges. The special case of this result for line graphs implies that, in any graph with an even number of edges, one can partition the edges into paths of length two. Perfect matchings may be used to provide another characterization of the claw-free graphs: they are exactly the graphs in which every connected induced subgraph of even order has a perfect matching.

Sumner's proof shows, more strongly, that in any connected claw-free graph one can find a pair of adjacent vertices the removal of which leaves the remaining graph connected. To show this, Sumner finds a pair of vertices u and v that are as far apart as possible in the graph, and chooses w to be a neighbor of v that is as far from u as possible; as he shows, neither v nor w can lie on any shortest path from any other node to u , so the removal of v and w leaves the remaining graph connected. Repeatedly removing matched pairs of vertices in this way forms a perfect matching in the given claw-free graph.

The same proof idea holds more generally if u is any vertex, v is any vertex that is maximally far from u , and w is any neighbor of v that is maximally far from u . Further, the removal of v and w from the graph does not change any of the other distances from u . Therefore, the process of forming a matching by finding and removing pairs vw that are maximally far from u may be performed by a single postorder traversal of a breadth first search tree of the graph, rooted at u , in linear time. Chrobak, Naor & Novick (1989) provide an alternative linear-time algorithm based on



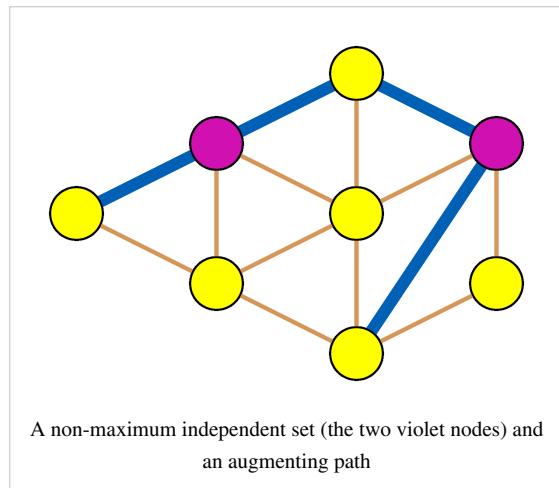
Sumner's proof that claw-free connected graphs of even order have perfect matchings: removing the two adjacent vertices v and w that are farthest from u leaves a connected subgraph, within which the same removal process may be repeated.

depth-first search, as well as efficient parallel algorithms for the same problem.

Faudree, Flandrin & Ryjáček (1997) list several related results, including the following: $(r - 1)$ -connected $K_{1,r}$ -free graphs of even order have perfect matchings for any $r \geq 2$; claw-free graphs of odd order with at most one degree-one vertex may be partitioned into an odd cycle and a matching; for any k that is at most half the minimum degree of a claw-free graph in which either k or the number of vertices is even, the graph has a k -factor; and, if a claw-free graph is $(2k + 1)$ -connected, then any k -edge matching can be extended to a perfect matching.

Independent sets

An independent set in a line graph corresponds to a matching in its underlying graph, a set of edges no two of which share an endpoint. As Edmonds (1965) showed, a maximum matching in any graph may be found in polynomial time; Sbihi (1980) extended this algorithm to one that computes a maximum independent set in any claw-free graph.^[5] Minty (1980) (corrected by Nakamura & Tamura 2001) independently provided an alternative extension of Edmonds' algorithms to claw-free graphs, that transforms the problem into one of finding a matching in an auxiliary graph derived from the input claw-free graph. Minty's approach may also be used to solve in polynomial time the more general problem of finding an independent set of maximum weight, and generalizations of these results to wider classes of graphs are also known.



A non-maximum independent set (the two violet nodes) and an augmenting path

As Sbihi observed, if I is an independent set in a claw-free graph, then any vertex of the graph may have at most two neighbors in I : three neighbors would form a claw. Sbihi calls a vertex *saturated* if it has two neighbors in I and *unsaturated* if it is not in I but has fewer than two neighbors in I . It follows from Sbihi's observation that if I and J are both independent sets, the graph induced by $I \cup J$ must have degree at most two; that is, it is a union of paths and cycles. In particular, if I is a non-maximum independent set, it differs from any maximum independent set by cycles and *augmenting paths*, induced paths which alternate between vertices in I and vertices not in I , and for which both endpoints are unsaturated. The symmetric difference of I with an augmenting path is a larger independent set; Sbihi's algorithm repeatedly increases the size of an independent set by searching for augmenting paths until no more can be found.

Searching for an augmenting path is complicated by the fact that a path may fail to be augmenting if it contains an edge between two vertices that are not in I , so that it is not an induced path. Fortunately, this can only happen in two cases: the two adjacent vertices may be the endpoints of the path, or they may be two steps away from each other; any other adjacency would lead to a claw. Adjacent endpoints may be avoided by temporarily removing the neighbors of v when searching for a path starting from a vertex v ; if no path is found, v can be removed from the graph for the remainder of the algorithm. Although Sbihi does not describe it in these terms, the problem remaining after this reduction may be described in terms of a switch graph, an undirected graph in which the edges incident to each vertex are partitioned into two subsets and in which paths through the vertex are constrained to use one edge from each subset. One may form a switch graph that has as its vertices the unsaturated and saturated vertices of the given claw-free graph, with an edge between two vertices of the switch graph whenever they are nonadjacent in the claw-free graph and there exists a length-two path between them that passes through a vertex of I . The two subsets of edges at each vertex are formed by the two vertices of I that these length-two paths pass through. A path in this switch graph between two unsaturated vertices corresponds to an augmenting path in the original graph. The switch graph has quadratic complexity, paths in it may be found in linear time, and $O(n)$ augmenting paths may need to be found over the course of the overall algorithm. Therefore, Sbihi's algorithm runs in $O(n^3)$ total time.

Coloring, cliques, and domination

A perfect graph is a graph in which the chromatic number and the size of the maximum clique are equal, and in which this equality persists in every induced subgraph. It is now known (the strong perfect graph theorem) that perfect graphs may be characterized as the graphs that do not have as induced subgraphs either an odd cycle or the complement of an odd cycle (a so-called *odd hole*). However, for many years this remained an unsolved conjecture, only proven for special subclasses of graphs. One of these subclasses was the family of claw-free graphs: it was discovered by several authors that claw-free graphs without odd cycles and odd holes are perfect. Perfect claw-free graphs may be recognized in polynomial time. In a perfect claw-free graph, the neighborhood of any vertex forms the complement of a bipartite graph. It is possible to color perfect claw-free graphs, or to find maximum cliques in them, in polynomial time.^[6]

If a claw-free graph is not perfect, it is NP-hard to find its largest clique. It is also NP-hard to find an optimal coloring of the graph, because (via line graphs) this problem generalizes the NP-hard problem of computing the chromatic index of a graph. For the same reason, it is NP-hard to find a coloring that achieves an approximation ratio better than 4/3. However, an approximation ratio of two can be achieved by a greedy coloring algorithm, because the chromatic number of a claw-free graph is greater than half its maximum degree.

Although not every claw-free graph is perfect, claw-free graphs satisfy another property, related to perfection. A graph is called domination perfect if it has a minimum dominating set that is independent, and if the same property holds in all of its induced subgraphs. Claw-free graphs have this property. To see this, let D be a dominating set in a claw-free graph, and suppose that v and w are two adjacent vertices in D ; then the set of vertices dominated by v but not by w must be a clique (else v would be the center of a claw). If every vertex in this clique is already dominated by at least one other member of D , then v can be removed producing a smaller independent dominating set, and otherwise v can be replaced by one of the undominated vertices in its clique producing a dominating set with fewer adjacencies. By repeating this replacement process one eventually reaches a dominating set no larger than D , so in particular when the starting set D is a minimum dominating set this process forms an equally small independent dominating set.^[7]

Despite this domination perfectness property, it is NP-hard to determine the size of the minimum dominating set in a claw-free graph. However, in contrast to the situation for more general classes of graphs, finding the minimum dominating set or the minimum connected dominating set in a claw-free graph is fixed-parameter tractable: it can be solved in time bounded by a polynomial in the size of the graph multiplied by an exponential function of the dominating set size.^[8]

Structure

Chudnovsky & Seymour (2005) overview a series of papers in which they prove a structure theory for claw-free graphs, analogous to the graph structure theorem for minor-closed graph families proven by Robertson and Seymour, and to the structure theory for perfect graphs that Chudnovsky, Seymour and their co-authors used to prove the strong perfect graph theorem. The theory is too complex to describe in detail here, but to give a flavor of it, it suffices to outline two of their results. First, for a special subclass of claw-free graphs which they call *quasi-line graphs* (equivalently, locally co-bipartite graphs), they state that every such graph has one of two forms:

1. A *fuzzy circular interval graph*, a class of graphs represented geometrically by points and arcs on a circle, generalizing proper circular arc graphs.
2. A graph constructed from a multigraph by replacing each edge by a *fuzzy linear interval graph*. This generalizes the construction of a line graph, in which every edge of the multigraph is replaced by a vertex. Fuzzy linear interval graphs are constructed in the same way as fuzzy circular interval graphs, but on a line rather than on a circle.

Chudnovsky and Seymour classify arbitrary connected claw-free graphs into one of the following:

1. Six specific subclasses of claw-free graphs. Three of these are line graphs, proper circular arc graphs, and the induced subgraphs of an icosahedron; the other three involve additional definitions.
2. Graphs formed in four simple ways from smaller claw-free graphs.
3. *Antiprismatic graphs*, a class of dense graphs defined as the claw-free graphs in which every four vertices induce a subgraph with at least two edges.

Much of the work in their structure theory involves a further analysis of antiprismatic graphs. The Schläfli graph, a claw-free strongly regular graph with parameters $srg(27, 16, 10, 8)$, plays an important role in this part of the analysis. This structure theory has led to new advances in polyhedral combinatorics and new bounds on the chromatic number of claw-free graphs, as well as to new fixed-parameter-tractable algorithms for dominating sets in claw-free graphs.

Notes

- [1] , p. 88.
- [2] , p. 89.
- [3] , p. 132.
- [4] , pp. 120–124.
- [5] , pp. 133–134.
- [6] , pp. 135–136.
- [7] , pp. 124–125.
- [8] ; .

References

- Beineke, L. W. (1968), "Derived graphs of digraphs", in Sachs, H.; Voss, H.-J.; Walter, H.-J., *Beiträge zur Graphentheorie*, Leipzig: Teubner, pp. 17–33.
- Chrobak, Marek; Naor, Joseph; Novick, Mark B. (1989), "Using bounded degree spanning trees in the design of efficient algorithms on claw-free graphs", in Dehne, F.; Sack, J.-R.; Santoro, N., *Algorithms and Data Structures: Workshop WADS '89, Ottawa, Canada, August 1989, Proceedings*, Lecture Notes in Comput. Sci. **382**, Berlin: Springer, pp. 147–162, doi: 10.1007/3-540-51542-9_13 (http://dx.doi.org/10.1007/3-540-51542-9_13).
- Chudnovsky, Maria; Robertson, Neil; Seymour, Paul; Thomas, Robin (2006), "The strong perfect graph theorem" (<http://people.math.gatech.edu/~thomas/PAP/spgc.pdf>), *Annals of Mathematics* **164** (1): 51–229, doi: 10.4007/annals.2006.164.51 (<http://dx.doi.org/10.4007/annals.2006.164.51>).
- Chudnovsky, Maria; Seymour, Paul (2005), "The structure of claw-free graphs" (http://www.math.princeton.edu/~mchudnov/claws_survey.pdf), *Surveys in combinatorics 2005*, London Math. Soc. Lecture Note Ser. **327**, Cambridge: Cambridge Univ. Press, pp. 153–171, MR 2187738 (<http://www.ams.org/mathscinet-getitem?mr=2187738>).
- Cygan, Marek; Philip, Geevarghese; Pilipczuk, Marcin; Pilipczuk, Michał; Wojtaszczyk, Jakub Onufry (2010). "Dominating set is fixed parameter tractable in claw-free graphs". arXiv: 1011.6239 (<http://arxiv.org/abs/1011.6239>).
- Edmonds, Jack (1965), "Paths, Trees and Flowers", *Canadian J. Math* **17** (0): 449–467, doi: 10.4153/CJM-1965-045-4 (<http://dx.doi.org/10.4153/CJM-1965-045-4>), MR 0177907 (<http://www.ams.org/mathscinet-getitem?mr=0177907>).
- Faudree, Ralph; Flandrin, Evelyne; Ryjáček, Zdeněk (1997), "Claw-free graphs — A survey", *Discrete Mathematics* **164** (1–3): 87–147, doi: 10.1016/S0012-365X(96)00045-3 ([http://dx.doi.org/10.1016/S0012-365X\(96\)00045-3](http://dx.doi.org/10.1016/S0012-365X(96)00045-3)), MR 1432221 (<http://www.ams.org/mathscinet-getitem?mr=1432221>).
- Hermelin, Danny; Mnich, Matthias; van Leeuwen, Erik Jan; Woeginger, Gerhard (2010). "Domination when the stars are out". arXiv: 1012.0012 (<http://arxiv.org/abs/1012.0012>).
- Itai, Alon; Rodeh, Michael (1978), "Finding a minimum circuit in a graph", *SIAM Journal on Computing* **7** (4): 413–423, doi: 10.1137/0207033 (<http://dx.doi.org/10.1137/0207033>), MR 0508603 (<http://www.ams.org/mathscinet-getitem?mr=0508603>).

- Kloks, Ton; Kratsch, Dieter; Müller, Haiko (2000), "Finding and counting small induced subgraphs efficiently", *Information Processing Letters* **74** (3–4): 115–121, doi: 10.1016/S0020-0190(00)00047-8 ([http://dx.doi.org/10.1016/S0020-0190\(00\)00047-8](http://dx.doi.org/10.1016/S0020-0190(00)00047-8)), MR 1761552 (<http://www.ams.org/mathscinet-getitem?mr=1761552>).
- Las Vergnas, M. (1975), "A note on matchings in graphs", *Cahiers du Centre d'Études de Recherche Opérationnelle* **17** (2-3-4): 257–260, MR 0412042 (<http://www.ams.org/mathscinet-getitem?mr=0412042>).
- Minty, George J. (1980), "On maximal independent sets of vertices in claw-free graphs", *Journal of Combinatorial Theory. Series B* **28** (3): 284–304, doi: 10.1016/0095-8956(80)90074-X ([http://dx.doi.org/10.1016/0095-8956\(80\)90074-X](http://dx.doi.org/10.1016/0095-8956(80)90074-X)), MR 579076 (<http://www.ams.org/mathscinet-getitem?mr=579076>).
- Nakamura, Daishin; Tamura, Akihisa (2001), "A revision of Minty's algorithm for finding a maximum weighted stable set of a claw-free graph" (<http://www.kurims.kyoto-u.ac.jp/preprint/file/RIMS1261.ps.gz>), *Journal of the Operations Research Society of Japan* **44** (2): 194–204.
- Palmer, Edgar M.; Read, Ronald C.; Robinson, Robert W. (2002), "Counting claw-free cubic graphs" (<http://www.cs.uga.edu/~rwr/publications/claw.pdf>), *SIAM Journal on Discrete Mathematics* **16** (1): 65–73, doi: 10.1137/S0895480194274777 (<http://dx.doi.org/10.1137/S0895480194274777>), MR 1972075 (<http://www.ams.org/mathscinet-getitem?mr=1972075>).
- Sbihi, Najiba (1980), "Algorithme de recherche d'un stable de cardinalité maximum dans un graphe sans étoile", *Discrete Mathematics* **29** (1): 53–76, doi: 10.1016/0012-365X(90)90287-R ([http://dx.doi.org/10.1016/0012-365X\(90\)90287-R](http://dx.doi.org/10.1016/0012-365X(90)90287-R)), MR 553650 (<http://www.ams.org/mathscinet-getitem?mr=553650>).
- Sumner, David P. (1974), "Graphs with 1-factors", *Proceedings of the American Mathematical Society* (American Mathematical Society) **42** (1): 8–12, doi: 10.2307/2039666 (<http://dx.doi.org/10.2307/2039666>), JSTOR 2039666 (<http://www.jstor.org/stable/2039666>), MR 0323648 (<http://www.ams.org/mathscinet-getitem?mr=0323648>).

External links

- Claw-free graphs (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_62.html), Information System on Graph Class Inclusions
- Mugan, Jonathan William; Weisstein, Eric W., "Claw-Free Graph" (<http://mathworld.wolfram.com/Claw-FreeGraph.html>), *MathWorld*.

Median graph

In graph theory, a division of mathematics, a **median graph** is an undirected graph in which every three vertices a , b , and c have a unique *median*: a vertex $m(a,b,c)$ that belongs to shortest paths between each pair of a , b , and c .

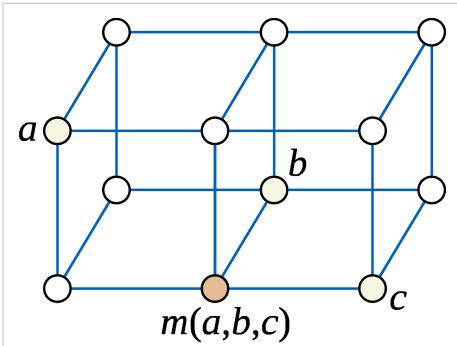
The concept of median graphs has long been studied, for instance by Birkhoff & Kiss (1947) or (more explicitly) by Avann (1961), but the first paper to call them "median graphs" appears to be Nebesk'y (1971). As Chung, Graham, and Saks write, "median graphs arise naturally in the study of ordered sets and discrete distributive lattices, and have an extensive literature". In phylogenetics, the Buneman graph representing all maximum parsimony evolutionary trees is a median graph.^[1] Median graphs also arise in social choice theory: if a set of alternatives has the structure of a median graph, it is possible to derive in an unambiguous way a majority preference among them.^[2]

Additional surveys of median graphs are given by Klavžar & Mulder (1999), Bandelt & Chepoi (2008), and Knuth (2008).

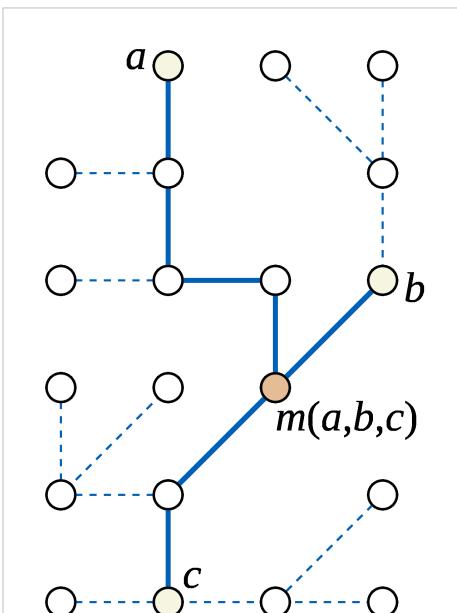
Examples

Every tree is a median graph.^[3] To see this, observe that in a tree, the union of the three shortest paths between pairs of the three vertices a , b , and c is either itself a path, or a subtree formed by three paths meeting at a single central node with degree three. If the union of the three paths is itself a path, the median $m(a,b,c)$ is equal to one of a , b , or c , whichever of these three vertices is between the other two in the path. If the subtree formed by the union of the three paths is not a path, the median of the three vertices is the central degree-three node of the subtree.

Additional examples of median graphs are provided by the grid graphs. In a grid graph, the coordinates of the median $m(a,b,c)$ can be found as the median of the coordinates of a , b , and c . Conversely, it turns out that, in every median graph, one may label the vertices by points in an integer lattice in such a way that medians can be calculated coordinatewise in this way.^[4]



The median of three vertices in a median graph

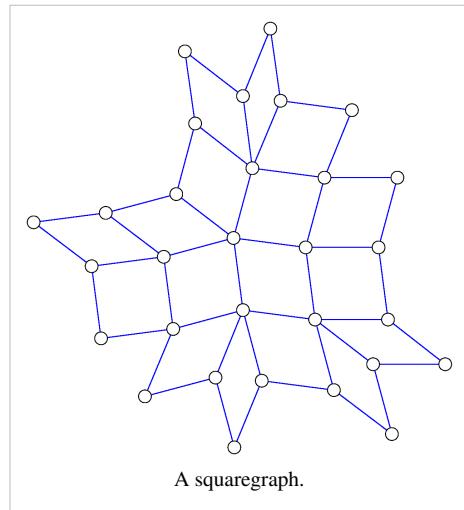


The median of three vertices in a tree, showing the subtree formed by the union of shortest paths between the vertices.

Squaregraphs, planar graphs in which all interior faces are quadrilaterals and all interior vertices have four or more incident edges, are another subclass of the median graphs.^[5] A polyomino is a special case of a squaregraph and therefore also forms a median graph.

The simplex graph $\kappa(G)$ of an arbitrary undirected graph G has a node for every clique (complete subgraph) of G ; two nodes are linked by an edge if the corresponding cliques differ by one vertex. The median of a given triple of cliques may be formed by using the majority rule to determine which vertices of the cliques to include; the simplex graph is a median graph in which this rule determines the median of each triple of vertices.

No cycle graph of length other than four can be a median graph, because every such cycle has three vertices a , b , and c such that the three shortest paths wrap all the way around the cycle without having a common intersection. For such a triple of vertices, there can be no median.



Equivalent definitions

In an arbitrary graph, for each two vertices a and b , the minimal number of edges between them is called their *distance*, denoted by $d(x,y)$. The *interval* of vertices that lie on shortest paths between a and b is defined as

$$I(a,b) = \{v \mid d(a,b) = d(a,v) + d(v,b)\}.$$

A median graph is defined by the property that, for every three vertices a , b , and c , these intervals intersect in a single point:

$$\text{For all } a, b, \text{ and } c, |I(a,b) \cap I(a,c) \cap I(b,c)| = 1.$$

Equivalently, for every three vertices a , b , and c one can find a vertex $m(a,b,c)$ such that the unweighted distances in the graph satisfy the equalities

- $d(a,b) = d(a,m(a,b,c)) + d(m(a,b,c),b)$
- $d(a,c) = d(a,m(a,b,c)) + d(m(a,b,c),c)$
- $d(b,c) = d(b,m(a,b,c)) + d(m(a,b,c),c)$

and $m(a,b,c)$ is the only vertex for which this is true.

It is also possible to define median graphs as the solution sets of 2-satisfiability problems, as the retracts of hypercubes, as the graphs of finite median algebras, as the Buneman graphs of Helly split systems, and as the graphs of windex 2; see the sections below.

Distributive lattices and median algebras

In lattice theory, the graph of a finite lattice has a vertex for each lattice element and an edge for each pair of elements in the covering relation of the lattice. Lattices are commonly presented visually via Hasse diagrams, which are drawings of graphs of lattices. These graphs, especially in the case of distributive lattices, turn out to be closely related to median graphs.

In a distributive lattice, Birkhoff's self-dual ternary median operation^[6]

$$m(a,b,c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \wedge (b \vee c),$$

satisfies certain key axioms, which it shares with the usual median of numbers in the range from 0 to 1 and with median algebras more generally:

- Idempotence: $m(a,a,b) = a$ for all a and b .
- Commutativity: $m(a,b,c) = m(a,c,b) = m(b,a,c) = m(b,c,a) = m(c,a,b) = m(c,b,a)$ for all a , b , and c .
- Distributivity: $m(a,m(b,c,d),e) = m(m(a,b,e),c,m(a,d,e))$ for all a , b , c , d , and e .
- Identity elements: $m(0,a,1) = a$ for all a .

The distributive law may be replaced by an associative law:^[7]

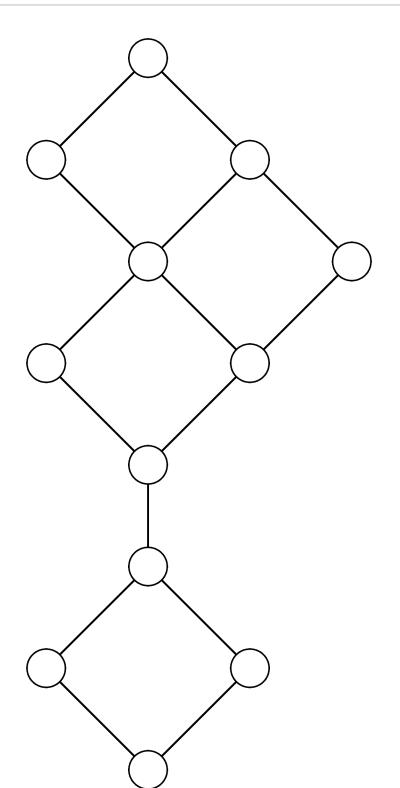
- Associativity: $m(x,w,m(y,w,z)) = m(m(x,w,y),w,z)$

The median operation may also be used to define a notion of intervals for distributive lattices:

$$I(a,b) = \{x \mid m(a,x,b) = x\} = \{x \mid a \wedge b \leq x \leq a \vee b\}.^{[8]}$$

The graph of a finite distributive lattice has an edge between vertices a and b whenever $I(a,b) = \{a,b\}$. For every two vertices a and b of this graph, the interval $I(a,b)$ defined in lattice-theoretic terms above consists of the vertices on shortest paths from a to b , and thus coincides with the graph-theoretic intervals defined earlier. For every three lattice elements a , b , and c , $m(a,b,c)$ is the unique intersection of the three intervals $I(a,b)$, $I(a,c)$, and $I(b,c)$.^[9] Therefore, the graph of an arbitrary finite distributive lattice is a median graph. Conversely, if a median graph G contains two vertices 0 and 1 such that every other vertex lies on a shortest path between the two (equivalently, $m(0,a,1) = a$ for all a), then we may define a distributive lattice in which $a \wedge b = m(a,0,b)$ and $a \vee b = m(a,1,b)$, and G will be the graph of this lattice.^[10]

Duffus & Rival (1983) characterize graphs of distributive lattices directly as diameter-preserving retracts of hypercubes. More generally, every median graph gives rise to a ternary operation m satisfying idempotence, commutativity, and distributivity, but possibly without the identity elements of a distributive lattice. Every ternary operation on a finite set that satisfies these three properties (but that does not necessarily have 0 and 1 elements) gives rise in the same way to a median graph.



The graph of a distributive lattice, drawn as a Hasse diagram.

Convex sets and Helly families

In a median graph, a set S of vertices is said to be convex if, for every two vertices a and b belonging to S , the whole interval $I(a,b)$ is a subset of S . Equivalently, given the two definitions of intervals above, S is convex if it contains every shortest path between two of its vertices, or if it contains the median of every set of three points at least two of which are from S . Observe that the intersection of every pair of convex sets is itself convex.^[11]

The convex sets in a median graph have the Helly property: if F is an arbitrary family of pairwise-intersecting convex sets, then all sets in F have a common intersection.^[12] For, if F has only three convex sets S , T , and U in it, with a in the intersection of the pair S and T , b in the intersection of the pair T and U , and c in the intersection of the pair S and U , then every shortest path from a to b must lie within T by convexity, and similarly every shortest path between the other two pairs of vertices must lie within the other two sets; but $m(a,b,c)$ belongs to paths between all three pairs of vertices, so it lies within all three sets, and forms part of their common intersection. If F has more than three convex sets in it, the result follows by induction on the number of sets, for one may replace an arbitrary pair of sets in F by their intersection, using the result for triples of sets to show that the replaced family is still pairwise intersecting.

A particularly important family of convex sets in a median graph, playing a role similar to that of halfspaces in Euclidean space, are the sets

$$W_{uv} = \{w \mid d(w,u) < d(w,v)\}$$

defined for each edge uv of the graph. In words, W_{uv} consists of the vertices closer to u than to v , or equivalently the vertices w such that some shortest path from v to w goes through u . To show that W_{uv} is convex, let $w_1 w_2 \dots w_k$ be an arbitrary shortest path that starts and ends within W_{uv} ; then w_2 must also lie within W_{uv} , for otherwise the two points $m_1 = m(u, w_1, w_k)$ and $m_2 = m(m_1, w_2 \dots w_k)$ could be shown (by considering the possible distances between the vertices) to be distinct medians of u , w_1 , and w_k , contradicting the definition of a median graph which requires medians to be unique. Thus, each successive vertex on a shortest path between two vertices of W_{uv} also lies within W_{uv} , so W_{uv} contains all shortest paths between its nodes, one of the definitions of convexity.

The Helly property for the sets W_{uv} plays a key role in the characterization of median graphs as the solution of 2-satisfiability instances, below.

2-satisfiability

Median graphs have a close connection to the solution sets of 2-satisfiability problems that can be used both to characterize these graphs and to relate them to adjacency-preserving maps of hypercubes.^[13]

A 2-satisfiability instance consists of a collection of Boolean variables and a collection of *clauses*, constraints on certain pairs of variables requiring those two variables to avoid certain combinations of values. Usually such problems are expressed in conjunctive normal form, in which each clause is expressed as a disjunction and the whole set of constraints is expressed as a conjunction of clauses, such as

$$(x_{11} \vee x_{12}) \wedge (x_{21} \vee x_{22}) \wedge \dots \wedge (x_{n1} \vee x_{n2}) \wedge \dots$$

A solution to such an instance is an assignment of truth values to the variables that satisfies all the clauses, or equivalently that causes the conjunctive normal form expression for the instance to become true when the variable values are substituted into it. The family of all solutions has a natural structure as a median algebra, where the median of three solutions is formed by choosing each truth value to be the majority function of the values in the three solutions; it is straightforward to verify that this median solution cannot violate any of the clauses. Thus, these solutions form a median graph, in which the neighbor of each solution is formed by negating a set of variables that are all constrained to be equal or unequal to each other.

Conversely, every median graph G may be represented in this way as the solution set to a 2-satisfiability instance. To find such a representation, create a 2-satisfiability instance in which each variable describes the orientation of one of

the edges in the graph (an assignment of a direction to the edge causing the graph to become directed rather than undirected) and each constraint allows two edges to share a pair of orientations only when there exists a vertex v such that both orientations lie along shortest paths from other vertices to v . Each vertex v of G corresponds to a solution to this 2-satisfiability instance in which all edges are directed towards v . Each solution to the instance must come from some vertex v in this way, where v is the common intersection of the sets W_{uv} for edges directed from w to u ; this common intersection exists due to the Helly property of the sets W_{uv} . Therefore, the solutions to this 2-satisfiability instance correspond one-for-one with the vertices of G .

Retracts of hypercubes

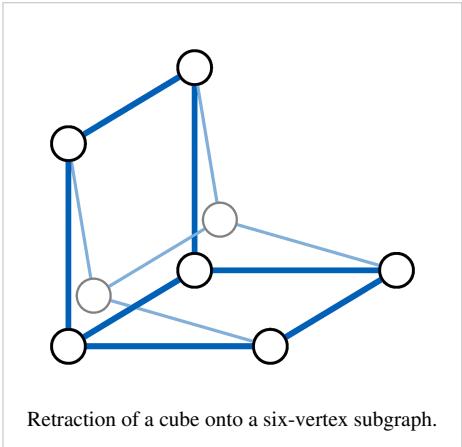
A *retraction* of a graph G is an adjacency-preserving map from G to one of its subgraphs. More precisely, it is graph homomorphism φ from G to itself such that $\varphi(v) = v$ for each vertex v in the subgraph $\varphi(G)$. The image of the retraction is called a *retract* of G . Retractions are examples of metric maps: the distance between $\varphi(v)$ and $\varphi(w)$, for every v and w , is at most equal to the distance between v and w , and is equal whenever v and w both belong to $\varphi(G)$. Therefore, a retract must be an *isometric subgraph* of G : distances in the retract equal those in G .

If G is a median graph, and a , b , and c are an arbitrary three vertices of a retract $\varphi(G)$, then $\varphi(m(a,b,c))$ must be a median of a , b , and c , and so must equal $m(a,b,c)$. Therefore, $\varphi(G)$ contains medians of all triples of its vertices, and must also be a median graph. In other words, the family of median graphs is closed under the retraction operation.^[14]

A hypercube graph, in which the vertices correspond to all possible k -bit bitvectors and in which two vertices are adjacent when the corresponding bitvectors differ in only a single bit, is a special case of a k -dimensional grid graph and is therefore a median graph. The median of three bitvectors a , b , and c may be calculated by computing, in each bit position, the majority function of the bits of a , b , and c . Since median graphs are closed under retraction, and include the hypercubes, every retract of a hypercube is a median graph.

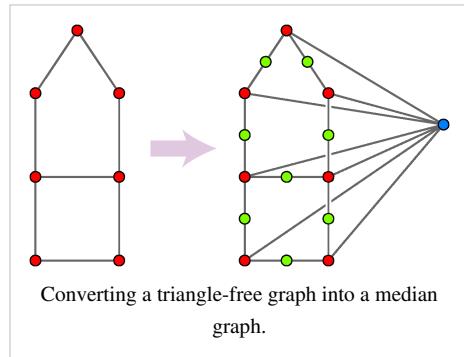
Conversely, every median graph must be the retract of a hypercube.^[15] This may be seen from the connection, described above, between median graphs and 2-satisfiability: let G be the graph of solutions to a 2-satisfiability instance; without loss of generality this instance can be formulated in such a way that no two variables are always equal or always unequal in every solution. Then the space of all truth assignments to the variables of this instance forms a hypercube. For each clause, formed as the disjunction of two variables or their complements, in the 2-satisfiability instance, one can form a retraction of the hypercube in which truth assignments violating this clause are mapped to truth assignments in which both variables satisfy the clause, without changing the other variables in the truth assignment. The composition of the retractions formed in this way for each of the clauses gives a retraction of the hypercube onto the solution space of the instance, and therefore gives a representation of G as the retract of a hypercube. In particular, median graphs are isometric subgraphs of hypercubes, and are therefore partial cubes. However, not all partial cubes are median graphs; for instance, a six-vertex cycle graph is a partial cube but is not a median graph.

As Imrich & Klavžar (2000) describe, an isometric embedding of a median graph into a hypercube may be constructed in time $O(m \log n)$, where n and m are the numbers of vertices and edges of the graph respectively.^[16]



Triangle-free graphs and recognition algorithms

The problems of testing whether a graph is a median graph, and whether a graph is triangle-free, both had been well studied when Imrich, Klavžar & Mulder (1999) observed that, in some sense, they are computationally equivalent.^[17] Therefore, the best known time bound for testing whether a graph is triangle-free, $O(m^{1.41})$,^[18] applies as well to testing whether a graph is a median graph, and any improvement in median graph testing algorithms would also lead to an improvement in algorithms for detecting triangles in graphs.

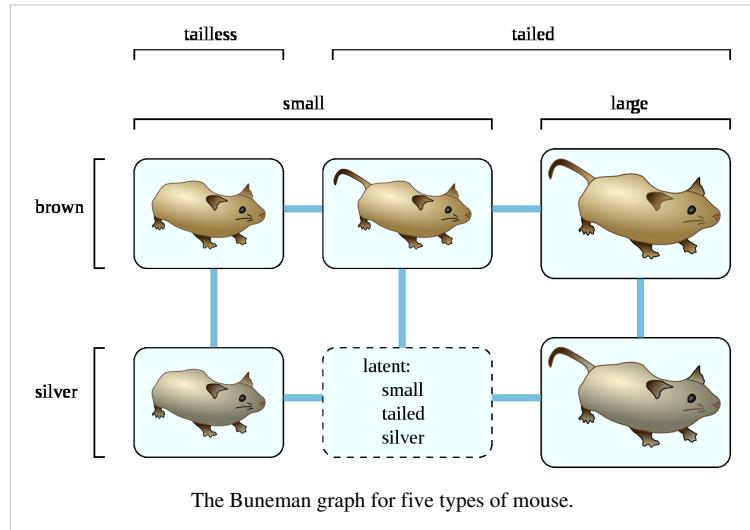


In one direction, suppose one is given as input a graph G , and must test whether G is triangle-free. From G , construct a new graph H having as vertices each set of zero, one, or two adjacent vertices of G . Two such sets are adjacent in H when they differ by exactly one vertex. An equivalent description of H is that it is formed by splitting each edge of G into a path of two edges, and adding a new vertex connected to all the original vertices of G . This graph H is by construction a partial cube, but it is a median graph only when G is triangle-free: if a, b , and c form a triangle in G , then $\{a,b\}$, $\{a,c\}$, and $\{b,c\}$ have no median in H , for such a median would have to correspond to the set $\{a,b,c\}$, but sets of three or more vertices of G do not form vertices in H . Therefore, G is triangle-free if and only if H is a median graph. In the case that G is triangle-free, H is its simplex graph. An algorithm to test efficiently whether H is a median graph could by this construction also be used to test whether G is triangle-free. This transformation preserves the computational complexity of the problem, for the size of H is proportional to that of G .

The reduction in the other direction, from triangle detection to median graph testing, is more involved and depends on the previous median graph recognition algorithm of Hagauer, Imrich & Klavžar (1999), which tests several necessary conditions for median graphs in near-linear time. The key new step involves using a breadth first search to partition the graph into levels according to their distances from some arbitrarily chosen root vertex, forming a graph in each level in which two vertices are adjacent if they share a common neighbor in the previous level, and searching for triangles in these graphs. The median of any such triangle must be a common neighbor of the three triangle vertices; if this common neighbor does not exist, the graph is not a median graph. If all triangles found in this way have medians, and the previous algorithm finds that the graph satisfies all the other conditions for being a median graph, then it must actually be a median graph. Note that this algorithm requires, not just the ability to test whether a triangle exists, but a list of all triangles in the level graph. In arbitrary graphs, listing all triangles sometimes requires $\Omega(m^{3/2})$ time, as some graphs have that many triangles, however Hagauer et al. show that the number of triangles arising in the level graphs of their reduction is near-linear, allowing the Alon et al. fast matrix multiplication based technique for finding triangles to be used.

Evolutionary trees, Buneman graphs, and Helly split systems

Phylogeny is the inference of evolutionary trees from observed characteristics of species; such a tree must place the species at distinct vertices, and may have additional *latent vertices*, but the latent vertices are required to have three or more incident edges and must also be labeled with characteristics. A characteristic is *binary* when it has only two possible values, and a set of species and their characteristics exhibit perfect phylogeny when there exists an evolutionary tree in which the vertices (species and latent vertices) labeled with any particular characteristic value form a contiguous subtree. If a tree with perfect phylogeny is not possible, it is often desired to find one exhibiting maximum parsimony, or equivalently, minimizing the number of times the endpoints of a tree edge have different values for one of the characteristics, summed over all edges and all characteristics.



Buneman (1971) described a method for inferring perfect phylogenies for binary characteristics, when they exist. His method generalizes naturally to the construction of a median graph for any set of species and binary characteristics, which has been called the *median network* or *Buneman graph*^[19] and is a type of phylogenetic network. Every maximum parsimony evolutionary tree embeds into the Buneman graph, in the sense that tree edges follow paths in the graph and the number of characteristic value changes on the tree edge is the same as the number in the corresponding path. The Buneman graph will be a tree if and only if a perfect phylogeny exists; this happens when there are no two incompatible characteristics for which all four combinations of characteristic values are observed.

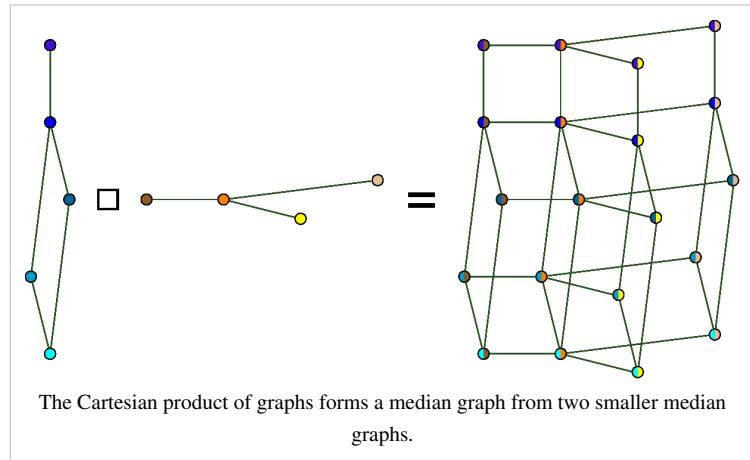
To form the Buneman graph for a set of species and characteristics, first, eliminate redundant species that are indistinguishable from some other species and redundant characteristics that are always the same as some other characteristic. Then, form a latent vertex for every combination of characteristic values such that every two of the values exist in some known species. In the example shown, there are small brown tailless mice, small silver tailless mice, small brown tailed mice, large brown tailed mice, and large silver tailed mice; the Buneman graph method would form a latent vertex corresponding to an unknown species of small silver tailed mice, because every pairwise combination (small and silver, small and tailed, and silver and tailed) is observed in some other known species. However, the method would not infer the existence of large brown tailless mice, because no mice are known to have both the large and tailless traits. Once the latent vertices are determined, form an edge between every pair of species or latent vertices that differ in a single characteristic.

One can equivalently describe a collection of binary characteristics as a *split system*, a family of sets having the property that the complement set of each set in the family is also in the family. This split system has a set for each characteristic value, consisting of the species that have that value. When the latent vertices are included, the resulting split system has the Helly property: every pairwise intersecting subfamily has a common intersection. In some sense median graphs are characterized as coming from Helly split systems: the pairs (W_{uv}, W_{vu}) defined for each edge uv of a median graph form a Helly split system, so if one applies the Buneman graph construction to this system no latent vertices will be needed and the result will be the same as the starting graph.

Bandelt et al. (1995) and Bandelt, Macaulay & Richards (2000) describe techniques for simplified hand calculation of the Buneman graph, and use this construction to visualize human genetic relationships.

Additional properties

- The Cartesian product of every two median graphs is another median graph. Medians in the product graph may be computed by independently finding the medians in the two factors, just as medians in grid graphs may be computed by independently finding the median in each linear dimension.
- The *windex* of a graph measures the amount of lookahead needed to optimally solve a problem in which one is given a sequence of graph vertices s_i , and must find as output another sequence of vertices t_i minimizing the sum of the distances $d(s_i, t_i)$ and $d(t_{i-1}, t_i)$. Median graphs are exactly the graphs that have *windex* 2. In a median graph, the optimal choice is to set $t_i = m(t_{i-1}, s_i, t_{i+1})$.
- The property of having a unique median is also called the *unique Steiner point property*. An optimal Steiner tree for three vertices a , b , and c in a median graph may be found as the union of three shortest paths, from a , b , and c to $m(a,b,c)$. Bandelt & Barthélémy (1984) study more generally the problem of finding the vertex minimizing the sum of distances to each of a given set of vertices, and show that it has a unique solution for any odd number of vertices in a median graph. They also show that this median of a set S of vertices in a median graph satisfies the Condorcet criterion for the winner of an election: compared to any other vertex, it is closer to a majority of the vertices in S .
- As with partial cubes more generally, every median graph with n vertices has at most $(n/2) \log_2 n$ edges. However, the number of edges cannot be too small: Klavžar, Mulder & Škrekovski (1998) prove that in every median graph the inequality $2n - m - k \leq 2$ holds, where m is the number of edges and k is the dimension of the hypercube that the graph is a retract of. This inequality is an equality if and only if the median graph contains no cubes. This is a consequence of another identity for median graphs: the Euler characteristic $\sum (-1)^{\dim(Q)}$ is always equal to one, where the sum is taken over all hypercube subgraphs Q of the given median graph.
- The only regular median graphs are the hypercubes.



Notes

- [1] ; .
- [2] ; .
- [3] , Proposition 1.26, p. 24.
- [4] This follows immediately from the characterization of median graphs as retracts of hypercubes, described below.
- [5] ; .
- [6] credit the definition of this operation to .
- [7] , p. 65, and exercises 75 and 76 on pp. 89–90. Knuth states that a simple proof that associativity implies distributivity remains unknown.
- [8] The equivalence between the two expressions in this equation, one in terms of the median operation and the other in terms of lattice operations and inequalities is Theorem 1 of .
- [9] , Theorem 2.
- [10] , p. 751.
- [11] calls such a set an *ideal*, but a convex set in the graph of a distributive lattice is not the same thing as an ideal of the lattice.
- [12] , Theorem 2.40, p. 77.
- [13] , Proposition 2.5, p. 8; ; , Theorem S, p. 72.
- [14] , Proposition 1.33, p. 27.
- [15] ; , Theorem 2.39, p. 76; , p. 74.

- [16] The technique, which culminates in Lemma 7.10 on p.218 of Imrich and Klavžar, consists of applying an algorithm of to list all 4-cycles in the graph G , forming an undirected graph having as its vertices the edges of G and having as its edges the opposite sides of a 4-cycle, and using the connected components of this derived graph to form hypercube coordinates. An equivalent algorithm is , Algorithm H, p. 69.
- [17] For previous median graph recognition algorithms, see , , and . For triangle detection algorithms, see , , and .
- [18] , based on fast matrix multiplication. Here m is the number of edges in the graph, and the big O notation hides a large constant factor; the best practical algorithms for triangle detection take time $O(m^{3/2})$. For median graph recognition, the time bound can be expressed either in terms of m or n (the number of vertices), as $m = O(n \log n)$.
- [19] described a version of this method for systems of characteristics not requiring any latent vertices, and gives the full construction. The Buneman graph name is given in and .

References

- Alon, Noga; Yuster, Raphael; Zwick, Uri (1995), "Color-coding", *Journal of the Association for Computing Machinery* **42** (4): 844–856, doi: 10.1145/210332.210337 (<http://dx.doi.org/10.1145/210332.210337>), MR 1411787 (<http://www.ams.org/mathscinet-getitem?mr=1411787>).
- Avann, S. P. (1961), "Metric ternary distributive semi-lattices", *Proceedings of the American Mathematical Society* (American Mathematical Society) **12** (3): 407–414, doi: 10.2307/2034206 (<http://dx.doi.org/10.2307/2034206>), JSTOR 2034206 (<http://www.jstor.org/stable/2034206>), MR 0125807 (<http://www.ams.org/mathscinet-getitem?mr=0125807>).
- Bandelt, Hans-Jürgen (1984), "Retracts of hypercubes", *Journal of Graph Theory* **8** (4): 501–510, doi: 10.1002/jgt.3190080407 (<http://dx.doi.org/10.1002/jgt.3190080407>), MR 0766499 (<http://www.ams.org/mathscinet-getitem?mr=0766499>).
- Bandelt, Hans-Jürgen; Barthélémy, Jean-Pierre (1984), "Medians in median graphs", *Discrete Applied Mathematics* **8** (2): 131–142, doi: 10.1016/0166-218X(84)90096-9 ([http://dx.doi.org/10.1016/0166-218X\(84\)90096-9](http://dx.doi.org/10.1016/0166-218X(84)90096-9)), MR 0743019 (<http://www.ams.org/mathscinet-getitem?mr=0743019>).
- Bandelt, Hans-Jürgen; Chepoi, V. (2008), "Metric graph theory and geometry: a survey" (http://www.lif-sud.univ-mrs.fr/~chepoi/survey_cm_bis.pdf), *Contemporary Mathematics*, to appear.
- Bandelt, Hans-Jürgen; Forster, P.; Sykes, B. C.; Richards, Martin B. (October 1, 1995), "Mitochondrial portraits of human populations using median networks" (<http://www.genetics.org/cgi/content/abstract/141/2/743>), *Genetics* **141** (2): 743–753, PMC 1206770 (<http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1206770>), PMID 8647407 (<http://www.ncbi.nlm.nih.gov/pubmed/8647407>).
- Bandelt, Hans-Jürgen; Forster, P.; Rohl, Arne (January 1, 1999), "Median-joining networks for inferring intraspecific phylogenies" (<http://mbe.oxfordjournals.org/cgi/content/abstract/16/1/37>), *Molecular Biology and Evolution* **16** (1): 37–48, doi: 10.1093/oxfordjournals.molbev.a026036 (<http://dx.doi.org/10.1093/oxfordjournals.molbev.a026036>), PMID 10331250 (<http://www.ncbi.nlm.nih.gov/pubmed/10331250>).
- Bandelt, Hans-Jürgen; Macaulay, Vincent; Richards, Martin B. (2000), "Median networks: speedy construction and greedy reduction, one simulation, and two case studies from human mtDNA", *Molecular Phylogenetics and Evolution* **16** (1): 8–28, doi: 10.1006/mpev.2000.0792 (<http://dx.doi.org/10.1006/mpev.2000.0792>), PMID 10877936 (<http://www.ncbi.nlm.nih.gov/pubmed/10877936>).
- Barthélémy, Jean-Pierre (1989), "From copair hypergraphs to median graphs with latent vertices", *Discrete Mathematics* **76** (1): 9–28, doi: 10.1016/0012-365X(89)90283-5 ([http://dx.doi.org/10.1016/0012-365X\(89\)90283-5](http://dx.doi.org/10.1016/0012-365X(89)90283-5)), MR 1002234 (<http://www.ams.org/mathscinet-getitem?mr=1002234>).
- Birkhoff, Garrett; Kiss, S. A. (1947), "A ternary operation in distributive lattices" (<http://projecteuclid.org/euclid.bams/1183510977>), *Bulletin of the American Mathematical Society* **53** (1): 749–752, doi: 10.1090/S0002-9904-1947-08864-9 (<http://dx.doi.org/10.1090/S0002-9904-1947-08864-9>), MR 0021540 (<http://www.ams.org/mathscinet-getitem?mr=0021540>).
- Buneman, P. (1971), "The recovery of trees from measures of dissimilarity", in Hodson, F. R.; Kendall, D. G.; Tautu, P. T., *Mathematics in the Archaeological and Historical Sciences*, Edinburgh University Press, pp. 387–395.

- Chepoi, V.; Dragan, F.; Vaxès, Y. (2002), "Center and diameter problems in planar quadrangulations and triangulations" (<http://portal.acm.org/citation.cfm?id=545381.545427>), *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pp. 346–355.
- Chepoi, V.; Facciullini, C.; Vaxès, Y. (2004), "Median problem in some plane triangulations and quadrangulations", *Computational Geometry: Theory & Applications* **27**: 193–210, doi: [10.1016/j.comgeo.2003.11.002](https://doi.org/10.1016/j.comgeo.2003.11.002) (<http://dx.doi.org/10.1016/j.comgeo.2003.11.002>).
- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM Journal on Computing* **14**: 210–223, doi: [10.1137/0214017](https://doi.org/10.1137/0214017) (<http://dx.doi.org/10.1137/0214017>), MR 0774940 (<http://www.ams.org/mathscinet-getitem?mr=0774940>).
- Chung, F. R. K.; Graham, R. L.; Saks, M. E. (1987), "Dynamic search in graphs" (<http://www.math.ucsd.edu/~fan/mypaps/fanpap/98dynamicsearch.pdf>), in Wilf, H., *Discrete Algorithms and Complexity (Kyoto, 1986)*, Perspectives in Computing **15**, New York: Academic Press, pp. 351–387, MR 0910939 (<http://www.ams.org/mathscinet-getitem?mr=0910939>).
- Chung, F. R. K.; Graham, R. L.; Saks, M. E. (1989), "A dynamic location problem for graphs" (<http://www.math.ucsd.edu/~fan/mypaps/fanpap/101location.pdf>), *Combinatorica* **9** (2): 111–132, doi: [10.1007/BF02124674](https://doi.org/10.1007/BF02124674) (<http://dx.doi.org/10.1007/BF02124674>).
- Day, William H. E.; McMorris, F. R. (2003), *Axiomatic Consensus [sic] Theory in Group Choice and Bioinformatics*, Society for Industrial and Applied Mathematics, pp. 91–94, ISBN 0-89871-551-2.
- Dress, A.; Hendy, M.; Huber, K.; Moulton, V. (1997), "On the number of vertices and edges of the Buneman graph", *Annals of Combinatorics* **1** (1): 329–337, doi: [10.1007/BF02558484](https://doi.org/10.1007/BF02558484) (<http://dx.doi.org/10.1007/BF02558484>), MR 1630739 (<http://www.ams.org/mathscinet-getitem?mr=1630739>).
- Dress, A.; Huber, K.; Moulton, V. (1997), "Some variations on a theme by Buneman", *Annals of Combinatorics* **1** (1): 339–352, doi: [10.1007/BF02558485](https://doi.org/10.1007/BF02558485) (<http://dx.doi.org/10.1007/BF02558485>), MR 1630743 (<http://www.ams.org/mathscinet-getitem?mr=1630743>).
- Duffus, Dwight; Rival, Ivan (1983), "Graphs orientable as distributive lattices", *Proceedings of the American Mathematical Society (American Mathematical Society)* **88** (2): 197–200, doi: [10.2307/2044697](https://doi.org/10.2307/2044697) (<http://dx.doi.org/10.2307/2044697>), JSTOR 2044697 (<http://www.jstor.org/stable/2044697>).
- Feder, T. (1995), *Stable Networks and Product Graphs*, Memoirs of the American Mathematical Society **555**.
- Hagauer, Johann; Imrich, Wilfried; Klavžar, Sandi (1999), "Recognizing median graphs in subquadratic time", *Theoretical Computer Science* **215** (1–2): 123–136, doi: [10.1016/S0304-3975\(97\)00136-9](https://doi.org/10.1016/S0304-3975(97)00136-9) ([http://dx.doi.org/10.1016/S0304-3975\(97\)00136-9](http://dx.doi.org/10.1016/S0304-3975(97)00136-9)), MR 1678773 (<http://www.ams.org/mathscinet-getitem?mr=1678773>).
- Hell, Pavol (1976), "Graph retractions", *Colloquio Internazionale sulle Teorie Combinatorie (Roma, 1973), Tomo II*, Atti dei Convegni Lincei **17**, Rome: Accad. Naz. Lincei, pp. 263–268, MR 0543779 (<http://www.ams.org/mathscinet-getitem?mr=0543779>).
- Imrich, Wilfried; Klavžar, Sandi (1998), "A convexity lemma and expansion procedures for bipartite graphs", *European Journal of Combinatorics* **19** (6): 677–686, doi: [10.1006/eujc.1998.0229](https://doi.org/10.1006/eujc.1998.0229) (<http://dx.doi.org/10.1006/eujc.1998.0229>), MR 1642702 (<http://www.ams.org/mathscinet-getitem?mr=1642702>).
- Imrich, Wilfried; Klavžar, Sandi (2000), *Product Graphs: Structure and Recognition*, Wiley, ISBN 0-471-37039-8, MR 788124 (<http://www.ams.org/mathscinet-getitem?mr=788124>).
- Imrich, Wilfried; Klavžar, Sandi; Mulder, Henry Martyn (1999), "Median graphs and triangle-free graphs", *SIAM Journal on Discrete Mathematics* **12** (1): 111–118, doi: [10.1137/S0895480197323494](https://doi.org/10.1137/S0895480197323494) (<http://dx.doi.org/10.1137/S0895480197323494>), MR 1666073 (<http://www.ams.org/mathscinet-getitem?mr=1666073>).
- Itai, A.; Rodeh, M. (1978), "Finding a minimum circuit in a graph", *SIAM Journal on Computing* **7** (4): 413–423, doi: [10.1137/0207033](https://doi.org/10.1137/0207033) (<http://dx.doi.org/10.1137/0207033>), MR 0508603 (<http://www.ams.org/mathscinet-getitem?mr=0508603>).
- Jha, Pranava K.; Slutzki, Giora (1992), "Convex-expansion algorithms for recognizing and isometric embedding of median graphs", *Ars Combinatoria* **34**: 75–92, MR 1206551 (<http://www.ams.org/mathscinet-getitem?mr=1206551>).

- mathscinet-getitem?mr=1206551).
- Klavžar, Sandi; Mulder, Henry Martyn (1999), "Median graphs: characterizations, location theory and related structures", *Journal of Combinatorial Mathematics and Combinatorial Computing* **30**: 103–127, MR 1705337 (<http://www.ams.org/mathscinet-getitem?mr=1705337>).
 - Klavžar, Sandi; Mulder, Henry Martyn; Škrekovski, Riste (1998), "An Euler-type formula for median graphs", *Discrete Mathematics* **187** (1): 255–258, doi: 10.1016/S0012-365X(98)00019-3 ([http://dx.doi.org/10.1016/S0012-365X\(98\)00019-3](http://dx.doi.org/10.1016/S0012-365X(98)00019-3)), MR 1630736 (<http://www.ams.org/mathscinet-getitem?mr=1630736>).
 - Knuth, Donald E. (2008), "Median algebras and median graphs", *The Art of Computer Programming*, IV, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions, Addison-Wesley, pp. 64–74, ISBN 978-0-321-53496-5.
 - Mulder, Henry Martyn (1980), " n -cubes and median graphs", *Journal of Graph Theory* **4** (1): 107–110, doi: 10.1002/jgt.3190040112 (<http://dx.doi.org/10.1002/jgt.3190040112>), MR 0558458 (<http://www.ams.org/mathscinet-getitem?mr=0558458>).
 - Mulder, Henry Martyn; Schrijver, Alexander (1979), "Median graphs and Helly hypergraphs", *Discrete Mathematics* **25** (1): 41–50, doi: 10.1016/0012-365X(79)90151-1 ([http://dx.doi.org/10.1016/0012-365X\(79\)90151-1](http://dx.doi.org/10.1016/0012-365X(79)90151-1)), MR 0522746 (<http://www.ams.org/mathscinet-getitem?mr=0522746>).
 - Nebesk'y, Ladislav (1971), "Median graphs", *Commentationes Mathematicae Universitatis Carolinae* **12**: 317–325, MR 0286705 (<http://www.ams.org/mathscinet-getitem?mr=0286705>).
 - Škrekovski, Riste (2001), "Two relations for median graphs", *Discrete Mathematics* **226** (1): 351–353, doi: 10.1016/S0012-365X(00)00120-5 ([http://dx.doi.org/10.1016/S0012-365X\(00\)00120-5](http://dx.doi.org/10.1016/S0012-365X(00)00120-5)), MR 1802603 (<http://www.ams.org/mathscinet-getitem?mr=1802603>).
 - Soltan, P.; Zambitskii, D.; Prisăcaru, C. (1973), *Extremal problems on graphs and algorithms of their solution* (in Russian), Chișinău: Știința.

External links

- Median graphs (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_211.html), Information System for Graph Class Inclusions.
- Network (<http://www.fluxus-engineering.com/sharenet.htm>), Free Phylogenetic Network Software. Network generates evolutionary trees and networks from genetic, linguistic, and other data.
- PhyloMurka (<http://sourceforge.net/projects/phylomurka>), open-source software for median network computations from biological data.

Graph isomorphism

Graph isomorphism

In graph theory, an **isomorphism of graphs** G and H is a bijection between the vertex sets of G and H

$$f: V(G) \rightarrow V(H)$$

such that any two vertices u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H . This kind of bijection is commonly called "edge-preserving bijection", in accordance with the general notion of isomorphism being a structure-preserving bijection.

If an isomorphism exists between two graphs, then the graphs are called **isomorphic** and we write $G \simeq H$. In the case when the bijection is a mapping of a graph onto itself, i.e., when G and H are one and the same graph, the bijection is called an automorphism of G .

The graph isomorphism is an equivalence relation on graphs and as such it partitions the class of all graphs into equivalence classes. A set of graphs isomorphic to each other is called an **isomorphism class of graphs**.

The two graphs shown below are isomorphic, despite their different looking drawings.

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(e) = 5$ $f(f) = 2$ $f(g) = 4$ $f(h) = 7$

Variations

In the above definition, graphs are understood to be undirected non-labeled non-weighted graphs. However, the notion of isomorphism may be applied to all other variants of the notion of graph, by adding the requirements to preserve the corresponding additional elements of structure: arc directions, edge weights, etc., with the following exception. When spoken about graph labeling with *unique labels*, commonly taken from the integer range $1, \dots, n$, where n is the number of the vertices of the graph, two labeled graphs are said to be isomorphic if the corresponding underlying unlabeled graphs are isomorphic.

Motivation

The formal notion of "isomorphism", e.g., of "graph isomorphism", captures the informal notion that some objects have "the same structure" if one ignores individual distinctions of "atomic" components of objects in question. Whenever individuality of "atomic" components (vertices and edges, for graphs) is important for correct representation of whatever is modeled by graphs, the model is refined by imposing additional restrictions on the structure, and other mathematical objects are used: digraphs, labeled graphs, colored graphs, rooted trees and so on. The isomorphism relation may also be defined for all these generalizations of graphs: the isomorphism bijection

must preserve the elements of structure which define the object type in question: arcs, labels, vertex/edge colors, the root of the rooted tree, etc.

The notion of "graph isomorphism" allows us to distinguish graph properties inherent to the structures of graphs themselves from properties associated with graph representations: graph drawings, data structures for graphs, graph labelings, etc. For example, if a graph has exactly one cycle, then all graphs in its isomorphism class also have exactly one cycle. On the other hand, in the common case when the vertices of a graph are (*represented by*) the integers 1, 2, ..., N, then the expression

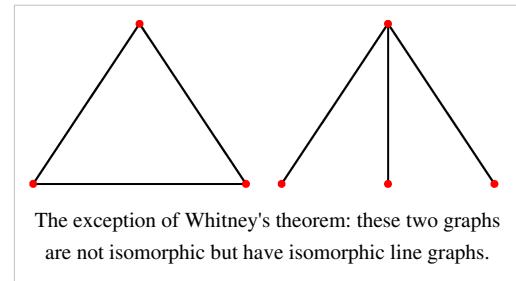
$$\sum_{v \in V(G)} v \cdot \deg v$$

may be different for two isomorphic graphs.

Whitney theorem

Main article: Whitney graph isomorphism theorem

The **Whitney graph isomorphism theorem**, shown by H. Whitney, states that two connected graphs are isomorphic if and only if their line graphs are isomorphic, with a single exception: K_3 , the complete graph on three vertices, and the complete bipartite graph $K_{1,3}$, which are not isomorphic but both have K_3 as their line graph. The Whitney graph theorem can be extended to hypergraphs.^[1]



Recognition of graph isomorphism

Main article: Graph isomorphism problem

While graph isomorphism may be studied in a classical mathematical way, as exemplified by the Whitney theorem, it is recognized that it is a problem to be tackled with an algorithmic approach. The computational problem of determining whether two finite graphs are isomorphic is called the graph isomorphism problem.

Its practical applications include primarily cheminformatics, mathematical chemistry (identification of chemical compounds), and electronic design automation (verification of equivalence of various representations of the design of an electronic circuit).

The graph isomorphism problem is one of few standard problems in computational complexity theory belonging to NP, but not known to belong to either of its well-known (and, if P ≠ NP, disjoint) subsets: P and NP-complete. It is one of only two, out of 12 total, problems listed in Garey & Johnson (1979) whose complexity remains unresolved, the other being integer factorization. It is however known that if the problem is NP-complete then the polynomial hierarchy collapses to a finite level.

Its generalization, the subgraph isomorphism problem, is known to be NP-complete.

The main areas of research for the problem are design of fast algorithms and theoretical investigations of its computational complexity, both for the general problem and for special classes of graphs.

Notes

[1] Dirk L. Vertigan, Geoffrey P. Whittle: A 2-Isomorphism Theorem for Hypergraphs. *J. Comb. Theory, Ser. B* 71(2): 215–230. 1997.

References

- Garey, Michael R.; Johnson, David S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, ISBN 0-7167-1045-5

Graph isomorphism problem

List of unsolved problems in computer science

Is the graph isomorphism problem in P, or NP-complete, or neither?

The **graph isomorphism problem** is the computational problem of determining whether two finite graphs are isomorphic.

Besides its practical importance, the graph isomorphism problem is a curiosity in computational complexity theory as it is one of a very small number of problems belonging to NP neither known to be solvable in polynomial time nor NP-complete: it is one of only 12 such problems listed by Garey & Johnson (1979), and one of only two of that list whose complexity remains unresolved (the other being integer factorization).^[1] As of 2008[2] the best algorithm (Luks, 1983) has run time $2^{O(\sqrt{n} \log n)}$ for graphs with n vertices.

It is known that the graph isomorphism problem is in the low hierarchy of class NP, which implies that it is not NP-complete unless the polynomial time hierarchy collapses to its second level.^[3]

At the same time, isomorphism for many special classes of graphs can be solved in polynomial time, and in practice graph isomorphism can often be solved efficiently.

This problem is a special case of the subgraph isomorphism problem, which is known to be NP-complete. It is also known to be a special case of the non-abelian hidden subgroup problem over the symmetric group.

State of the art

The best current theoretical algorithm is due to Eugene Luks (1983), and is based on the earlier work by Luks (1981), Babai & Luks (1982), combined with a *subfactorial* algorithm due to Zemlyachenko (1982). The algorithm relies on the classification of finite simple groups. Without CFSG, a slightly weaker bound $2^{O(\sqrt{n} \log_2 n)}$ was obtained first for strongly regular graphs by László Babai (1980), and then extended to general graphs by Babai & Luks (1982). Improvement of the exponent \sqrt{n} is a major open problem; for strongly regular graphs this was done by Spielman (1996). For hypergraphs of bounded rank, a subexponential upper bound matching the case of graphs, was recently obtained by Babai & Codenotti (2008).

On a side note, the graph isomorphism problem is computationally equivalent to the problem of computing the automorphism group of a graph, and is weaker than the permutation group isomorphism problem, and the permutation group intersection problem. For the latter two problems, Babai, Kantor and Luks (1983) obtained complexity bounds similar to that for the graph isomorphism.^[4]

There are several competing practical algorithms for graph isomorphism, due to McKay (1981), Schmidt & Druffel (1976), Ullman (1976), etc. While they seem to perform well on random graphs, a major drawback of these algorithms is their exponential time performance in the worst case.^[5]

Solved special cases

A number of important special cases of the graph isomorphism problem have efficient, polynomial-time solutions:

- Trees^[6]
- Planar graphs (In fact, planar graph isomorphism is in log space, a class contained in P.)
- Interval graphs
- Permutation graphs
- Partial k -trees
- Bounded-parameter graphs
 - Graphs of bounded genus^[7] (Note: planar graphs are graphs of genus 0)
 - Graphs of bounded degree
 - Graphs with bounded eigenvalue multiplicity
 - k -Contractible graphs (a generalization of bounded degree and bounded genus)^[8]
 - Color-preserving isomorphism of colored graphs with bounded color multiplicity (i.e., at most k vertices have the same color for a fixed k) is in class NC, which is a subclass of P.^[9]

Complexity class GI

Since the graph isomorphism problem is neither known to be NP-complete nor to be tractable, researchers have sought to gain insight into the problem by defining a new class **GI**, the set of problems with a polynomial-time Turing reduction to the graph isomorphism problem. If in fact the graph isomorphism problem is solvable in polynomial time, **GI** would equal **P**.

As is common for complexity classes within the polynomial time hierarchy, a problem is called **GI-hard** if there is a polynomial-time Turing reduction from any problem in **GI** to that problem, i.e., a polynomial-time solution to a GI-hard problem would yield a polynomial-time solution to the graph isomorphism problem (and so all problems in **GI**). A problem X is called complete for **GI**, or **GI-complete**, if it is both GI-hard and a polynomial-time solution to the GI problem would yield a polynomial-time solution to X .

The graph isomorphism problem is contained in both **NP** and co-**AM**. GI is contained in and low for Parity P, as well as contained in the potentially much smaller class SPP^[10]. That it lies in Parity P means that the graph isomorphism problem is no harder than determining whether a polynomial-time nondeterministic Turing machine has an even or odd number of accepting paths. GI is also contained in and low for ZPP^{NP}. This essentially means that an efficient Las Vegas algorithm with access to an NP oracle can solve graph isomorphism so easily that it gains no power from being given the ability to do so in constant time.

GI-complete and GI-hard problems

Isomorphism of other objects

There are a number of classes of mathematical objects for which the problem of isomorphism is a GI-complete problem. A number of them are graphs endowed with additional properties or restrictions:

- digraphs
- labelled graphs, with the proviso that an isomorphism is not required to preserve the labels, but only the equivalence relation consisting of pairs of vertices with the same label
- "polarized graphs" (made of a complete graph K_m and an empty graph K_n plus some edges connecting the two; their isomorphism must preserve the partition)
- 2-colored graphs
- explicitly given finite structures
- multigraphs

- hypergraphs
- finite automata
- Markov Decision Processes^[11]
- commutative class 3 nilpotent (i.e., $xyz = 0$ for every elements x, y, z) semigroups
- finite rank associative algebras over a fixed algebraically closed field with zero squared radical and commutative factor over the radical^[12]
- context-free grammars
- balanced incomplete block designs
- Recognizing combinatorial isomorphism of convex polytopes represented by vertex-facet incidences.^[13]

This list is incomplete; you can help by expanding it^[2].

GI-complete classes of graphs

A class of graphs is called GI-complete if recognition of isomorphism for graphs from this subclass is a GI-complete problem. The following classes are GI-complete:

- connected graphs
- graphs of diameter 2 and radius 1
- directed acyclic graphs
- regular graphs.
- bipartite graphs without non-trivial strongly regular subgraphs
- bipartite Eulerian graphs
- bipartite regular graphs
- line graphs
- chordal graphs
- regular self-complementary graphs
- polytopal graphs of general, simple, and simplicial convex polytopes in arbitrary dimensions

This list is incomplete; you can help by expanding it^[2].

Many classes of digraphs are also GI-complete.

Other GI-complete problems

There are other nontrivial GI-complete problems in addition to isomorphism problems.

- The recognition of self-complementarity of a graph or digraph.^[14]
- A clique problem for a class of so-called M -graphs. It is shown that finding an isomorphism for n -vertex graphs is equivalent to finding an n -clique in an M -graph of size n^2 . This fact is interesting because the problem of finding an $(n - \varepsilon)$ -clique in a M -graph of size n^2 is NP-complete for arbitrarily small positive ε .
- The problem of homeomorphism of 2-complexes.^[15]

GI-hard problems

- The problem of counting the number of isomorphisms between two graphs is polynomial-time equivalent to the problem of telling whether even one exists.^[16]
- The problem of deciding whether two convex polytopes given by either the V-description or H-description are projectively or affinely isomorphic. The latter means existence of a projective or affine map between the spaces that contain the two polytopes (not necessarily of the same dimension) which induces a bijection between the polytopes.

Program checking

Blum and Kannan^[17] have shown a program checker for graph isomorphism. Suppose P is a claimed polynomial-time procedure that checks if two graphs are isomorphic, but it is not trusted. To check if G and H are isomorphic:

- Ask P whether G and H are isomorphic.
 - If the answer is "yes":
 - Attempt to construct an isomorphism using P as subroutine. Mark a vertex u in G and v in H , and modify the graphs to make them distinctive (with a small local change). Ask P if the modified graphs are isomorphic. If no, change v to a different vertex. Continue searching.
 - Either the isomorphism will be found (and can be verified), or P will contradict itself.
 - If the answer is "no":
 - Perform the following 100 times. Choose randomly G or H , and randomly permute its vertices. Ask P if the graph is isomorphic to G and H . (As in AM protocol for graph nonisomorphism).
 - If any of the tests are failed, judge P as invalid program. Otherwise, answer "no".

This procedure is polynomial-time and gives the correct answer if P is a correct program for graph isomorphism. If P is not a correct program, but answers correctly on G and H , the checker will either give the correct answer, or detect invalid behaviour of P . If P is not a correct program, and answers incorrectly on G and H , the checker will detect invalid behaviour of P with high probability, or answer wrong with probability 2^{-100} .

Notably, P is used only as a blackbox.

Applications

In cheminformatics and in mathematical chemistry, graph isomorphism testing is used to identify a chemical compound within a chemical database.^[18] Also, in organic mathematical chemistry graph isomorphism testing is useful for generation of molecular graphs and for computer synthesis.

Chemical database search is an example of graphical data mining, where the graph canonization approach is often used.^[19] In particular, a number of identifiers for chemical substances, such as SMILES and InChI, designed to provide a standard and human-readable way to encode molecular information and to facilitate the search for such information in databases and on the web, use canonization step in their computation, which is essentially the canonization of the graph which represents the molecule.

In electronic design automation graph isomorphism is the basis of the Layout Versus Schematic (LVS) circuit design step, which is a verification whether the electric circuits represented by a circuit schematic and an integrated circuit layout are the same.

Notes

- [1] The latest one resolved was minimum-weight triangulation, proved to be NP-complete in 2006. .
- [2] http://en.wikipedia.org/w/index.php?title=Graph_isomorphism_problem&action=edit
- [3] Uwe Schöning, "Graph isomorphism is in the low hierarchy", Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, 1987, 114–124; also: *Journal of Computer and System Sciences*, vol. 37 (1988), 312–323
- [4] László Babai, William Kantor, Eugene Luks, Computational complexity and the classification of finite simple groups (<http://portal.acm.org/citation.cfm?id=1382437.1382817>), Proc. 24th FOCS (1983), pp. 162-171.
- [5] P. Foggia, C. Sansone, M. Vento, A Performance Comparison of Five Algorithms for Graph Isomorphism (http://www.engr.uconn.edu/~vkk06001/GraphIsomorphism/Papers/VF_SD_NAUTY_Ullman_Experiments.pdf), Proc. 3rd IAPR-TC15 Workshop Graph-Based Representations in Pattern Recognition, 2001, pp. 188-199.
- [6] P.J. Kelly, "A congruence theorem for trees" Pacific J. Math., 7 (1957) pp. 961–968; .
- [7] : .
- [8] Gary L. Miller: Isomorphism Testing and Canonical Forms for k -Contractible Graphs (A Generalization of Bounded Valence and Bounded Genus). Proc. Int. Conf. on Foundations of Computer Theory, 1983, pp. 310–327 (*Lecture Notes in Computer Science*, vol. 158, full paper in:

- Information and Control*, 56(1–2):1–20, 1983.)
- [9] Eugene Luks, "Parallel algorithms for permutation groups and graph isomorphism", Proc. IEEE Symp. Foundations of Computer Science, 1986, 292–302
- [10] http://qwiki.stanford.edu/wiki/Complexity_Zoo:S#spp
- [11] "On the hardness of finding symmetries in Markov decision processes", by SM Narayananmurthy, B Ravindran (<http://www.cse.iitm.ac.in/~ravi/papers/Shravan-ICML08.pdf>), Proceedings of the Twenty Fifth International Conference on Machine Learning (ICML 2008), pp. 688–696.
- [12] D.Yu.Grigor'ev, "Complexity of "wild" matrix problems and of isomorphism of algebras and graphs", *Journal of Mathematical Sciences*, Volume 22, Number 3, 1983, pp. 1285–1289, (translation of a 1981 Russian language article)
- [13] Volker Kaibel, Alexander Schwartz, "On the Complexity of Polytope Isomorphism Problems" (<http://eprintweb.org/S/authors/All/ka/Kaibel/16>), *Graphs and Combinatorics*, 19 (2):215 — 230, 2003.
- [14] Colbourn M.J., Colbourn Ch.J. "Graph isomorphism and self-complementary graphs", *SIGACT News*, 1978, vol. 10, no. 1, 25–29.
- [15] J. Shawe-Taylor, T.Pisanski, "Homeomorphism of 2-Complexes is Graph Isomorphism Complete", *SIAM Journal on Computing*, 23 (1994) 120 – 132.
- [16] R. Mathon, "A note on the graph isomorphism counting problem", *Information Processing Letters*, 8 (1979) pp. 131–132; .
- [17] Designing Programs that Check their Work (<ftp://ftp.cis.upenn.edu/pub/kannan/jacm.ps.gz>)
- [18] Christophe-André Mario Irniger (2005) "Graph Matching: Filtering Databases of Graphs Using Machine Learning", ISBN 1-58603-557-6
- [19] "Mining Graph Data", by Diane J. Cook, Lawrence B. Holder (2007) ISBN 0-470-07303-9, pp. 120–122, section 6.2.1. "Canonical Labeling" (http://books.google.com/books?id=bHGy0_H0g8QC&pg=PA119&dq=%22canonical+labeling%22+graphs#PPA120,M1)

References

- Aho, Alfred V.; Hopcroft, John; Ullman, Jeffrey D. (1974), *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley.
- Arvind, Vikraman; Köbler, Johannes (2000), "Graph isomorphism is low for ZPP(NP) and other lowness results.", *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **1770**, Springer-Verlag, pp. 431–442, ISBN 3-540-67141-2, OCLC 43526888 (<http://www.worldcat.org/oclc/43526888>).
- Arvind, Vikraman; Kurur, Piyush P. (2006), "Graph isomorphism is in SPP", *Information and Computation* **204** (5): 835–852, doi: 10.1016/j.ic.2006.02.002 (<http://dx.doi.org/10.1016/j.ic.2006.02.002>).
- Babai, László; Codenotti, Paolo (2008), "Isomorphism of Hypergraphs of Low Rank in Moderately Exponential Time", *FOCS '08: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, pp. 667–676, ISBN 978-0-7695-3436-7.
- Babai, László; Grigoryev, D. Yu.; Mount, David M. (1982), "Isomorphism of graphs with bounded eigenvalue multiplicity", *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pp. 310–324, doi: 10.1145/800070.802206 (<http://dx.doi.org/10.1145/800070.802206>), ISBN 0-89791-070-2.
- Bodlaender, Hans (1990), "Polynomial algorithms for graph isomorphism and chromatic index on partial k -trees", *Journal of Algorithms* **11** (4): 631–643, doi: 10.1016/0196-6774(90)90013-5 ([http://dx.doi.org/10.1016/0196-6774\(90\)90013-5](http://dx.doi.org/10.1016/0196-6774(90)90013-5)).
- Booth, Kellogg S.; Colbourn, C. J. (1977), *Problems polynomially equivalent to graph isomorphism*, Technical Report CS-77-04, Computer Science Department, University of Waterloo.
- Booth, Kellogg S.; Lueker, George S. (1979), "A linear time algorithm for deciding interval graph isomorphism", *Journal of the ACM* **26** (2): 183–195, doi: 10.1145/322123.322125 (<http://dx.doi.org/10.1145/322123.322125>).
- Boucher, C.; Loker, D. (2006), *Graph isomorphism completeness for perfect graphs and subclasses of perfect graphs* (<http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-32.pdf>), University of Waterloo, Technical Report CS-2006-32.
- Colbourn, C. J. (1981), "On testing isomorphism of permutation graphs", *Networks* **11**: 13–21, doi: 10.1002/net.3230110103 (<http://dx.doi.org/10.1002/net.3230110103>).
- Filotti, I. S.; Mayer, Jack N. (1980), "A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus", *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pp. 236–243, doi: 10.1145/800141.804671 (<http://dx.doi.org/10.1145/800141.804671>), ISBN 0-89791-017-6.

- Garey, Michael R.; Johnson, David S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, ISBN 978-0-7167-1045-5, OCLC 11745039 (<http://www.worldcat.org/oclc/11745039>).
- Hopcroft, John; Wong, J. (1974), "Linear time algorithm for isomorphism of planar graphs", *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pp. 172–184, doi: 10.1145/800119.803896 (<http://dx.doi.org/10.1145/800119.803896>).
- Köbler, Johannes; Schöning, Uwe; Torán, Jacobo (1992), "Graph isomorphism is low for PP", *Computational Complexity* **2** (4): 301–330, doi: 10.1007/BF01200427 (<http://dx.doi.org/10.1007/BF01200427>).
- Köbler, Johannes; Schöning, Uwe; Torán, Jacobo (1993), *The Graph Isomorphism Problem: Its Structural Complexity*, Birkhäuser, ISBN 978-0-8176-3680-7, OCLC 246882287 (<http://www.worldcat.org/oclc/246882287>).
- Kozen, Dexter (1978), "A clique problem equivalent to graph isomorphism", *ACM SIGACT News* **10** (2): 50–52, doi: 10.1145/990524.990529 (<http://dx.doi.org/10.1145/990524.990529>).
- Luks, Eugene M. (1982), "Isomorphism of graphs of bounded valence can be tested in polynomial time", *Journal of Computer and System Sciences* **25**: 42–65, doi: 10.1016/0022-0000(82)90009-5 ([http://dx.doi.org/10.1016/0022-0000\(82\)90009-5](http://dx.doi.org/10.1016/0022-0000(82)90009-5)).
- McKay, Brendan D. (1981), "Practical graph isomorphism" (<http://cs.anu.edu.au/~bdm/nauty/PGI/>), *Congressus Numerantium* **30**: 45–87, 10th. Manitoba Conference on Numerical Mathematics and Computing (Winnipeg, 1980).
- Miller, Gary (1980), "Isomorphism testing for graphs of bounded genus", *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pp. 225–235, doi: 10.1145/800141.804670 (<http://dx.doi.org/10.1145/800141.804670>), ISBN 0-89791-017-6.
- Schmidt, Douglas C.; Druffel, Larry E. (1976), "A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices", *J. ACM (ACM)* **23** (3): 433–445, doi: 10.1145/321958.321963 (<http://dx.doi.org/10.1145/321958.321963>), ISSN 0004-5411 (<http://www.worldcat.org/issn/0004-5411>).
- Spielman, Daniel A. (1996), "Faster isomorphism testing of strongly regular graphs", *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, ACM, pp. 576–584, ISBN 978-0-89791-785-8.
- Ullman, Julian R. (1976), "An algorithm for subgraph isomorphism", *Journal of the ACM* **23**: 31–42, doi: 10.1145/321921.321925 (<http://dx.doi.org/10.1145/321921.321925>).

Surveys and monographs

- Read, Ronald C.; Corneil, Derek G. (1977), "The graph isomorphism disease", *Journal of Graph Theory* **1** (4): 339–363, doi: 10.1002/jgt.3190010410 (<http://dx.doi.org/10.1002/jgt.3190010410>), MR 0485586 (<http://www.ams.org/mathscinet-getitem?mr=0485586>).
- Gati, G. "Further annotated bibliography on the isomorphism disease." – *Journal of Graph Theory* 1979, 3, 95–109.
- Zemlyachenko, V. N.; Korneenko, N. M.; Tyshkevich, R. I. (1985), "Graph isomorphism problem", *Journal of Mathematical Sciences* **29** (4): 1426–1481, doi: 10.1007/BF02104746 (<http://dx.doi.org/10.1007/BF02104746>). (Translated from *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V. A. Steklova AN SSSR* (Records of Seminars of the Leningrad Department of Steklov Institute of Mathematics of the USSR Academy of Sciences), Vol. 118, pp. 83–158, 1982.)
- Arvind, V.; Jacobo Torán (2005), "Isomorphism Testing: Perspectives and Open Problems" (<http://theorie.informatik.uni-ulm.de/Personen/toran/beatcs/column86.pdf>), *Bulletin of the European Association for Theoretical Computer Science* (no. 86): 66–84 . (A brief survey of open questions related to the isomorphism problem for graphs, rings and groups.)

- Köbler, Johannes; Uwe Schöning, Jacobo Torán (1993), *Graph Isomorphism Problem: The Structural Complexity*, Birkhäuser Verlag, ISBN 0-8176-3680-3, OCLC 246882287 (<http://www.worldcat.org/oclc/246882287>) . (*From the book cover:* The books focuses on the issue of the computational complexity of the problem and presents several recent results that provide a better understanding of the relative position of the problem in the class NP as well as in other complexity classes.)
- Johnson, David S. (2005), "The NP-Completeness Column", *ACM Transactions on Algorithms* 1 (no. 1): 160–176, doi: 10.1145/1077464.1077476 (<http://dx.doi.org/10.1145/1077464.1077476>). (This 24th edition of the Column discusses the state of the art for the open problems from the book *Computers and Intractability* and previous columns, in particular, for Graph Isomorphism.)
- Torán, Jacobo; Fabian Wagner (2009), "The Complexity of Planar Graph Isomorphism" (<http://theorie.informatik.uni-ulm.de/Personen/toran/beatcs/column97.pdf>), *Bulletin of the European Association for Theoretical Computer Science* (no. 97) .

Software

- Graph Isomorphism (<http://www.cs.sunysb.edu/~algorith/files/graph-isomorphism.shtml>), review of implementations, The Stony Brook Algorithm Repository (<http://www.cs.sunysb.edu/~algorith>).

Graph canonization

In graph theory, a branch of mathematics, **graph canonization** is the problem finding a canonical form of a given graph G . A canonical form is a labeled graph $\text{Canon}(G)$ that is isomorphic to G , such that every graph that is isomorphic to G has the same canonical form as G . Thus, from a solution to the graph canonization problem, one could also solve the problem of graph isomorphism: to test whether two graphs G and H are isomorphic, compute their canonical forms $\text{Canon}(G)$ and $\text{Canon}(H)$, and test whether these two canonical forms are identical.

The canonical form of a graph is an example of a complete graph invariant: every two isomorphic graphs have the same canonical form, and every two non-isomorphic graphs have different canonical forms. Conversely, every complete invariant of graphs may be used to construct a canonical form. The vertex set of an n -vertex graph may be identified with the integers from 1 to n , and using such an identification a canonical form of a graph may also be described as a permutation of its vertices. Canonical forms of a graph are also called **canonical labelings**, and graph canonization is also sometimes known as **graph canonicalization**.

Computational complexity

Clearly, the graph canonization problem is at least as computationally hard as the graph isomorphism problem. In fact, graph isomorphism is even AC^0 -reducible to graph canonization. However it is still an open question whether the two problems are polynomial time equivalent.

While existence of (deterministic) polynomial algorithms for Graph Isomorphism is still an open problem in the computational complexity theory, in 1977 Laszlo Babai reported that with probability at least $1 - \exp(-O(n))$, a simple vertex classification algorithm after only two refinement steps produces a canonical labeling of a graph chosen uniformly at random from the set of all n -vertex graphs. Small modifications and an added depth-first search step produce canonical labeling of such uniformly-chosen random graphs in linear expected time. This result sheds some light on the fact why many reported graph isomorphism algorithms behave well in practice. This was an important breakthrough in probabilistic complexity theory which became widely known in its manuscript form and which was still cited as an "unpublished manuscript" long after it was reported at a symposium.

A commonly known canonical form is the lexicographically smallest graph within the isomorphism class, which is the graph of the class with lexicographically smallest adjacency matrix considered as a linear string. However, the

computation of the lexicographically smallest graph is NP-hard.

For trees, a concise polynomial canonization algorithm requiring $O(n)$ space is presented in. Begin by labeling each vertex with the string 01. Iteratively for each non-leaf x remove the leading 0 and trailing 1 from x 's label; then sort x 's label along with the labels of all adjacent leaves in lexicographic order. Concatenate these sorted labels, add back a leading 0 and trailing 1, make this the new label of x , and delete the adjacent leaves. If there are two vertices remaining, concatenate their labels in lexicographic order.

Applications

Graph canonization is the essence of many graph isomorphism algorithms.

A common application of graph canonization is in graphical data mining, in particular in chemical database applications.

A number of identifiers for chemical substances, such as SMILES and InChI, designed to provide a standard and human-readable way to encode molecular information and to facilitate the search for such information in databases and on the web, use canonization step in their computation, which is essentially the canonization of the graph which represents the molecule.

References

Subgraph isomorphism problem

In theoretical computer science, the **subgraph isomorphism** problem is a computational task in which two graphs G and H are given as input, and one must determine whether G contains a subgraph that is isomorphic to H . Subgraph isomorphism is a generalization of both the maximum clique problem and the problem of testing whether a graph contains a Hamiltonian cycle, and is therefore NP-complete.^[1] However certain other cases of subgraph isomorphism may be solved in polynomial time.

Sometimes the name **subgraph matching** is also used for the same problem. This name puts emphasis on finding such a subgraph as opposed to the bare decision problem.

Decision problem and computational complexity

To prove subgraph isomorphism is NP-complete, it must be formulated as a decision problem. The input to the decision problem is a pair of graphs G and H . The answer to the problem is positive if H is isomorphic to a subgraph of G , and negative otherwise.

Formal question:

Let $G = (V, E)$, $H = (V', E')$ be graphs. Is there a subgraph $G_0 = (V_0, E_0) : V_0 \subseteq V, E_0 = E \cap (V_0 \times V_0)$ such that $G_0 \cong H$? I.e., does there exist an $f: V_0 \rightarrow V'$ such that $(v_1, v_2) \in E_0 \Leftrightarrow (f(v_1), f(v_2)) \in E'$?

The proof of subgraph isomorphism being NP-complete is simple and based on reduction of the clique problem, an NP-complete decision problem in which the input is a single graph G and a number k , and the question is whether G contains a complete subgraph with k vertices. To translate this to a subgraph isomorphism problem, simply let H be the complete graph K_k ; then the answer to the subgraph isomorphism problem for G and H is equal to the answer to the clique problem for G and k . Since the clique problem is NP-complete, this polynomial-time many-one reduction shows that subgraph isomorphism is also NP-complete.

An alternative reduction from the Hamiltonian cycle problem translates a graph G which is to be tested for Hamiltonicity into the pair of graphs G and H , where H is a cycle having the same number of vertices as G . Because the Hamiltonian cycle problem is NP-complete even for planar graphs, this shows that subgraph isomorphism remains NP-complete even in the planar case.

Subgraph isomorphism is a generalization of the graph isomorphism problem, which asks whether G is isomorphic to H : the answer to the graph isomorphism problem is true if and only if G and H both have the same numbers of vertices and edges and the subgraph isomorphism problem for G and H is true. However the complexity-theoretic status of graph isomorphism remains an open question.

In the context of the Aanderaa–Karp–Rosenberg conjecture on the query complexity of monotone graph properties, Gröger (1992) showed that any subgraph isomorphism problem has query complexity $\Omega(n^{3/2})$; that is, solving the subgraph isomorphism requires an algorithm to check the presence or absence in the input of $\Omega(n^{3/2})$ different edges in the graph.^[2]

Algorithms

Ullmann (1976) describes a recursive backtracking procedure for solving the subgraph isomorphism problem. Although its running time is, in general, exponential, it takes polynomial time for any fixed choice of H (with a polynomial that depends on the choice of H). When G is a planar graph and H is fixed, the running time of subgraph isomorphism can be reduced to linear time.

Applications

As subgraph isomorphism has been applied in the area of cheminformatics to find similarities between chemical compounds from their structural formula; often in this area the term **substructure search** is used. Typically a query structure is defined as SMARTS, a SMILES extension.

The closely related problem of counting the number of isomorphic copies of a graph H in a larger graph G has been applied to pattern discovery in databases, the bioinformatics of protein-protein interaction networks, and in exponential random graph methods for mathematically modeling social networks.

Ohlrich et al. (1993) describe an application of subgraph isomorphism in the computer-aided design of electronic circuits. Subgraph matching is also a substep in graph rewriting (the most runtime-intensive), and thus offered by graph rewrite tools.

The problem is also of interest in artificial intelligence, where it considered part of an array of pattern matching in graphs problems; an extension of subgraph isomorphism known as graph mining is also of interest in that area.^[3]

Notes

[1] The original paper that proves the Cook–Levin theorem already showed subgraph isomorphism to be NP-complete, using a reduction from 3-SAT involving cliques.

[2] Here Ω invokes Big Omega notation.

[3] <http://www.aaai.org/Papers/Symposia/Fall/2006/FS-06-02/FS06-02-007.pdf>; expanded version at <https://e-reports-ext.llnl.gov/pdf/332302.pdf>

References

- Cook, S. A. (1971), "The complexity of theorem-proving procedures" (<http://4mhz.de/cook.html>), *Proc. 3rd ACM Symposium on Theory of Computing*, pp. 151–158, doi: 10.1145/800157.805047 (<http://dx.doi.org/10.1145/800157.805047>).
- Eppstein, David (1999), "Subgraph isomorphism in planar graphs and related problems" (<http://www.cs.brown.edu/publications/jgaa/accepted/99/Eppstein99.3.3.pdf>), *Journal of Graph Algorithms and Applications* 3 (3):

- 1–27, arXiv: cs.DS/9911003 (<http://arxiv.org/abs/cs.DS/9911003>), doi: 10.7155/jgaa.00014 (<http://dx.doi.org/10.7155/jgaa.00014>).
- Garey, Michael R.; Johnson, David S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, ISBN 0-7167-1045-5. A1.4: GT48, pg.202.
 - Gröger, Hans Dietmar (1992), "On the randomized complexity of monotone graph properties" (http://www.inf.u-szeged.hu/actacybernetica/edb/vol10n3/pdf/Groger_1992_ActaCybernetica.pdf), *Acta Cybernetica* **10** (3): 119–127.
 - Kuramochi, Michihiro; Karypis, George (2001), "Frequent subgraph discovery", *1st IEEE International Conference on Data Mining*, p. 313, doi: 10.1109/ICDM.2001.989534 (<http://dx.doi.org/10.1109/ICDM.2001.989534>), ISBN 0-7695-1119-8.
 - Ohlrich, Miles; Ebeling, Carl; Ginting, Eka; Sather, Lisa (1993), "SubGemini: identifying subcircuits using a fast subgraph isomorphism algorithm", *Proceedings of the 30th international Design Automation Conference*, pp. 31–37, doi: 10.1145/157485.164556 (<http://dx.doi.org/10.1145/157485.164556>), ISBN 0-89791-577-1.
 - Pržulj, N.; Corneil, D. G.; Jurisica, I. (2006), "Efficient estimation of graphlet frequency distributions in protein–protein interaction networks", *Bioinformatics* **22** (8): 974–980, doi: 10.1093/bioinformatics/btl030 (<http://dx.doi.org/10.1093/bioinformatics/btl030>), PMID 16452112 (<http://www.ncbi.nlm.nih.gov/pubmed/16452112>).
 - Snijders, T. A. B.; Pattison, P. E.; Robins, G.; Handcock, M. S. (2006), "New specifications for exponential random graph models", *Sociological Methodology* **36** (1): 99–153, doi: 10.1111/j.1467-9531.2006.00176.x (<http://dx.doi.org/10.1111/j.1467-9531.2006.00176.x>).
 - Ullmann, Julian R. (1976), "An algorithm for subgraph isomorphism", *Journal of the ACM* **23** (1): 31–42, doi: 10.1145/321921.321925 (<http://dx.doi.org/10.1145/321921.321925>).
 - Jamil, Hasan (2011), "Computing Subgraph Isomorphic Queries using Structural Unification and Minimum Graph Structures", *26th ACM Symposium on Applied Computing*, pp. 1058–1063.

Color-coding

This article is about a technique in the design of graph algorithms. For the use of color to display information, see color code.

In computer science and graph theory, the method of **color-coding**^{[1][2]} efficiently finds k -vertex simple paths, k -vertex cycles, and other small subgraphs within a given graph using probabilistic algorithms, which can then be derandomized and turned into deterministic algorithms. This method shows that many subcases of the subgraph isomorphism problem (an NP-complete problem) can in fact be solved in polynomial time.

The theory and analysis of the color-coding method was proposed in 1994 by Noga Alon, Raphael Yuster, and Uri Zwick.

Results

The following results can be obtained through the method of color-coding:

- For every fixed constant k , if a graph $G = (V, E)$ contains a simple cycle of size k , then such cycle can be found in:
 - $O(V^\omega)$ expected time, or
 - $O(V^\omega \log V)$ worst-case time, where ω is the exponent of matrix multiplication.^[3]
- For every fixed constant k , and every graph $G = (V, E)$ that is in any nontrivial minor-closed graph family (e.g., a planar graph), if G contains a simple cycle of size k , then such cycle can be found in:
 - $O(V)$ expected time, or
 - $O(V \log V)$ worst-case time.
- If a graph $G = (V, E)$ contains a subgraph isomorphic to a bounded treewidth graph which has $O(\log V)$ vertices, then such a subgraph can be found in polynomial time.

The method

To solve the problem of finding a subgraph $H = (V_H, E_H)$ in a given graph $G = (V, E)$, where H can be a path, a cycle, or any bounded treewidth graph where $|V_H| = O(\log V)$, the method of color-coding begins by randomly coloring each vertex of G with $k = |V_H|$ colors, and then tries to find a colorful copy of H in colored G . Here, a graph is colorful if every vertex in it is colored with a distinct color. This method works by repeating (1) random coloring a graph and (2) finding colorful copy of the target subgraph, and eventually the target subgraph can be found if the process is repeated a sufficient number of times.

Suppose H becomes colorful with some non-zero probability p . It immediately follows that if the random coloring is repeated $\frac{1}{p}$ times, then H is expected to become colorful once. Note that though p is small, it is shown that if $|V_H| = O(\log V)$, p is only polynomially small. Suppose again there exists an algorithm such that, given a graph G and a coloring which maps each vertex of G to one of the k colors, it finds a copy of colorful H , if one exists, within some runtime $O(r)$. Then the expected time to find a copy of H in G , if one exists, is $O\left(\frac{r}{p}\right)$.

Sometimes it is also desirable to use a more restricted version of colorfulness. For example, in the context of finding cycles in planar graphs, it is possible to develop an algorithm that finds well-colored cycles. Here, a cycle is well-colored if its vertices are colored by consecutive colors.

Example

An example would be finding a simple cycle of length k in graph $G = (V, E)$.

By applying random coloring method, each simple cycle has a probability of $k!/k^k > \frac{1}{e^k}$ to become colorful, since there are k^k ways of coloring the k vertices on the path, among which there are $k!$ colorful occurrences. Then an algorithm (described below) of runtime $O(V^\omega)$ can be adopted to find colorful cycles in the randomly colored graph G . Therefore, it takes $e^k \cdot O(V^\omega)$ overall time to find a simple cycle of length k in G . The colorful cycle-finding algorithm works by first finding all pairs of vertices in V that are connected by a simple path of length $k - 1$, and then checking whether the two vertices in each pair are connected. Given a coloring function $c : V \rightarrow \{1, \dots, k\}$ to color graph G , enumerate all partitions of the color set $\{1, \dots, k\}$ into two subsets C_1, C_2 of size $k/2$ each. Note that V can be divided into V_1 and V_2 accordingly, and let G_1 and G_2 denote the subgraphs induced by V_1 and V_2 respectively. Then, recursively find colorful paths of length $k/2 - 1$ in each of G_1 and G_2 . Suppose the boolean matrix A_1 and A_2 represent the connectivity of each pair of vertices in G_1 and G_2 by a colorful path, respectively, and let B be the matrix describing the adjacency relations between vertices of V_1 and those of V_2 , the boolean product $A_1 B A_2$ gives all pairs of vertices in V that are connected by a colorful path of length $k - 1$. Thus, the recursive relation of matrix multiplications is $t(k) \leq 2^k \cdot t(k/2)$, which yields a runtime of $2^{O(k)} \cdot V^\omega \in O(V^\omega)$. Although this algorithm finds only the end points of the colorful path, another algorithm by Alon and Naor^[4] that finds colorful paths themselves can be incorporated into it.

Derandomization

The derandomization of color-coding involves enumerating possible colorings of a graph G , such that the randomness of coloring G is no longer required. For the target subgraph H in G to be discoverable, the enumeration has to include at least one instance where the H is colorful. To achieve this, enumerating a k -perfect family F of hash functions from $\{1, 2, \dots, |V|\}$ to $\{1, 2, \dots, k\}$ is sufficient. By definition, F is k -perfect if for every subset S of $\{1, 2, \dots, |V|\}$ where $|S| = k$, there exists a hash function $h \in F$ such that $h : S \rightarrow \{1, 2, \dots, k\}$ is perfect. In other words, there must exist a hash function in F that colors any given k vertices with k distinct colors.

There are several approaches to construct such a k -perfect hash family:

1. The best explicit construction is by Moni Naor, Leonard J. Schulman, and Aravind Srinivasan,^[5] where a family of size $e^k k^{O(\log k)} \log |V|$ can be obtained. This construction does not require the target subgraph to exist in the original subgraph finding problem.
2. Another explicit construction by Jeanette P. Schmidt and Alan Siegel^[6] yields a family of size $2^{O(k)} \log^2 |V|$.
3. Another construction that appears in the original paper of Noga Alon et al. can be obtained by first building a k -perfect family that maps $\{1, 2, \dots, |V|\}$ to $\{1, 2, \dots, k^2\}$, followed by building another k -perfect family that maps $\{1, 2, \dots, k^2\}$ to $\{1, 2, \dots, k\}$. In the first step, it is possible to construct such a family with $2n \log k$ random bits that are almost $2 \log k$ -wise independent,^{[7][8]} and the sample space needed for generating those random bits can be as small as $k^{O(1)} \log |V|$. In the second step, it has been shown by Jeanette P. Schmidt and Alan Siegel that the size of such k -perfect family can be $2^{O(k)}$. Consequently, by composing the k -perfect families from both steps, a k -perfect family of size $2^{O(k)} \log |V|$ that maps from

In the case of $|V| \leq k$, a k -perfect family of hash functions from $\{1, 2, \dots, |V|\}$ to $\{1, 2, \dots, k!\}$ is needed. A sufficient k -perfect family which maps from $\{1, 2, \dots, |V|\}$ to $\{1, 2, \dots, k^k\}$ can be constructed in a way similar to the approach 3 above (the first step). In particular, it is done by using $n k \log k$ random bits that are almost $k \log k$ independent, and the size of the resulting k -perfect family will be $k^{O(k)} \log |V|$. The derandomization of color-coding method can be easily parallelized, yielding efficient NC algorithms.

Applications

Recently, color-coding has attracted much attention in the field of bioinformatics. One example is the detection of signaling pathways in protein-protein interaction (PPI) networks. Another example is to discover and to count the number of motifs in PPI networks. Studying both signaling pathways and motifs allows a deeper understanding of the similarities and differences of many biological functions, processes, and structures among organisms.

Due to the huge amount of gene data that can be collected, searching for pathways or motifs can be highly time consuming. However, by exploiting the color-coding method, the motifs or signaling pathways with $k = O(\log n)$ vertices in a network G with n vertices can be found very efficiently in polynomial time. Thus, this enables us to explore more complex or larger structures in PPI networks. More details can be found in.^{[9][10]}

References

- [1] Alon, N., Yuster, R., and Zwick, U. 1994. Color-coding: a new method for finding simple paths, cycles and other small subgraphs within large graphs. In Proceedings of the Twenty-Sixth Annual ACM Symposium on theory of Computing (Montreal, Quebec, Canada, May 23–25, 1994). STOC '94. ACM, New York, NY, 326–335. DOI= <http://doi.acm.org/10.1145/195058.195179>
- [2] Alon, N., Yuster, R., and Zwick, U. 1995. Color-coding. J. ACM 42, 4 (Jul. 1995), 844–856. DOI= <http://doi.acm.org/10.1145/210332.210337>
- [3] Coppersmith–Winograd Algorithm
- [4] Alon, N. and Naor, M. 1994 Derandomization, Witnesses for Boolean Matrix Multiplication and Construction of Perfect Hash Functions. Technical Report. UMI Order Number: CS94-11., Weizmann Science Press of Israel.
- [5] Naor, M., Schulman, L. J., and Srinivasan, A. 1995. Splitters and near-optimal derandomization. In Proceedings of the 36th Annual Symposium on Foundations of Computer Science (October 23–25, 1995). FOCS. IEEE Computer Society, Washington, DC, 182.
- [6] Schmidt, J. P. and Siegel, A. 1990. The spatial complexity of oblivious k-probe Hash functions. SIAM J. Comput. 19, 5 (Sep. 1990), 775–786. DOI= <http://dx.doi.org/10.1137/0219054>
- [7] Naor, J. and Naor, M. 1990. Small-bias probability spaces: efficient constructions and applications. In Proceedings of the Twenty-Second Annual ACM Symposium on theory of Computing (Baltimore, Maryland, United States, May 13–17, 1990). H. Ortiz, Ed. STOC '90. ACM, New York, NY, 213–223. DOI= <http://doi.acm.org/10.1145/100216.100244>
- [8] Alon, N., Goldreich, O., Hastad, J., and Peralta, R. 1990. Simple construction of almost k-wise independent random variables. In Proceedings of the 31st Annual Symposium on Foundations of Computer Science (October 22–24, 1990). SFCS. IEEE Computer Society, Washington, DC, 544–553 vol.2. DOI= <http://dx.doi.org/10.1109/FSCS.1990.89575>
- [9] Alon, N., Dao, P., Hajirasouliha, I., Hormozdiari, F., and Sahinalp, S. C. 2008. Biomolecular network motif counting and discovery by color coding. Bioinformatics 24, 13 (Jul. 2008), i241–i249. DOI= <http://dx.doi.org/10.1093/bioinformatics/btn163>
- [10] Hüffner, F., Wernicke, S., and Zichner, T. 2008. Algorithm Engineering for Color-Coding with Applications to Signaling Pathway Detection. Algorithmica 52, 2 (Aug. 2008), 114–132. DOI= <http://dx.doi.org/10.1007/s00453-007-9008-7>

Induced subgraph isomorphism problem

In complexity theory and graph theory, **induced subgraph isomorphism** is an NP-complete decision problem that involves finding a given graph as an induced subgraph of a larger graph.

Problem statement

Formally, the problem takes as input two graphs $G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$, where the number of vertices in V_1 can be assumed to be less than or equal to the number of vertices in V_2 . G_1 is isomorphic to an induced subgraph of G_2 if there is an injective function f which maps the vertices of G_1 to vertices of G_2 such that for all pairs of vertices x, y in V_1 , edge (x, y) is in E_1 if and only if the edge $(f(x), f(y))$ is in E_2 . The answer to the decision problem is yes if this function f exists, and no otherwise.

This is different from the subgraph isomorphism problem in that the absence of an edge in G_1 implies that the corresponding edge in G_2 must also be absent. In subgraph isomorphism, these "extra" edges in G_2 may be present.

Computational complexity

The complexity of induced subgraph isomorphism separates outerplanar graphs from their generalization series-parallel graphs: it may be solved in polynomial time for 2-connected outerplanar graphs, but is NP-complete for 2-connected series-parallel graphs.

Special cases

The special case of finding a long path as an induced subgraph of a hypercube has been particularly well-studied, and is called the snake-in-the-box problem. The maximum independent set problem is also an induced subgraph isomorphism problem in which one seeks to find a large independent set as an induced subgraph of a larger graph, and the maximum clique problem is an induced subgraph isomorphism problem in which one seeks to find a large clique graph as an induced subgraph of a larger graph.

Differences with the subgraph isomorphism problem

Although the induced subgraph isomorphism problem seems only slightly different from the subgraph isomorphism problem, the "induced" restriction introduces changes large enough that we can witness differences from a computational complexity point of view.

For example, the subgraph isomorphism problem is NP-complete on connected proper interval graphs and on connected bipartite permutation graphs, but the **induced** subgraph isomorphism problem can be solved in polynomial time on these two classes.

Moreover, the induced subtree isomorphism problem (i.e. the induced subgraph isomorphism problem where G_2 is restricted to be a tree) can be solved in polynomial time on interval graphs, while the subtree isomorphism problem is NP-complete on proper interval graphs.

References

Maximum common subgraph isomorphism problem

In complexity theory, **maximum common subgraph-isomorphism (MCS)** is an optimization problem that is known to be NP-hard. The formal description of the problem is as follows:

Maximum common subgraph-isomorphism(G_1, G_2)

- Input: Two graphs G_1 and G_2 .
- Question: What is the largest subgraph of G_1 isomorphic to a subgraph of G_2 ?

The associated decision problem, i.e., given G_1 , G_2 and an integer k , deciding whether G_1 contains a subgraph of at least k edges isomorphic to a subgraph of G_2 is NP-complete.

One possible solution for this problem is to build a modular product graph, in which the largest clique represents a solution for the MCS problem.

MCS algorithms have a long tradition in cheminformatics and pharmacophore mapping.

References

- Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5. A1.4: GT48, pg.202.

Graph decomposition and graph minors

Graph partition

In mathematics, the **graph partition** problem is defined on data represented in the form of a graph $G = (V, E)$, with V vertices and E edges, such that it is possible to partition G into smaller components with specific properties. For instance, a k -way partition divides the vertex set into k smaller components. A good partition is defined as one in which the number of edges running between separated components is small. Uniform graph partition is a type of graph partitioning problem that consists of dividing a graph into components, such that the components are of about the same size and there are few connections between the components. Important applications of graph partitioning include scientific computing, partitioning various stages of a VLSI design circuit and task scheduling in multi-processor systems. Recently, the graph partition problem has gained importance due to its application for clustering and detection of cliques in social, pathological and biological networks. A survey on the recent trends in computational methods and applications can be found in.

Problem complexity

Typically, graph partition problems fall under the category of NP-hard problems. Solutions to these problems are generally derived using heuristics and approximation algorithms. However, uniform graph partitioning or a balanced graph partition problem can be shown to be NP-complete to approximate within any finite factor. Even for special graph classes such as trees and grids, no reasonable approximation algorithms exist, unless P=NP. Grids are a particularly interesting case since they model the graphs resulting from Finite Element Model (FEM) simulations. When not only the number of edges between the components is approximated, but also the sizes of the components, it can be shown that no reasonable fully polynomial algorithms exist for these graphs.

Problem

Consider a graph $G = (V, E)$, where V denotes the set of n vertices and E the set of edges. For a (k, v) balanced partition problem, the objective is to partition G into k components of at most size $v \cdot (n/k)$, while minimizing the capacity of the edges between separate components. Also, given G and an integer $k > 1$, partition V into k parts (subsets) V_1, V_2, \dots, V_k such that the parts are disjoint and have equal size, and the number of edges with endpoints in different parts is minimized. Such partition problems have been discussed in literature as bicriteria-approximation or resource augmentation approaches. A common extension is to hypergraphs, where an edge can connect more than two vertices. A hyperedge is not cut if all vertices are in one partition, and cut exactly once otherwise, no matter how many vertices are on each side. This usage is common in electronic design automation.

Analysis

For a specific $(k, 1 + \varepsilon)$ balanced partition problem, we seek to find a minimum cost partition of G into k components with each component containing maximum of $(1 + \varepsilon) \cdot (n/k)$ nodes. We compare the cost of this approximation algorithm to the cost of a $(k, 1)$ cut, wherein each of the k components must have exactly the same size of (n/k) nodes each, thus being a more restricted problem. Thus,

$$\max_i |V_i| \leq (1 + \varepsilon) \left\lceil \frac{|V|}{k} \right\rceil.$$

We already know that $(2, 1)$ cut is the minimum bisection problem and it is NP complete. Next we assess a 3-partition problem wherein $n = 3k$, which is also bounded in polynomial time. Now, if we assume that we have an finite approximation algorithm for $(k, 1)$ -balanced partition, then, either the 3-partition instance can be solved using the balanced $(k, 1)$ partition in G or it cannot be solved. If the 3-partition instance can be solved, then $(k, 1)$ -balanced partitioning problem in G can be solved without cutting any edge. Otherwise if the 3-partition instance cannot be solved, the optimum $(k, 1)$ -balanced partitioning in G will cut at least one edge. An approximation algorithm with finite approximation factor has to differentiate between these two cases. Hence, it can solve the 3-partition problem which is a contradiction under the assumption that $P = NP$. Thus, it is evident that $(k, 1)$ -balanced partitioning problem has no polynomial time approximation algorithm with finite approximation factor unless $P = NP$.

The planar separator theorem states that any n -vertex planar graph can be partitioned into roughly equal parts by the removal of $O(\sqrt{n})$ vertices. This is not a partition in the sense described above, because the partition set consists of vertices rather than edges. However, the same result also implies that every planar graph of bounded degree has a balanced cut with $O(\sqrt{n})$ edges.

Graph partition methods

Since graph partitioning is a hard problem, practical solutions are based on heuristics. There are two broad categories of methods, local and global. Well known local methods are the Kernighan–Lin algorithm, and Fiduccia-Mattheyses algorithms, which were the first effective 2-way cuts by local search strategies. Their major drawback is the arbitrary initial partitioning of the vertex set, which can affect the final solution quality. Global approaches rely on properties of the entire graph and do not rely on an arbitrary initial partition. The most common example is spectral partitioning, where a partition is derived from the spectrum of the adjacency matrix.

Multi-level methods

A multi-level graph partitioning algorithm works by applying one or more stages. Each stage reduces the size of the graph by collapsing vertices and edges, partitions the smaller graph, then maps back and refines this partition of the original graph. A wide variety of partitioning and refinement methods can be applied within the overall multi-level scheme. In many cases, this approach can give both fast execution times and very high quality results. One widely used example of such an approach is METIS, a graph partitioner, and hMETIS, the corresponding partitioner for hypergraphs.

Spectral partitioning and spectral bisection

Given a graph with adjacency matrix A , where an entry A_{ij} implies an edge between node i and j , and degree matrix D , which is a diagonal matrix, where each diagonal entry of a row i , d_{ii} , represents the node degree of node i . The Laplacian of the matrix L is defined as $L = D - A$. Now, a ratio-cut partition for graph $G = (V, E)$ is defined as a partition of V into disjoint U , and W , such that cost of $\text{cut}(U, W)/(|U||W|)$ is minimized.

In such a scenario, the second smallest eigenvalue (λ) of L , yields a lower bound on the optimal cost (c) of ratio-cut partition with $c \geq \lambda/n$. The eigenvector (V) corresponding to λ , called the Fiedler vector, bisects the graph into only two communities based on the sign of the corresponding vector entry. Division into a larger number of communities is usually achieved by repeated bisection, but this does not always give satisfactory results. The examples in Figures 1,2 illustrate the spectral bisection approach.

Minimum cut partitioning however fails when the number of communities to be partitioned, or the partition sizes are unknown. For instance, optimizing the cut size for free group sizes puts all vertices in the same community. Additionally, cut size may be the wrong thing to minimize since a good division is not just one with small number of edges between communities. This motivated the use of Modularity (Q) as a metric to optimize a balanced graph partition. The example in Figure 3 illustrates 2 instances of the same graph such that in (a) modularity (Q) is the partitioning metric and in (b), ratio-cut is the partitioning metric. However, it is well-known that Q suffers a resolution limit, producing unreliable results when dealing with small communities. In this context, Surprise has been proposed as an alternative approach for evaluating the quality of a partition.

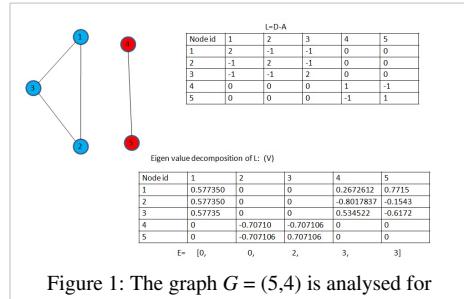


Figure 1: The graph $G = (5,4)$ is analysed for spectral bisection. The linear combination of the smallest two eigenvectors leads to $[1 1 1 1 1]$ ' having an eigen value = 0.

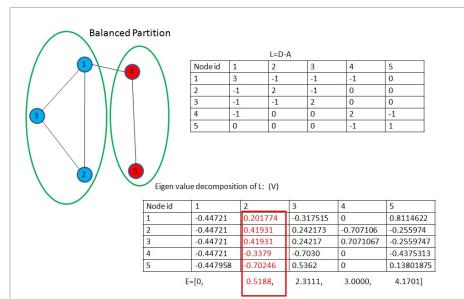


Figure 2: The graph $G = (5,5)$ illustrates that the Fiedler vector in red bisects the graph into two communities, one with vertices $\{1,2,3\}$ with positive entries in the vector space, and the other community has vertices $\{4,5\}$ with negative vector space entries.

Other graph partition methods

Spin models have been used for clustering of multivariate data wherein similarities are translated into coupling strengths. The properties of ground state spin configuration can be directly interpreted as communities. Thus, a graph is partitioned to minimize the Hamiltonian of the partitioned graph. The Hamiltonian (H) is derived by assigning the following partition rewards and penalties.

- Reward internal edges between nodes of same group (same spin)
- Penalize missing edges in same group
- Penalize existing edges between different groups
- Reward non-links between different groups.

Additionally, Kernel PCA based Spectral clustering takes a form of least squares Support Vector Machine framework, and hence it becomes possible to project the data entries to a kernel induced feature space that has maximal variance, thus implying a high separation between the projected communities

Some methods express graph partitioning as a multi-criteria optimization problem which can be solved using local methods expressed in a game theoretic framework where each node makes a decision on the partition it chooses.^[1]

Software tools

One of the first publicly available software packages called Chaco is due to Hendrickson and Leland. As most of the publicly available software packages, Chaco implements the multilevel approach outlined above and basic local search algorithms. Moreover, they implement spectral partitioning techniques.

METIS is a graph partitioning family by Karypis and Kumar. kMetis is focused on partitioning speed and hMetis, which is a hypergraph partitioner, aims at partition quality. ParMetis is a parallel implementation of the Metis graph partitioning algorithm.

PaToH is also a widely used hypergraph partitioner that produces high quality partitions.

Scotch is graph partitioning framework by Pellegrini. It uses recursive multilevel bisection and includes sequential as well as parallel partitioning techniques.

Jostle is a sequential and parallel graph partitioning solver developed by Chris Walshaw. The commercialized version of this partitioner is known as NetWorks.

If a model of the communication network is available, then Jostle and Scotch are able to take this model into account for the partitioning process.

Party implements the Bubble/shape-optimized framework and the Helpful Sets algorithm.

The software packages DibaP and its MPI-parallel variant PDibaP by Meyerhenke implement the Bubble framework using diffusion; DibaP also uses AMG-based techniques for coarsening and solving linear systems arising in the diffusive approach.

Sanders and Schulz released a graph partitioning package KaHIP (Karlsruhe High Quality Partitioning) focusing on solution quality. It implements for example flow-based methods, more-localized local searches and several parallel and sequential meta-heuristics.

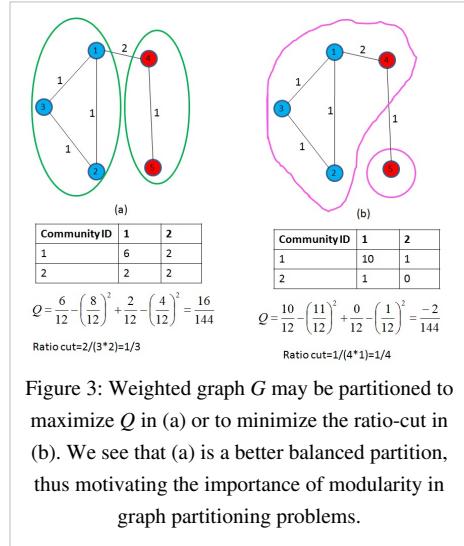


Figure 3: Weighted graph G may be partitioned to maximize Q in (a) or to minimize the ratio-cut in (b). We see that (a) is a better balanced partition, thus motivating the importance of modularity in graph partitioning problems.

To address the load balancing problem in parallel applications, distributed versions of the sequential partitioners Metis, Jostle and Scotch have been developed. Wikipedia:Citation needed

The tools Parkway by Trifunovic and Knottenbelt as well as Zoltan by Devine et al. focus on hypergraph partitioning.

List of free open-source frameworks:

Name	License	Brief info
Chaco	GPL	software package implementing spectral techniques and the multilevel approach
DiBaP	*	graph partitioning based on multilevel techniques, algebraic multigrid as well as graph based diffusion
Jostle	*	multilevel partitioning techniques and diffusive load-balancing, sequential and parallel
KaHIP	GPL	several parallel and sequential meta-heuristics, guarantees the balance constraint
kMetis	Apache 2.0	graph partitioning package based on multilevel techniques and k-way local search
Mondriaan	LGPL	matrix partitioner to partition rectangular sparse matrices
PaToH	BSD	multilevel hypergraph partitioning
Parkway	*	parallel multilevel hypergraph partitioning
Scotch	CeCILL-C	implements multilevel recursive bisection as well as diffusion techniques, sequential and parallel
Zoltan	BSD	hypergraph partitioning

References

- [1] Kurve, Griffin, Kesidis (2011) "A graph partitioning game for distributed simulation of networks" Proceedings of the 2011 International Workshop on Modeling, Analysis, and Control of Complex Networks: 9 -16

External links

- Chamberlain, Bradford L. (1998). "Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations" (<http://masters.donntu.edu.ua/2006/fvti/krasnokutskaya/library/generals.pdf>)

Bibliography

- Charles-Edmond Bichot, Patrick Siarry (2011). *Graph Partitioning: Optimisation and Applications* (http://cebichot.netne.net/graph_partitioning_book/). ISTE – Wiley. p. 384. ISBN 978-1848212336.
- Feldmann, Andreas Emil (2012). *Balanced Partitioning of Grids and Related Graphs: A Theoretical Study of Data Distribution in Parallel Finite Element Model Simulations* (http://www.pw.ethz.ch/people/research_group/andemil/personal/thesis.pdf). Goettingen, Germany: Cuvillier Verlag. p. 218. ISBN 978-3954041251. An exhaustive analysis of the problem from a theoretical point of view.
- BW Kernighan, S Lin (1970). "An efficient heuristic procedure for partitioning graphs" (<http://www.ece.wisc.edu/~adavoodi/teaching/756-old/papers/kl.pdf>). *Bell System Technical Journal*. One of the early fundamental works in the field. However, performance is $O(n^2)$, so it is no longer commonly used.
- CM Fiduccia, RM Mattheyses (1982). "A Linear-Time Heuristic for Improving Network Partitions" (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1585498). *Design Automation Conference*. A later variant that is linear time, very commonly used, both by itself and as part of multilevel partitioning, see below.
- G Karypis, V Kumar (1999). "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs" (<http://glaros.dtc.umn.edu/gkhome/node/107>). *Siam Journal on Scientific Computing*. Multi-level partitioning is the current state of the art. This paper also has good explanations of many other methods, and comparisons of the various methods on a wide variety of problems.

- Karypis, G., Aggarwal, R., Kumar, V., and Shekhar, S. (March 1999). "Multilevel hypergraph partitioning applications in VLSI domain" (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=748202). *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7 (1): pp. 69–79. doi: 10.1109/92.748202 (<http://dx.doi.org/10.1109/92.748202>). Graph partitioning (and in particular, hypergraph partitioning) has many applications to IC design.
- S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi (13 May 1983). "Optimization by Simulated Annealing" (<http://www.sciencemag.org/cgi/content/abstract/220/4598/671>). *Science* 220 (4598): 671–680. doi: 10.1126/science.220.4598.671 (<http://dx.doi.org/10.1126/science.220.4598.671>). PMID 17813860 (<http://www.ncbi.nlm.nih.gov/pubmed/17813860>). Simulated annealing can be used as well.
- Hagen, L. and Kahng, A.B. (September 1992). "New spectral methods for ratio cut partitioning and clustering" (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=159993). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 11, (9): 1074–1085. doi: 10.1109/43.159993 (<http://dx.doi.org/10.1109/43.159993>).. There is a whole class of *spectral partitioning* methods, which use the Eigenvectors of the Laplacian of the connectivity graph. You can see a demo of this (<http://www.stanford.edu/~dgleich/demos/matlab/spectral/spectral.html>), using Matlab.

Kernighan–Lin algorithm

This article is about the heuristic algorithm for the graph partitioning problem. For a heuristic for the traveling salesperson problem, see Lin–Kernighan heuristic.

Kernighan–Lin is a $O(n^2 \log(n))$ heuristic algorithm for solving the graph partitioning problem. The algorithm has important applications in the layout of digital circuits and components in VLSI.

Description

Let $G(V, E)$ be a graph, and let V be the set of nodes and E the set of edges. The algorithm attempts to find a partition of V into two disjoint subsets A and B of equal size, such that the sum T of the weights of the edges between nodes in A and B is minimized. Let I_a be the *internal cost* of a , that is, the sum of the costs of edges between a and other nodes in A , and let E_a be the *external cost* of a , that is, the sum of the costs of edges between a and nodes in B . Furthermore, let

$$D_a = E_a - I_a$$

be the difference between the external and internal costs of a . If a and b are interchanged, then the reduction in cost is

$$T_{old} - T_{new} = D_a + D_b - 2c_{a,b}$$

where $c_{a,b}$ is the cost of the possible edge between a and b .

The algorithm attempts to find an optimal series of interchange operations between elements of A and B which maximizes $T_{old} - T_{new}$ and then executes the operations, producing a partition of the graph to A and B .

Pseudocode

See

```

1  function Kernighan–Lin( $G(V, E)$ ):
2      determine a balanced initial partition of the nodes into sets  $A$  and  $B$ 
3       $A1 := A$ ;  $B1 := B$ 
4      do
5          compute  $D$  values for all  $a$  in  $A1$  and  $b$  in  $B1$ 

```

```

6      for (n := 1 to /2)
7          find a[i] from A1 and b[j] from B1, such that g[n] = D[a[i + D[b[j]] - 2*c[a[i]][b[j]] is maximal
8          move a[i] to B1 and b[j] to A1
9          remove a[i] and b[j] from further consideration in this pass
10         update D values for the elements of A1 = A1 \ a[i] and B1 = B1 \ b[j]
11     end for
12     find k which maximizes g_max, the sum of g[1],...,g[k]
13     if (g_max > 0) then
14         Exchange a[1],a[2],...,a[k] with b[1],b[2],...,b[k]
15     until (g_max <= 0)
16 return G(V,E)

```

References

Tree decomposition

This article is about tree structure of graphs. For decomposition of graphs into trees, see [Graph_theory#Decomposition_problems](#).

In graph theory, a **tree decomposition** is a mapping of a graph into a tree that can be used to define the treewidth of the graph and speed up solving certain computational problems on the graph.

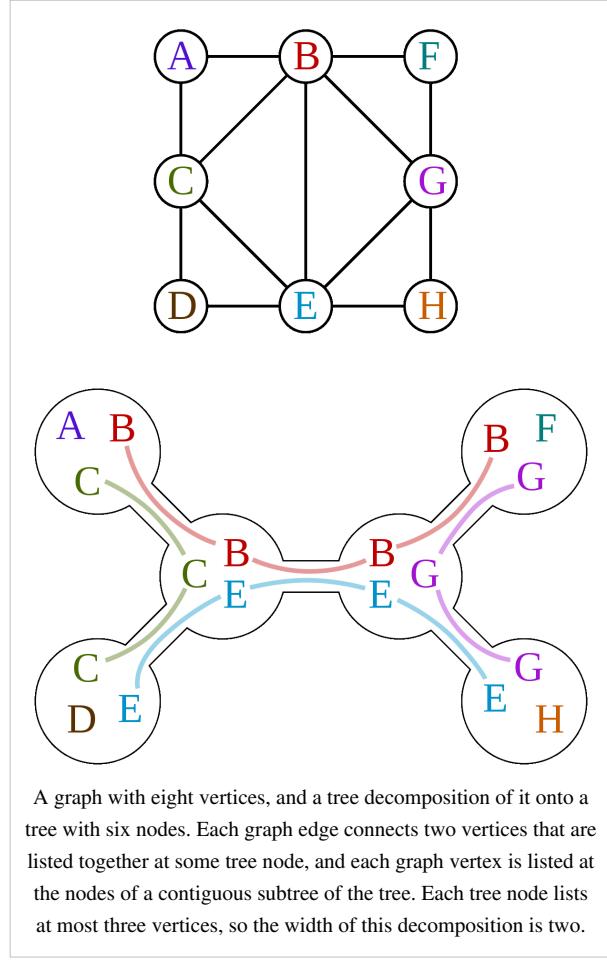
In machine learning, tree decompositions are also called **junction trees**, **clique trees**, or **join trees**; they play an important role in problems like probabilistic inference, constraint satisfaction, query optimization, and matrix decomposition.

The concept of tree decompositions was originally introduced by Rudolf Halin (1976). Later it was rediscovered by Neil Robertson and Paul Seymour (1984) and has since been studied by many other authors.^[1]

Definition

Intuitively, a tree decomposition represents the vertices of a given graph G as subtrees of a tree, in such a way that vertices in the given graph are adjacent only when the corresponding subtrees intersect. Thus, G forms a subgraph of the intersection graph of the subtrees. The full intersection graph is a chordal graph.

Each subtree associates a graph vertex with a set of tree nodes. To define this formally, we represent each tree node as the set of vertices associated with it. Thus, given a graph $G = (V, E)$, a tree decomposition is a pair (X, T) , where $X = \{X_1, \dots, X_n\}$ is a family of subsets of V , and T is a tree whose nodes are the subsets X_i , satisfying the following properties:^[2]



1. The union of all sets X_i equals V . That is, each graph vertex is associated with at least one tree node.
2. For every edge (v, w) in the graph, there is a subset X_i that contains both v and w . That is, vertices are adjacent in the graph only when the corresponding subtrees have a node in common.
3. If X_i and X_j both contain a vertex v , then all nodes X_k of the tree in the (unique) path between X_i and X_j contain v as well. That is, the nodes associated with vertex v form a connected subset of T . This is also known as coherence, or the *running intersection property*. It can be stated equivalently that if X_i , X_j and X_k are nodes, and X_k is on the path from X_i to X_j , then $X_i \cap X_j \subseteq X_k$.

The tree decomposition of a graph is far from unique; for example, a trivial tree decomposition contains all vertices of the graph in its single root node.

A tree decomposition in which the underlying tree is a path graph is called a path decomposition, and the width parameter derived from these special types of tree decompositions is known as pathwidth.

Treewidth

Main article: Treewidth

The *width* of a tree decomposition is the size of its largest set X_i minus one. The treewidth $\text{tw}(G)$ of a graph G is the minimum width among all possible tree decompositions of G . In this definition, the size of the largest set is diminished by one in order to make the treewidth of a tree equal to one. Treewidth may also be defined from other structures than tree decompositions, including chordal graphs, brambles, and havens.

It is NP-complete to determine whether a given graph G has treewidth at most a given variable k . However, when k is any fixed constant, the graphs with treewidth k can be recognized, and a width k tree decomposition constructed for them, in linear time. The time dependence of this algorithm on k is exponential.

Dynamic programming

At the beginning of the 1970s, it was observed that a large class of combinatorial optimization problems defined on graphs could be efficiently solved by non serial dynamic programming as long as the graph had a bounded dimension,^[3] a parameter related to treewidth. Later, several authors independently observed at the end of the 1980s^[4] that many algorithmic problems that are NP-complete for arbitrary graphs may be solved efficiently by dynamic programming for graphs of bounded treewidth, using the tree-decompositions of these graphs.

As an example, consider the problem of finding the maximum independent set in a graph of treewidth k . To solve this problem, first choose one of the nodes of the tree decomposition to be the root, arbitrarily. For a node X_i of the tree decomposition, let D_i be the union of the sets X_j descending from X_i . For an independent set $S \subset X_i$, let $A(S, i)$ denote the size of the largest independent subset I of D_i such that $I \cap X_i = S$. Similarly, for an adjacent pair of nodes X_i and X_j , with X_i farther from the root of the tree than X_j , and an independent set $S \subset X_i \cap X_j$, let $B(S, i, j)$ denote the size of the largest independent subset I of D_i such that $I \cap X_i \cap X_j = S$. We may calculate these A and B values by a bottom-up traversal of the tree:

$$A(S, i) = |S| + \sum_j (B(S \cap X_j, j, i) - |S \cap X_j|)$$

$$B(S, i, j) = \max_{\substack{S' \subset X_i \\ S = S' \cap X_j}} A(S', i)$$

where the sum in the calculation of $A(S, i)$ is over the children of node X_i .

At each node or edge, there are at most 2^k sets S for which we need to calculate these values, so if k is a constant then the whole calculation takes constant time per edge or node. The size of the maximum independent set is the largest value stored at the root node, and the maximum independent set itself can be found (as is standard in dynamic programming algorithms) by backtracking through these stored values starting from this largest value. Thus, in graphs of bounded treewidth, the maximum independent set problem may be solved in linear time. Similar

algorithms apply to many other graph problems.

This dynamic programming approach is used in machine learning via the junction tree algorithm for belief propagation in graphs of bounded treewidth. It also plays a key role in algorithms for computing the treewidth and constructing tree decompositions: typically, such algorithms have a first step that approximates the treewidth, constructing a tree decomposition with this approximate width, and then a second step that performs dynamic programming in the approximate tree decomposition to compute the exact value of the treewidth.

Notes

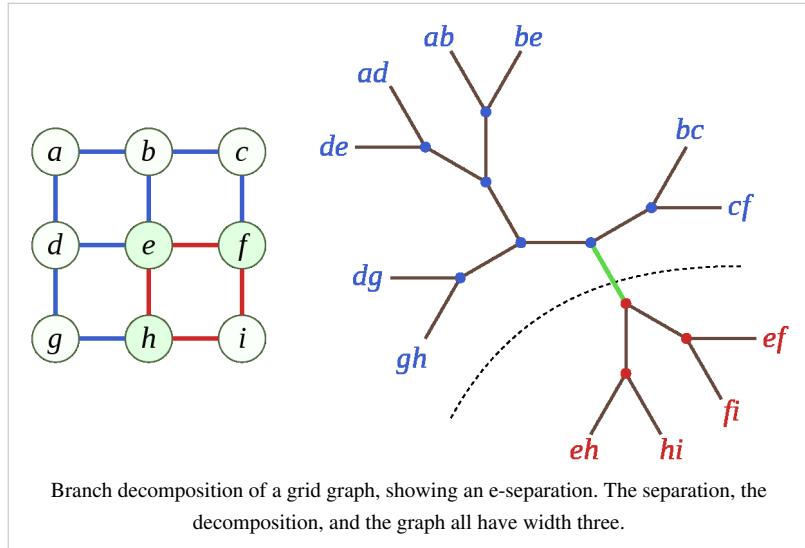
- [1] pp.354–355
- [2] section 12.3
- [3] Bertelé & Brioschi (1972).
- [4] ;;.

References

- Arnborg, S.; Corneil, D.; Proskurowski, A. (1987), "Complexity of finding embeddings in a k -tree", *SIAM Journal on Matrix Analysis and Applications* **8** (2): 277–284, doi: 10.1137/0608024 (<http://dx.doi.org/10.1137/0608024>).
- Arnborg, S.; Proskurowski, A. (1989), "Linear time algorithms for NP-hard problems restricted to partial k -trees", *Discrete Applied Mathematics* **23** (1): 11–24, doi: 10.1016/0166-218X(89)90031-0 ([http://dx.doi.org/10.1016/0166-218X\(89\)90031-0](http://dx.doi.org/10.1016/0166-218X(89)90031-0)).
- Bern, M. W.; Lawler, E. L.; Wong, A. L. (1987), "Linear-time computation of optimal subgraphs of decomposable graphs", *Journal of Algorithms* **8** (2): 216–235, doi: 10.1016/0196-6774(87)90039-3 ([http://dx.doi.org/10.1016/0196-6774\(87\)90039-3](http://dx.doi.org/10.1016/0196-6774(87)90039-3)).
- Bertelé, Umberto; Brioschi, Francesco (1972), *Nonserial Dynamic Programming*, Academic Press, ISBN 0-12-093450-7.
- Bodlaender, Hans L. (1988), "Dynamic programming on graphs with bounded treewidth", *Proc. 15th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **317**, Springer-Verlag, pp. 105–118, doi: 10.1007/3-540-19488-6_110 (http://dx.doi.org/10.1007/3-540-19488-6_110).
- Bodlaender, Hans L. (1996), "A linear time algorithm for finding tree-decompositions of small treewidth", *SIAM Journal on Computing* **25** (6): 1305–1317, doi: 10.1137/S0097539793251219 (<http://dx.doi.org/10.1137/S0097539793251219>).
- Diestel, Reinhard (2005), *Graph Theory* (<http://www.math.uni-hamburg.de/home/diestel/books/graph-theory/>) (3rd ed.), Springer, ISBN 3-540-26182-6.
- Halin, Rudolf (1976), "S-functions for graphs", *Journal of Geometry* **8**: 171–186, doi: 10.1007/BF01917434 (<http://dx.doi.org/10.1007/BF01917434>).
- Robertson, Neil; Seymour, Paul D. (1984), "Graph minors III: Planar tree-width", *Journal of Combinatorial Theory, Series B* **36** (1): 49–64, doi: 10.1016/0095-8956(84)90013-3 ([http://dx.doi.org/10.1016/0095-8956\(84\)90013-3](http://dx.doi.org/10.1016/0095-8956(84)90013-3)).

Branch-decomposition

In graph theory, a **branch-decomposition** of an undirected graph G is a hierarchical clustering of the edges of G , represented by an unrooted binary tree T with the edges of G as its leaves. Removing any edge from T partitions the edges of G into two subgraphs, and the width of the decomposition is the maximum number of shared vertices of any pair of subgraphs formed in this way. The **branchwidth** of G is the minimum width of any branch-decomposition of G ; branchwidth is closely related to tree-width and many graph optimization problems may be solved efficiently for graphs of small branchwidth. Branch-decompositions and branchwidth may also be generalized from graphs to matroids.



Definitions

An unrooted binary tree is a connected undirected graph with no cycles in which each non-leaf node has exactly three neighbors. A branch-decomposition may be represented by an unrooted binary tree T , together with a bijection between the leaves of T and the edges of the given graph $G = (V, E)$. If e is any edge of the tree T , then removing e from T partitions it into two subtrees T_1 and T_2 . This partition of T into subtrees induces a partition of the edges associated with the leaves of T into two subgraphs G_1 and G_2 of G . This partition of G into two subgraphs is called an **e-separation**.

The width of an e-separation is the number of vertices of G that are incident both to an edge of E_1 and to an edge of E_2 ; that is, it is the number of vertices that are shared by the two subgraphs G_1 and G_2 . The width of the branch-decomposition is the maximum width of any of its e-separations. The branchwidth of G is the minimum width of a branch-decomposition of G .

Relation to treewidth

Branch-decompositions of graphs are closely related to tree decompositions, and branch-width is closely related to tree-width: the two quantities are always within a constant factor of each other. In particular, in the paper in which they introduced branch-width, Neil Robertson and Paul Seymour^[1] showed that for a graph G with tree-width k and branchwidth $b > 1$,

$$b - 1 \leq k \leq \left\lfloor \frac{3}{2}b \right\rfloor - 1.$$

Carving width

Carving width is a concept defined similarly to branch width, except with edges replaced by vertices and vice versa. A carving decomposition is an unrooted binary tree with each leaf representing a vertex in the original graph, and the width of a cut is the number (or total weight in a weighted graph) of edges that are incident to a vertex in both subtrees.

Branch width algorithms typically work by reducing to an equivalent carving width problem. In particular, the carving width of the medial graph of a graph is exactly twice the branch width of the original graph.

Algorithms and complexity

It is NP-complete to determine whether a graph G has a branch-decomposition of width at most k , when G and k are both considered as inputs to the problem. However, the graphs with branchwidth at most k form a minor-closed family of graphs,^[2] from which it follows that computing the branchwidth is fixed-parameter tractable: there is an algorithm for computing optimal branch-decompositions whose running time, on graphs of branchwidth k for any fixed constant k , is linear in the size of the input graph.^[3]

For planar graphs, the branchwidth can be computed exactly in polynomial time. This in contrast to treewidth for which the complexity on planar graphs is a well known open problem. The original algorithm for planar branchwidth, by Paul Seymour and Robin Thomas, took time $O(n^2)$ on graphs with n vertices, and their algorithm for constructing a branch decomposition of this width took time $O(n^4)$. This was later sped up to $O(n^3)$.^[4]

As with treewidth, branchwidth can be used as the basis of dynamic programming algorithms for many NP-hard optimization problems, using an amount of time that is exponential in the width of the input graph or matroid.^[5] For instance, Cook & Seymour (2003) apply branchwidth-based dynamic programming to a problem of merging multiple partial solutions to the travelling salesman problem into a single global solution, by forming a sparse graph from the union of the partial solutions, using a spectral clustering heuristic to find a good branch-decomposition of this graph, and applying dynamic programming to the decomposition. Fomin & Thilikos (2006) argue that branchwidth works better than treewidth in the development of fixed-parameter-tractable algorithms on planar graphs, for multiple reasons: branchwidth may be more tightly bounded by a function of the parameter of interest than the bounds on treewidth, it can be computed exactly in polynomial time rather than merely approximated, and the algorithm for computing it has no large hidden constants.

Generalization to matroids

It is also possible to define a notion of branch-decomposition for matroids that generalizes branch-decompositions of graphs.^[6] A branch-decomposition of a matroid is a hierarchical clustering of the matroid elements, represented as an unrooted binary tree with the elements of the matroid at its leaves. An e-separation may be defined in the same way as for graphs, and results in a partition of the set M of matroid elements into two subsets A and B . If ρ denotes the rank function of the matroid, then the width of an e-separation is defined as $\rho(A) + \rho(B) - \rho(M) + 1$, and the width of the decomposition and the branchwidth of the matroid are defined analogously. The branchwidth of a graph and the branchwidth of the corresponding graphic matroid may differ: for instance, the three-edge path graph and the three-edge star have different branchwidths, 2 and 1 respectively, but they both induce the same graphic matroid with branchwidth 1. However, for graphs that are not trees, the branchwidth of the graph is equal to the branchwidth of its associated graphic matroid.^[7] The branchwidth of a matroid is equal to the branchwidth of its dual matroid, and in particular this implies that the branchwidth of any planar graph that is not a tree is equal to that of its dual.

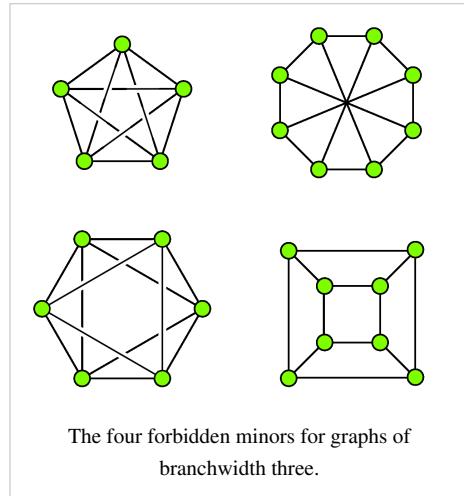
Branchwidth is an important component of attempts to extend the theory of graph minors to matroid minors: although treewidth can also be generalized to matroids, and plays a bigger role than branchwidth in the theory of graph minors, branchwidth has more convenient properties in the matroid setting. Robertson and Seymour conjectured that the matroids representable over any particular finite field are well-quasi-ordered, analogously to the

Robertson–Seymour theorem for graphs, but so far this has been proven only for the matroids of bounded branchwidth.^[8] Additionally, if a minor-closed family of matroids representable over a finite field does not include the graphic matroids of all planar graphs, then there is a constant bound on the branchwidth of the matroids in the family, generalizing similar results for minor-closed graph families.^[9]

For any fixed constant k , the matroids with branchwidth at most k can be recognized in polynomial time by an algorithm that has access to the matroid via an independence oracle.

Forbidden minors

By the Robertson–Seymour theorem, the graphs of branchwidth k can be characterized by a finite set of forbidden minors. The graphs of branchwidth 0 are the matchings; the minimal forbidden minors are a two-edge path graph and a triangle graph (or the two-edge cycle, if multigraphs rather than simple graphs are considered). The graphs of branchwidth 1 are the graphs in which each connected component is a star; the minimal forbidden minors for branchwidth 1 are the triangle graph (or the two-edge cycle, if multigraphs rather than simple graphs are considered) and the three-edge path graph. The graphs of branchwidth 2 are the graphs in which each biconnected component is a series-parallel graph; the only minimal forbidden minor is the complete graph K_4 on four vertices.^[1] A graph has branchwidth three if and only if it has treewidth three and does not have the cube graph as a minor; therefore, the four minimal forbidden minors are three of the four forbidden minors for treewidth three (the graph of the octahedron, the complete graph K_5 , and the Wagner graph) together with the cube graph.^[10]



Forbidden minors have also been studied for matroid branchwidth, despite the lack of a full analogue to the Robertson–Seymour theorem in this case. A matroid has branchwidth one if and only if every element is either a loop or a coloop, so the unique minimal forbidden minor is the uniform matroid $U(2,3)$, the graphic matroid of the triangle graph. A matroid has branchwidth two if and only if it is the graphic matroid of a graph of branchwidth two, so its minimal forbidden minors are the graphic matroid of K_4 and the non-graphic matroid $U(2,4)$. The matroids of branchwidth three are not well-quasi-ordered without the additional assumption of representability over a finite field, but nevertheless the matroids with any finite bound on their branchwidth have finitely many minimal forbidden minors, all of which have a number of elements that is at most exponential in the branchwidth.^[11]

Notes



Wikimedia Commons has media related to [Tree decomposition](#).

[1] , Theorem 5.1, p. 168.

[2] , Theorem 4.1, p. 164.

[3] . describe an algorithm with improved dependence on k , $(2\sqrt{3})^k$, at the expense of an increase in the dependence on the number of vertices from linear to quadratic.

[4] Gu & Tamaki (2008).

[5] ; .

[6] . Section 12, "Tangles and Matroids", pp. 188–190.

[7] ; .

[8] ; .

[9] ; .

[10] . The fourth forbidden minor for treewidth three, the pentagonal prism, has the cube graph as a minor, so it is not minimal for branchwidth three.

[11] ;.

References

- Bodlaender, Hans L.; Thilikos, Dimitrios M. (1997), "Constructive linear time algorithms for branchwidth", *Proc. 24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, Lecture Notes in Computer Science **1256**, Springer-Verlag, pp. 627–637, doi: 10.1007/3-540-63165-8_217 (http://dx.doi.org/10.1007/3-540-63165-8_217).
- Bodlaender, Hans L.; Thilikos, Dimitrios M. (1999), "Graphs with branchwidth at most three", *Journal of Algorithms* **32** (2): 167–194, doi: 10.1006/jagm.1999.1011 (<http://dx.doi.org/10.1006/jagm.1999.1011>).
- Cook, William; Seymour, Paul D. (2003), "Tour merging via branch-decomposition" (<http://www.cs.utk.edu/~langston/projects/papers/tmerge.pdf>), *INFORMS Journal on Computing* **15** (3): 233–248, doi: 10.1287/ijoc.15.3.233.16078 (<http://dx.doi.org/10.1287/ijoc.15.3.233.16078>).
- Fomin, Fedor V.; Thilikos, Dimitrios M. (2006), "Dominating sets in planar graphs: branch-width and exponential speed-up", *SIAM Journal on Computing* **36** (2): 281, doi: 10.1137/S0097539702419649 (<http://dx.doi.org/10.1137/S0097539702419649>).
- Fomin, Fedor V.; Mazoit, Frédéric; Todinca, Ioan (2009), "Computing branchwidth via efficient triangulations and blocks" (<http://hal.archives-ouvertes.fr/hal-00390623/>), *Discrete Applied Mathematics* **157** (12): 2726–2736, doi: 10.1016/j.dam.2008.08.009 (<http://dx.doi.org/10.1016/j.dam.2008.08.009>).
- Geelen, Jim; Gerards, Bert; Robertson, Neil; Whittle, Geoff (2003), "On the excluded minors for the matroids of branch-width k ", *Journal of Combinatorial Theory, Series B* **88** (2): 261–265, doi: 10.1016/S0095-8956(02)00046-1 ([http://dx.doi.org/10.1016/S0095-8956\(02\)00046-1](http://dx.doi.org/10.1016/S0095-8956(02)00046-1)).
- Geelen, Jim; Gerards, Bert; Whittle, Geoff (2002), "Branch-width and well-quasi-ordering in matroids and graphs", *Journal of Combinatorial Theory, Series B* **84** (2): 270–290, doi: 10.1006/jctb.2001.2082 (<http://dx.doi.org/10.1006/jctb.2001.2082>).
- Geelen, Jim; Gerards, Bert; Whittle, Geoff (2006), "Towards a structure theory for matrices and matroids" (http://www.icm2006.org/proceedings/Vol_III/contents/ICM_Vol_3_41.pdf), *Proc. International Congress of Mathematicians III*, pp. 827–842.
- Geelen, Jim; Gerards, Bert; Whittle, Geoff (2007), "Excluding a planar graph from $\text{GF}(q)$ -representable matroids" (<http://www.math.uwaterloo.ca/~jfgeelen/publications/grid.pdf>), *Journal of Combinatorial Theory, Series B* **97** (6): 971–998, doi: 10.1016/j.jctb.2007.02.005 (<http://dx.doi.org/10.1016/j.jctb.2007.02.005>).
- Gu, Qian-Ping; Tamaki, Hisao (July 2008), "Optimal branch-decomposition of planar graphs in $O(n^3)$ time", *ACM Transactions on Algorithms* **4** (3): 30:1–30:13, doi: 10.1145/1367064.1367070 (<http://dx.doi.org/10.1145/1367064.1367070>).
- Hall, Rhiannon; Oxley, James; Semple, Charles; Whittle, Geoff (2002), "On matroids of branch-width three", *Journal of Combinatorial Theory, Series B* **86** (1): 148–171, doi: 10.1006/jctb.2002.2120 (<http://dx.doi.org/10.1006/jctb.2002.2120>).
- Hicks, Illya V. (2000), *Branch Decompositions and their Applications* (<http://www.caam.rice.edu/caam/trs/2000/TR00-17.ps>), Ph.D. thesis, Rice University.
- Hicks, Illya V.; McMurray, Nolan B., Jr. (2007), "The branchwidth of graphs and their cycle matroids", *Journal of Combinatorial Theory, Series B* **97** (5): 681–692, doi: 10.1016/j.jctb.2006.12.007 (<http://dx.doi.org/10.1016/j.jctb.2006.12.007>).
- Hliněný, Petr (2003), "On matroid properties definable in the MSO logic", *Proc. 28th International Symposium on Mathematical Foundations of Computer Science (MFCS '03)*, Lecture Notes in Computer Science **2747**, Springer-Verlag, pp. 470–479, doi: 10.1007/978-3-540-45138-9_41 (http://dx.doi.org/10.1007/978-3-540-45138-9_41).
- Hliněný, Petr; Whittle, Geoff (2006), "Matroid tree-width" (<http://www.fi.muni.cz/~hlineny/Research/papers/matr-tw-final.pdf>), *European Journal of Combinatorics* **27** (7): 1117–1128, doi:

10.1016/j.ejc.2006.06.005 (<http://dx.doi.org/10.1016/j.ejc.2006.06.005>).

- Addendum and corrigendum: Hliněný, Petr; Whittle, Geoff (2009), "Addendum to matroid tree-width", *European Journal of Combinatorics* **30** (4): 1036–1044, doi: 10.1016/j.ejc.2008.09.028 (<http://dx.doi.org/10.1016/j.ejc.2008.09.028>).
- Mazoit, Frédéric; Thomassé, Stéphan (2007), "Branchwidth of graphic matroids" (<http://hal.archives-ouvertes.fr/docs/00/04/09/28/PDF/Branchwidth.pdf>), in Hilton, Anthony; Talbot, John, *Surveys in Combinatorics 2007*, London Mathematical Society Lecture Note Series **346**, Cambridge University Press, p. 275.
- Oum, Sang-il; Seymour, Paul (2007), "Testing branch-width", *Journal of Combinatorial Theory, Series B* **97** (3): 385–393, doi: 10.1016/j.jctb.2006.06.006 (<http://dx.doi.org/10.1016/j.jctb.2006.06.006>), MR 2305892 (<http://www.ams.org/mathscinet-getitem?mr=2305892>).
- Robertson, Neil; Seymour, Paul D. (1991), "Graph minors. X. Obstructions to tree-decomposition", *Journal of Combinatorial Theory* **52** (2): 153–190, doi: 10.1016/0095-8956(91)90061-N ([http://dx.doi.org/10.1016/0095-8956\(91\)90061-N](http://dx.doi.org/10.1016/0095-8956(91)90061-N)).
- Seymour, Paul D.; Thomas, Robin (1994), "Call routing and the ratcatcher", *Combinatorica* **14** (2): 217–241, doi: 10.1007/BF01215352 (<http://dx.doi.org/10.1007/BF01215352>).

Path decomposition

In graph theory, a **path decomposition** of a graph G is, informally, a representation of G as a "thickened" path graph, and the **pathwidth** of G is a number that measures how much the path was thickened to form G . More formally, a path-decomposition is a sequence of subsets of vertices of G such that the endpoints of each edge appear in one of the subsets and such that each vertex appears in a contiguous subsequence of the subsets, and the pathwidth is one less than the size of the largest set in such a decomposition. Pathwidth is also known as **interval thickness** (one less than the maximum clique size in an interval supergraph of G), **vertex separation number**, or **node searching number**.

Pathwidth and path-decompositions are closely analogous to treewidth and tree decompositions. They play a key role in the theory of graph minors: the families of graphs that are closed under graph minors and do not include all forests may be characterized as having bounded pathwidth, and the "vortices" appearing in the general structure theory for minor-closed graph families have bounded pathwidth. Pathwidth, and graphs of bounded pathwidth, also have applications in VLSI design, graph drawing, and computational linguistics.

It is NP-hard to find the pathwidth of arbitrary graphs, or even to approximate it accurately. However, the problem is fixed-parameter tractable: testing whether a graph has pathwidth k can be solved in an amount of time that depends linearly on the size of the graph but superexponentially on k . Additionally, for several special classes of graphs, such as trees, the pathwidth may be computed in polynomial time without dependence on k . Many problems in graph algorithms may be solved efficiently on graphs of bounded pathwidth, by using dynamic programming on a path-decomposition of the graph. Path decomposition may also be used to measure the space complexity of dynamic programming algorithms on graphs of bounded treewidth.

Definition

In the first of their famous series of papers on graph minors, Neil Robertson and Paul Seymour (1983) define a path-decomposition of a graph G to be a sequence of subsets X_i of vertices of G , with two properties:

1. For each edge of G , there exists an i such that both endpoints of the edge belong to subset X_i , and
2. For every three indices $i \leq j \leq k$, $X_i \cap X_k \subseteq X_j$.

The second of these two properties is equivalent to requiring that the subsets containing any particular vertex form a contiguous subsequence of the whole sequence. In the language of the later papers in Robertson and Seymour's graph minor series, a path-decomposition is a tree decomposition (X, T) in which the underlying tree T of the decomposition is a path graph.

The width of a path-decomposition is defined in the same way as for tree-decompositions, as $\max_i |X_i| - 1$, and the pathwidth of G is the minimum width of any path-decomposition of G . The subtraction of one from the size of X_i in this definition makes little difference in most applications of pathwidth, but is used to make the pathwidth of a path graph be equal to one.

Alternative characterizations

As Bodlaender (1998) describes, pathwidth can be characterized in many equivalent ways.

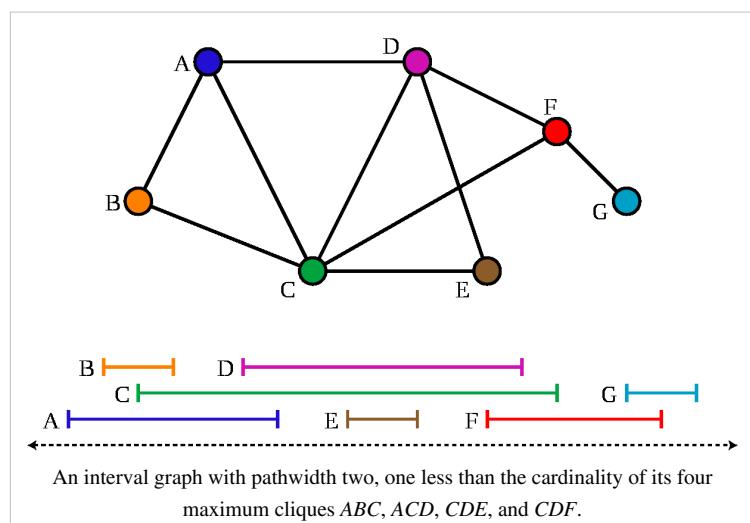
Gluing sequences

A path decomposition can be described as a sequence of graphs G_i that are glued together by identifying pairs of vertices from consecutive graphs in the sequence, such that the result of performing all of these gluings is G . The graphs G_i may be taken as the induced subgraphs of the sets X_i in the first definition of path decompositions, with two vertices in successive induced subgraphs being glued together when they are induced by the same vertex in G , and in the other direction one may recover the sets X_i as the vertex sets of the graphs G_i . The width of the path decomposition is then one less than the maximum number of vertices in one of the graphs G_i .

Interval thickness

The pathwidth of any graph G is equal to one less than the smallest clique number of an interval graph that contains G as a subgraph.^[1] That is, for every path decomposition of G one can find an interval supergraph of G , and for every interval supergraph of G one can find a path decomposition of G , such that the width of the decomposition is one less than the clique number of the interval graph.

In one direction, suppose a path decomposition of G is given. Then one may represent the nodes of the decomposition as points on a line (in path order) and represent each vertex v as a closed interval having these points as endpoints. In this way, the path decomposition nodes containing v correspond to the representative points in the interval for v . The intersection graph of the intervals formed from the vertices of G is an interval graph that contains G as a subgraph. Its maximal cliques are given by the sets of intervals containing the representative points, and its maximum clique size is one plus the pathwidth of G .



In the other direction, if G is a subgraph of an interval graph with clique number $p + 1$, then G has a path decomposition of width p whose nodes are given by the maximal cliques of the interval graph. For instance, the interval graph shown with its interval representation in the figure has a path decomposition with five nodes, corresponding to its five maximal cliques ABC , ACD , CDE , CDF , and FG ; the maximum clique size is three and the width of this path decomposition is two.

This equivalence between pathwidth and interval thickness is closely analogous to the equivalence between treewidth and the minimum clique number (minus one) of a chordal graph of which the given graph is a subgraph. Interval graphs are a special case of chordal graphs, and chordal graphs can be represented as intersection graphs of subtrees of a common tree generalizing the way that interval graphs are intersection graphs of subpaths of a path.

Vertex separation number

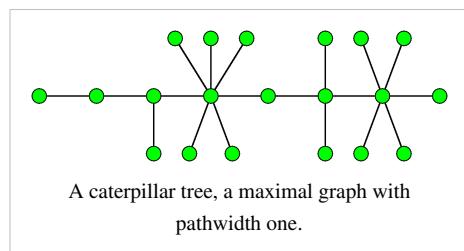
Suppose that the vertices of a graph G are linearly ordered. Then the vertex separation number of G is the smallest number s such that, for each vertex v , at most s vertices are earlier than v in the ordering but that have v or a later vertex as a neighbor. The vertex separation number of G is the minimum vertex separation number of any linear ordering of G . The vertex separation number was defined by Ellis, Sudborough & Turner (1983), and is equal to the pathwidth of G .^[2] This follows from the earlier equivalence with interval graph clique numbers: if G is a subgraph of an interval graph I , represented (as in the figure) in such a way that all interval endpoints are distinct, then the ordering of the left endpoints of the intervals of I has vertex separation number one less than the clique number of I . And in the other direction, from a linear ordering of G one may derive an interval representation in which the left endpoint of the interval for a vertex v is its position in the ordering and the right endpoint is the position of the neighbor of v that comes last in the ordering.

Node searching number

The node searching game on a graph is a form of pursuit-evasion in which a set of searchers collaborate to track down a fugitive hiding in a graph. The searchers are placed on vertices of the graph while the fugitive may be in any edge of the graph, and the fugitive's location and moves are hidden from the searchers. In each turn, some or all of the searchers may move (arbitrarily, not necessarily along edges) from one vertex to another, and then the fugitive may move along any path in the graph that does not pass through a searcher-occupied vertex. The fugitive is caught when both endpoints of his edge are occupied by searchers. The node searching number of a graph is the minimum number of searchers needed to ensure that the fugitive can be guaranteed to be caught, no matter how he moves. As Kirousis & Papadimitriou (1985) show, the node searching number of a graph equals its interval thickness. The optimal strategy for the searchers is to move the searchers so that in successive turns they form the separating sets of a linear ordering with minimal vertex separation number.

Bounds

Every n -vertex graph with pathwidth k has at most $k(n - k + (k - 1)/2)$ edges, and the maximal pathwidth- k graphs (graphs to which no more edges can be added without increasing the pathwidth) have exactly this many edges. A maximal pathwidth- k graph must be either a k -path or a k -caterpillar, two special kinds of k -tree. A k -tree is a chordal graph with exactly $n - k$ maximal cliques, each containing $k + 1$ vertices; in a k -tree that is not itself a $(k + 1)$ -clique, each maximal clique either separates the graph into two or more components, or it contains a single leaf vertex, a vertex that belongs to only a single maximal clique. A k -path is a k -tree with at most two leaves, and a k -caterpillar is a k -tree that can be partitioned into a k -path and a set of k -leaves each adjacent to a separator k -clique of the k -path. In particular the maximal graphs of pathwidth one are exactly the caterpillar trees.



Since path-decompositions are a special case of tree-decompositions, the pathwidth of any graph is greater than or equal to its treewidth. The pathwidth is also less than or equal to the cutwidth, the minimum number of edges that crosses any cut between lower-numbered and higher-numbered vertices in an optimal linear arrangement of the vertices of a graph; this follows because the vertex separation number, the number of lower-numbered vertices with higher-numbered neighbors, can at most equal the number of cut edges.^[3] For similar reasons, the cutwidth is at most the pathwidth times the maximum degree of the vertices in a given graph.^[4]

Any n -vertex forest has pathwidth $O(\log n)$.^[5] For, in a forest, one can always find a constant number of vertices the removal of which leaves a forest that can be partitioned into two smaller subforests with at most $2n/3$ vertices each. A linear arrangement formed by recursively partitioning each of these two subforests, placing the separating vertices between them, has logarithmic vertex searching number. The same technique, applied to a tree-decomposition of a graph, shows that, if the treewidth of an n -vertex graph G is t , then the pathwidth of G is $O(t \log n)$.^[6] Since outerplanar graphs, series-parallel graphs, and Halin graphs all have bounded treewidth, they all also have at most logarithmic pathwidth.

As well as its relations to treewidth, pathwidth is also related to clique-width and cutwidth, via line graphs; the line graph $L(G)$ of a graph G has a vertex for each edge of G and two vertices in $L(G)$ are adjacent when the corresponding two edges of G share an endpoint. Any family of graphs has bounded pathwidth if and only if its line graphs have bounded linear clique-width, where linear clique-width replaces the disjoint union operation from clique-width with the operation of adjoining a single new vertex. If a connected graph with three or more vertices has maximum degree three, then its cutwidth equals the vertex separation number of its line graph.

In any planar graph, the pathwidth is at most proportional to the square root of the number of vertices.^[7] One way to find a path-decomposition with this width is (similarly to the logarithmic-width path-decomposition of forests described above) to use the planar separator theorem to find a set of $O(\sqrt{n})$ vertices the removal of which separates the graph into two subgraphs of at most $2n/3$ vertices each, and concatenate recursively-constructed path decompositions for each of these two subgraphs. The same technique applies to any class of graphs for which a similar separator theorem holds.^[8] Since, like planar graphs, the graphs in any fixed minor-closed graph family have separators of size $O(\sqrt{n})$, it follows that the pathwidth of the graphs in any fixed minor-closed family is again $O(\sqrt{n})$. For some classes of planar graphs, the pathwidth of the graph and the pathwidth of its dual graph must be within a constant factor of each other: bounds of this form are known for biconnected outerplanar graphs^[9] and for polyhedral graphs.^[10] For 2-connected planar graphs, the pathwidth of the dual graph less than the pathwidth of the line graph. It remains open whether the pathwidth of a planar graph and its dual are always within a constant factor of each other in the remaining cases.

In some classes of graphs, it has been proven that the pathwidth and treewidth are always equal to each other: this is true for cographs, permutation graphs, the complements of comparability graphs, and the comparability graphs of interval orders.

In any cubic graph, or more generally any graph with maximum vertex degree three, the pathwidth is at most $n/6 + o(n)$, where n is the number of vertices in the graph. There exist cubic graphs with pathwidth $0.082n$, but it is not known how to reduce this gap between this lower bound and the $n/6$ upper bound.

Computing path-decompositions

It is NP-complete to determine whether the pathwidth of a given graph is at most k , when k is a variable given as part of the input.^[1] The best known worst-case time bounds for computing the pathwidth of arbitrary n -vertex graphs are of the form $O(2^n n^c)$ for some constant c . Nevertheless several algorithms are known to compute path-decompositions more efficiently when the pathwidth is small, when the class of input graphs is limited, or approximately.

Fixed-parameter tractability

Pathwidth is fixed-parameter tractable: for any constant k , it is possible to test whether the pathwidth is at most k , and if so to find a path-decomposition of width k , in linear time. In general, these algorithms operate in two phases. In the first phase, the assumption that the graph has pathwidth k is used to find a path-decomposition or tree-decomposition that is not optimal, but whose width can be bounded as a function of k . In the second phase, a dynamic programming algorithm is applied to this decomposition in order to find the optimal decomposition. However, the time bounds for known algorithms of this type are exponential in k^2 , impractical except for the smallest values of k .^[11] For the case $k = 2$ an explicit linear-time algorithm based on a structural decomposition of pathwidth-2 graphs is given by de Fluiter (1997).

Special classes of graphs

Bodlaender (1994) surveys the complexity of computing the pathwidth on various special classes of graphs. Determining whether the pathwidth of a graph G is at most k remains NP-complete when G is restricted to bounded-degree graphs, planar graphs, planar graphs of bounded degree, chordal graphs, chordal dominoes,^[12] the complements of comparability graphs, and bipartite distance-hereditary graphs. It follows immediately that it is also NP-complete for the graph families that contain the bipartite distance-hereditary graphs, including the bipartite graphs, chordal bipartite graphs, distance-hereditary graphs, and circle graphs.

However, the pathwidth may be computed in linear time for trees and forests.^[1] It may also be computed in polynomial time for graphs of bounded treewidth including series-parallel graphs, outerplanar graphs, and Halin graphs, as well as for split graphs,^[13] for the complements of chordal graphs,^[14] for permutation graphs, for cographs, for circular-arc graphs, for the comparability graphs of interval orders, and of course for interval graphs themselves, since in that case the pathwidth is just one less than the maximum number of intervals covering any point in an interval representation of the graph.

Approximation algorithms

It is NP-hard to approximate the pathwidth of a graph to within an additive constant. The best known approximation ratio of a polynomial time approximation algorithm for pathwidth is $O((\log n)^{3/2})$. For earlier approximation algorithms for pathwidth, see Bodlaender et al. (1992) and Guha (2000). For approximations on restricted classes of graphs, see Kloks & Bodlaender (1992).

Graph minors

A minor of a graph G is another graph formed from G by contracting edges, removing edges, and removing vertices. Graph minors have a deep theory in which several important results involve pathwidth.

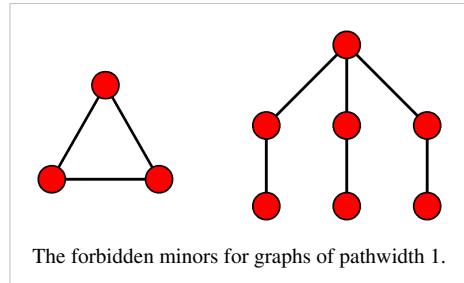
Excluding a forest

If a family F of graphs is closed under taking minors (every minor of a member of F is also in F), then by the Robertson–Seymour theorem F can be characterized as the graphs that do not have any minor in X , where X is a finite set of forbidden minors. For instance, Wagner's theorem states that the planar graphs are the graphs that have neither the complete graph K_5 nor the complete bipartite graph $K_{3,3}$ as minors. In many cases, the properties of F and the properties of X are closely related, and the first such result of this type was by Robertson & Seymour (1983), and relates bounded pathwidth with the existence of a forest in the family of forbidden minors. Specifically, define a family F of graphs to have *bounded pathwidth* if there exists a constant p such that every graph in F has pathwidth at most p . Then, a minor-closed family F has bounded pathwidth if and only if the set X of forbidden minors for F includes at least one forest.

In one direction, this result is straightforward to prove: if X does not include at least one forest, then the X -minor-free graphs do not have bounded pathwidth. For, in this case, the X -minor-free graphs include all forests, and in particular they include the perfect binary trees. But a perfect binary tree with $2k+1$ levels has pathwidth k , so in this case the X -minor-free-graphs have unbounded pathwidth. In the other direction, if X contains an n -vertex forest, then the X -minor-free graphs have pathwidth at most $n - 2$.^[15]

Obstructions to bounded pathwidth

The property of having pathwidth at most p is, itself, closed under taking minors: if G has a path-decomposition with width at most p , then the same path-decomposition remains valid if any edge is removed from G , and any vertex can be removed from G and from its path-decomposition without increasing the width. Contraction of an edge, also, can be accomplished without increasing the width of the decomposition, by merging the sub-paths representing the two endpoints of the contracted edge. Therefore, the graphs of pathwidth at most p can be characterized by a set X_p of excluded minors.^[16]



The forbidden minors for graphs of pathwidth 1.

Although X_p necessarily includes at least one forest, it is not true that all graphs in X_p are forests: for instance, X_1 consists of two graphs, a seven-vertex tree and the triangle K_3 . However, the set of trees in X_p may be precisely characterized: these trees are exactly the trees that can be formed from three trees in X_{p-1} by connecting a new root vertex by an edge to an arbitrarily chosen vertex in each of the three smaller trees. For instance, the seven-vertex tree in X_1 is formed in this way from the two-vertex tree (a single edge) in X_0 . Based on this construction, the number of forbidden minors in X_p can be shown to be at least $(p!)^2$. The complete set X_2 of forbidden minors for pathwidth-2 graphs has been computed; it contains 110 different graphs.

Structure theory

The graph structure theorem for minor-closed graph families states that, for any such family F , the graphs in F can be decomposed into clique-sums of graphs that can be embedded onto surfaces of bounded genus, together with a bounded number of apexes and vortices for each component of the clique-sum. An apex is a vertex that may be adjacent to any other vertex in its component, while a vortex is a graph of bounded pathwidth that is glued into one of the faces of the bounded-genus embedding of a component. The cyclic ordering of the vertices around the face into which a vortex is embedded must be compatible with the path decomposition of the vortex, in the sense that breaking the cycle to form a linear ordering must lead to an ordering with bounded vertex separation number. This theory, in which pathwidth is intimately connected to arbitrary minor-closed graph families, has important algorithmic applications.

Applications

VLSI

In VLSI design, the vertex separation problem was originally studied as a way to partition circuits into smaller subsystems, with a small number of components on the boundary between the subsystems.

Ohtsuki et al. (1979) use interval thickness to model the number of tracks needed in a one-dimensional layout of a VLSI circuit, formed by a set of modules that need to be interconnected by a system of nets. In their model, one forms a graph in which the vertices represent nets, and in which two vertices are connected by an edge if their nets both connect to the same module; that is, if the modules and nets are interpreted as forming the nodes and hyperedges of a hypergraph then the graph formed from them is its line graph. An interval representation of a supergraph of this line graph, together with a coloring of the supergraph, describes an arrangement of the nets along a system of horizontal tracks (one track per color) in such a way that the modules can be placed along the tracks in a linear order and connect to the appropriate nets. The fact that interval graphs are perfect graphs implies that the number of colors needed, in an optimal arrangement of this type, is the same as the clique number of the interval completion of the net graph.

Gate matrix layout is a specific style of CMOS VLSI layout for Boolean logic circuits. In gate matrix layouts, signals are propagated along "lines" (vertical line segments) while each gate of the circuit is formed by a sequence of device features that lie along a horizontal line segment. Thus, the horizontal line segment for each gate must cross the vertical segments for each of the lines that form inputs or outputs of the gate. As in the layouts of Ohtsuki et al. (1979), a layout of this type that minimizes the number of vertical tracks on which the lines are to be arranged can be found by computing the pathwidth of a graph that has the lines as its vertices and pairs of lines sharing a gate as its edges. The same algorithmic approach can also be used to model folding problems in programmable logic arrays.^[17]

Graph drawing

Pathwidth has several applications to graph drawing:

- The minimal graphs that have a given crossing number have pathwidth that is bounded by a function of their crossing number.
- The number of parallel lines on which the vertices of a tree can be drawn with no edge crossings (under various natural restrictions on the ways that adjacent vertices can be placed with respect to the sequence of lines) is proportional to the pathwidth of the tree.
- A k -crossing h -layer drawing of a graph G is a placement of the vertices of G onto h distinct horizontal lines, with edges routed as monotonic polygonal paths between these lines, in such a way that there are at most k crossings. The graphs with such drawings have pathwidth that is bounded by a function of h and k . Therefore, when h and k are both constant, it is possible in linear time to determine whether a graph has a k -crossing h -layer drawing.
- A graph with n vertices and pathwidth p can be embedded into a three-dimensional grid of size $p \times p \times n$ in such a way that no two edges (represented as straight line segments between grid points) intersect each other. Thus, graphs of bounded pathwidth have embeddings of this type with linear volume.

Compiler design

In the compilation of high-level programming languages, pathwidth arises in the problem of reordering sequences of straight-line code (that is, code with no control flow branches or loops) in such a way that all the values computed in the code can be placed in machine registers instead of having to be spilled into main memory. In this application, one represents the code to be compiled as a directed acyclic graph in which the nodes represent the input values to the code and the values computed by the operations within the code. An edge from node x to node y in this DAG represents the fact that value x is one of the inputs to operation y . A topological ordering of the vertices of this DAG represents a valid reordering of the code, and the number of registers needed to evaluate the code in a given ordering

is given by the vertex separation number of the ordering.

For any fixed number w of machine registers, it is possible to determine in linear time whether a piece of straight-line code can be reordered in such a way that it can be evaluated with at most w registers. For, if the vertex separation number of a topological ordering is at most w , the minimum vertex separation among all orderings can be no larger, so the undirected graph formed by ignoring the orientations of the DAG described above must have pathwidth at most w . It is possible to test whether this is the case, using the known fixed-parameter-tractable algorithms for pathwidth, and if so to find a path-decomposition for the undirected graph, in linear time given the assumption that w is a constant. Once a path decomposition has been found, a topological ordering of width w (if one exists) can be found using dynamic programming, again in linear time.

Linguistics

Kornai & Tuza (1992) describe an application of path-width in natural language processing. In this application, sentences are modeled as graphs, in which the vertices represent words and the edges represent relationships between words; for instance if an adjective modifies a noun in the sentence then the graph would have an edge between those two words. Due to the limited capacity of human short-term memory, Kornai and Tuza argue that this graph must have bounded pathwidth (more specifically, they argue, pathwidth at most six), for otherwise humans would not be able to parse speech correctly.

Exponential algorithms

Many problems in graph algorithms may be solved efficiently on graphs of low pathwidth, by using dynamic programming on a path-decomposition of the graph. For instance, if a linear ordering of the vertices of an n -vertex graph G is given, with vertex separation number w , then it is possible to find the maximum independent set of G in time $O(2^w n)$. On graphs of bounded pathwidth, this approach leads to fixed-parameter tractable algorithms, parametrized by the pathwidth. Such results are not frequently found in the literature because they are subsumed by similar algorithms parametrized by the treewidth; however, pathwidth arises even in treewidth-based dynamic programming algorithms in measuring the space complexity of these algorithms.

The same dynamic programming method also can be applied to graphs with unbounded pathwidth, leading to algorithms that solve unparametrized graph problems in exponential time. For instance, combining this dynamic programming approach with the fact that cubic graphs have pathwidth $n/6 + o(n)$ shows that, in a cubic graph, the maximum independent set can be constructed in time $O(2^{n/6 + o(n)})$, faster than previous known methods. A similar approach leads to improved exponential-time algorithms for the maximum cut and minimum dominating set problems in cubic graphs, and for several other NP-hard optimization problems.^[18]

Notes

[1] , Theorem 29, p. 13.

[2] ; , Theorem 51.

[3] , Lemma 3 p.99; , Theorem 47, p. 24.

[4] , Lemma 1, p. 99; , Theorem 49, p. 24.

[5] , Theorem 5, p. 99; , Theorem 66, p. 30. gives a tighter upper bound of $\log_3(2n + 1)$ on the pathwidth of an n -vertex forest.

[6] , Theorem 6, p. 100; , Corollary 24, p.10.

[7] , Corollary 23, p. 10.

[8] , Theorem 20, p. 9.

[9] ; .

[10] ; .

[11] , p.12.

[12] . A chordal domino is a chordal graph in which every vertex belongs to at most two maximal cliques.

[13] ; .

[14] credits this result to the 1993 Ph.D. thesis of Ton Kloks; Garbe's polynomial time algorithm for comparability graphs of interval orders generalizes this result, since any chordal graph must be a comparability graph of this type.

- [15] ; ; .
- [16] ; ; , p. 8.
- [17] ; .
- [18] ; .

References

- Alon, Noga; Seymour, Paul; Thomas, Robin (1990), "A separator theorem for graphs with an excluded minor and its applications", *Proc. 22nd ACM Symp. on Theory of Computing (STOC 1990)*, pp. 293–299, doi: 10.1145/100216.100254 (<http://dx.doi.org/10.1145/100216.100254>), ISBN 0897913612.
- Amini, Omid; Huc, Florian; Pérennes, Stéphane (2009), "On the path-width of planar graphs", *SIAM Journal on Discrete Mathematics* **23** (3): 1311–1316, doi: 10.1137/060670146 (<http://dx.doi.org/10.1137/060670146>).
- Arnborg, Stefan (1985), "Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey", *BIT* **25** (1): 2–23, doi: 10.1007/BF01934985 (<http://dx.doi.org/10.1007/BF01934985>).
- Arnborg, Stefan; Corneil, Derek G.; Proskurowski, Andrzej (1987), "Complexity of finding embeddings in a $k\$$ -tree", *SIAM Journal on Algebraic and Discrete Methods* **8** (2): 277–284, doi: 10.1137/0608024 (<http://dx.doi.org/10.1137/0608024>).
- Aspvall, Bengt; Proskurowski, Andrzej; Telle, Jan Arne (2000), "Memory requirements for table computations in partial k -tree algorithms", *Algorithmica* **27** (3): 382–394, doi: 10.1007/s004530010025 (<http://dx.doi.org/10.1007/s004530010025>).
- Berge, Claude (1967), "Some classes of perfect graphs", *Graph Theory and Theoretical Physics*, New York: Academic Press, pp. 155–165.
- Bienstock, Dan; Robertson, Neil; Seymour, Paul; Thomas, Robin (1991), "Quickly excluding a forest", *Journal of Combinatorial Theory, Series B* **52** (2): 274–283, doi: 10.1016/0095-8956(91)90068-U ([http://dx.doi.org/10.1016/0095-8956\(91\)90068-U](http://dx.doi.org/10.1016/0095-8956(91)90068-U)).
- Björklund, Andreas; Husfeldt, Thore (2008), "Exact algorithms for exact satisfiability and number of perfect matchings", *Algorithmica* **52** (2): 226–249, doi: 10.1007/s00453-007-9149-8 (<http://dx.doi.org/10.1007/s00453-007-9149-8>).
- Bodlaender, Hans L. (1994), "A tourist guide through treewidth", in Dassow, Jürgen; Kelemenová, Alisa, *Developments in Theoretical Computer Science (Proc. 7th International Meeting of Young Computer Scientists, Smolenice, 16–20 November 1992)*, Topics in Computer Mathematics **6**, Gordon and Breach, pp. 1–20.
- Bodlaender, Hans L. (1996), "A linear-time algorithm for finding tree-decompositions of small treewidth", *SIAM Journal on Computing* **25** (6): 1305–1317, doi: 10.1137/S0097539793251219 (<http://dx.doi.org/10.1137/S0097539793251219>).
- Bodlaender, Hans L. (1998), "A partial k -arboretum of graphs with bounded treewidth", *Theoretical Computer Science* **209** (1–2): 1–45, doi: 10.1016/S0304-3975(97)00228-4 ([http://dx.doi.org/10.1016/S0304-3975\(97\)00228-4](http://dx.doi.org/10.1016/S0304-3975(97)00228-4)).
- Bodlaender, Hans L.; Fomin, Fedor V. (2002), "Approximation of pathwidth of outerplanar graphs", *Journal of Algorithms* **43** (2): 190–200, doi: 10.1016/S0196-6774(02)00001-9 ([http://dx.doi.org/10.1016/S0196-6774\(02\)00001-9](http://dx.doi.org/10.1016/S0196-6774(02)00001-9)).
- Bodlaender, Hans L.; Gilbert, John R.; Hafsteinsson, Hjálmtýr; Kloks, Ton (1992), "Approximating treewidth, pathwidth, and minimum elimination tree height", *Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science **570**, pp. 1–12, doi: 10.1007/3-540-55121-2_1 (http://dx.doi.org/10.1007/3-540-55121-2_1), ISBN 978-3-540-55121-8.
- Bodlaender, Hans L.; Gustadt, Jens; Telle, Jan Arne (1998), "Linear-time register allocation for a fixed number of registers" (<http://www.ii.uib.no/~telle/bib/BGT.pdf>), *Proc. 9th ACM–SIAM Symposium on Discrete Algorithms (SODA '98)*, pp. 574–583.

- Bodlaender, Hans L.; Kloks, Ton (1996), "Efficient and constructive algorithms for the pathwidth and treewidth of graphs", *Journal of Algorithms* **21** (2): 358–402, doi: 10.1006/jagm.1996.0049 (<http://dx.doi.org/10.1006/jagm.1996.0049>).
- Bodlaender, Hans L.; Kloks, Ton; Kratsch, Dieter (1993), "Treewidth and pathwidth of permutation graphs", *Proc. 20th International Colloquium on Automata, Languages and Programming (ICALP 1993)*, Lecture Notes in Computer Science **700**, Springer-Verlag, pp. 114–125, doi: 10.1007/3-540-56939-1_66 (http://dx.doi.org/10.1007/3-540-56939-1_66), ISBN 978-3-540-56939-8.
- Bodlaender, Hans L.; Möhring, Rolf H. (1990), "The pathwidth and treewidth of cographs", *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **447**, Springer-Verlag, pp. 301–309, doi: 10.1007/3-540-52846-6_99 (http://dx.doi.org/10.1007/3-540-52846-6_99), ISBN 978-3-540-52846-3.
- Cattell, Kevin; Dinneen, Michael J.; Fellows, Michael R. (1996), "A simple linear-time algorithm for finding path-decompositions of small width", *Information Processing Letters* **57** (4): 197–203, doi: 10.1016/0020-0190(95)00190-5 ([http://dx.doi.org/10.1016/0020-0190\(95\)00190-5](http://dx.doi.org/10.1016/0020-0190(95)00190-5)).
- Coudert, David; Huc, Florian; Mazauric, Dorian (1998), "A distributed algorithm for computing and updating the process number of a forest", *Proc. 22nd Int. Symp. Distributed Computing*, Lecture Notes in Computer Science **5218**, Springer-Verlag, pp. 500–501, arXiv: 0806.2710 (<http://arxiv.org/abs/0806.2710>), doi: 10.1007/978-3-540-87779-0_36 (http://dx.doi.org/10.1007/978-3-540-87779-0_36), ISBN 978-3-540-87778-3.
- Coudert, David; Huc, Florian; Sereni, Jean-Sébastien (2007), "Pathwidth of outerplanar graphs", *Journal of Graph Theory* **55** (1): 27–41, doi: 10.1002/jgt.20218 (<http://dx.doi.org/10.1002/jgt.20218>).
- Diestel, Reinhard (1995), "Graph Minors I: a short proof of the path-width theorem", *Combinatorics, Probability and Computing* **4** (1): 27–30, doi: 10.1017/S0963548300001450 (<http://dx.doi.org/10.1017/S0963548300001450>).
- Diestel, Reinhard; Kühn, Daniela (2005), "Graph minor hierarchies", *Discrete Applied Mathematics* **145** (2): 167–182, doi: 10.1016/j.dam.2004.01.010 (<http://dx.doi.org/10.1016/j.dam.2004.01.010>).
- Demaine, Erik D.; Hajiaghayi, MohammadTaghi; Kawarabayashi, Ken-ichi (2005), "Algorithmic graph minor theory: decomposition, approximation, and coloring", *Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005)*, pp. 637–646, doi: 10.1109/SFCS.2005.14 (<http://dx.doi.org/10.1109/SFCS.2005.14>), ISBN 0-7695-2468-0.
- Downey, Rod G.; Fellows, Michael R. (1999), *Parameterized Complexity*, Springer-Verlag, ISBN 0-387-94883-X.
- Dujmović, V.; Fellows, M.R.; Kitching, M.; Liotta, G.; McCartin, C.; Nishimura, N.; Ragde, P.; Rosamond, F. et al. (2008), "On the parameterized complexity of layered graph drawing", *Algorithmica* **52** (2): 267–292, doi: 10.1007/s00453-007-9151-1 (<http://dx.doi.org/10.1007/s00453-007-9151-1>).
- Dujmović, Vida; Morin, Pat; Wood, David R. (2003), "Path-width and three-dimensional straight-line grid drawings of graphs" (<http://cg.scs.carleton.ca/~vida/pubs/papers/DMW-GD02.pdf>), *Proc. 10th International Symposium on Graph Drawing (GD 2002)*, Lecture Notes in Computer Science **2528**, Springer-Verlag, pp. 42–53.
- Ellis, J. A.; Sudborough, I. H.; Turner, J. S. (1983), "Graph separation and search number", *Proc. 1983 Allerton Conf. on Communication, Control, and Computing*. As cited by Monien & Sudborough (1988).
- Ellis, J. A.; Sudborough, I. H.; Turner, J. S. (1994), "The vertex separation and search number of a tree", *Information and Computation* **113** (1): 50–79, doi: 10.1006/inco.1994.1064 (<http://dx.doi.org/10.1006/inco.1994.1064>).
- Feige, Uriel; Hajiaghayi, Mohammadtaghi; Lee, James R. (2005), "Improved approximation algorithms for minimum-weight vertex separators", *Proc. 37th ACM Symposium on Theory of Computing (STOC 2005)*, pp. 563–572, doi: 10.1145/1060590.1060674 (<http://dx.doi.org/10.1145/1060590.1060674>),

ISBN 1581139608.

- Fellows, Michael R.; Langston, Michael A. (1989), "On search decision and the efficiency of polynomial-time algorithms", *Proc. 21st ACM Symposium on Theory of Computing*, pp. 501–512, doi: 10.1145/73007.73055 (<http://dx.doi.org/10.1145/73007.73055>), ISBN 0897913078.
- Ferreira, Afonso G.; Song, Siang W. (1992), "Achieving optimality for gate matrix layout and PLA folding: a graph theoretic approach", *Proc. 1st Latin American Symposium on Theoretical Informatics (LATIN '92)*, Lecture Notes in Computer Science **583**, Springer-Verlag, pp. 139–153, doi: 10.1007/BFb0023825 (<http://dx.doi.org/10.1007/BFb0023825>), ISBN 3-540-55284-7.
- de Fluiter, Babette (1997), *Algorithms for Graphs of Small Treewidth* (<http://igitur-archive.library.uu.nl/dissertations/01847381/full.pdf>), Ph.D. thesis, Utrecht University, ISBN 90-393-1528-0.
- Fomin, Fedor V. (2003), "Pathwidth of planar and line graphs", *Graphs and Combinatorics* **19** (1): 91–99, doi: 10.1007/s00373-002-0490-z (<http://dx.doi.org/10.1007/s00373-002-0490-z>).
- Fomin, Fedor V.; Høie, Kjartan (2006), "Pathwidth of cubic graphs and exact algorithms", *Information Processing Letters* **97** (5): 191–196, doi: 10.1016/j.ipl.2005.10.012 (<http://dx.doi.org/10.1016/j.ipl.2005.10.012>).
- Fomin, Fedor V.; Kratsch, Dieter; Todinca, Ioan; Villanger, Yngve (2008), "Exact algorithms for treewidth and minimum fill-in", *SIAM Journal on Computing* **38** (3): 1058–1079, doi: 10.1137/050643350 (<http://dx.doi.org/10.1137/050643350>).
- Fomin, Fedor V.; Thilikos, Dimitrios M. (2007), "On self duality of pathwidth in polyhedral graph embeddings", *Journal of Graph Theory* **55** (1): 42–54, doi: 10.1002/jgt.20219 (<http://dx.doi.org/10.1002/jgt.20219>).
- Garbe, Renate (1995), "Tree-width and path-width of comparability graphs of interval orders", *Proc. 20th International Workshop Graph-Theoretic Concepts in Computer Science (WG'94)*, Lecture Notes in Computer Science **903**, Springer-Verlag, pp. 26–37, doi: 10.1007/3-540-59071-4_35 (http://dx.doi.org/10.1007/3-540-59071-4_35), ISBN 978-3-540-59071-2.
- Golovach, P. A. (1993), "The cutwidth of a graph and the vertex separation number of the line graph", *Discrete Mathematics and Applications* **3** (5): 517–522, doi: 10.1515/dma.1993.3.5.517 (<http://dx.doi.org/10.1515/dma.1993.3.5.517>).
- Guha, Sudipto (2000), "Nested graph dissection and approximation algorithms", *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS 2000)* **0**: 126, doi: 10.1109/SFCS.2000.892072 (<http://dx.doi.org/10.1109/SFCS.2000.892072>), ISBN 0-7695-0850-2.
- Gurski, Frank; Wanke, Egon (2007), "Line graphs of bounded clique-width", *Discrete Mathematics* **307** (22): 2734–2754, doi: 10.1016/j.disc.2007.01.020 (<http://dx.doi.org/10.1016/j.disc.2007.01.020>).
- Gusted, Jens (1993), "On the pathwidth of chordal graphs", *Discrete Applied Mathematics* **45** (3): 233–248, doi: 10.1016/0166-218X(93)90012-D ([http://dx.doi.org/10.1016/0166-218X\(93\)90012-D](http://dx.doi.org/10.1016/0166-218X(93)90012-D)).
- Habib, Michel; Möhring, Rolf H. (1994), "Treewidth of cocomparability graphs and a new order-theoretic parameter", *Order* **11** (1): 47–60, doi: 10.1007/BF01462229 (<http://dx.doi.org/10.1007/BF01462229>).
- Hliněny, Petr (2003), "Crossing-number critical graphs have bounded path-width", *Journal of Combinatorial Theory, Series B* **88** (2): 347–367, doi: 10.1016/S0095-8956(03)00037-6 ([http://dx.doi.org/10.1016/S0095-8956\(03\)00037-6](http://dx.doi.org/10.1016/S0095-8956(03)00037-6)).
- Kashiwabara, T.; Fujisawa, T. (1979), "NP-completeness of the problem of finding a minimum-clique-number interval graph containing a given graph as a subgraph", *Proc. International Symposium on Circuits and Systems*, pp. 657–660.
- Kinnersley, Nancy G. (1992), "The vertex separation number of a graph equals its path-width", *Information Processing Letters* **42** (6): 345–350, doi: 10.1016/0020-0190(92)90234-M ([http://dx.doi.org/10.1016/0020-0190\(92\)90234-M](http://dx.doi.org/10.1016/0020-0190(92)90234-M)).
- Kinnersley, Nancy G.; Langston, Michael A. (1994), "Obstruction set isolation for the gate matrix layout problem", *Discrete Applied Mathematics* **54** (2–3): 169–213, doi: 10.1016/0166-218X(94)90021-3 ([http://dx.doi.org/10.1016/0166-218X\(94\)90021-3](http://dx.doi.org/10.1016/0166-218X(94)90021-3)).

- doi.org/10.1016/0166-218X(94)90021-3).
- Kirousis, Lefteris M.; Papadimitriou, Christos H. (1985), "Interval graphs and searching" (<http://lca.ceid.upatras.gr/~kirousis/publications/j31.pdf>), *Discrete Mathematics* **55** (2): 181–184, doi: 10.1016/0012-365X(85)90046-9 ([http://dx.doi.org/10.1016/0012-365X\(85\)90046-9](http://dx.doi.org/10.1016/0012-365X(85)90046-9)).
 - Kloks, Ton; Bodlaender, Hans L. (1992), "Approximating treewidth and pathwidth of some classes of perfect graphs", *Proc. 3rd International Symposium on Algorithms and Computation (ISAAC'92)*, Lecture Notes in Computer Science **650**, Springer-Verlag, pp. 116–125, doi: 10.1007/3-540-56279-6_64 (http://dx.doi.org/10.1007/3-540-56279-6_64), ISBN 978-3-540-56279-5.
 - Kloks, T.; Bodlaender, H.; Müller, H.; Kratsch, D. (1993), "Computing treewidth and minimum fill-in: all you need are the minimal separators", *Proc. 1st European Symposium on Algorithms (ESA'93) (Lecture Notes in Computer Science)* **726**, Springer-Verlag, pp. 260–271, doi: 10.1007/3-540-57273-2_61 (http://dx.doi.org/10.1007/3-540-57273-2_61).
 - Kloks, Ton; Kratsch, Dieter; Müller, H. (1995), "Dominoes", *Proc. 20th International Workshop Graph-Theoretic Concepts in Computer Science (WG'94)*, Lecture Notes in Computer Science **903**, Springer-Verlag, pp. 106–120, doi: 10.1007/3-540-59071-4_41 (http://dx.doi.org/10.1007/3-540-59071-4_41), ISBN 978-3-540-59071-2.
 - Kneis, Joachim; Mölle, Daniel; Richter, Stefan; Rossmanith, Peter (2005), "Algorithms based on the treewidth of sparse graphs", *Proc. 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2005)*, Lecture Notes in Computer Science **3787**, Springer-Verlag, pp. 385–396, doi: 10.1007/11604686_34 (http://dx.doi.org/10.1007/11604686_34), ISBN 978-3-540-31000-6.
 - Korach, Ephraim; Solel, Nir (1993), "Tree-width, path-width, and cutwidth", *Discrete Applied Mathematics* **43** (1): 97–101, doi: 10.1016/0166-218X(93)90171-J ([http://dx.doi.org/10.1016/0166-218X\(93\)90171-J](http://dx.doi.org/10.1016/0166-218X(93)90171-J)).
 - Kornai, András; Tuza, Zsolt (1992), "Narrowness, path-width, and their application in natural language processing", *Discrete Applied Mathematics* **36** (1): 87–92, doi: 10.1016/0166-218X(92)90208-R ([http://dx.doi.org/10.1016/0166-218X\(92\)90208-R](http://dx.doi.org/10.1016/0166-218X(92)90208-R)).
 - Lengauer, Thomas (1981), "Black-white pebbles and graph separation", *Acta Informatica* **16** (4): 465–475, doi: 10.1007/BF00264496 (<http://dx.doi.org/10.1007/BF00264496>).
 - Lopez, Alexander D.; Law, Hung-Fai S. (1980), "A dense gate matrix layout method for MOS VLSI", *IEEE Transactions on Electron Devices*, ED-27 (8): 1671–1675, doi: 10.1109/T-ED.1980.20086 (<http://dx.doi.org/10.1109/T-ED.1980.20086>), Also in the joint issue, *IEEE Journal of Solid-State Circuits* **15** (4): 736–740, 1980, doi: 10.1109/JSSC.1980.1051462 (<http://dx.doi.org/10.1109/JSSC.1980.1051462>).
 - Miller, George A. (1956), "The Magical Number Seven, Plus or Minus Two" (<http://www.musanim.com/miller1956/>), *Psychological Review* **63** (2): 81–97, doi: 10.1037/h0043158 (<http://dx.doi.org/10.1037/h0043158>), PMID 13310704 (<http://www.ncbi.nlm.nih.gov/pubmed/13310704>).
 - Möhring, Rolf H. (1990), "Graph problems related to gate matrix layout and PLA folding", in Tinhofer, G.; Mayr, E.; Noltemeier, H. et al., *Computational Graph Theory*, Computing Supplementum **7**, Springer-Verlag, pp. 17–51, ISBN 3-211-82177-5 .
 - Monien, B.; Sudborough, I. H. (1988), "Min cut is NP-complete for edge weighted trees", *Theoretical Computer Science* **58** (1–3): 209–229, doi: 10.1016/0304-3975(88)90028-X ([http://dx.doi.org/10.1016/0304-3975\(88\)90028-X](http://dx.doi.org/10.1016/0304-3975(88)90028-X)).
 - Ohtsuki, Tatsuo; Mori, Hajimu; Kuh, Ernest S.; Kashiwabara, Toshinobu; Fujisawa, Toshio (1979), "One-dimensional logic gate assignment and interval graphs", *IEEE Transactions on Circuits and Systems* **26** (9): 675–684, doi: 10.1109/TCS.1979.1084695 (<http://dx.doi.org/10.1109/TCS.1979.1084695>).
 - Peng, Sheng-Lung; Ho, Chin-Wen; Hsu, Tsan-sheng; Ko, Ming-Tat; Tang, Chuan Yi (1998), "A linear-time algorithm for constructing an optimal node-search strategy of a tree" (<http://www.springerlink.com/content/lamc6dynulxv7a8n/>), *Proc. 4th Int. Conf. Computing and Combinatorics (COCOON'98)*, Lecture Notes in Computer Science **1449**, Springer-Verlag, pp. 197–205.

- Proskurowski, Andrzej; Telle, Jan Arne (1999), "Classes of graphs with restricted interval models" (<http://www.emis.ams.org/journals/DMTCS/volumes/abstracts/pdfpapers/dm030404.pdf>), *Discrete Mathematics and Theoretical Computer Science* **3**: 167–176.
- Robertson, Neil; Seymour, Paul (1983), "Graph minors. I. Excluding a forest", *Journal of Combinatorial Theory, Series B* **35** (1): 39–61, doi: 10.1016/0095-8956(83)90079-5 ([http://dx.doi.org/10.1016/0095-8956\(83\)90079-5](http://dx.doi.org/10.1016/0095-8956(83)90079-5)).
- Robertson, Neil; Seymour, Paul (2003), "Graph minors. XVI. Excluding a non-planar graph", *Journal of Combinatorial Theory, Series B* **89** (1): 43–76, doi: 10.1016/S0095-8956(03)00042-X ([http://dx.doi.org/10.1016/S0095-8956\(03\)00042-X](http://dx.doi.org/10.1016/S0095-8956(03)00042-X)).
- Robertson, Neil; Seymour, Paul D. (2004), "Graph Minors. XX. Wagner's conjecture", *Journal of Combinatorial Theory, Series B* **92** (2): 325–357, doi: 10.1016/j.jctb.2004.08.001 (<http://dx.doi.org/10.1016/j.jctb.2004.08.001>).
- Scheffler, Petra (1990), "A linear algorithm for the pathwidth of trees", in Bodendiek, R.; Henn, R., *Topics in Combinatorics and Graph Theory*, Physica-Verlag, pp. 613–620.
- Scheffler, Petra (1992), "Optimal embedding of a tree into an interval graph in linear time", in Nešetřil, Jaroslav; Fiedler, Miroslav, *Fourth Czechoslovakian Symposium on Combinatorics, Graphs and Complexity*, Elsevier.
- Skodinis, Konstantin (2000), "Computing optimal linear layouts of trees in linear time", *Proc. 8th European Symposium on Algorithms (ESA 2000)*, Lecture Notes in Computer Science **1879**, Springer-Verlag, pp. 403–414, doi: 10.1007/3-540-45253-2_37 (http://dx.doi.org/10.1007/3-540-45253-2_37), ISBN 978-3-540-41004-1.
- Skodinis, Konstantin (2003), "Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time", *Journal of Algorithms* **47** (1): 40–59, doi: 10.1016/S0196-6774(02)00225-0 ([http://dx.doi.org/10.1016/S0196-6774\(02\)00225-0](http://dx.doi.org/10.1016/S0196-6774(02)00225-0)).
- Suchan, Karol; Todinca, Ioan (2007), "Pathwidth of circular-arc graphs", *Proc. 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2007)*, Lecture Notes in Computer Science **4769**, Springer-Verlag, pp. 258–269, doi: 10.1007/978-3-540-74839-7_25 (http://dx.doi.org/10.1007/978-3-540-74839-7_25).
- Suderman, Matthew (2004), "Pathwidth and layered drawings of trees" (<http://cgm.cs.mcgill.ca/~msuder/schools/mcgill/research/trees/SOCS-02-8.pdf>), *International Journal of Computational Geometry and Applications* **14** (3): 203–225, doi: 10.1142/S0218195904001433 (<http://dx.doi.org/10.1142/S0218195904001433>).
- Takahashi, Atsushi; Ueno, Shuichi; Kajitani, Yoji (1994), "Minimal acyclic forbidden minors for the family of graphs with bounded path-width", *Discrete Mathematics* **127** (1–3): 293–304, doi: 10.1016/0012-365X(94)90092-2 ([http://dx.doi.org/10.1016/0012-365X\(94\)90092-2](http://dx.doi.org/10.1016/0012-365X(94)90092-2)).

Planar separator theorem

In graph theory, the **planar separator theorem** is a form of isoperimetric inequality for planar graphs, that states that any planar graph can be split into smaller pieces by removing a small number of vertices. Specifically, the removal of $O(\sqrt{n})$ vertices from an n -vertex graph (where the O invokes big O notation) can partition the graph into disjoint subgraphs each of which has at most $2n/3$ vertices.

A weaker form of the separator theorem with $O(\sqrt{n} \log n)$ vertices in the separator instead of $O(\sqrt{n})$ was originally proven by Ungar (1951), and the form with the tight asymptotic bound on the separator size was first proven by Lipton & Tarjan (1979). Since their work, the separator theorem has been reproven in several different ways, the constant in the $O(\sqrt{n})$ term of the theorem has been improved, and it has been extended to certain classes of nonplanar graphs.

Repeated application of the separator theorem produces a separator hierarchy which may take the form of either a tree decomposition or a branch-decomposition of the graph. Separator hierarchies may be used to devise efficient divide and conquer algorithms for planar graphs, and dynamic programming on these hierarchies can be used to devise exponential time and fixed-parameter tractable algorithms for solving NP-hard optimization problems on these graphs. Separator hierarchies may also be used in nested dissection, an efficient variant of Gaussian elimination for solving sparse systems of linear equations arising from finite element methods.

Statement of the theorem

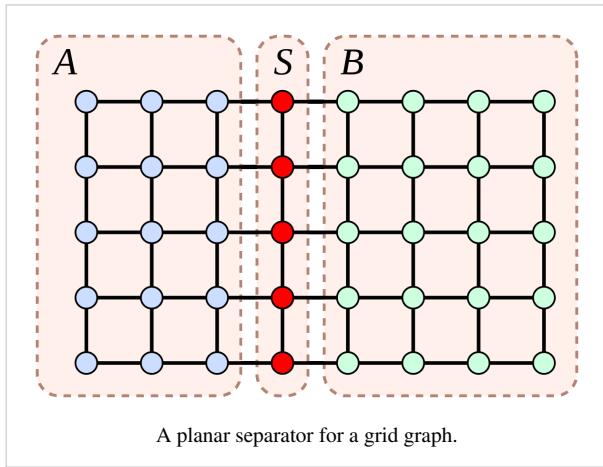
As it is usually stated, the separator theorem states that, in any n -vertex planar graph $G = (V, E)$, there exists a partition of the vertices of G into three sets A , S , and B , such that each of A and B has at most $2n/3$ vertices, S has $O(\sqrt{n})$ vertices, and there are no edges with one endpoint in A and one endpoint in B . It is not required that A or B form connected subgraphs of G . S is called the **separator** for this partition.

An equivalent formulation is that the edges of any n -vertex planar graph G may be subdivided into two edge-disjoint subgraphs G_1 and G_2 in such a way that both subgraphs have at least $n/3$ vertices and such that the intersection of the vertex sets of the two subgraphs has $O(\sqrt{n})$ vertices in it. Such a partition is known as a **separation**. If a separation is given, then the intersection of the vertex sets forms a separator, and the vertices that belong to one subgraph but not the other form the separated subsets of at most $2n/3$ vertices. In the other direction, if one is given a partition into three sets A , S , and B that meet the conditions of the planar separator theorem, then one may form a separation in which the edges with an endpoint in A belong to G_1 , the edges with an endpoint in B belong to G_2 , and the remaining edges (with both endpoints in S) are partitioned arbitrarily.

The constant $2/3$ in the statement of the separator theorem is arbitrary and may be replaced by any other number in the open interval $(1/2, 1)$ without changing the form of the theorem: a partition into more equal subsets may be obtained from a less-even partition by repeatedly splitting the larger sets in the uneven partition and regrouping the resulting connected components.

Example

Consider a grid graph with r rows and c columns; the number n of vertices equals rc . For instance, in the illustration, $r = 5$, $c = 8$, and $n = 40$. If r is odd, there is a single central row, and otherwise there are two rows equally close to the center; similarly, if c is odd, there is a single central column, and otherwise there are two columns equally close to the center. Choosing S to be any of these central rows or columns, and removing S from the graph, partitions the graph into two smaller connected subgraphs A and B , each of which has at most $n/2$ vertices. If $r \leq c$ (as in the illustration), then choosing a central column will give a separator S with $r \leq \sqrt{n}$ vertices, and similarly if $c \leq r$ then choosing a central row will give a separator with at most \sqrt{n} vertices. Thus, every grid graph has a separator S of size at most \sqrt{n} , the removal of which partitions it into two connected components, each of size at most $n/2$.^[1]



A planar separator for a grid graph.

The planar separator theorem states that a similar partition can be constructed in any planar graph. The case of arbitrary planar graphs differs from the case of grid graphs in that the separator has size $O(\sqrt{n})$ but may be larger than \sqrt{n} , the bound on the size of the two subsets A and B (in the most common versions of the theorem) is $2n/3$ rather than $n/2$, and the two subsets A and B need not themselves form connected subgraphs.

Constructions

Breadth-first layering

Lipton & Tarjan (1979) augment the given planar graph by additional edges, if necessary, so that it becomes maximal planar (every face in a planar embedding is a triangle). They then perform a breadth-first search, rooted at an arbitrary vertex v , and partition the vertices into levels by their distance from v . If l_1 is the median level (the level such that the numbers of vertices at higher and lower levels are both at most $n/2$) then there must be levels l_0 and l_2 that are $O(\sqrt{n})$ steps above and below l_1 respectively and that contain $O(\sqrt{n})$ vertices, respectively, for otherwise there would be more than n vertices in the levels near l_1 . They show that there must be a separator S formed by the union of l_0 and l_2 , the endpoints e of an edge of G that does not belong to the breadth-first search tree and that lies between the two levels, and the vertices on the two breadth-first search tree paths from e back up to level l_0 . The size of the separator S constructed in this way is at most $\sqrt{8\sqrt{n}}$, or approximately $2.83\sqrt{n}$. The vertices of the separator and the two disjoint subgraphs can be found in linear time.

This proof of the separator theorem applies as well to weighted planar graphs, in which each vertex has a non-negative cost. The graph may be partitioned into three sets A , S , and B such that A and B each have at most $2/3$ of the total cost and S has $O(\sqrt{n})$ vertices, with no edges from A to B . By analysing a similar separator construction more carefully, Djidjev (1982) shows that the bound on the size of S can be reduced to $\sqrt{6\sqrt{n}}$, or approximately $2.45\sqrt{n}$.

Holzer et al. (2009) suggest a simplified version of this approach: they augment the graph to be maximal planar and construct a breadth first search tree as before. Then, for each edge e that is not part of the tree, they form a cycle by combining e with the tree path that connects its endpoints. They then use as a separator the vertices of one of these cycles. Although this approach cannot be guaranteed to find a small separator for planar graphs of high diameter, their experiments indicate that it outperforms the Lipton–Tarjan and Djidjev breadth-first layering methods on many types of planar graph.

Simple cycle separators

For a graph that is already maximal planar it is possible to show a stronger construction of a **simple cycle separator**, a cycle of small length such that the inside and the outside of the cycle (in the unique planar embedding of the graph) each have at most $2n/3$ vertices. Miller (1986) proves this (with a separator size of $\sqrt{8\sqrt{n}}$) by using the Lipton–Tarjan technique for a modified version of breadth first search in which the levels of the search form simple cycles.

Alon, Seymour & Thomas (1994) prove the existence of simple cycle separators more directly: they let C be a cycle of at most $\sqrt{8\sqrt{n}}$ vertices, with at most $2n/3$ vertices outside C , that forms as even a partition of inside and outside as possible, and they show that these assumptions force C to be a separator. For otherwise, the distances within C must equal the distances in the disk enclosed by C (a shorter path through the interior of the disk would form part of the boundary of a better cycle). Additionally, C must have length exactly $\sqrt{8\sqrt{n}}$, as otherwise it could be improved by replacing one of its edges by the other two sides of a triangle. If the vertices in C are numbered (in clockwise order) from 1 to $\sqrt{8\sqrt{n}}$, and vertex i is matched up with vertex $\sqrt{8\sqrt{n}} - i + 1$, then these matched pairs can be connected by vertex-disjoint paths within the disk, by a form of Menger's theorem for planar graphs. However, the total length of these paths would necessarily exceed n , a contradiction. With some additional work they show by a similar method that there exists a simple cycle separator of size at most $(3/\sqrt{2})\sqrt{n}$, approximately $2.12\sqrt{n}$.

Djidjev & Venkatesan (1997) further improved the constant factor in the simple cycle separator theorem to $1.97\sqrt{n}$. Their method can also find simple cycle separators for graphs with non-negative vertex weights, with separator size at most $2\sqrt{n}$, and can generate separators with smaller size at the expense of a more uneven partition of the graph. In 2-connected planar graphs that are not maximal, there exist simple cycle separators with size proportional to the Euclidean norm of the vector of face lengths that can be found in near-linear time.

Circle separators

According to the Koebe–Andreev–Thurston circle-packing theorem, any planar graph may be represented by a packing of circular disks in the plane with disjoint interiors, such that two vertices in the graph are adjacent if and only if the corresponding pair of disks are mutually tangent. As Miller et al. (1997) show, for such a packing, there exists a circle that has at most $3n/4$ disks touching or inside it, at most $3n/4$ disks touching or outside it, and that crosses $O(\sqrt{n})$ disks.

To prove this, Miller et al. use stereographic projection to map the packing onto the surface of a unit sphere in three dimensions. By choosing the projection carefully, the center of the sphere can be made into a centerpoint of the disk centers on its surface, so that any plane through the center of the sphere partitions it into two halfspaces that each contain or cross at most $3n/4$ of the disks. If a plane through the center is chosen uniformly at random, a disk will be crossed with probability proportional to its radius. Therefore, the expected number of disks that are crossed is proportional to the sum of the radii of the disks. However, the sum of the squares of the radii is proportional to the total area of the disks, which is at most the total surface area of the unit sphere, a constant. An argument involving Jensen's inequality shows that, when the sum of squares of n non-negative real numbers is bounded by a constant, the sum of the numbers themselves is $O(\sqrt{n})$. Therefore, the expected number of disks crossed by a random plane is $O(\sqrt{n})$ and there exists a plane that crosses at most that many disks. This plane intersects the sphere in a great circle, which projects back down to a circle in the plane with the desired properties. The $O(\sqrt{n})$ disks crossed by this circle correspond to the vertices of a planar graph separator that separates the vertices whose disks are inside the circle from the vertices whose disks are outside the circle, with at most $3n/4$ vertices in each of these two subsets.

This method leads to a randomized algorithm that finds such a separator in linear time, and a less-practical deterministic algorithm with the same linear time bound. By analyzing this algorithm carefully using known bounds on the packing density of circle packings, it can be shown to find separators of size at most

$$\sqrt{\frac{2\pi}{\sqrt{3}}} \left(\frac{1 + \sqrt{3}}{2\sqrt{2}} + o(1) \right) \sqrt{n} \approx 1.84\sqrt{n}.$$

Although this improved separator size bound comes at the expense of a more-uneven partition of the graph, Spielman & Teng (1996) argue that it provides an improved constant factor in the time bounds for nested dissection compared to the separators of Alon, Seymour & Thomas (1990). The size of the separators it produces can be further improved, in practice, by using a nonuniform distribution for the random cutting planes.

The stereographic projection in the Miller et al. argument can be avoided by considering the smallest circle containing a constant fraction of the centers of the disks and then expanding it by a constant picked uniformly in the range [1,2]. It is easy to argue, as in Miller et al., that the disks intersecting the expanded circle form a valid separator, and that, in expectation, the separator is of the right size. The resulting constants are somewhat worse.

Spectral partitioning

Spectral clustering methods, in which the vertices of a graph are grouped by the coordinates of the eigenvectors of matrices derived from the graph, have long been used as a heuristic for graph partitioning problems for nonplanar graphs.^[2] As Spielman & Teng (2007) show, spectral clustering can also be used to derive an alternative proof for a weakened form of the planar separator theorem that applies to planar graphs with bounded degree. In their method, the vertices of a given planar graph are sorted by the second coordinates of the eigenvectors of the Laplacian matrix of the graph, and this sorted order is partitioned at the point that minimizes the ratio of the number of edges cut by the partition to the number of vertices on the smaller side of the partition. As they show, every planar graph of bounded degree has a partition of this type in which the ratio is $O(1/\sqrt{n})$. Although this partition may not be balanced, repeating the partition within the larger of the two sides and taking the union of the cuts formed at each repetition will eventually lead to a balanced partition with $O(\sqrt{n})$ edges. The endpoints of these edges form a separator of size $O(\sqrt{n})$.

Edge separators

A variation of the planar separator theorem involves **edge separators**, small sets of edges forming a cut between two subsets A and B of the vertices of the graph. The two sets A and B must each have size at most a constant fraction of the number n of vertices of the graph (conventionally, both sets have size at most $2n/3$), and each vertex of the graph belongs to exactly one of A and B . The separator consists of the edges that have one endpoint in A and one endpoint in B . Bounds on the size of an edge separator involve the degree of the vertices as well as the number of vertices in the graph: the planar graphs in which one vertex has degree $n - 1$, including the wheel graphs and star graphs, have no edge separator with a sublinear number of edges, because any edge separator would have to include all the edges connecting the high degree vertex to the vertices on the other side of the cut. However, every planar graph with maximum degree Δ has an edge separator of size $O(\sqrt{(\Delta n)})$.^[3]

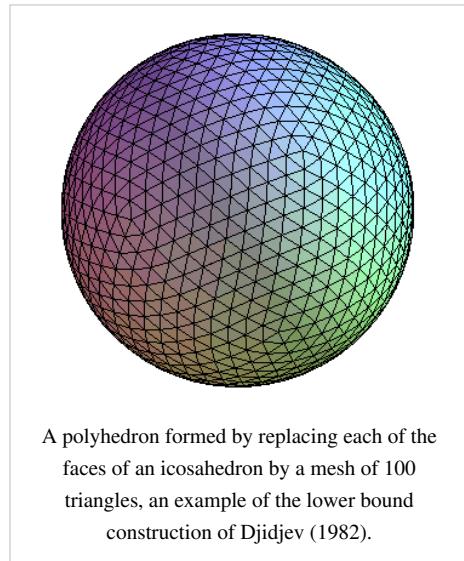
A simple cycle separator in the dual graph of a planar graph forms an edge separator in the original graph.^[4] Applying the simple cycle separator theorem of Gazit & Miller (1990) to the dual graph of a given planar graph strengthens the $O(\sqrt{(\Delta n)})$ bound for the size of an edge separator by showing that every planar graph has an edge separator whose size is proportional to the Euclidean norm of its vector of vertex degrees.

Papadimitriou & Sideri (1996) describe a polynomial time algorithm for finding the smallest edge separator that partitions a graph G into two subgraphs of equal size, when G is an induced subgraph of a grid graph with no holes or with a constant number of holes. However, they conjecture that the problem is NP-complete for arbitrary planar graphs, and they show that the complexity of the problem is the same for grid graphs with arbitrarily many holes as it is for arbitrary planar graphs.

Lower bounds

In a $\sqrt{n} \times \sqrt{n}$ grid graph, a set S of $s < \sqrt{n}$ points can enclose a subset of at most $s(s - 1)/2$ grid points, where the maximum is achieved by arranging S in a diagonal line near a corner of the grid. Therefore, in order to form a separator that separates at least $n/3$ of the points from the remaining grid, s needs to be at least $\sqrt{(2n/3)}$, approximately $0.82\sqrt{n}$.

There exist n -vertex planar graphs (for arbitrarily large values of n) such that, for every separator S that partitions the remaining graph into subgraphs of at most $2n/3$ vertices, S has at least $\sqrt{(4\pi\sqrt{3})\sqrt{n}}$ vertices, approximately $1.56\sqrt{n}$. The construction involves approximating a sphere by a convex polyhedron, replacing each of the faces of the polyhedron by a triangular mesh, and applying isoperimetric theorems for the surface of the sphere.



Separator hierarchies

Separators may be combined into a **separator hierarchy** of a planar graph, a recursive decomposition into smaller graphs. A separator hierarchy may be represented by a binary tree in which the root node represents the given graph itself, and the two children of the root are the roots of recursively constructed separator hierarchies for the induced subgraphs formed from the two subsets A and B of a separator.

A separator hierarchy of this type forms the basis for a tree decomposition of the given graph, in which the set of vertices associated with each tree node is the union of the separators on the path from that node to the root of the tree. Since the sizes of the graphs go down by a constant factor at each level of the tree, the upper bounds on the sizes of the separators also go down by a constant factor at each level, so the sizes of the separators on these paths add in a geometric series to $O(\sqrt{n})$. That is, a separator formed in this way has width $O(\sqrt{n})$, and can be used to show that every planar graph has treewidth $O(\sqrt{n})$.

Constructing a separator hierarchy directly, by traversing the binary tree top down and applying a linear-time planar separator algorithm to each of the induced subgraphs associated with each node of the binary tree, would take a total of $O(n \log n)$ time. However, it is possible to construct an entire separator hierarchy in linear time, by using the Lipton–Tarjan breadth-first layering approach and by using appropriate data structures to perform each partition step in sublinear time.

If one forms a related type of hierarchy based on separations instead of separators, in which the two children of the root node are roots of recursively constructed hierarchies for the two subgraphs G_1 and G_2 of a separation of the given graph, then the overall structure forms a branch-decomposition instead of a tree decomposition. The width of any separation in this decomposition is, again, bounded by the sum of the sizes of the separators on a path from any node to the root of the hierarchy, so any branch-decomposition formed in this way has width $O(\sqrt{n})$ and any planar graph has branchwidth $O(\sqrt{n})$. Although many other related graph partitioning problems are NP-complete, even for planar graphs, it is possible to find a minimum-width branch-decomposition of a planar graph in polynomial time.

By applying methods of Alon, Seymour & Thomas (1994) more directly in the construction of branch-decompositions, Fomin & Thilikos (2006a) show that every planar graph has branchwidth at most $2.12\sqrt{n}$, with the same constant as the one in the simple cycle separator theorem of Alon et al. Since the treewidth of any graph is at most $3/2$ its branchwidth, this also shows that planar graphs have treewidth at most $3.18\sqrt{n}$.

Other classes of graphs

Some sparse graphs do not have separators of sublinear size: in an expander graph, deleting up to a constant fraction of the vertices still leaves only one connected component.^[5]

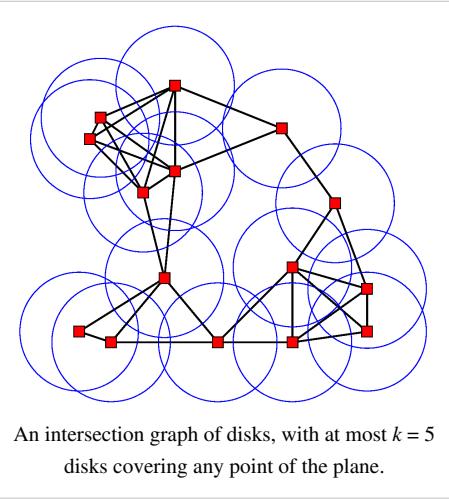
Possibly the earliest known separator theorem is a result of Jordan (1869) that any tree can be partitioned into subtrees of at most $2n/3$ vertices each by the removal of a single vertex. In particular, the vertex that minimizes the maximum component size has this property, for if it did not then its neighbor in the unique large subtree would form an even better partition. By applying the same technique to a tree decomposition of an arbitrary graph, it is possible to show that any graph has a separator of size at most equal to its treewidth.

If a graph G is not planar, but can be embedded on a surface of genus g , then it has a separator with $O((gn)^{1/2})$ vertices. Gilbert, Hutchinson & Tarjan (1984) prove this by using a similar approach to that of Lipton & Tarjan (1979). They group the vertices of the graph into breadth-first levels and find two levels the removal of which leaves at most one large component consisting of a small number of levels. This remaining component can be made planar by removing a number of breadth-first paths proportional to the genus, after which the Lipton–Tarjan method can be applied to the remaining planar graph. The result follows from a careful balancing of the size of the removed two levels against the number of levels between them. If the graph embedding is given as part of the input, its separator can be found in linear time. Graphs of genus g also have edge separators of size $O((g\Delta n)^{1/2})$.

Graphs of bounded genus form an example of a family of graphs closed under the operation of taking minors, and separator theorems also apply to arbitrary minor-closed graph families. In particular, if a graph family has a forbidden minor with h vertices, then it has a separator with $O(h\sqrt{n})$ vertices, and such a separator can be found in time $O(n^{1+\varepsilon})$ for any $\varepsilon > 0$.^[6]

The circle separator method of Miller et al. (1997) generalizes to the intersection graphs of any system of d -dimensional balls with the property that any point in space is covered by at most some constant number k of balls, to k -nearest-neighbor graphs in d dimensions, and to the graphs arising from finite element meshes. The sphere separators constructed in this way partition the input graph into subgraphs of at most $n(d+1)/(d+2)$ vertices. The size of the separators for k -ply ball intersection graphs and for k -nearest-neighbor graphs is $O(k^{1/d}n^{1-1/d})$.

Applications



Divide and conquer algorithms

Separator decompositions can be of use in designing efficient divide and conquer algorithms for solving problems on planar graphs. As an example, one problem that can be solved in this way is to find the shortest cycle in a weighted planar digraph. This may be solved by the following steps:

- Partition the given graph G into three subsets S, A, B according to the planar separator theorem
- Recursively search for the shortest cycles in A and B
- Use Dijkstra's algorithm to find, for each s in S , the shortest cycle through s in G .
- Return the shortest of the cycles found by the above steps.

The time for the two recursive calls to A and B in this algorithm is dominated by the time to perform the $O(\sqrt{n})$ calls to Dijkstra's algorithm, so this algorithm finds the shortest cycle in $O(n^{3/2} \log n)$ time.

A faster algorithm for the same shortest cycle problem, running in time $O(n \log^3 n)$, was given by Wulff-Nilsen (2009). His algorithm uses the same separator-based divide and conquer structure, but uses simple cycle separators rather than arbitrary separators, so that the vertices of S belong to a single face of the graphs inside and outside the cycle separator. He then replaces the $O(\sqrt{n})$ separate calls to Dijkstra's algorithm with more sophisticated algorithms to find shortest paths from all vertices on a single face of a planar graph and to combine the distances from the two subgraphs. For weighted but undirected planar graphs, the shortest cycle is equivalent to the minimum cut in the dual graph and can be found in $O(n \log \log n)$ time, and the shortest cycle in an unweighted undirected planar graph (its girth) may be found in time $O(n)$. (However, the faster algorithm for unweighted graphs is not based on the separator theorem.)

Frederickson proposed another faster algorithm for single source shortest paths by implementing separator theorem in planar graphs in 1986.^[7] This is an improvement of Dijkstra's algorithm with iterative search on a carefully selected subset of the vertices. This version takes $O(n \sqrt{(\log n)})$ time in an n -vertex graph. Separators are used to find a division of a graph, that is, a partition of the edge-set into two or more subsets, called regions. A node is said to be contained in a region if some edge of the region is incident to the node. A node contained in more than one region is called a boundary node of the regions containing it. The method uses the notion of a r -division of an n -node graph that is a graph division into $O(n/r)$ regions, each containing $O(r)$ nodes including $O(\sqrt{r})$ boundary nodes. Frederickson showed that an r -division can be found in $O(n \log n)$ time by recursive application of separator theorem.

The sketch of his algorithm to solve the problem is as follows.

1. Preprocessing Phase: Partition the graph into carefully selected subsets of vertices and determine the shortest paths between all pairs of vertices in these subsets, where intermediate vertices on this path are not in the subset. This phase requires a planar graph G_0 to be transformed into G with no vertex having degree greater than 3. From a corollary of Euler's formula, the number of vertices in the resulting graph will be $n \leq 6n_0 - 12$, where n_0 is the number of vertices in G_0 . This phase also ensures the following properties of a suitable r -division. A suitable r -division of a planar graph is an r -division such that,

- each boundary vertex is contained in at most three regions, and
- any region that is not connected consists of connected components, all of which share boundary vertices with exactly the same set of either one or two connected regions.

2. Search Phase:

- Main Thrust: Find Shortest distances from the source to each vertex in the subset. When a vertex v in the subset is closed, $d(w)$ must be updated for all vertices w in the subset such that a path exists from v to w .
- Mop-up: Determine shortest distances to every remaining vertex.

Henzinger et. al. extended Frederickson's r -division technique for the single source shortest path algorithm in planar graphs for nonnegative edge-lengths and proposed a linear time algorithm.^[8] Their method generalizes Frederickson's notion of graph-divisions such that now an (r,s) -division of an n -node graph be a division into $O(n/r)$ regions, each containing $r^{O(1)}$ nodes, each having at most s boundary nodes. If an (r, s) -division is repeatedly divided into smaller regions, that is called get a recursive division. This algorithm uses approximately $\log^* n$ levels of divisions. The recursive division is represented by a rooted tree whose leaves are labeled by distinct edge of G . The root of the tree represents the region consisting of full- G , the children of the root represent the subregions into which that region is divided and so on. Each leaf (atomic region) represents a region containing exactly one edge.

Nested dissection is a separator based divide and conquer variation of Gaussian elimination for solving sparse symmetric systems of linear equations with a planar graph structure, such as the ones arising from the finite element method. It involves finding a separator for the graph describing the system of equations, recursively eliminating the variables in the two subproblems separated from each other by the separator, and then eliminating the variables in the separator. The fill-in of this method (the number of nonzero coefficients of the resulting Cholesky decomposition of the matrix) is $O(n \log n)$,^[9] allowing this method to be competitive with iterative methods for the same problems.

Klein, Mozes and Weimann [10] gave an $O(n \log^2 n)$ -time, linear-space algorithm to find the shortest path distances from s to all nodes for a directed planar graph with positive and negative arc-lengths containing no negative cycles. Their algorithm uses planar graph separators to find a Jordan curve C that passes through $O(\sqrt{n})$ nodes (and no arcs) such that between $n/3$ and $2n/3$ nodes are enclosed by C . Nodes through which C passes are boundary nodes. The original graph G is separated into two subgraphs G_0 and G_1 by cutting the planar embedding along C and duplicating the boundary nodes. For $i = 0$ and 1 , in G_i the boundary nodes lie on the boundary of a single face F_i .

The overview of their approach is given below.

- Recursive call: The first stage recursively computes the distances from r within G_i for $i = 0, 1$.
- Intra-part boundary-distances: For each graph G_i compute all distances in G_i between boundary nodes. This takes $O(n \log n)$ time.
- Single-source inter-part boundary distances: A shortest path in G passes back and forth between G_0 and G_1 to compute the distances in G from r to all the boundary nodes. Alternating iterations use the all-boundary-distances in $\$G_0$ and $\$G_1$. The number of iterations is $O(\sqrt{n})$, so the overall time for this stage is $O(n \alpha(n))$ where $\alpha(n)$ is the inverse Ackerman function.
- Single-source inter-part distances: The distances computed in the previous stages are used, together with a Dijkstra computation within a modified version of each G_i , to compute the distances in G from r to all the nodes. This stage takes $O(n \log n)$ time.
- Rerooting single-source distances: The distances from r in G are transformed into nonnegative lengths, and again Dijkstra's algorithm is used to compute distances from s . This stage requires $O(n \log n)$ time.

An important part of this algorithm is the use of Price Functions and Reduced Lengths. For a directed graph G with arc-lengths $\iota(\cdot)$, a price function is a function φ from the nodes of G to the real numbers. For an arc uv , the reduced length with respect to φ is $\iota\varphi(uv) = \iota(uv) + \varphi(u) - \varphi(v)$. A feasible price function is a price function that induces nonnegative reduced lengths on all arcs of G . It is useful in transforming a shortest-path problem involving positive and negative lengths into one involving only nonnegative lengths, which can then be solved using Dijkstra's algorithm.

The separator based divide and conquer paradigm has also been used to design data structures for dynamic graph algorithms^[11] and point location, algorithms for polygon triangulation, shortest paths,^[12] and the construction of nearest neighbor graphs, and approximation algorithms for the maximum independent set of a planar graph.

Exact solution of NP-hard optimization problems

By using dynamic programming on a tree decomposition or branch-decomposition of a planar graph, many NP-hard optimization problems may be solved in time exponential in \sqrt{n} or $\sqrt{n} \log n$. For instance, bounds of this form are known for finding maximum independent sets, Steiner trees, and Hamiltonian cycles, and for solving the travelling salesman problem on planar graphs.^[13] Similar methods involving separator theorems for geometric graphs may be used to solve Euclidean travelling salesman problem and Steiner tree construction problems in time bounds of the same form.

For parameterized problems that admit a kernelization that preserves planarity and reduces the input graph to a kernel of size linear in the input parameter, this approach can be used to design fixed-parameter tractable algorithms the running time of which depends polynomially on the size of the input graph and exponentially on \sqrt{k} , where k is the parameter of the algorithm. For instance, time bounds of this form are known for finding vertex covers and dominating sets of size k .^[14]

Approximation algorithms

Lipton & Tarjan (1980) observed that the separator theorem may be used to obtain polynomial time approximation schemes for NP-hard optimization problems on planar graphs such as finding the maximum independent set. Specifically, by truncating a separator hierarchy at an appropriate level, one may find a separator of size $O(n/\sqrt{\log n})$ the removal of which partitions the graph into subgraphs of size $c \log n$, for any constant c . By the four-color theorem, there exists an independent set of size at least $n/4$, so the removed nodes form a negligible fraction of the maximum independent set, and the maximum independent sets in the remaining subgraphs can be found independently in time exponential in their size. By combining this approach with later linear-time methods for separator hierarchy construction and with table lookup to share the computation of independent sets between isomorphic subgraphs, it can be made to construct independent sets of size within a factor of $1 - O(1/\sqrt{\log n})$ of optimal, in linear time. However, for approximation ratios even closer to 1 than this factor, a later approach of Baker (1994) (based on tree-decomposition but not on planar separators) provides better tradeoffs of time versus approximation quality.

Similar separator-based approximation schemes have also been used to approximate other hard problems such as vertex cover.^[15] Arora et al. (1998) use separators in a different way to approximate the travelling salesman problem for the shortest path metric on weighted planar graphs; their algorithm uses dynamic programming to find the shortest tour that, at each level of a separator hierarchy, crosses the separator a bounded number of times, and they show that as the crossing bound increases the tours constructed in this way have lengths that approximate the optimal tour.

Graph compression

Separators have been used as part of data compression algorithms for representing planar graphs and other separable graphs using a small number of bits. The basic principle of these algorithms is to choose a number k and repeatedly subdivide the given planar graph using separators into $O(n/k)$ subgraphs of size at most k , with $O(n/\sqrt{k})$ vertices in the separators. With an appropriate choice of k (at most proportional to the logarithm of n) the number of non-isomorphic k -vertex planar subgraphs is significantly less than the number of subgraphs in the decomposition, so the graph can be compressed by constructing a table of all the possible non-isomorphic subgraphs and representing each subgraph in the separator decomposition by its index into the table. The remainder of the graph, formed by the separator vertices, may be represented explicitly or by using a recursive version of the same data structure. Using this method, planar graphs and many more restricted families of planar graphs may be encoded using a number of bits that is information-theoretically optimal: if there are P_n n -vertex graphs in the family of graphs to be represented, then an individual graph in the family can be represented using only $(1 + o(n))\log_2 P_n$ bits. It is also possible to construct representations of this type in which one may test adjacency between vertices, determine the degree of a vertex, and list neighbors of vertices in constant time per query, by augmenting the table of subgraphs with additional tabular information representing the answers to the queries.

Universal graphs

A universal graph for a family F of graphs is a graph that contains every member of F as a subgraphs. Separators can be used to show that the n -vertex planar graphs have universal graphs with n vertices and $O(n^{3/2})$ edges.^[16]

The construction involves a strengthened form of the separator theorem in which the size of the three subsets of vertices in the separator does not depend on the graph structure: there exists a number c , the magnitude of which at most a constant times \sqrt{n} , such that the vertices of every n -vertex planar graph can be separated into subsets A , S , and B , with no edges from A to B , with $|S| = c$, and with $|A| = |B| = (n - c)/2$. This may be shown by using the usual form of the separator theorem repeatedly to partition the graph until all the components of the partition can be arranged into two subsets of fewer than $n/2$ vertices, and then moving vertices from these subsets into the separator as necessary until it has the given size.

Once a separator theorem of this type is shown, it can be used to produce a separator hierarchy for n -vertex planar graphs that again does not depend on the graph structure: the tree-decomposition formed from this hierarchy has width $O(\sqrt{n})$ and can be used for any planar graph. The set of all pairs of vertices in this tree-decomposition that both belong to a common node of the tree-decomposition forms a trivially perfect graph with $O(n^{3/2})$ vertices that contains every n -vertex planar graph as a subgraph. A similar construction shows that bounded-degree planar graphs have universal graphs with $O(n \log n)$ edges, where the constant hidden in the O notation depends on the degree bound. Any universal graph for planar graphs (or even for trees of unbounded degree) must have $\Omega(n \log n)$ edges, but it remains unknown whether this lower bound or the $O(n^{3/2})$ upper bound is tight for universal graphs for arbitrary planar graphs.

Notes

- [1] . Instead of using a row or column of a grid graph, George partitions the graph into four pieces by using the union of a row and a column as a separator.
- [2] ; .
- [3] proved this result for 2-connected planar graphs, and extended it to all planar graphs.
- [4] ; .
- [5] ; .
- [6] . For earlier work on separators in minor-closed families see , , and .
- [7] Greg n. Frederickson, Fast algorithms for shortest paths in planar graphs, with applications, SIAM J. Computing, pp. 1004-1022, 1987.
- [8] Monika R. Henzinger , Philip Klein , Satish Rao , Sairam Subramanian, \textit{Faster shortest-path algorithms for planar graphs}, Journal of Computer and System Science, Vol. 55, Issue 1, August 1997.
- [9] ; .
- [10] Philip N. Klein, Shay Mozes and Oren Weimann, Shortest Paths in Directed Planar Graphs with Negative Lengths: a Linear-Space $O(n \log^2 n)$ -Time Algorithm}, Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, 2009.
- [11] ; .
- [12] ; .
- [13] ; ; ; .
- [14] ; .
- [15] ; .
- [16] ; ; .

References

- Alber, Jochen; Fernau, Henning; Niedermeier, Rolf (2003), "Graph separators: A parameterized view", *Journal of Computer and System Sciences* **67** (4): 808–832, doi: 10.1016/S0022-0000(03)00072-2 ([http://dx.doi.org/10.1016/S0022-0000\(03\)00072-2](http://dx.doi.org/10.1016/S0022-0000(03)00072-2)).
- Alon, Noga; Seymour, Paul; Thomas, Robin (1990), "A separator theorem for nonplanar graphs", *J. Amer. Math. Soc.* **3** (4): 801–808, doi: 10.1090/S0894-0347-1990-1065053-0 (<http://dx.doi.org/10.1090/S0894-0347-1990-1065053-0>).
- Alon, Noga; Seymour, Paul; Thomas, Robin (1994), "Planar separators", *SIAM Journal on Discrete Mathematics* **7** (2): 184–193, doi: 10.1137/S0895480191198768 (<http://dx.doi.org/10.1137/S0895480191198768>).
- Arora, Sanjeev; Grigni, Michelangelo; Karger, David; Klein, Philip; Woloszyn, Andrzej (1998), "A polynomial-time approximation scheme for weighted planar graph TSP" (<http://portal.acm.org/citation.cfm?id=314613.314632>), *Proc. 9th ACM-SIAM Symposium on Discrete algorithms (SODA '98)*, pp. 33–41.
- Babai, L.; Chung, F. R. K.; Erdős, P.; Graham, R. L.; Spencer, J. H. (1982), "On graphs which contain all sparse graphs" (http://renyi.hu/~p_erdos/1982-12.pdf), in Rosa, Alexander; Sabidussi, Gert; Turgeon, Jean, *Theory and practice of combinatorics: a collection of articles honoring Anton Kotzig on the occasion of his sixtieth birthday*, Annals of Discrete Mathematics **12**, pp. 21–26.
- Baker, Brenda S. (1994), "Approximation algorithms for NP-complete problems on planar graphs", *Journal of the ACM* **41** (1): 153–180, doi: 10.1145/174644.174650 (<http://dx.doi.org/10.1145/174644.174650>).

- Bar-Yehuda, R.; Even, S. (1982), "On approximating a vertex cover for planar graphs", *Proc. 14th ACM Symposium on Theory of Computing (STOC '82)*, pp. 303–309, doi: 10.1145/800070.802205 (<http://dx.doi.org/10.1145/800070.802205>), ISBN 0-89791-070-2.
- Bern, Marshall (1990), "Faster exact algorithms for Steiner trees in planar networks", *Networks* **20** (1): 109–120, doi: 10.1002/net.3230200110 (<http://dx.doi.org/10.1002/net.3230200110>).
- Bhatt, Sandeep N.; Chung, Fan R. K.; Leighton, F. T.; Rosenberg, Arnold L. (1989), "Universal graphs for bounded-degree trees and planar graphs" (<http://www.math.ucsd.edu/~fan/mypaps/fanpap/116universal.pdf>), *SIAM Journal on Discrete Mathematics* **2** (2): 145, doi: 10.1137/0402014 (<http://dx.doi.org/10.1137/0402014>).
- Blandford, Daniel K.; Blelloch, Guy E.; Kash, Ian A. (2003), "Compact representations of separable graphs" (<http://www.cs.cornell.edu/~kash/papers/BBK03.pdf>), *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*, pp. 679–688.
- Blelloch, Guy E.; Farzan, Arash (2010), "Succinct representations of separable graphs", in Amir, Amihood; Parida, Laxmi, *Proc. 21st Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science **6129**, Springer-Verlag, pp. 138–150, doi: 10.1007/978-3-642-13509-5_13 (http://dx.doi.org/10.1007/978-3-642-13509-5_13), ISBN 978-3-642-13508-8.
- Chalermsook, Parinya; Fakcharoenphol, Jittat; Nanongkai, Danupon (2004), "A deterministic near-linear time algorithm for finding minimum cuts in planar graphs" (<http://chalermsook.googlepages.com/mincut.ps>), *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, pp. 828–829.
- Chang, Hsien-Chih; Lu, Hsueh-I, "Computing the girth of a planar graph in linear time", *SIAM Journal on Computing*: To appear, arXiv: 1104.4892 (<http://arxiv.org/abs/1104.4892>).
- Chiba, Norishige; Nishizeki, Takao; Saito, Nobuji (1981), "Applications of the Lipton and Tarjan planar separator theorem" (<http://www.kurims.kyoto-u.ac.jp/~kyodo/kokyuroku/contents/pdf/0427-11.pdf>), *J. Inform. Process* **4** (4): 203–207.
- Chung, Fan R. K. (1990), "Separator theorems and their applications" (<http://www.math.ucsd.edu/~fan/mypaps/fanpap/117separatorthms.pdf>), in Korte, Bernhard; Lovász, László; Prömel, Hans Jürgen et al., *Paths, Flows, and VLSI-Layout*, Algorithms and Combinatorics **9**, Springer-Verlag, pp. 17–34, ISBN 978-0-387-52685-0 .
- Deineko, Vladimir G.; Klinz, Bettina; Woeginger, Gerhard J. (2006), "Exact algorithms for the Hamiltonian cycle problem in planar graphs", *Operations Research Letters* **34** (3): 269–274, doi: 10.1016/j.orl.2005.04.013 (<http://dx.doi.org/10.1016/j.orl.2005.04.013>).
- Diks, K.; Djidjev, H. N.; Sýkora, O.; Vrt'o, I. (1993), "Edge separators of planar and outerplanar graphs with applications", *Journal of Algorithms* **14** (2): 258–279, doi: 10.1006/jagm.1993.1013 (<http://dx.doi.org/10.1006/jagm.1993.1013>).
- Djidjev, H. N. (1982), "On the problem of partitioning planar graphs", *SIAM Journal on Algebraic and Discrete Methods* **3** (2): 229–240, doi: 10.1137/0603022 (<http://dx.doi.org/10.1137/0603022>).
- Djidjev, Hristo N.; Venkatesan, Shankar M. (1997), "Reduced constants for simple cycle graph separation", *Acta Informatica* **34** (3): 231–243, doi: 10.1007/s002360050082 (<http://dx.doi.org/10.1007/s002360050082>).
- Donath, W. E.; Hoffman, A. J. (1972), "Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices", *IBM Techn. Disclosure Bull.* **15**: 938–944. As cited by Spielman & Teng (2007).
- Dorn, Frederic; Penninkx, Eelko; Bodlaender, Hans L.; Fomin, Fedor V. (2005), "Efficient exact algorithms on planar graphs: exploiting sphere cut branch decompositions", *Proc. 13th European Symposium on Algorithms (ESA '05)*, Lecture Notes in Computer Science **3669**, Springer-Verlag, pp. 95–106, doi: 10.1007/11561071_11 (http://dx.doi.org/10.1007/11561071_11), ISBN 978-3-540-29118-3.
- Eppstein, David; Galil, Zvi; Italiano, Giuseppe F.; Spencer, Thomas H. (1996), "Separator based sparsification. I. Planarity testing and minimum spanning trees", *Journal of Computer and System Sciences* **52** (1): 3–27, doi:

- 10.1006/jcss.1996.0002 (<http://dx.doi.org/10.1006/jcss.1996.0002>).
- Eppstein, David; Galil, Zvi; Italiano, Giuseppe F.; Spencer, Thomas H. (1998), "Separator-based sparsification. II. Edge and vertex connectivity", *SIAM Journal on Computing* **28**: 341, doi: 10.1137/S0097539794269072 (<http://dx.doi.org/10.1137/S0097539794269072>).
 - Eppstein, David; Miller, Gary L.; Teng, Shang-Hua (1995), "A deterministic linear time algorithm for geometric separators and its applications" (<http://www.ics.uci.edu/~eppstein/pubs/EppMilTen-FI-95.ps.gz>), *Fundamenta Informaticae* **22** (4): 309–331.
 - Erdős, Paul; Graham, Ronald; Szemerédi, Endre (1976), "On sparse graphs with dense long paths" (http://www.renyi.hu/~p_erdos/1976-26.pdf), *Computers and mathematics with applications*, Oxford: Pergamon, pp. 365–369.
 - Fiedler, Miroslav (1973), "Algebraic connectivity of graphs", *Czechoslovak Math. J.* **23** (98): 298–305. As cited by Spielman & Teng (2007).
 - Fomin, Fedor V.; Thilikos, Dimitrios M. (2006a), "New upper bounds on the decomposability of planar graphs" (<http://users.uoa.gr/~sedthilk/papers/planar.pdf>), *Journal of Graph Theory* **51** (1): 53–81, doi: 10.1002/jgt.20121 (<http://dx.doi.org/10.1002/jgt.20121>).
 - Fomin, Fedor V.; Thilikos, Dimitrios M. (2006b), "Dominating sets in planar graphs: branch-width and exponential speed-up", *SIAM Journal on Computing* **36** (2): 281, doi: 10.1137/S0097539702419649 (<http://dx.doi.org/10.1137/S0097539702419649>).
 - Frieze, Alan; Miller, Gary L.; Teng, Shang-Hua (1992), "Separator based parallel divide and conquer in computational geometry" (<http://www.math.cmu.edu/~af1p/Texfiles/sep.pdf>), *Proc. 4th ACM Symposium on Parallel Algorithms and Architecture (SPAA '92)*, pp. 420–429, doi: 10.1145/140901.141934 (<http://dx.doi.org/10.1145/140901.141934>), ISBN 0-89791-483-X.
 - Gazit, Hillel; Miller, Gary L. (1990), "Planar separators and the Euclidean norm" (<http://www.cs.cmu.edu/~glmiller/Publications/Papers/GaMi90.pdf>), *Proc. International Symposium on Algorithms (SIGAL'90)*, Lecture Notes in Computer Science **450**, Springer-Verlag, pp. 338–347, doi: 10.1007/3-540-52921-7_83 (http://dx.doi.org/10.1007/3-540-52921-7_83), ISBN 978-3-540-52921-7.
 - George, J. Alan (1973), "Nested dissection of a regular finite element mesh", *SIAM Journal on Numerical Analysis* **10** (2): 345–363, doi: 10.1137/0710032 (<http://dx.doi.org/10.1137/0710032>), JSTOR 2156361 (<http://www.jstor.org/stable/2156361>).
 - Gilbert, John R.; Hutchinson, Joan P.; Tarjan, Robert E. (1984), "A separator theorem for graphs of bounded genus", *Journal of Algorithms* **5** (3): 391–407, doi: 10.1016/0196-6774(84)90019-1 ([http://dx.doi.org/10.1016/0196-6774\(84\)90019-1](http://dx.doi.org/10.1016/0196-6774(84)90019-1)).
 - Gilbert, John R.; Tarjan, Robert E. (1986), "The analysis of a nested dissection algorithm", *Numerische Mathematik* **50** (4): 377–404, doi: 10.1007/BF01396660 (<http://dx.doi.org/10.1007/BF01396660>).
 - Goodrich, Michael T. (1995), "Planar separators and parallel polygon triangulation", *Journal of Computer and System Sciences* **51** (3): 374–389, doi: 10.1006/jcss.1995.1076 (<http://dx.doi.org/10.1006/jcss.1995.1076>).
 - Gremban, Keith D.; Miller, Gary L.; Teng, Shang-Hua (1997), "Moments of inertia and graph separators" (<http://www.cs.cmu.edu/~glmiller/Publications/Papers/GrMiTe94.pdf>), *Journal of Combinatorial Optimization* **1** (1): 79–104, doi: 10.1023/A:1009763020645 (<http://dx.doi.org/10.1023/A:1009763020645>).
 - Har-Peled, Sariel (2011), *A Simple Proof of the Existence of a Planar Separator*, arXiv: 1105.0103 (<http://arxiv.org/abs/1105.0103>).
 - He, Xin; Kao, Ming-Yang; Lu, Hsueh-I (2000), "A fast general methodology for information-theoretically optimal encodings of graphs", *SIAM Journal on Computing* **30** (3): 838–846, doi: 10.1137/S0097539799359117 (<http://dx.doi.org/10.1137/S0097539799359117>).
 - Holzer, Martin; Schulz, Frank; Wagner, Dorothea; Prasinos, Grigorios; Zaroliagis, Christos (2009), "Engineering planar separator algorithms" (<http://digbib.ubka.uni-karlsruhe.de/eva/ira/2005/20>), *Journal of Experimental Algorithmics* **14**: 1.5–1.31, doi: 10.1145/1498698.1571635 (<http://dx.doi.org/10.1145/1498698.1571635>).

- Jordan, Camille (1869), "Sur les assemblages des lignes" (<http://resolver.sub.uni-goettingen.de/purl?GDZPPN002153998>), *Journal für die reine und angewandte Mathematik* **70**: 185–190, As cited by Miller et al. (1997).
- Kawarabayashi, Ken-Ichi; Reed, Bruce (2010), "A separator theorem in minor-closed classes", *Proc. 51st Annual IEEE Symposium on Foundations of Computer Science*.
- Klein, Philip; Rao, Satish; Rauch, Monika; Subramanian, Sairam (1994), "Faster shortest-path algorithms for planar graphs", *Proc. 26th ACM Symposium on Theory of Computing (STOC '94)*, pp. 27–37, doi: 10.1145/195058.195092 (<http://dx.doi.org/10.1145/195058.195092>), ISBN 0-89791-663-8.
- Łącki, Jakub; Sankowski, Piotr (2011), "Min-Cuts and Shortest Cycles in Planar Graphs in $O(n \log \log n)$ Time", *Proc. 19th Annual European Symposium on Algorithms*, Lecture Notes in Computer Science **6942**, Springer-Verlag, pp. 155–166, arXiv: 1104.4890 (<http://arxiv.org/abs/1104.4890>), doi: 10.1007/978-3-642-23719-5_14 (http://dx.doi.org/10.1007/978-3-642-23719-5_14), ISBN 978-3-642-23718-8.
- Lipton, Richard J.; Rose, Donald J.; Tarjan, Robert E. (1979), "Generalized nested dissection", *SIAM Journal on Numerical Analysis* **16** (2): 346–358, doi: 10.1137/0716027 (<http://dx.doi.org/10.1137/0716027>), JSTOR 2156840 (<http://www.jstor.org/stable/2156840>).
- Lipton, Richard J.; Tarjan, Robert E. (1979), "A separator theorem for planar graphs", *SIAM Journal on Applied Mathematics* **36** (2): 177–189, doi: 10.1137/0136016 (<http://dx.doi.org/10.1137/0136016>).
- Lipton, Richard J.; Tarjan, Robert E. (1980), "Applications of a planar separator theorem", *SIAM Journal on Computing* **9** (3): 615–627, doi: 10.1137/0209046 (<http://dx.doi.org/10.1137/0209046>).
- Miller, Gary L. (1986), "Finding small simple cycle separators for 2-connected planar graphs" (<http://www.cs.cmu.edu/~glmiller/Publications/Papers/Mi87.pdf>), *Journal of Computer and System Sciences* **32** (3): 265–279, doi: 10.1016/0022-0000(86)90030-9 ([http://dx.doi.org/10.1016/0022-0000\(86\)90030-9](http://dx.doi.org/10.1016/0022-0000(86)90030-9)).
- Miller, Gary L.; Teng, Shang-Hua; Thurston, William; Vavasis, Stephen A. (1997), "Separators for sphere-packings and nearest neighbor graphs", *J. ACM* **44** (1): 1–29, doi: 10.1145/256292.256294 (<http://dx.doi.org/10.1145/256292.256294>).
- Miller, Gary L.; Teng, Shang-Hua; Thurston, William; Vavasis, Stephen A. (1998), "Geometric separators for finite-element meshes", *SIAM Journal on Scientific Computing* **19** (2): 364–386, doi: 10.1137/S1064827594262613 (<http://dx.doi.org/10.1137/S1064827594262613>).
- Pach, János; Agarwal, Pankaj K. (1995), "Lipton–Tarjan Separator Theorem", *Combinatorial Geometry*, John Wiley & Sons, pp. 99–102.
- Papadimitriou, C. H.; Sideri, M. (1996), "The bisection width of grid graphs", *Theory of Computing Systems* **29** (2): 97–110, doi: 10.1007/BF01305310 (<http://dx.doi.org/10.1007/BF01305310>).
- Plotkin, Serge; Rao, Satish; Smith, Warren D. (1994), "Shallow excluded minors and improved graph decompositions" (<http://portal.acm.org/citation.cfm?id=314625>), *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA '94)*, pp. 462–470.
- Reed, Bruce; Wood, David R. (2009), "A linear-time algorithm to find a separator in a graph excluding a minor", *ACM Transactions on Algorithms* **5** (4): 1–16, doi: 10.1145/1597036.1597043 (<http://dx.doi.org/10.1145/1597036.1597043>).
- Seymour, Paul D.; Thomas, Robin (1994), "Call routing and the ratcatcher", *Combinatorica* **14** (2): 217–241, doi: 10.1007/BF01215352 (<http://dx.doi.org/10.1007/BF01215352>).
- Spielman, Daniel A.; Teng, Shang-Hua (1996), "Disk packings and planar separators" (<http://www.cs.yale.edu/homes/spielman/PAPERS/planarSep.pdf>), *Proc. 12th ACM Symposium on Computational Geometry (SCG '96)*, pp. 349–358, doi: 10.1145/237218.237404 (<http://dx.doi.org/10.1145/237218.237404>), ISBN 0-89791-804-5.
- Spielman, Daniel A.; Teng, Shang-Hua (2007), "Spectral partitioning works: Planar graphs and finite element meshes", *Linear Algebra and its Applications* **421** (2–3): 284–305, doi: 10.1016/j.laa.2006.07.020 (<http://dx.doi.org/10.1016/j.laa.2006.07.020>).

doi.org/10.1016/j.laa.2006.07.020.

- Sýkora, Ondrej; Vrt'o, Imrich (1993), "Edge separators for graphs of bounded genus with applications", *Theoretical Computer Science* **112** (2): 419–429, doi: 10.1016/0304-3975(93)90031-N ([http://dx.doi.org/10.1016/0304-3975\(93\)90031-N](http://dx.doi.org/10.1016/0304-3975(93)90031-N)).
- Tazari, Siamak; Müller-Hannemann, Matthias (2009), "Shortest paths in linear time on minor-closed graph classes, with an application to Steiner tree approximation", *Discrete Applied Mathematics* **157** (4): 673–684, doi: 10.1016/j.dam.2008.08.002 (<http://dx.doi.org/10.1016/j.dam.2008.08.002>).
- Ungar, Peter (1951), "A theorem on planar graphs", *Journal of the London Mathematical Society* **1** (4): 256, doi: 10.1112/jlms/s1-26.4.256 (<http://dx.doi.org/10.1112/jlms/s1-26.4.256>).
- Weimann, Oren; Yuster, Raphael (2010), "Computing the girth of a planar graph in $O(n \log n)$ time", *SIAM Journal on Discrete Mathematics* **24** (2): 609, doi: 10.1137/090767868 (<http://dx.doi.org/10.1137/090767868>).
- Wulff-Nilsen, Christian (2009), *Girth of a planar digraph with real edge weights in $O(n \log^3 n)$ time*, arXiv: 0908.0697 (<http://arxiv.org/abs/0908.0697>).

Graph minors

In graph theory, an undirected graph H is called a **minor** of the graph G if H can be formed from G by deleting edges and vertices and by contracting edges.

The theory of graph minors began with Wagner's theorem that a graph is planar if and only if its minors do not include the complete graph K_5 nor the complete bipartite graph $K_{3,3}$.^[1] The Robertson–Seymour theorem implies that an analogous forbidden minor characterization exists for every property of graphs that is preserved by deletions and edge contractions.^[2] For every fixed graph H , it is possible to test whether H is a minor of an input graph G in polynomial time; together with the forbidden minor characterization this implies that every graph property preserved by deletions and contractions may be recognized in polynomial time.

Other results and conjectures involving graph minors include the graph structure theorem, according to which the graphs that do not have H as a minor may be formed by gluing together simpler pieces, and Hadwiger's conjecture relating the inability to color a graph to the existence of a large complete graph as a minor of it. Important variants of graph minors include the topological minors and immersion minors.

Definitions

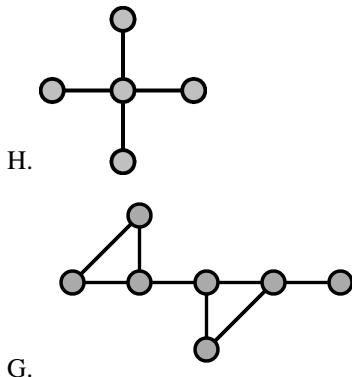
An edge contraction is an operation which removes an edge from a graph while simultaneously merging the two vertices it used to connect. An undirected graph H is a minor of another undirected graph G if a graph isomorphic to H can be obtained from G by contracting some edges, deleting some edges, and deleting some isolated vertices. The order in which a sequence of such contractions and deletions is performed on G does not affect the resulting graph H .

Graph minors are often studied in the more general context of matroid minors. In this context, it is common to assume that all graphs are connected, with self-loops and multiple edges allowed (that is, they are multigraphs rather than simple graphs); the contraction of a loop and the deletion of a cut-edge are forbidden operations. This point of view has the advantage that edge deletions leave the rank of a graph unchanged, and edge contractions always reduce the rank by one.

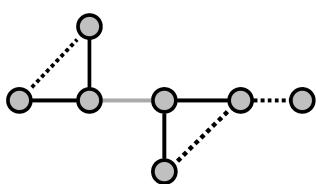
In other contexts (such as with the study of pseudoforests) it makes more sense to allow the deletion of a cut-edge, and to allow disconnected graphs, but to forbid multigraphs. In this variation of graph minor theory, a graph is always simplified after any edge contraction to eliminate its self-loops and multiple edges.^[3]

Example

In the following example, graph H is a minor of graph G :



The following diagram illustrates this. First construct a subgraph of G by deleting the dashed edges (and the resulting isolated vertex), and then contract the gray edge (merging the two vertices it connects):



Major results and conjectures

It is straightforward to verify that the graph minor relation forms a partial order on the isomorphism classes of undirected graphs: it is transitive (a minor of a minor of G is a minor of G itself), and G and H can only be minors of each other if they are isomorphic because any nontrivial minor operation removes edges or vertices. A deep result by Neil Robertson and Paul Seymour states that this partial order is actually a well-quasi-ordering: if an infinite list G_1, G_2, \dots of finite graphs is given, then there always exist two indices $i < j$ such that G_i is a minor of G_j . Another equivalent way of stating this is that any set of graphs can have only a finite number of minimal elements under the minor ordering.^[4] This result proved a conjecture formerly known as Wagner's conjecture, after Klaus Wagner; Wagner had conjectured it long earlier, but only published it in 1970.^[5]

In the course of their proof, Seymour and Robertson also prove the graph structure theorem in which they determine, for any fixed graph H , the rough structure of any graph which does not have H as a minor. The statement of the theorem is itself long and involved, but in short it establishes that such a graph must have the structure of a clique-sum of smaller graphs that are modified in small ways from graphs embedded on surfaces of bounded genus. Thus, their theory establishes fundamental connections between graph minors and topological embeddings of graphs.^[6]

For any graph H , the simple H -minor-free graphs must be sparse, which means that the number of edges is less than some constant multiple of the number of vertices. More specifically, if H has h vertices, then a simple n -vertex simple H -minor-free graph can have at most $O(nh\sqrt{\log h})$ edges, and some K_h -minor-free graphs have at least this many edges.^[7] Thus, if H has h vertices, then H -minor-free graphs have average degree $O(h\sqrt{\log h})$ and furthermore degeneracy $O(h\sqrt{\log h})$. Additionally, the H -minor-free graphs have a separator theorem similar to the planar separator theorem for planar graphs: for any fixed H , and any n -vertex H -minor-free graph G , it is possible to find a subset of $O(\sqrt{n})$ vertices whose removal splits G into two (possibly disconnected) subgraphs with at most $2n/3$ vertices per subgraph.^[8] Even stronger, for any fixed H , H -minor-free graphs have treewidth $O(\sqrt{n})$.

The Hadwiger conjecture in graph theory proposes that if a graph G does not contain a minor isomorphic to the complete graph on k vertices, then G has a proper coloring with $k - 1$ colors. The case $k = 5$ is a restatement of the four color theorem. The Hadwiger conjecture has been proven for $k \leq 6$, but is unknown in the general case.

Bollobás, Catlin & Erdős (1980) call it “one of the deepest unsolved problems in graph theory.” Another result relating the four-color theorem to graph minors is the snark theorem announced by Robertson, Sanders, Seymour, and Thomas, a strengthening of the four-color theorem conjectured by W. T. Tutte and stating that any bridgeless 3-regular graph that requires four colors in an edge coloring must have the Petersen graph as a minor.^[9]

Minor-closed graph families

For more details on minor-closed graph families, including a list of some, see Robertson–Seymour theorem.

Many families of graphs have the property that every minor of a graph in F is also in F ; such a class is said to be *minor-closed*. For instance, in any planar graph, or any embedding of a graph on a fixed topological surface, neither the removal of edges nor the contraction of edges can increase the genus of the embedding; therefore, planar graphs and the graphs embeddable on any fixed surface form minor-closed families.

If F is a minor-closed family, then (because of the well-quasi-ordering property of minors) among the graphs that do not belong to F there is a finite set X of minor-minimal graphs. These graphs are forbidden minors for F : a graph belongs to F if and only if it does not contain as a minor any graph in X . That is, every minor-closed family F can be characterized as the family of X -minor-free graphs for some finite set X of forbidden minors. The best-known example of a characterization of this type is Wagner's theorem characterizing the planar graphs as the graphs having neither K_5 nor $K_{3,3}$ as minors.

In some cases, the properties of the graphs in a minor-closed family may be closely connected to the properties of their excluded minors. For example a minor-closed graph family F has bounded pathwidth if and only if its forbidden minors include a forest, F has bounded tree-depth if and only if its forbidden minors include a disjoint union of path graphs, F has bounded treewidth if and only if its forbidden minors include a planar graph,^[10] and F has bounded local treewidth (a functional relationship between diameter and treewidth) if and only if its forbidden minors include an apex graph (a graph that can be made planar by the removal of a single vertex).^[11] If H can be drawn in the plane with only a single crossing (that is, it has crossing number one) then the H -minor-free graphs have a simplified structure theorem in which they are formed as clique-sums of planar graphs and graphs of bounded treewidth.^[12] For instance, both K_5 and $K_{3,3}$ have crossing number one, and as Wagner showed the K_5 -free graphs are exactly the 3-clique-sums of planar graphs and the eight-vertex Wagner graph, while the $K_{3,3}$ -free graphs are exactly the 2-clique-sums of planar graphs and K_5 .^[13]

Variations

Topological minors

A graph H is called a **topological minor** of a graph G if a subdivision of H is isomorphic to a subgraph of G . It is easy to see that every topological minor is also a minor. The converse however is not true in general (for instance the complete graph K_5 in the Petersen graph is a minor but not a topological one), but holds for graphs with maximum degree not greater than three.

The topological minor relation is not a well-quasi-ordering on the set of finite graphs^[14] and hence the result of Robertson and Seymour does not apply to topological minors. However it is straightforward to construct finite forbidden topological minor characterizations from finite forbidden minor characterizations by replacing every branch set with k outgoing edges by every tree on k leaves that has down degree at least two.

Immersion minor

A graph operation called *lifting* is central in a concept called *immersions*. The *lifting* is an operation on adjacent edges. Given three vertices v , u , and w , where (v,u) and (u,w) are edges in the graph, the lifting of vw , or equivalent of (v,u) , (u,w) is the operation that deletes the two edges (v,u) and (u,w) and adds the edge (v,w) . In the case where (v,w) already was present, v and w will now be connected by more than one edge, and hence this operation is intrinsically a multi-graph operation.

In the case where a graph H can be obtained from a graph G by a sequence of lifting operations (on G) and then finding an isomorphic subgraph, we say that H is an immersion minor of G . There is yet another way of defining immersion minors, which is equivalent to the lifting operation. We say that H is an immersion minor of G if there exists an injective mapping from vertices in H to vertices in G where the images of adjacent elements of H are connected in G by edge-disjoint paths.

The immersion minor relation is a well-quasi-ordering on the set of finite graphs and hence the result of Robertson and Seymour applies to immersion minors. This furthermore means that every immersion minor-closed family is characterized by a finite family of forbidden immersion minors.

In graph drawing, immersion minors arise as the planarizations of non-planar graphs: from a drawing of a graph in the plane, with crossings, one can form an immersion minor by replacing each crossing point by a new vertex, and in the process also subdividing each crossed edge into a path. This allows drawing methods for planar graphs to be extended to non-planar graphs.^[15]

Shallow minors

A shallow minor of a graph G is a minor in which the edges of G that were contracted to form the minor form a collection of disjoint subgraphs with low diameter. Shallow minors interpolate between the theories of graph minors and subgraphs, in that shallow minors with high depth coincide with the usual type of graph minor, while the shallow minors with depth zero are exactly the subgraphs. They also allow the theory of graph minors to be extended to classes of graphs such as the 1-planar graphs that are not closed under taking minors.^[16]

Algorithms

The problem of deciding whether a graph G contains H as a minor is NP-complete in general; for instance, if H is a cycle graph with the same number of vertices as G , then H is a minor of G if and only if G contains a Hamiltonian cycle. However, when G is part of the input but H is fixed, it can be solved in polynomial time. More specifically, the running time for testing whether H is a minor of G in this case is $O(n^3)$, where n is the number of vertices in G and the big O notation hides a constant that depends superexponentially on H ; since the original Graph Minors result, this algorithm has been improved to $O(n^2)$ time. Thus, by applying the polynomial time algorithm for testing whether a given graph contains any of the forbidden minors, it is possible to recognize the members of any minor-closed family in polynomial time. However, in order to apply this result constructively, it is necessary to know what the forbidden minors of the graph family are.

Notes

- [1] , p. 77; .
- [2] , theorem 4, p. 78; .
- [3] is inconsistent about whether to allow self-loops and multiple adjacencies: he writes on p. 76 that "parallel edges and loops are allowed" but on p. 77 he states that "a graph is a forest if and only if it does not contain the triangle K_3 as a minor", true only for simple graphs.
- [4] , Chapter 12: Minors, Trees, and WQO; .
- [5] , p. 76.
- [6] , pp. 80–82; .
- [7] ; ; .
- [8] ; ; .
- [9] ; .
- [10] , Theorem 9, p. 81; .
- [11] ; .
- [12] ; .
- [13] ; ; .
- [14] Ding (1996).
- [15] Buchheim et al. (2014).
- [16] , pp. 319–321.

References

- Alon, Noga; Seymour, Paul; Thomas, Robin (1990), "A separator theorem for nonplanar graphs" (<http://www.ams.org/journals/jams/1990-03-04/S0894-0347-1990-1065053-0/home.html>), *Journal of the American Mathematical Society* **3** (4): 801–808, doi: 10.2307/1990903 (<http://dx.doi.org/10.2307/1990903>), JSTOR 1990903 (<http://www.jstor.org/stable/1990903>), MR 1065053 (<http://www.ams.org/mathscinet-getitem?mr=1065053>).
- Bollobás, B.; Catlin, P. A.; Erdős, Paul (1980), "Hadwiger's conjecture is true for almost every graph" (http://www2.renyi.hu/~p_erdos/1980-10.pdf), *European Journal of Combinatorics* **1**: 195–199.
- Buchheim, Christoph; Chimani, Markus; Gutwenger, Carsten; Jünger, Michael; Mutzel, Petra (2014), "Crossings and planarization", in Tamassia, Roberto, *Handbook of Graph Drawing and Visualization*, Discrete Mathematics and its Applications (Boca Raton), CRC Press, Boca Raton, FL.
- Demaine, Erik D.; Hajiaghayi, MohammadTaghi (2004), "Diameter and treewidth in minor-closed graph families, revisited" (http://erikdemaine.org/papers/DiameterTreewidth_Algorithmica/), *Algorithmica* **40** (3): 211–215, doi: 10.1007/s00453-004-1106-1 (<http://dx.doi.org/10.1007/s00453-004-1106-1>).
- Demaine, Erik D.; Hajiaghayi, MohammadTaghi; Thilikos, Dimitrios M. (2002), "1.5-Approximation for treewidth of graphs excluding a graph with one crossing as a minor", *Proc. 5th International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX 2002)*, Lecture Notes in Computer Science **2462**, Springer-Verlag, pp. 67–80, doi: 10.1007/3-540-45753-4_8 (http://dx.doi.org/10.1007/3-540-45753-4_8)
- Diestel, Reinhard (2005), *Graph Theory* (http://www.math.uni-hamburg.de/home/diestel/books/graph_theory/) (3rd ed.), Berlin, New York: Springer-Verlag, ISBN 978-3-540-26183-4.
- Ding, Guoli (1996), "Excluding a long double path minor", *Journal of Combinatorial Theory, Series B* **66** (1): 11–23, doi: 10.1006/jctb.1996.0002 (<http://dx.doi.org/10.1006/jctb.1996.0002>), MR 1368512 (<http://www.ams.org/mathscinet-getitem?mr=1368512>).
- Eppstein, David (2000), "Diameter and treewidth in minor-closed graph families", *Algorithmica* **27**: 275–291, arXiv: math.CO/9907126 (<http://arxiv.org/abs/math.CO/9907126>), doi: 10.1007/s004530010020 (<http://dx.doi.org/10.1007/s004530010020>), MR 2001c:05132 (<http://www.ams.org/mathscinet-getitem?mr=2001c:05132>).
- Fellows, Michael R.; Langston, Michael A. (1988), "Nonconstructive tools for proving polynomial-time decidability", *Journal of the ACM* **35** (3): 727–739, doi: 10.1145/44483.44491 (<http://dx.doi.org/10.1145/44483.44491>).

- Grohe, Martin (2003), "Local tree-width, excluded minors, and approximation algorithms" (<http://arxiv.org/abs/math/0001128>), *Combinatorica* **23** (4): 613–632, doi: 10.1007/s00493-003-0037-9 (<http://dx.doi.org/10.1007/s00493-003-0037-9>).
- Hadwiger, Hugo (1943), "Über eine Klassifikation der Streckenkomplexe", *Vierteljschr. Naturforsch. Ges. Zürich* **88**: 133–143.
- Hall, Dick Wick (1943), "A note on primitive skew curves", *Bulletin of the American Mathematical Society* **49** (12): 935–936, doi: 10.1090/S0002-9904-1943-08065-2 (<http://dx.doi.org/10.1090/S0002-9904-1943-08065-2>).
- Kawarabayashi, Ken-ichi; Kobayashi, Yusuke; Reed, Bruce (March 2012), "The disjoint paths problem in quadratic time", *Journal of Combinatorial Theory, Series B* **102** (2): 424–435, doi: 10.1016/j.jctb.2011.07.004 (<http://dx.doi.org/10.1016/j.jctb.2011.07.004>)
- Kostochka, Alexandr V. (1982), "The minimum Hadwiger number for graphs with a given mean degree of vertices", *Metody Diskret. Analiz.* (in Russian) **38**: 37–58.
- Kostochka, Alexandr V. (1984), "Lower bound of the Hadwiger number of graphs by their average degree", *Combinatorica* **4**: 307–316, doi: 10.1007/BF02579141 (<http://dx.doi.org/10.1007/BF02579141>).
- Lovász, László (2006), "Graph minor theory", *Bulletin of the American Mathematical Society* **43** (1): 75–86, doi: 10.1090/S0273-0979-05-01088-8 (<http://dx.doi.org/10.1090/S0273-0979-05-01088-8>).
- Mader, Wolfgang (1967), "Homomorphieeigenschaften und mittlere Kantendichte von Graphen", *Mathematische Annalen* **174** (4): 265–268, doi: 10.1007/BF01364272 (<http://dx.doi.org/10.1007/BF01364272>).
- Nešetřil, Jaroslav; Ossona de Mendez, Patrice (2012), *Sparsity: Graphs, Structures, and Algorithms*, Algorithms and Combinatorics **28**, Springer, pp. 62–65, doi: 10.1007/978-3-642-27875-4 (<http://dx.doi.org/10.1007/978-3-642-27875-4>), ISBN 978-3-642-27874-7, MR 2920058 (<http://www.ams.org/mathscinet-getitem?mr=2920058>).
- Pegg, Ed, Jr. (2002), "Book Review: The Colossal Book of Mathematics" (<http://www.ams.org/notices/200209/rev-pegg.pdf>), *Notices of the American Mathematical Society* **49** (9): 1084–1086, doi: 10.1109/TED.2002.1003756 (<http://dx.doi.org/10.1109/TED.2002.1003756>).
- Plotkin, Serge; Rao, Satish; Smith, Warren D. (1994), "Shallow excluded minors and improved graph decompositions" (<http://www.stanford.edu/~plotkin/lminors.ps>), *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms (SODA 1994)*, pp. 462–470.
- Reed, Bruce; Wood, David R. (2009), "A linear-time algorithm to find a separator in a graph excluding a minor", *ACM Transactions on Algorithms* **5** (4): Article 39, doi: 10.1145/1597036.1597043 (<http://dx.doi.org/10.1145/1597036.1597043>).
- Robertson, Neil; Seymour, Paul (1983), "Graph minors. I. Excluding a forest", *Journal of Combinatorial Theory, Series B* **35** (1): 39–61, doi: 10.1016/0095-8956(83)90079-5 ([http://dx.doi.org/10.1016/0095-8956\(83\)90079-5](http://dx.doi.org/10.1016/0095-8956(83)90079-5)).
- Robertson, Neil; Seymour, Paul D. (1986), "Graph minors. V. Excluding a planar graph", *Journal of Combinatorial Theory, Series B* **41** (1): 92–114, doi: 10.1016/0095-8956(86)90030-4 ([http://dx.doi.org/10.1016/0095-8956\(86\)90030-4](http://dx.doi.org/10.1016/0095-8956(86)90030-4)).
- Robertson, Neil; Seymour, Paul D. (1993), "Excluding a graph with one crossing", in Robertson, Neil; Seymour, Paul, *Graph Structure Theory: Proc. AMS-IMS-SIAM Joint Summer Research Conference on Graph Minors*, Contemporary Mathematics **147**, American Mathematical Society, pp. 669–675.
- Robertson, Neil; Seymour, Paul D. (1995), "Graph Minors. XIII. The disjoint paths problem", *Journal of Combinatorial Theory, Series B* **63** (1): 65–110, doi: 10.1006/jctb.1995.1006 (<http://dx.doi.org/10.1006/jctb.1995.1006>).
- Robertson, Neil; Seymour, Paul D. (2003), "Graph Minors. XVI. Excluding a non-planar graph", *Journal of Combinatorial Theory, Series B* **89** (1): 43–76, doi: 10.1016/S0095-8956(03)00042-X ([http://dx.doi.org/10.1016/S0095-8956\(03\)00042-X](http://dx.doi.org/10.1016/S0095-8956(03)00042-X)).

- Robertson, Neil; Seymour, Paul D. (2004), "Graph Minors. XX. Wagner's conjecture", *Journal of Combinatorial Theory, Series B* **92** (2): 325–357, doi: 10.1016/j.jctb.2004.08.001 (<http://dx.doi.org/10.1016/j.jctb.2004.08.001>).
- Robertson, Neil; Seymour, Paul; Thomas, Robin (1993), "Hadwiger's conjecture for K_6 -free graphs" (<http://www.math.gatech.edu/~thomas/PAP/hadwiger.pdf>), *Combinatorica* **13**: 279–361, doi: 10.1007/BF01202354 (<http://dx.doi.org/10.1007/BF01202354>).
- Thomas, Robin (1999), "Recent excluded minor theorems for graphs" (<http://people.math.gatech.edu/~thomas/PAP/bcc.pdf>), *Surveys in combinatorics, 1999 (Canterbury)*, London Math. Soc. Lecture Note Ser. **267**, Cambridge: Cambridge Univ. Press, pp. 201–222, MR 1725004 (<http://www.ams.org/mathscinet-getitem?mr=1725004>).
- Thomason, Andrew (1984), "An extremal function for contractions of graphs", *Mathematical Proceedings of the Cambridge Philosophical Society* **95** (2): 261–265, doi: 10.1017/S0305004100061521 (<http://dx.doi.org/10.1017/S0305004100061521>).
- Thomason, Andrew (2001), "The extremal function for complete minors", *Journal of Combinatorial Theory, Series B* **81** (2): 318–338, doi: 10.1006/jctb.2000.2013 (<http://dx.doi.org/10.1006/jctb.2000.2013>).
- Wagner, Klaus (1937a), "Über eine Eigenschaft der ebenen Komplexe", *Math. Ann.* **114**: 570–590, doi: 10.1007/BF01594196 (<http://dx.doi.org/10.1007/BF01594196>).
- Wagner, Klaus (1937b), "Über eine Erweiterung des Satzes von Kuratowski", *Deutsche Mathematik* **2**: 280–285.

External links

- Weisstein, Eric W., "Graph Minor" (<http://mathworld.wolfram.com/GraphMinor.html>), *MathWorld*.

Courcelle's theorem

In the study of graph algorithms, **Courcelle's theorem** is the statement that every graph property definable in monadic second-order logic can be decided in linear time on graphs of bounded treewidth. The result was first proved by Bruno Courcelle in 1990 and is considered the archetype of algorithmic meta-theorems.

Formulations

Vertex sets

In one variation of monadic second-order graph logic known as MSO₁, the graph is described by a set of vertices V and a binary adjacency relation adj(.,.), and the restriction to monadic logic means that the graph property in question may be defined in terms of sets of vertices of the given graph, but not in terms of sets of edges or of tuples of vertices.

As an example, the property of a graph being colorable with three colors (represented by three sets of vertices R , G , and B) may be defined by the monadic second-order formula

$$\exists R, G, B. \quad (\forall v \in V. (v \in R \vee v \in G \vee v \in B)) \wedge \\ (\forall u, v \in V. ((u \in R \wedge v \in R) \vee (u \in G \wedge v \in G) \vee (u \in B \wedge v \in B)) \rightarrow \neg \text{adj}(u, v)).$$

The first part of this formula ensures that the three color classes cover all the vertices of the graph, and the second ensures that they each form an independent set. (It would also be possible to add clauses to the formula to ensure that the three color classes are disjoint, but this makes no difference to the result.) Thus, by Courcelle's theorem, 3-colorability of graphs of bounded treewidth may be tested in linear time.

For this variation of graph logic, Courcelle's theorem can be extended from treewidth to clique-width: for every fixed MSO_1 property P , and every fixed bound b on the clique-width of a graph, there is a linear-time algorithm for testing whether a graph of clique-width at most b has property P .

Edge sets

Courcelle's theorem may also be used with a stronger variation of monadic second-order logic known as MSO_2 . In this formulation, a graph is represented by a set V of vertices, a set E of edges, and an incidence relation between vertices and edges. This variation allows quantification over sets of vertices or edges, but not over more complex relations on tuples of vertices or edges.

For instance, the property of having a Hamiltonian cycle may be expressed in MSO_2 by describing the cycle as a set of edges that includes exactly two edges incident to each vertex, such that every nonempty proper subset of vertices has an edge in the cycle with exactly one endpoint in the subset. However, Hamiltonicity cannot be expressed in MSO_1 .^[1]

Modular congruences

Another direction for extending Courcelle's theorem concerns logical formulas that include predicates for counting the size of the test. In this context, it is not possible to perform arbitrary arithmetic operations on set sizes, nor even to test whether two sets have the same size. However, MSO_1 and MSO_2 can be extended to logics called CMSO_1 and CMSO_2 , that include for every two constants q and r a predicate $\text{card}_{q,r}(S)$ which tests whether the cardinality of set S is congruent to r modulo q . Courcelle's theorem can be extended to these logics.

Proof strategy

The typical approach to proving Courcelle's theorem involves the construction of a finite bottom-up tree automaton that performs a tree decomposition of the graph to recognize it.

Satisfiability and Seese's theorem

The satisfiability problem for a formula of monadic second-order logic is the problem of determining whether there exists at least one graph (possibly within a restricted family of graphs) for which the formula is true. For arbitrary graph families, and arbitrary formulas, this problem is undecidable. However, satisfiability of MSO_2 formulas is decidable for the graphs of bounded treewidth, and satisfiability of MSO_1 formulas is decidable for graphs of bounded clique-width. The proof involves building a tree automaton for the formula and then testing whether the automaton has an accepting path.

As a partial converse, Seese (1991) proved that, whenever a family of graphs has a decidable MSO_2 satisfiability problem, the family must have bounded treewidth. The proof is based on a theorem of Robertson and Seymour that the families of graphs with unbounded treewidth have arbitrarily large grid minors. Seese also conjectured that every family of graphs with a decidable MSO_1 satisfiability problem must have bounded clique-width; this has not been proven, but a weakening of the conjecture that replaces MSO_1 by CMSO_1 is true.

Applications

Grohe (2001) used Courcelle's theorem to show that computing the crossing number of a graph G is fixed-parameter tractable with a quadratic dependence on the size of G , improving a cubic-time algorithm based on the Robertson–Seymour theorem. An additional later improvement to linear time by Kawarabayashi & Reed (2007) follows the same approach. If the given graph G has small treewidth, Courcelle's theorem can be applied directly to this problem. On the other hand, if G has large treewidth, then it contains a large grid minor, within which the graph can be simplified while leaving the crossing number unchanged. Grohe's algorithm performs these simplifications

until the remaining graph has a small treewidth, and then applies Courcelle's theorem to solve the reduced subproblem.

Gottlob & Lee (2007) observed that Courcelle's theorem applies to several problems of finding minimum multi-way cuts in a graph, when the structure formed by the graph and the set of cut pairs has bounded treewidth. As a result they obtain a fixed-parameter tractable algorithm for these problems, parameterized by a single parameter, treewidth, improving previous solutions that had combined multiple parameters.

In computational topology, Burton & Downey (2014) extend Courcelle's theorem from MSO_2 to a form of monadic second-order logic on simplicial complexes of bounded dimension that allows quantification over simplices of any fixed dimension. As a consequence, they show how to compute certain quantum invariants of 3-manifolds as well as how to solve certain problems in discrete Morse theory efficiently, when the manifold has a triangulation (avoiding degenerate simplices) whose dual graph has small treewidth.^[2]

Methods based on Courcelle's theorem have also been applied to database theory, knowledge representation and reasoning, automata theory, and model checking.

References

[1] , Proposition 5.13, p. 338 (<http://books.google.com/books?id=JpIhAwAAQBAJ&pg=PA338>).

[2] . Short communication, International Congress of Mathematicians, 2014.

Robertson–Seymour theorem

In graph theory, the **Robertson–Seymour theorem** (also called the **graph minor theorem**) states that the undirected graphs, partially ordered by the graph minor relationship, form a well-quasi-ordering. Equivalently, every family of graphs that is closed under minors can be defined by a finite set of forbidden minors, in the same way that Wagner's theorem characterizes the planar graphs as being the graphs that do not have the complete graph K_5 and the complete bipartite graph $K_{3,3}$ as minors.

The Robertson–Seymour theorem is named after mathematicians Neil Robertson and Paul D. Seymour, who proved it in a series of twenty papers spanning over 500 pages from 1983 to 2004.^[1] Before its proof, the statement of the theorem was known as **Wagner's conjecture** after the German mathematician Klaus Wagner, although Wagner said he never conjectured it.

A weaker result for trees is implied by Kruskal's tree theorem, which was conjectured in 1937 by Andrew Vázsonyi and proved in 1960 independently by Joseph Kruskal and S. Tarkowski.^[2]

Statement

A minor of an undirected graph G is any graph that may be obtained from G by a sequence of zero or more contractions of edges of G and deletions of edges and vertices of G . The minor relationship forms a partial order on the set of all distinct finite undirected graphs, as it obeys the three axioms of partial orders: it is reflexive (every graph is a minor of itself), transitive (a minor of a minor of G is itself a minor of G), and antisymmetric (if two graphs G and H are minors of each other, then they must be isomorphic). However, if graphs that are isomorphic may nonetheless be considered as distinct objects, then the minor ordering on graphs forms a preorder, a relation that is reflexive and transitive but not necessarily antisymmetric.^[3]

A preorder is said to form a well-quasi-ordering if it contains neither an infinite descending chain nor an infinite antichain. For instance, the usual ordering on the non-negative integers is a well-quasi-ordering, but the same ordering on the set of all integers is not, because it contains the infinite descending chain $0, -1, -2, -3\dots$

The Robertson–Seymour theorem states that finite undirected graphs and graph minors form a well-quasi-ordering. It is obvious that the graph minor relationship does not contain any infinite descending chain, because each contraction or deletion reduces the number of edges and vertices of the graph (a non-negative integer). The nontrivial part of the theorem is that there are no infinite antichains, infinite sets of graphs that are all unrelated to each other by the minor ordering. If S is a set of graphs, and M is a subset of S containing one representative graph for each equivalence class of minimal elements (graphs that belong to S but for which no proper minor belongs to S), then M forms an antichain; therefore, an equivalent way of stating the theorem is that, in any infinite set S of graphs, there must be only a finite number of non-isomorphic minimal elements.

Another equivalent form of the theorem is that, in any infinite set S of graphs, there must be a pair of graphs one of which is a minor of the other. The statement that every infinite set has finitely many minimal elements implies this form of the theorem, for if there are only finitely many minimal elements, then each of the remaining graphs must belong to a pair of this type with one of the minimal elements. And in the other direction, this form of the theorem implies the statement that there can be no infinite antichains, because an infinite antichain is a set that does not contain any pair related by the minor relation.

Forbidden minor characterizations

A family F of graphs is said to be closed under the operation of taking minors if every minor of a graph in F also belongs to F . If F is a minor-closed family, then let S be the set of graphs that are not in F (the complement of F). According to the Robertson–Seymour theorem, there exists a finite set H of minimal elements in S . These minimal elements form a forbidden graph characterization of F : the graphs in F are exactly the graphs that do not have any graph in H as a minor.^[4] The members of H are called the **excluded minors** (or **forbidden minors**, or **minor-minimal obstructions**) for the family F .

For example, the planar graphs are closed under taking minors: contracting an edge in a planar graph, or removing edges or vertices from the graph, cannot destroy its planarity. Therefore, the planar graphs have a forbidden minor characterization, which in this case is given by Wagner's theorem: the set H of minor-minimal nonplanar graphs contains exactly two graphs, the complete graph K_5 and the complete bipartite graph $K_{3,3}$, and the planar graphs are exactly the graphs that do not have a minor in the set $\{K_5, K_{3,3}\}$.

The existence of forbidden minor characterizations for all minor-closed graph families is an equivalent way of stating the Robertson–Seymour theorem. For, suppose that every minor-closed family F has a finite set H of minimal forbidden minors, and let S be any infinite set of graphs. Define F from S as the family of graphs that do not have a minor in S . Then F is minor-closed and has a finite set H of minimal forbidden minors. Let C be the complement of F . S is a subset of C since S and F are disjoint, and H are the minimal graphs in C . Consider a graph G in H . G cannot have a proper minor in S since G is minimal in C . At the same time, G must have a minor in S , since otherwise G would be an element in F . Therefore, G is an element in S , i.e., H is a subset of S , and all other graphs in S have a minor among the graphs in H , so H is the finite set of minimal elements of S .

For the other implication, assume that every set of graphs has a finite subset of minimal graphs and let a minor-closed set F be given. We want to find a set H of graphs such that a graph is in F if and only if it does not have a minor in H . Let E be the graphs which are not minors of any graph in F , and let H be the finite set of minimal graphs in E . Now, let an arbitrary graph G be given. Assume first that G is in F . G cannot have a minor in H since G is in F and H is a subset of E . Now assume that G is not in F . Then G is not a minor of any graph in F , since F is minor-closed. Therefore, G is in E , so G has a minor in H .

Examples of minor-closed families

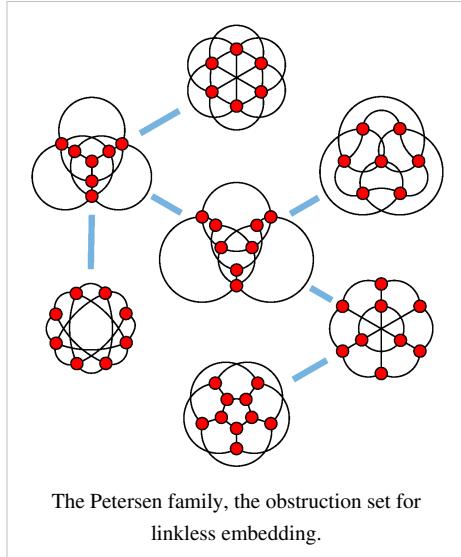
Main article: [Forbidden graph characterization](#)

The following sets of finite graphs are minor-closed, and therefore (by the Robertson–Seymour theorem) have forbidden minor characterizations:

- forests, linear forests (disjoint unions of path graphs), pseudoforests, and cactus graphs;
- planar graphs, outerplanar graphs, apex graphs (formed by adding a single vertex to a planar graph), toroidal graphs, and the graphs that can be embedded on any fixed two-dimensional manifold;
- graphs that are linklessly embeddable in Euclidean 3-space, and graphs that are knotlessly embeddable in Euclidean 3-space;
- graphs with a feedback vertex set of size bounded by some fixed constant; graphs with Colin de Verdière graph invariant bounded by some fixed constant; graphs with treewidth, pathwidth, or branchwidth bounded by some fixed constant.

Obstruction sets

Some examples of finite obstruction sets were already known for specific classes of graphs before the Robertson–Seymour theorem was proved. For example, the obstruction for the set of all forests is the loop graph (or, if one restricts to simple graphs, the cycle with three vertices). This means that a graph is a forest if and only if none of its minors is the loop (or, the cycle with three vertices, respectively). The sole obstruction for the set of paths is the tree with four vertices, one of which has degree 3. In these cases, the obstruction set contains a single element, but in general this is not the case. Wagner's theorem states that a graph is planar if and only if it has neither K_5 nor $K_{3,3}$ as a minor. In other words, the set $\{K_5, K_{3,3}\}$ is an obstruction set for the set of all planar graphs, and in fact the unique minimal obstruction set. A similar theorem states that K_4 and $K_{2,3}$ are the forbidden minors for the set of outerplanar graphs.



Although the Robertson–Seymour theorem extends these results to arbitrary minor-closed graph families, it is not a complete substitute for these results, because it does not provide an explicit description of the obstruction set for any family. For example, it tells us that the set of toroidal graphs has a finite obstruction set, but it does not provide any such set. The complete set of forbidden minors for toroidal graphs remains unknown, but contains at least 16000 graphs.

Polynomial time recognition

The Robertson–Seymour theorem has an important consequence in computational complexity, due to the proof by Robertson and Seymour that, for each fixed graph G , there is a polynomial time algorithm for testing whether larger graphs have G as a minor. The running time of this algorithm can be expressed as a cubic polynomial in the size of the larger graph (although there is a constant factor in this polynomial that depends superpolynomially on the size of G), which has been improved to quadratic time by Kawarabayashi, Kobayashi, and Reed. As a result, for every minor-closed family F , there is polynomial time algorithm for testing whether a graph belongs to F : simply check, for each of the forbidden minors for F , whether the given graph contains that forbidden minor.^[5]

However, this method requires a specific finite obstruction set to work, and the theorem does not provide one. The theorem proves that such a finite obstruction set exists, and therefore the problem is polynomial because of the above

algorithm. However, the algorithm can be used in practice only if such a finite obstruction set is provided. As a result, the theorem proves that the problem can be solved in polynomial time, but does not provide a concrete polynomial-time algorithm for solving it. Such proofs of polynomiality are non-constructive: they prove polynomiality of problems without providing an explicit polynomial-time algorithm.^[6] In many specific cases, checking whether a graph is in a given minor-closed family can be done more efficiently: for example, checking whether a graph is planar can be done in linear time.

Fixed-parameter tractability

For graph invariants with the property that, for each k , the graphs with invariant at most k are minor-closed, the same method applies. For instance, by this result, treewidth, branchwidth, and pathwidth, vertex cover, and the minimum genus of an embedding are all amenable to this approach, and for any fixed k there is a polynomial time algorithm for testing whether these invariants are at most k , in which the exponent in the running time of the algorithm does not depend on k . A problem with this property, that it can be solved in polynomial time for any fixed k with an exponent that does not depend on k , is known as fixed-parameter tractable.

However, this method does not directly provide a single fixed-parameter-tractable algorithm for computing the parameter value for a given graph with unknown k , because of the difficulty of determining the set of forbidden minors. Additionally, the large constant factors involved in these results make them highly impractical. Therefore, the development of explicit fixed-parameter algorithms for these problems, with improved dependence on k , has continued to be an important line of research.

Finite form of the graph minor theorem

Friedman, Robertson & Seymour (1987) showed that the following theorem exhibits the independence phenomenon by being *unprovable* in various formal systems that are much stronger than Peano arithmetic, yet being *provable* in systems much weaker than ZFC:

Theorem: For every positive integer n , there is an integer m so large that if G_1, \dots, G_m is a sequence of finite undirected graphs,

where each G_i has size at most $n+i$, then $G_j \leq G_k$ for some $j < k$.

(Here, the *size* of a graph is the total number of its nodes and edges, and \leq denotes the minor ordering.)

Notes

[1] : .

[2] : , Section 3.3, pp. 78–79.

[3] E.g., see , Section 2, "well-quasi-orders".

[4] , Corollary 2.1.1; , Theorem 4, p. 78.

[5] ; , Theorem 2.1.4 and Corollary 2.1.5; , Theorem 11, p. 83.

[6] ; , Section 6.

References

- Bienstock, Daniel; Langston, Michael A. (1995), "Algorithmic implications of the graph minor theorem" (<http://www.cs.utk.edu/~langston/courses/cs594-fall2003/BL.pdf>), *Network Models*, Handbooks in Operations Research and Management Science 7, pp. 481–502, doi: 10.1016/S0927-0507(05)80125-2 ([http://dx.doi.org/10.1016/S0927-0507\(05\)80125-2](http://dx.doi.org/10.1016/S0927-0507(05)80125-2)).
- Chambers, J. (2002), *Hunting for torus obstructions*, M.Sc. thesis, Department of Computer Science, University of Victoria.
- Diestel, Reinhard (2005), "Minors, Trees, and WQO" (<http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/preview/Ch12.pdf>), *Graph Theory* (Electronic Edition 2005 ed.), Springer, pp. 326–367.

- Fellows, Michael R.; Langston, Michael A. (1988), "Nonconstructive tools for proving polynomial-time decidability", *Journal of the ACM* **35** (3): 727–739, doi: 10.1145/44483.44491 (<http://dx.doi.org/10.1145/44483.44491>).
- Friedman, Harvey; Robertson, Neil; Seymour, Paul (1987), "The metamathematics of the graph minor theorem", in Simpson, S., *Logic and Combinatorics*, Contemporary Mathematics **65**, American Mathematical Society, pp. 229–261.
- Kawarabayashi, Ken-ichi; Kobayashi, Yusuke; Reed, Bruce (2012), "The disjoint paths problem in quadratic time" (http://research.nii.ac.jp/~k_keniti/quaddp1.pdf), *Journal of Combinatorial Theory, Series B* **102** (2): 424–435, doi: 10.1016/j.jctb.2011.07.004 (<http://dx.doi.org/10.1016/j.jctb.2011.07.004>).
- Lovász, László (2005), "Graph Minor Theory", *Bulletin of the American Mathematical Society (New Series)* **43** (1): 75–86, doi: 10.1090/S0273-0979-05-01088-8 (<http://dx.doi.org/10.1090/S0273-0979-05-01088-8>).
- Robertson, Neil; Seymour, Paul (1983), "Graph Minors. I. Excluding a forest", *Journal of Combinatorial Theory, Series B* **35** (1): 39–61, doi: 10.1016/0095-8956(83)90079-5 ([http://dx.doi.org/10.1016/0095-8956\(83\)90079-5](http://dx.doi.org/10.1016/0095-8956(83)90079-5)).
- Robertson, Neil; Seymour, Paul (1995), "Graph Minors. XIII. The disjoint paths problem", *Journal of Combinatorial Theory, Series B* **63** (1): 65–110, doi: 10.1006/jctb.1995.1006 (<http://dx.doi.org/10.1006/jctb.1995.1006>).
- Robertson, Neil; Seymour, Paul (2004), "Graph Minors. XX. Wagner's conjecture", *Journal of Combinatorial Theory, Series B* **92** (2): 325–357, doi: 10.1016/j.jctb.2004.08.001 (<http://dx.doi.org/10.1016/j.jctb.2004.08.001>).

External links

- Weisstein, Eric W., "Robertson-Seymour Theorem" (<http://mathworld.wolfram.com/Robertson-SeymourTheorem.html>), *MathWorld*.

Bidimensionality

Bidimensionality theory characterizes a broad range of graph problems (**bidimensional**) that admit efficient approximate, fixed-parameter or kernel solutions in a broad range of graphs. These graph classes include planar graphs, map graphs, bounded-genus graphs and graphs excluding any fixed minor. In particular, bidimensionality theory builds on the graph minor theory of Robertson and Seymour by extending the mathematical results and building new algorithmic tools. The theory was introduced in the work of Demaine, Fomin, Hajiaghayi, and Thilikos.

Definition

A parameterized problem Π is a subset of $\Gamma^* \times \mathbb{N}$ for some finite alphabet Γ . An instance of a parameterized problem consists of (x, k) , where k is called the parameter.

A parameterized problem Π is *minor-bidimensional* if

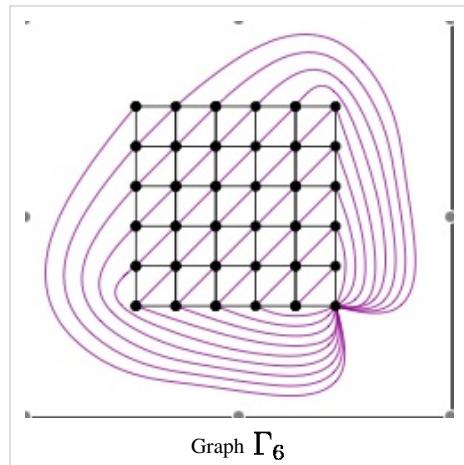
1. For any pair of graphs H, G , such that H is a minor of G and integer k , $(G, k) \in \Pi$ yields that $(H, k) \in \Pi$. In other words, contracting or deleting an edge in a graph G cannot increase the parameter; and
2. there is $\delta > 0$ such that for every $(r \times r)$ -grid R , $(R, k) \notin \Pi$ for every $k \leq \delta r^2$. In other words, the value of the solution on R should be at least δr^2 .

Examples of minor-bidimensional problems are the parameterized versions of vertex cover, feedback vertex set, minimum maximal matching, and longest path.

Let Γ_r be the graph obtained from the $(r \times r)$ -grid by triangulating internal faces such that all internal vertices become of degree 6, and then one corner of degree two joined by edges with all vertices of the external face. A parameterized problem Π is *contraction-bidimensional* if

1. For any pair of graphs H, G , such that H is a contraction of G and integer k , $(G, k) \in \Pi$ yields that $(H, k) \in \Pi$. In other words, contracting an edge in a graph G cannot increase the parameter; and
2. there is $\delta > 0$ such that $(\Gamma_r, k) \notin \Pi$ for every $k \leq \delta r^2$.

Examples of contraction-bidimensional problems are dominating set, connected dominating set, max-leaf spanning tree, and edge dominating set.



Excluded grid theorems

All algorithmic applications of bidimensionality are based on the following combinatorial property: either the treewidth of a graph is small, or the graph contains a large grid as a minor or contraction. More precisely,

1. There is a function f such that every graph G excluding a fixed h -vertex graph as a minor and of treewidth at least $f(h)r$ contains $(r \times r)$ -grid as a minor.
2. There is a function g such that every graph G excluding a fixed h -vertex apex graph as a minor and of treewidth at least $g(h)r$ can be edge-contracted to Γ_r .

Halin's grid theorem is an analogous excluded grid theorem for infinite graphs.

Subexponential parameterized algorithms

Let Π be a minor-bidimensional problem such that for any graph G excluding some fixed graph as a minor and of treewidth at most t , deciding whether $(G, k) \in \Pi$ can be done in time $2^{O(t)} \cdot |G|^{O(1)}$. Then for every graph G excluding some fixed graph as a minor, deciding whether $(G, k) \in \Pi$ can be done in time $2^{O(\sqrt{k})} \cdot |G|^{O(1)}$.

Similarly, for contraction-bidimensional problems, for graph G excluding some fixed apex graph as a minor, inclusion $(G, k) \in \Pi$ can be decided in time $2^{O(\sqrt{k})} \cdot |G|^{O(1)}$.

Thus many bidimensional problems like Vertex Cover, Dominating Set, k-Path, are solvable in time $2^{O(\sqrt{k})} \cdot |G|^{O(1)}$ on graphs excluding some fixed graph as a minor.

Polynomial time approximation schemes

Bidimensionality theory has been used to obtain polynomial-time approximation schemes for many bidimensional problems. If a minor (contraction) bidimensional problem has several additional properties then the problem poses efficient polynomial-time approximation schemes on (apex) minor-free graphs.

In particular, by making use of bidimensionality, it was shown that feedback vertex set, vertex cover, connected vertex cover, cycle packing, diamond hitting set, maximum induced forest, maximum induced bipartite subgraph and maximum induced planar subgraph admit an EPTAS on H-minor-free graphs. Edge dominating set, dominating set, r-dominating set, connected dominating set, r-scattered set, minimum maximal matching, independent set, maximum full-degree spanning tree, maximum induced at most d-degree subgraph, maximum internal spanning tree, induced matching, triangle packing, partial r-dominating set and partial vertex cover admit an EPTAS on apex-minor-free graphs.

Kernelization

Main article: Kernelization

A parameterized problem with a parameter k is said to admit a linear vertex kernel if there is a polynomial time reduction, called a kernelization algorithm, that maps the input instance to an equivalent instance with at most $O(k)$ vertices.

Every minor-bidimensional problem Π with additional properties, namely, with the separation property and with finite integer index, has a linear vertex kernel on graphs excluding some fixed graph as a minor. Similarly, every contraction-bidimensional problem Π with the separation property and with finite integer index has a linear vertex kernel on graphs excluding some fixed apex graph as a minor.

Notes

References

- Demaine, Erik D.; Fomin, Fedor V.; Hajiaghayi, MohammadTaghi; Thilikos, Dimitrios M. (2005), "Subexponential parameterized algorithms on bounded-genus graphs and H -minor-free graphs", *J. ACM* **52** (6): 866–893, doi: 10.1145/1101821.1101823 (<http://dx.doi.org/10.1145/1101821.1101823>).
- Demaine, Erik D.; Fomin, Fedor V.; Hajiaghayi, MohammadTaghi; Thilikos, Dimitrios M. (2004), "Bidimensional parameters and local treewidth", *SIAM Journal on Discrete Mathematics* **18** (3): 501–511, doi: 10.1137/S0895480103433410 (<http://dx.doi.org/10.1137/S0895480103433410>).
- Demaine, Erik D.; Hajiaghayi, MohammadTaghi (2005), "Bidimensionality: new connections between FPT algorithms and PTASs", *16th ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pp. 590–601.
- Demaine, Erik D.; Hajiaghayi, MohammadTaghi (2008), "Linearity of grid minors in treewidth with applications through bidimensionality", *Combinatorica* **28** (1): 19–36, doi: 10.1007/s00493-008-2140-4 (<http://dx.doi.org/>

- 10.1007/s00493-008-2140-4).
- Demaine, Erik D.; Hajiaghayi, MohammadTaghi (2008), "The bidimensionality theory and its algorithmic applications", *The Computer Journal* **51** (3): 332–337, doi: 10.1093/comjnl/bxm033 (<http://dx.doi.org/10.1093/comjnl/bxm033>).
 - Diestel, R. (2004), "A short proof of Halin's grid theorem", *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* **74**: 237–242, doi: 10.1007/BF02941538 (<http://dx.doi.org/10.1007/BF02941538>), MR 2112834 (<http://www.ams.org/mathscinet-getitem?mr=2112834>).
 - Fomin, Fedor V.; Golovach, Petr A.; Thilikos, Dimitrios M. (2009), "Contraction Bidimensionality: The Accurate Picture", *17th Annual European Symposium on Algorithms (ESA 2009)*, Lecture Notes in Computer Science **5757**, pp. 706–717, doi: 10.1007/978-3-642-04128-0_63 (http://dx.doi.org/10.1007/978-3-642-04128-0_63).
 - Fomin, Fedor V.; Lokshtanov, Daniel; Raman, Venkatesh; Saurabh, Saket (2010), "Bidimensionality and EPTAS", arXiv: 1005.5449 (<http://arxiv.org/abs/1005.5449>). *Proc. 22nd ACM/SIAM Symposium on Discrete Algorithms (SODA 2011)*, pp. 748–759.
 - Fomin, Fedor V.; Lokshtanov, Daniel; Saurabh, Saket; Thilikos, Dimitrios M. (2010), "Bidimensionality and Kernels", *21st ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, pp. 503–510.

Article Sources and Contributors

Graph theory *Source:* <http://en.wikipedia.org/w/index.php?oldid=623813468> *Contributors:* 4368a, APH, Aaditya 7, Aaronzat, Abeg92, Agro1986, Ajweinstein, Aknxy, Aks1521, Akutagawa10, Alan Au, Alansohn, Alcidesfonseca, Alex Bakharev, Allens, Almi, Altenmann, Ams80, Andre Engels, Andrea105, Andreas Kaufmann, Andris, Ankithbhatt, Anna Lincoln, Anonymous Dissident, Anthonymow12, Aquae, Arbor, Arvindn, Astrophil, AxelBoldt, Ayda D, Bact, Bananastalktome, Bazuz, Beda42, Bender235, Bereziny, Berteun, Bezug, Bg9989, Bkell, Blair Azzopardi, BlckKnght, Boleslav Bobcik, Booyabazooka, Brad7777, Brent Gulanowski, Brick Thrower, Brona, Bumbulski, C S, CRGreathouse, Camembert, CanadianLinuxUser, Caramdir, Cerber, CeserB, Chalst, Charles Matthews, Chas zzz brown, Chmod007, ChrisGualtieri, Conversion script, CorpX, Csl77, Cyborgbadger, D.Lazard, D75304, DFRussia, Dafyddg, Damien Karras, Daniel Mietchen, Daniel Quinlan, David Eppstein, Davidfstr, Dawn Bard, Dbenbenn, Delaszk, DerHexer, Dgrant, Dicklyon, Diego UFCG, Dina, Disaviam, Discospinster, Dittaeva, Doctorborzball, Domesticeinginerd, Doradus, Dr.S.Ramachandran, Duncharris, Dycedarg, Dysprosia, Dze27, ElBenevolente, Eleuther, Elf, Eric.weigle, Ettrig, Eugeneiiim, Faizan, Favonian, Faysalf, Fchristo, Feeshboy, Fivelittlemonkeys, Fred Bradstadt, Fredrik, FvdP, GGordonWorleyIII, GTBacchus, Gaius Cornelius, Gandalf61, Garyzx, Geometry guy, George Burgess, Gianfranco, Giftlite, GiveAFishABone, Glinos, Gmelli, GoShow, Gold4444, Goochelaara, Gragragna, Graham87, GraphTheoryPwns, GregorB, Grog, Gutza, Gyan, HMSSolent, Hannes Eder, Hans Adler, Hazmat2, Hbruhn, Headbomb, Henning Makholm, Hirzel, Idiosyncratic-bumblebee, Ignatzmice, Igodard, InoShiro, Jakob Voss, Jalpar75, Jarble, Jaxl, JeLuF, Jean.julius, Jeronimo, Jheuristic, Jim1138, Jinna, Jmencisom, Joel B. Lewis, Joespiff, JohnBlackburne, Johnsop, Jojib fb, Jon Awbrey, Jon har, Jonsafar, Jorge Stolfi, Jwissick, Kalogeropoulos, Karl E. V. Palmen, KellyCoinGuy, Kilom691, King Bee, Knutux, Kope, Kpjas, Kruusamägi, LC, LOL, Lanem, Ldecoba, Liao, Linas, Looxix, Low-frequency storn, Lpgeffen, Lupin, MER-C, Madewokherd, Magister Mathematicae, Magni, Maherite, Mailer diablo, Mani1, Marek69, Marianoceccowski, Mark Foskey, Mark Renier, Massly, MathMartin, Matthiaspaul, Matusz, Maurobio, Maxime.Deboscchere, Maxwell bernard, McKay, Meekohi, MelbourneStar, Melchoir, Michael Hardy, Michael Sloane, Miguel, Mild Bill Hiccup, Miym, Mpkr, Morph, Msh210, MusikAnimal, Mn, Myaw, Myasuda, MynameisJayden, Naerbin, Nanshu, Nasnema, Neth Hussain, Nichitch, Ntimp, OMурго, Obankston, Obradovic Goran, Ohnoitsjamie, Oleg Alexandrov, Oli Filth, Omigotanaccount, Onorem, Optim, OrangeDog, Oroso, Oskar Flordal, Outraged duck, Pashute, Patrick, Paul August, Paul Murray, PaulHadley, PaulTananbaum, Pcb21, Peter Kwok, Photoniique, Piano non troppo, PierreAbbat, Pinethicket, Poliron, Poor Yorick, Populus, Powerthirst123, Protonk, Pstanton, Pucky, Pugget, Quaeler, Qutezue, R'n'B, RUvard, Radagast3, RandomAct, Ratioincate, Renice, Requestion, Restname, Rev3nrant, RexNL, Rjwilmsi, Rmiesen, Roachmeister, Robert Merkel, Robin klein, RobinK, Ronz, Ruud Koot, SCEhardt, Sacredmint, SakeUPenn, Sagan, Sardanaphalus, Shanes, Shd, Shepazu, Shikhar1986, Shizhao, Sibian, SixWingedSeraph, Slawekb, SlumdogAramis, Smoke73, Sofia karampataki, Solitude, Sonia, Spacefarer, Spitfire8520, StaticElectricity, Stochata, Sundar, Sverdrup, TakuyaMurata, Tangi-tamma, Tarocards, Taxipom, Tkma, Template namespace initialisation script, The Cave Troll, The Isiah, Thesilverbail, Thv, Titanic4000, Tmusgrove, Tobias Bergemann, Tollyabolly, Tomo, Tompwy, Trinitrix, Tyir, Tyler McHenry, Uncle Dick, Usien6, Vacio, Vonkje, Watersmeetfreak, Wavelength, Whiteknox, Whyfish, Wikiisgreat123, Womiller99, Wraithful, Wshun, Wsu-dm-jb, Wsu-f, XJaM, Xdenizen, Xiong, Xnn, XxjwuxX, Yecril, Ylloh, Yloreander, Youngster68, Zaslav, Zero000, Zoicon5, Zsoftua, Zundark, Александр, Канеюк, 515 anonymous edits

Glossary of graph theory *Source:* <http://en.wikipedia.org/w/index.php?oldid=623397675> *Contributors:* 3mta3, A1kmm, Aarond144, Achab, Alexey Muranov, Algebraist, Altenmann, Altoid, Andrej.gric, Anonymous Dissident, Archelon, ArnoldReinholt, Bender235, Bethnim, Bkell, Booyabazooka, Brick Thrower, Brona, Buenasdiaz, Charles Matthews, ChrisGualtieri, Citrus538, Closedmouth, Csaracho, D6, DVanDyck, Damian Yerrick, David Eppstein, DavidRideout, Dbenbenn, Deljr, Dcoetze, Deltaheadron, Denisarona, Doc honcho, Doradus, Dysprosia, Edward, Eggwadi, El C, Elwikipedista, EmanueleMinotto, Eric119, Ferris37, GGordonWorleyIII, GTBacchus, Gaius Cornelius, Giftlite, Grubber, Happynomad, Headbomb, HorsePunchKid, Hyacinth, Ivan Štambuk, JLeander, JZacharyG, Jalal0, Jfizzell, Jmerm, JoernenB, John of Reading, Jokes Free4Me, Joriki, Justin W Smith, Jw489kent, Jwandres, Jérôme, Kjoonlee, Kope, Lansey, Lasuncty, Linas, Markhurd, Mastergreg82, MathMartin, MattGiuci, Maximus Rex, Mcld, Me and, MentorMentorum, Mggreenbe, Michael Hardy, Michael@nosivad.com, Mikhail Dvorkin, Miym, Morgoth106, Mzamora2, N8wilson, Nonenmac, Oleg Alexandrov, Ott2, Patrick, Paul August, PaulTananbaum, Peter Kwok, Pmq20, Pojo, Populus, Prunesqualer, Quaeler, Quintopia, Quixplusone, R'n'B, Ratfox, Rdvdijk, Reina riemann, RekishiEJ, Rich Farmbrough, Rick Norwood, Ricky81682, Rsdetsch, Ruud Koot, Salgueiro, Salix alba, SanitySolipsism, Scarp, Shadowjams, SixWingedSeraph, Skaraoke, SofjaKovalevskaja, Spanningtree, Starcrab, Stux, Sunayana, Sundar, Szebenisz, TakuyaMurata, The Transliterator, TheSolomon, Thechao, Thehotelambush, Tizio, Tomo, Twri, Vonkje, Warumwarum, Whorush, Wikipelli, Wshun, XJaM, Xdenizen, Xiong, Yath, Yecril, Yitzhak, Zaslav, 158 anonymous edits

Undirected graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=624611861> *Contributors:* 3ICE, A.amitkumar, ABF, Abdull, Ahorsteimer, Aittas, Ajraddatz, Aknorals, Akutagawa10, Algont, Altenmann, Amintora, Anabus, Anand jeyahar, Andrewsky00, Andros 1337, Andyman100, Athenray, Barra, BenRG, Bethnim, Bhadani, BiT, Bkell, Bobo192, Booyabazooka, Borgx, Brentdax, Burnin1134, CRGreathouse, Can sleep, clown will eat me, Catgut, Chbdorsett, Chuckley, CentroBabbage, Ch'marr, Chewings72, Chinassur, Chmod007, Chris the speller, Chronist, Citrus538, Cfjsyntropy, Comflake pirate, Corti, Crisófilax, Cwkmil, Cybercoba, D.Lazard, DARTH SIDIOUS 2, DRAGON BOOSTER, Danrah, David Eppstein, Davidfstr, Dbenbenn, Dcoetze, Ddxc, Ddebfam, Debeourlaus, Den fättradre ankan, Deur, Dicklyon, Discospinster, Dj3, Dockfish, Doradus, Dreamster, Dtrebbien, Dureo, Dysprosia, E mraedarab, Edgars2007, Editor70, Eken, Erhudy, Eric Lengyel, Falcon8765, Gaius Cornelius, Gandalf61, Gauge, Gene.arboit, Grolmusz, Gutin, Gwaihir, HMSSolent, Hairy Dude, Hannes Eder, Hans Adler, Hans Dunkelberg, Harp, Headbomb, Henry Delforn (old), Huynl, Ijdje, Ili Kr., Ilya, Inka, 888, Insanity Incarnate, J.delanoy, JNW, JP.Martin-Flatin, JaconaFrere, Jason Quinn, Jeff Erickson, Jiang, Joeblakesley, Joel B. Lewis, Jokes Free4Me, Jon Awbrey, Jon har, JonDePlume, Joriki, Josve05a, Jpeeling, Jpiw, JuPitEer, Jwandres, Karada, Karlscherer3, Kerrick Staley, Kine, Kingmash, Knutux, Kracekumar, Kruusamägi, Kuru, L Kensington, Lambyte, Liao, Libcub, LuckyWizard, Lugia2453, MER-C, MRG90, Maghnus, Magister Mathematicae, Mahanga, Manning Bartlett, Marc van Leeuwen, Maschen, Mate2code, Materialscientist, MathMartin, Matt Crypto, Mayoaranathan, McGeddon, McKay, Mdd, Michael Hardy, Michael Sloane, MikeBorkowski, Minder2k, Mindmatrix, Mitmaro, Miym, Mountain, Mrjohncummins, Myasuda, Myimyoward16, Nat2, Neilc, Netrapri, Nihonjoe, Nina Cerutti, Nowhither, Nsk92, Ohnoitsjamie, Oliphant, Oxymoron03, Pacu, Paleorhith, Patrick, Paul August, PaulTananbaum, Peter Kwok, Peter, Phegy81, PhotoBox, Pinethicket, Possum, Powerthirst123, Prunesqualer, Quaeler, R'n'B, Radagast3, RandomAct, RaseaC, Repied, Requestion, Rettetast, Rgcegg, Rich Farmbrough, Rjwilmsi, RobertBergersen, Robertsteadian, RobinK, Royerloic, Salix alba, Sdrucker, Shadowjams, Shanel, Siddhant, Silversmith, SixWingedSeraph, Someguy1221, SophomoricPedant, Sswn, Stevertigo, Stevetihi, Struthious Bandersnatch, Stux, Suchap, Super-Magician, Svick, THEN WHO WAS PHONE?, Tamfang, Tangi-tamma, Tbc2, Tempodivalse, Tgv8925, The Anome, Theone256, TimBentley, Timflutre, Timrollpickering, Tman159, Tobias Bergemann, Tom harrison, Tomhubard, Tomo, Tompwy, Tomruen, Tosha, Tslocum, Twri, Tyw7, UKoch, Ulric1313, Urdutext, Vacio, Vanischenu, Vaughan Pratt, VictorPorton, Voedin, Void-995, W, Wandler2, Wesley Moy, West.andrew.g, Wgntuner, White Trillium, WikiDao, Wikidsp, Wilking1979, Wknight94, Wshun, XJaM, Xavexgoem, Xiong, Yath, Yecril, Yitzhak, Ylloh, Yloreander, Zachlinton, Zaslav, Zeneck.k, Zero0000, Zocky, Zven, Канеюк, Пика Пика, 472 anonymous edits

Directed graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=620890846> *Contributors:* Alesiomi, Altenmann, Andreas Kaufmann, Anne Bauval, BiT, Bkell, Booyabazooka, Bryan.burgers, Calle, Catgut, Corruptcopper, David Eppstein, Delcypher, Dzied Bulbush, Einkil, Giftlite, Grue, Hamaryns, Headbomb, Henry Delforn (old), HughD, JP.Martin-Flatin, Jarble, Joerg Bader, Justin W Smith, Linas, Llorenzi, M-le-mot-dit, Marcuse7, Mark Renier, Mark viking, Meno25, Michael Hardy, Miym, Ms.wiki.us, Nbarth, NuclearWarfare, PaulTananbaum, Pcap, R'n'B, Ricardo Ferreira de Oliveira, Rinix, Saranavan, Shauncutts, Sinuhe20, SixWingedSeraph, SofjaKovalevskaja, Stepa, Twri, Vadmium, Wavelength, Werddemer, WookieInHeat, Zaslav, چارلز, ۴۷ anonymous edits

Directed acyclic graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=622920019> *Contributors:* Andreas Toth, Ap, Brick Thrower, Bryan Derksen, C-sonic, CRGreathouse, Calculuslover, Chris Howard, Colonies Chris, Comps, Damian Yerrick, Davehi1, David Eppstein, Dcoetze, Ddxc, Deflective, Delirium, DoostdarWKP, Doradus, Edward, Emre D, Ebpr123, Esap, Farisori, Frederic Y Bois, Gambette, Giftlite, Greenrd, Hannes Eder, Henrygb, Homerjay, IcedNut, Jaredwf, Jerset77, Jmr, Jonon, Kaapih, Kieran, Kristjan.Jonasson, Kwamikagami, Marc Girod, MathMartin, Mauritsmaartendejong, Mckaysalisbury, Michael Hardy, Midgley, Mindmatrix, MisterSheik, Mitchan, Mpolonot, NavarroJ, Nbarth, NoblerSavager, Ort43v, Oskar Sigvardsson, Patrick, PaulTananbaum, Pleasantville, Radagast3, RexNL, Rjwilmsi, Roy.ba, Samohyl Jan, Sanya, Sartak, Smyth, Stephenbez, StevePowell, TakuyaMurata, Template namespace initialisation script, Tingga, Trevor MacInnis, Trovatore, Twri, Vonkje, Watcher, Wile E. Heresiarch, Wren337, Zaslav, Zat'n'kel, Zero0000, 87 anonymous edits

Computer representations of graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=619783663> *Contributors:* 31stCenturyMatt, A Aleg, Aaronzat, Aizquier, Alink, Andreas Kaufmann, Any Key, AvicAWB, Avoided, Bg9989, Bluebus, Booyabazooka, Bruyninc, C4Cypher, Chochop, Chrisholland, Cooldudefx, Cubancigar1, Cybercoba, DRAGON BOOSTER, David Eppstein, Dcoetze, Dyspersia, Electro, EmilJ, Epimetheus, Excirial, FedericoMenaQuintero, Gallando, Giftlite, Gmelli, Graphicalx, Gvanrossum, Hariva, Hobsonlane, Joe Decker, Jojib fb, Jon Awbrey, JonHarder, Jorge Stolfi, Juliancolton, Kate4341, Kazuobn, Kbrose, KellyCoinGuy, Kendrich Hang, Klothor, Kristjan.Jonasson, Labraun90, Liao, Maria kishore, Max Terry, NerdysNK, Nmz787, Obradovic Goran, Ovi 1, P0nc, Pbirmie, Pieleric, QuiteUnusual, R. S. Shaw, RG2, Rabberski, Rborrego, Rd232, Rhanekom, Ruud Koot, Sae1962, Saimhe, Salix alba, ScNewcastle, ScaledLizard, SimonFuhrmann, Simonfairfax, SiobhanHansa, Skippydo, Spydermanga, Steven Hudak, Stphung, Tflot, Timwi, Tyir, UKoch, Zoicon5, ZorilloIII, ZweiOhren, 150 anonymous edits

Adjacency list *Source:* <http://en.wikipedia.org/w/index.php?oldid=621644718> *Contributors:* Andreas Kaufmann, Ash211, Aviggiano, Beetstra, Bobbertface, Booyabazooka, Chimarkine, Chmod007, Chris the speller, Cobi, Craig Pemberton, David Eppstein, Dcoetze, Dessaya, Dminkovsky, Dtbullock, Dysprosia, Fgnievinski, Fredrik, Garyzx, Giftlite, Hariva, Hobsonlane, Iridescent, Jamelan, Justin W Smith, Jwpurple, Kku, Krazelke, MathMartin, Michael Hardy, NSR, Oleg Alexandrov, Only2sea, Patmorin, Qwertys, Rdude, Ricardo Ferreira de Oliveira, SPTWriter, Sabrinamagers, Schneelocke, Serketan, ThE cRaCkEr, Twri, Velella, Venkatraghavang, 31 anonymous edits

Adjacency matrix *Source:* <http://en.wikipedia.org/w/index.php?oldid=599466571> *Contributors:* Abdull, Abha Jain, Aleph4, Arthouse, AxelBoldt, Beetstra, BenFrantzDale, Bender235, Bender2k14, Bitsianshash, Bkell, Booyabazooka, Burn, Calle, Chburnett, Chnopodiaceous, David Eppstein, Dcoetze, Debresser, Dreadstar, Dysprosia, ElBenevolente, Felix Hoffmann, Fgnievinski, Fredrik, Garyzx, Gauge, Giftlite, Headbomb, Hu12, JackSchmidt, JamesBWatson, Jean Honorio, Jokers, John of Reading, Jpbownen, Juffi, Kneufeld, Kompot 3, LokiClock, MarkSweep, Mate2code, MathMartin, Matěj Grabovský, Mbogelund, Mdrine, Michael Hardy, Miym, More, Natlewis, Oleg Alexandrov, Olenielsen, Only2sea, Patmorin, Paulish, Periergeia, Phils, Reina riemann, Rgbboer, Rhett, Rich Farmbrough, RobinK, SPTWriter, Salgueiro, Schneelocke, Senfo, Shipli4560, Snetff, Snowcream, Squids and Chips, Slawomir Bialy, TakuyaMurata, Tamfang, Thackstr, Tim Q. Wells, Timendum, Tomo, Tomruen, Ttzz, Twri, Wcherow, X7q, Yoav HaCohen, YuryKirienko, Zaslav, ۹۸۰, ۸۶ anonymous edits

Implicit graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=607161263> *Contributors:* Andreas Kaufmann, Cybercobra, David Eppstein, Michael Hardy, Olsonist, PhnomPencil, Rjwilmsi, Twri, 2 anonymous edits

Depth-first search *Source:* <http://en.wikipedia.org/w/index.php?oldid=615134410> *Contributors:* A5b, ABCD, Aadnan.tufail, Andre Engels, Antiuser, Apalsola, Apanag, Atkins, Bbi5291, Beetstra, BenRG, Braddwinter, Bryan Derksen, Bubba73, Cesarc, Citrus538, Clacker, Craig Barkhouse, Cst17, Curtmack, DHN, David Eppstein, Davideoth, Dcoetze, Dllu, Dreske, Duncan Hove, Dysprosia, EdiTOr, EdLee, ErikvanB, Fkodama, Fraggle81, Frecklefoot, Frizzil, G3moon, Giftlite, Gcschoryu, Gurch, Gurpreetkaur88, Gustavb, Hansamurai, Hemanshu, Hermel, Itai, Ixfd64, Jaredwf, Jef41341, Jeltz, Jerf, Jnl, Jonon, Jtle515, Justin Mauger, Justin W Smith, Kate, Kdau, Kesla, Koertea, Kromped, LittleDan, Mark viking, MarkSweep, Marked, Martynas Patasius, Materialscientist, McKay, Mcrypt, Mentifisto, Michael Hardy, Miknight, Mild Bill Hiccup, Miles, Mipadi, Miym, Moosebumps, MusicScience, Musiphil, Nizonstolz, Nuno Tavares, Nvrmnd, Patmorin, Pmueller, Poor Yorick, Pukeye, Qrancik, Qwertys, Qx2020, Readytohelp, Regnaron, Rich Farmbrough, Rick Norwood, Rror, Rtcasey, RXS, Ryuuunoshounen, SPTWriter, Shuro, Smallman12q, Snowolf, Srrgei, Staszek Lem, Stefan, Steverapaport, Stumpy, Subbh83, Svick, Tauwasser, Taxipom, Thegeneralguy, Thesilverbail, Thumperward, TimBentley, Vroo, Vsh3r, Waldir, Wavelength, Yonestar, Yuide, Z10x, 256 anonymous edits

Breadth-first search *Source:* <http://en.wikipedia.org/w/index.php?oldid=618525331> *Contributors:* 28bytes, A5b, Adam Zivner, Akmenon, AllenDowney, Amelio Vázquez, Andre Engels, Antkro, Arkenflame, Asp-GentoLinux, Bazonka, BenFrantzDale, Bender05, Bender250, Bkkbrad, BlueNovember, Bmatheby, Bonadea, Braindrain0000, CBM, Cesarc, Cheethdj, Clacker, Corti, Cscott, CyberShadow, Cybercobra, DHN, Davepape, David Eppstein, DavidCary, Davideoth, Dbeatty, Dcoetze, Dedicatedecoy, Dianna, Dileep98490, Dysprosia, Earobinson, Ec29, Edaelon, Elyazalmahfouz, Erdogany, EricTalevich, ErikvanB, Ffaarr, Fkodama, Folletto, Franken44, Frizzil, Galaxiaad, Gdr, GeeksHaveFeelings, Gelwood, GeorgeBills, Giftlite, Gilliam, Headbomb, Hmwith, IRP, J Di, Jacobkwitkoski, Jan Winnicki, Jbellessa87, Jerf, Jldwig, Jgloran, Justin W Smith, Kdau, Kdnewton, Kenyon, Kesla, Kiand, Kostmo, LOL, Lemontea, LittleDan, Liuofficial, Lixino, Loading, Lvsmart, M.aznaveli, Madmardigan53, Mahue, MarkSweep, Martynas Patasius, Matt.smart, Meld, Mentifisto, Michael Hardy, Michael Sloane, Mia, Miym, Mmtux, Mrwojo, Musiphil, Mxn, Night Gyr, Nikkimaria, Nisargatanna, NotAnonymous0, Nuno Tavares, Nvrmnd, Oblatenhaller, Ott2, P b1999, Patmorin, Pcap, Philip Trueman, Phoe6, Polveroj, Pome, Poor Yorick, Qrancik, Qwertys, Ragzouken, Regnaron, RobinK, Robost, Saurabhbnanerji, Secfan, Smallman12q, Smij, Sonett72, Spoon!, Staszek Lem, Steverapaport, Szabolcs Nagy, TWiStErRob, TakuyaMurata, Thesilverbail, Timwi, Tooto, Tracytheta, Valery.vv, Veleta, VictorAnyakin, Viebel, W3bbo, Wtmitchell, Xodarap00, Zad68, Тиверополник, 302 anonymous edits

Lexicographic breadth-first search *Source:* <http://en.wikipedia.org/w/index.php?oldid=581598754> *Contributors:* A3 nm, Arpi Ter-Araqelyan, Bob5972, David Eppstein, Headbomb, 6 anonymous edits

Iterative deepening depth-first search *Source:* <http://en.wikipedia.org/w/index.php?oldid=608372065> *Contributors:* AP Shinobi, Arabani, Arvindn, BenFrantzDale, Bryan Derksen, Buptcharie, Cesarc, CharlesGillingham, Codercrux, Derek Ross, Doug Bell, Dysprosia, Emrys, ErikvanB, Firenu, Frikle, GHemsley, Haoyo, Harshaljahagirdar, Ilmari Karonen, JMCorey, Jafet, Jamelan, Kesla, Malcohol, Mariam Asatryan, Marked, Metaxal, Michael Sloane, Msmith0957, Nbarth, Pgarn002, Pipasharto, PoisonedQuill, Quadrescence, Qwertys, Regnaron, Rjwilmsi, Rubenlagus, Solberg, Styler, Thesisilverbail, Wolfkeeper, 42 anonymous edits

Topological sorting *Source:* <http://en.wikipedia.org/w/index.php?oldid=623226071> *Contributors:* A3 nm, Abaumgar, Aeons, AlainD, Apanag, Armine badalyan, Arvindn, Atlantia, BACBKA, Baby112342, BjarneH, Bluegrass, Bosmon, Branonn, CJLL, Wright, Captainfranz, Churnett, Cleet77, Charles Matthews, Chenopodiaceous, Chub, Craxic, Cs77, Cybercobra, David Eppstein, Dcoetze, Delaszk, DmitiTrix, Dmitry Dzhuz, DomQ, Esmond.pitt, Fresheneesz, Gmaxwell, Greennrd, GregorB, Grimboy, Guray9000, Gurt Posh, Hannes Eder, Headbomb, Imnotminkus, Jarble, Jim Huggins, John of Reading, Johnwon, JokesFree4Me, Jonsafari, Joy, Karada, Kenyon, Kingpin13, Kristjan.Jonasson, Leland McLemes, Maneeshsinha, Mark viking, Matthew Woodcraft, Mggreenbe, Michael Hardy, Mikofski, Miym, Mordomo, Nithns43, Obradovic Goran, Oliphanta, Oskar Sigvardsson, Pascal666, Pasixxx, Pekinisn, Planetscape, Quarl, Qwertys, Rattatosk, RazorICE, RobinK, Slaniel, Stumpy, Sundar, Surturz, TSornalingam, Taxipom, Tobias Bergemann, TomasRiker, Unbitwise, Uranographer, Vonbrand, Vromascanus, Workaphobia, Zundark, ىل، 109 anonymous edits

Application: Dependency graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=620063135> *Contributors:* Abdull, Antonielly, Bryan.burgers, ChrisDailey, Colin Greene, Cthulhu88, David Eppstein, Dmcreeary, Edward, Ego White Tray, GrAfIT, Greenrd, Headbomb, Ibasota, Jarble, Jason Quinn, Linas, Niceguyedc, Quenhitran, SAE1962, Sam Pointon, SummerWithMorons, Tetha, Twri, Vadmium, 24 anonymous edits

Connected components *Source:* <http://en.wikipedia.org/w/index.php?oldid=612040537> *Contributors:* Alansohn, Arlekean, Arthena, Asturius, AxelBoldt, DPoon, DavePeixotto, David Eppstein, Dcoetze, Dicklyon, Dricherby, Edemaine, Eggwadi, Fæ, Giftlite, Graue, Headbomb, Hermel, Igor Yalovecky, Jhbdel, JoergenB, Jpbown, Kostmo, LokiClock, Mashiah Davidson, Meekohi, Mikaeay, Msh210, PerryTachett, Pklosek, R'n'B, RichardVeryard, Ross m mcconnell, Rswarbrick, Rxxtreme, Shadowjams, Staszek Lem, Stdazi, Subversive.sound, Titodutta, Twri, Wavelength, Wcherowi, Zundark, 33 anonymous edits

Edge connectivity *Source:* <http://en.wikipedia.org/w/index.php?oldid=622757810> *Contributors:* AmirOnWiki, Booyabazooka, CBM, David Eppstein, Flamholz, Headbomb, Inozem, Justin W Smith, KD5TVI, Larsborn, Mgcl, Mike Fikes, R'n'B, Radagast3, Sjcoosten, Sopoforic, WikHead, 10 anonymous edits

Vertex connectivity *Source:* <http://en.wikipedia.org/w/index.php?oldid=620487398> *Contributors:* A3 nm, David Eppstein, Dcoetze, Douglas R. White, Giftlite, Headbomb, Justin W Smith, McKay, Mike Fikes, Miym, Nonenmac, Radagast3, Ratfox, Rjwilmsi, Sopoforic, Zynwyx, 12 anonymous edits

Menger's theorems on edge and vertex connectivity *Source:* <http://en.wikipedia.org/w/index.php?oldid=610376017> *Contributors:* A3 nm, Arthena, Asiantuntja, BeteNoir, Bigbluefish, Blankfaze, Charles Matthews, Cykerway, David Eppstein, Delaszk, Douglas R. White, Dricherby, Giftlite, Headbomb, Heinzi.at, Jason Quinn, Kope, MathMartin, Michael Hardy, Nonenmac, Oleg Alexandrov, Prsephone1674, RDBury, Rjwilmsi, Schmittz, Seb35, Silverfish, Stewbasic, 8 anonymous edits

Ear decomposition *Source:* <http://en.wikipedia.org/w/index.php?oldid=607874171> *Contributors:* Colonies Chris, David Eppstein, Fy sk, Michael Hardy, Rjwilmsi, Tmigler, 8 anonymous edits

Algorithms for 2-edge-connected components *Source:* <http://en.wikipedia.org/w/index.php?oldid=620486584> *Contributors:* Algebraista, At-par, BearMachine, Bedwyr, Booyabazooka, Creidieki, DaedalusInfinity, DarkMarsSasha, David Eppstein, Dcoetze, El Roih, Flyrev, Giftlite, GregorB, Headbomb, Herr Satz, Igor Yalovecky, Joshuancooper, Kerrick Staley, Klundarr, Leen Droogendijk, McKay, Mesoderm, Mjaredd, Phys, Purdygb, Rjwilmsi, Sun Creator, Vanish2, XJaM, Zaslav, 42 anonymous edits

Algorithms for 2-vertex-connected components *Source:* <http://en.wikipedia.org/w/index.php?oldid=621620575> *Contributors:* Bike756, David Eppstein, Dcoetze, Edemaine, Eggwadi, FactorialG, Frantic Ant, Giftlite, Goooopy, Joriki, Joshxyz, Justin W Smith, Keenan Pepper, Lunae, MahmoudHashemi, Maliberty, Martarius, MathsPoetry, McKay, Michael Hardy, Ott2, Rmashhadi, SteinbDJ, Tizio, Xezbeth, Zyqqh, 24 anonymous edits

Algorithms for 3-vertex-connected components *Source:* <http://en.wikipedia.org/w/index.php?oldid=616019689> *Contributors:* A3 nm, Andreas Kaufmann, Auntoff, Cybercobra, David Eppstein, Eng 104*, Harrigan, Headbomb, Kakila, Khazar2, Kintaro, LilHelpa, MaxEnt, Pol098, Riedel, Rjwilmsi, Twri, 5 anonymous edits

Karger's algorithm for general vertex connectivity *Source:* <http://en.wikipedia.org/w/index.php?oldid=610551971> *Contributors:* Bruyninc, ColdthroatEskimo, Daiyuda, David Eppstein, Duncan.Hull, Ecnernwala, GregorB, Headbomb, Hiiiiiiiiiiiiiiiiiiiiii, Hoonose, John of Reading, JohnI, Kilom691, Larry V, Luc4, Magioladitis, Monsday, Qwertys, Sadads, Sweet tea van, Thore Husfeldt, Waffleguy4, Zackchase, 19 anonymous edits

Strongly connected components *Source:* <http://en.wikipedia.org/w/index.php?oldid=621625327> *Contributors:* 4v4l0n42, Aaron McDaid, Andreas Kaufmann, Ascánder, BernardZ, Booyabazooka, Cyberjac, DPoon, David Eppstein, Dcoetze, Giftlite, Gioto, Hairy Dude, Headbomb, Hvn0413, Integr8e, Jamie King, Jpbown, Justin W Smith, Martin Bravenboer, Mashiah Davidson, Matt Crypto, Nutcracker2007, Oleg Alexandrov, Plbogen, Pypmanneltjes, Qleariver, Quakor, R'n'B, RichardVeryard, Rjwilmsi, Sho Uemura, Silverfish, Sleske, Sytelus, Thijsswijs, Twri, 38 anonymous edits

Tarjan's strongly connected components algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=623882941> *Contributors:* Amanjain110893, Andreas Kaufmann, Arthas702, BlueNovember, CBM, Chadhutchins10, David Eppstein, Dcoetze, Digital Organism, Duplicit, Eclecticos, Flyer22, Ghazer, Giftlite, Grubber, Haboud1, Headbomb, Jedaily, Johan.de.Ruiter, Kejia, Kejv2, LilHelpa, Loufranco, MattGiua, Michael Veksler, Musiphil, Neilc, Ott2, Rich Farmbrough, Short000, Stephen70edwards, Svick, Swarmendu.biswas, Sytelus, Tharwen, VirtualDemon, Zitronenquetscher, 85 anonymous edits

Path-based strong component algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=610043696> *Contributors:* Andreas Kaufmann, Darguz Parsilvan, David Eppstein, DavidCBryant, LOL, Nutcracker2007, Oleg Alexandrov, Rjwilmsi, Ruud Koot, Vadmium, WPPatrol, 4 anonymous edits

Kosaraju's strongly connected components algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=587504275> *Contributors:* AmirOnWiki, Andre.bittar, Boggle3343, David Eppstein, Dcoetze, Emilkeyder, Favonian, Forbsey, Fridayda13, Funandtrvl, Giftlite, Headbomb, Justin W Smith, Matt Crypto, NickLewycky, Octahedron80, Omnipaedista, Phatsphere, Playswithfire, Prashmohan, Qwertys, RobinK, Ruud Koot, Stifle, Svick, 20 anonymous edits

Reachability *Source:* <http://en.wikipedia.org/w/index.php?oldid=606347771> *Contributors:* Brick Thrower, BrideOfKripenstein, CBM, David Eppstein, Delicious carbuncle, Devanden, Eecs574group6, Furby100, Gbint, Harrigan, LilHelpa, Matt Cook, Mccaskey, PaulTanenbaum, Qetuth, R'n'B, Rjwilmsi, Rp, Skier Dude, TheEternalVortex, TheNightFly, Twri, Verticalgroup, 4 anonymous edits

Transitive closure *Source:* <http://en.wikipedia.org/w/index.php?oldid=616496963> *Contributors:* AL SAM, AmirOnWiki, Artdadam0, Arthur Rubin, Awaterl, Bender2k14, Bgwhite, Borislav, CBM, CBM2, CRGreathouse, Charles Matthews, Classicaleton, Creidieki, Danwizard208, David Eppstein, Dcoetzee, Dmitri L. Slabk., Dreadstar, DuaneLAnderson, Fropuff, Gifflite, Girlwithglasses, Gregbard, Henry Delforn (old), JRSpriggs, Jamelan, Joriki, Kirtag Hratiba, KnightRider, Loadmaster, LungZeno, Lyonsam, Matt Crypto, Mhss, Michael Hardy, MountainGoat8, NeilFraser, Neonfreon, Obradovic Goran, Patrick, PaulTanenbaum, Plasticphilosopher, Plustgarten, Populus, Quickwik, Salix alba, Sdrucker, ShelfSkewed, Tayste, Tijfo098, Timwi, Tobias Bergemann, Tomaxer, Vunkac, Vpieterse, Wizeguytristram, Yavoh, 29 anonymous edits

Transitive reduction *Source:* <http://en.wikipedia.org/w/index.php?oldid=602726846> *Contributors:* A3 nm, Algebraist, Arthur Rubin, CRGreathouse, Charles Matthews, Cuaxdon, David Cooke, David Eppstein, Dcoetzee, EmilJ, Gifflite, Greenrd, Lyonsam, Michael Hardy, Nbarth, PaulTanenbaum, Rjwilmis, Rp, Salix alba, W1r3d2, 16 anonymous edits

Application: 2-satisfiability *Source:* <http://en.wikipedia.org/w/index.php?oldid=621346870> *Contributors:* Aleph4, Andris, Baumbaron, Booyabazooka, Bsilverthorn, C. lorenz, CBM, Chalst, Charles Matthews, CharlotteWebb, Chayant, Creidieki, David Eppstein, Dcoetzee, EagleFan, EdH, Fratrep, GoingBatty, Gregbard, Headbomb, Hermel, Igorpak, Imranaf, Jacobolus, Leibniz, Magioladitis, MarkusQ, Mboverload, Mets501, Miym, Niteowlneils, Oliphant, Pushpendera, Rjwilmis, RobinK, Schnellocke, SiddMahan, Sun Creator, Tijfo098, Twin Bird, Vegaswikan, Wavelength, Yaronf, Ylloh, Zundark, 22 anonymous edits

Shortest path problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=622752068> *Contributors:* Aaron Rotenberg, Alcidesfonseca, Alexander Anoprienko, Alfredo J. Herrera Lago, Altenmann, Amine.marref, Andreas Kaufmann, Andris, Anomie, Artyom Kalinin, AxelBoldt, B4hand, BenFrantzDale, Booyabazooka, Brian0918, C. lorenz, Caesura, Cairomax, Camw, Choeß, Chris Capoccia, Cipher1024, Cthe, Daveagg, David Eppstein, Dcoetzee, Deanshp, Delaszk, Denisarona, Devis, Dmforci, Download, Dysprosia, ESKog, Edepot, Essess, Fgnievinski, Fæ, Gaius Cornelius, General Wesc, Gifflite, Glrx, Graph Theory page blanker, Happyuk, Hari6389, Hariva, Headbomb, JMP EAX, Jarble, Jason.surratt, JeLuF, Jochen Burghardt, Justi W Smith, Kbrose, Kope, LC, Lourakis, Lpgeffen, Lqs, Lzur, MadScientistVX, Marc van Leeuwen, Marcelkcs, Mathiasstck, Mcld, Metaprimer, Michael Hardy, Mindmatrix, MiroBrada, MisterSheik, Mjs1991, Mkroeger, MoreNet, Nethgirb, Nullzero, Nuno Tavares, Ohnoitsjamie, Oliphant, OverlordQ, Phelanpt, Piginorsch, Pmlineditor, Qwertys, Rasmusdf, Rich Farmbrough, Rjwilmis, Robert Geisberger, RobinK, Robmcoll, RomanSpa, Ruud Koot, Shd, Sho Uemura, Squire55, Super48paul, Suzhouhuwuye, Taemyr, Templatetypedef, TerraFrost, The Anome, Tommyjb, ToneDaBass, Tremsill, Wolfkeeper, Xperimental, Yacs, 129 anonymous edits

Dijkstra's algorithm for single-source shortest paths with positive edge lengths *Source:* <http://en.wikipedia.org/w/index.php?oldid=624621426> *Contributors:* 4v4l0n42, 90 Auto, Ajim, Abu adam, Adamarnesen, Adamianash, AgadaUrbanit, Agthorr, Ahyl1, Airlacorn2, Aladdin.chettouh, AlanUS, Alanb, AlcoholVat, Alex.mccarthy, Allan speck, Alquantor, Altenmann, Alyssaq, Amenel, Andreasneumann, Angus Lepper, Anog, Apalamarchuk, Aragorn2, Arjun G. Menon, Arrenlex, Arsstyleh, AxelBoldt, Aydee, B3virq3b, B6s, BACbKA, B4'4, Bcnof, Beatle Fab Four, Behco, BenFrantzDale, BernardZ, Bgwithe, Bkell, Blueshining, Boemanneke, Borgx, Brona, CGamesPlay, CambridgeBayWeather, Charles Matthews, Chehabz, Choess, Christopher Parham, Cicconetti, Cincutprabu, Clementi, Coralnimzu, Crazy george, Crefrog, Css, Csurgine, Ctppc, Cyde, Danmaz74, DarrylNester, Daveagg, Davekaminski, David Eppstein, Davub, Dcoetze, Decrypt3, Deflective, Diego UFCC, Digwuren, Dionyzia, Dmforci, Dmitrif66, DorisSmith, Dosman, Dougher, Dr.Koljan, Dreske, Drostic, Dudzcom, Dysprosia, Edemaime, ElonNarai, Erel Segal, Eric Burnett, Esrogs, Ewedistrict, Ezrakily, Ezubaric, Faure.thomas, Foobaz, Foxj, FrankTobia, Frankrod44, Frap, Fresheneesz, GRuban, Gaiacarra, Galoubet, Gauravxpress, Gerel, Geron82, Gerrit, Gifflite, Gordonnovak, GraemeL, Graham87, Grantstevens, GregorB, Guanaco, Gutza, Hadal, Haham hanuka, Hao2lian, Happyuk, Harish victory, Hell112342, HereToHelp, Harry2, Huazheng, I am One of Many, Ibmu, IgushevEdward, IkuusameFan, Illnab1024, Iridescent, Itai, JBocco, JForget, Jacobolus, Jarble, Jaredwf, Jason.Rafe.Miller, Jellyworld, Jeltz, Jewillico, Jheiv, Jim1138, Jochen Burghardt, Joelimlimit, JohnBlackburne, Jongman.koo, Joniscool98, Jorvis, Julesed, JuliusJ. Gonera, Justin W Smith, K3rb, Kbkb, Kejiana, Kesla, Kh naba, King of Hearts, Kku, Kndiaye, Kooo, Kostmo, LC, LOL, Laurinkus, Lavaka, Lawpj, Leonard G., Lone boatman, LordArtemis, LunaticFringe, Mahanga, Mameisam, MarkSweep, Martynas Patasius, Materialscientist, MathMartin, Mathiasstck, MattGiua, Matusz, Mccraig, Mcculley, Megharajv, MementoVivere, Merlion444, Mgreebne, Michael Hardy, Michael Sloane, Mikeo, Mikrosian Akademija 2, Milcke, MindAfterMath, Mkws13, Mr.TAMER.Shlash, MrOllie, Muon, MusicScience, Mwarren.us, Nanobear, NetRoller 3D, Nethgirb, Nixdorf, Noogz, Norm mit, Obradovic Goran, Obscurans, Oleg Alexandrov, Oliphant, Olivernina, Optikos, Owen, Peasaep, Peatar, PesoSwe, Pilode, Pijo, Pol098, PoliticalJunkie, Possum, ProBoj!, Pseudomonas, Pshanka, Pskjs, Pxtermre75, Quenhtiran, Quidquam, RISHARTHA, Radim Bacá, Rami R, RamiWissa, Recognizance, Reliableforever, Rhanekom, Rjwilmis, Robert Southworth, RodrigoCamargo, RoyBoy, Ruud Koot, Ryan Roos, Ryangerard, Ryanli, SQGibbon, Sambyless, Sarkar12, Scorintha, Seanhan, Sephiroth storm, Shd, Sheepatgrass, Shizny, Shuroo, Sidonath, SiobhanHansa, Sk2613, Slakr, Smied, Sokari, Someone else, Soytuny, Spencer, Sprhodes, Stdazi, SteveJothen, Subh83, Sundar, Svick, T0ljan, Tehwikipwnerer, Tesse, Thayts, The Arbitr, TheRingess, Thijswijs, Thom2729, ThomasGHenry, Timwi, Tobei, Tomisti, Torla42, Trunks175, Turketwh, VTBassMatt, Vecrumba, Vellela, Venkataram95, Vevek, Watcher, Wavelength, Wierdy1024, WikiSlasher, Wikipelli, Wildcat dunny, Williamyf, Woshiqiqye, Wphamilton, X7q, Xerox 5B, Ycl6, Yujianzhao, Yutsi, Yworo, ZeroOne, Zhaladshar, Zr2d2, 673 anonymous edits

Bellman–Ford algorithm for single-source shortest paths allowing negative edge lengths *Source:* <http://en.wikipedia.org/w/index.php?oldid=624609501> *Contributors:* Aaron Rotenberg, Abednigo, Aednichols, Aene, Agthorr, Aladdin.chettouh, AlexCovarrubias, Altenmann, Anabus, Andris, Arlekean, B3virq3b, Backslash Forwardslash, BenFrantzDale, Bjozen, Bkell, BlankVerse, Brona, Brookie, CBM, Carlwitt, Charles Matthews, CiaPan, Ciphergoth, David Eppstein, Dcoetze, Docu, Drdevil44, Ecb29, Enochlae, Eprb123, Ferengi, FrankTobia, Fredrik, Fvw, Gadfium, Gauravxpress, Gifflite, GregorB, Guahnala, Happyuk, Headbomb, Heineman, Helix4, Iceblock, Instanton, Itai, J.delanoy, Jamelan, Jaredwf, Jason.Rafe.Miller, Jellyworld, Josteinaj, Justin W Smith, Konstable, LOL, Lavl7, Linket, Lone boatman, Magioladitis, Mario77Zelda, Mathimike, Mazin07, Mcld, Michael Hardy, Miym, Monsday, N Shar, Narozu, Nihiltres, Nils Grimsmo, Nostalgius, Nullzero, Orbst, P b1999, PanLevan, Pion, Pjrm, Poor Yorick, Posix4e, Proton, Pskjs, Quuxplusone, Qwertys, Rjwilmis, RobinK, Rspeer, Ruud Koot, SQL, Salix alba, Sam Hocevar, Shuroo, Sigkill, Skizzik, Solon.KR, SpyMagician, Stdazi, Stderd, Stern, Str82n01, Tomo, ToneDaBass, Tsunanet, Tvidas, Ucucha, Waldir, Wavelength, Williamyf, Wmahan, Writer130, Zholadas, 205 anonymous edits

Johnson's algorithm for all-pairs shortest paths in sparse graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=564595231> *Contributors:* Abtinb, AlexTG, Algebra123230, Altenmann, Bob5972, Brahle, Charles Matthews, Cyrus Grisham, Danielx, David Eppstein, Drilnoth, Gaius Cornelius, Jarble, Josteinaj, Karl-Henner, Kl4m, MarkSweep, MicGrigni, MorganGreen, Nanobear, Netvor, Octahedron80, Ruud Koot, Twexcom, Welsh, 23 anonymous edits

Floyd–Warshall algorithm for all-pairs shortest paths in dense graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=622593242> *Contributors:* 16@r, Aenima23, AlanUS, AlexandreZ, Algebra123230, Altenmann, Anharrington, Aureooms, Barfooz, Beardybloke, Buagg, C. Siebert, CBM, Cachedio, CiaPan, Closedmouth, Cuijmpier, Daveagg, Davekaminski, David Eppstein, Dcoetze, Dfrankow, Dittymathew, Donhalcon, DrAndrewColes, Exercisephys, Fatal1955, Fintler, Frankrod44, Gaius Cornelius, Gifflite, Greenleaf, Greenmatter, GregorB, Gutworth, Harrigan, Hemant19cse, Hjffreyer, Intgr, J.Finkelstein, JLaTondre, Jarble, Jaredwf, Jellyworld, Jerryobject, Jixani, Jochen Burghardt, Joy, Juancarlosgolos, Julian Mendez, Justin W Smith, Kanitani, Kenyon, Kesla, KitMarlow, Kletos, Leycey, LiDaobing, Luv2run, Mac1421, Magioladitis, Makyen, MarkSweep, Martinkunev, Md.raftabuddin, Mellum, Michael Hardy, Minghong, Mini-Geek, Minority Report, Mqchen, Msg555, Mwk soul, Nanobear, Netvor, Nicapicella, Nishanthj, Nneonnie, Nostalgius, Obradovic Goran, Oliver, Opium, Pandemias, Phil Boswell, Pilotguy, Pjrm, Polymerbringer, Poor Yorick, Pvza85, Pxtermre75, Quuxplusone, Qwertys, Rabarberski, Raknarf44, RobinK, Roman Munich, Ropez, Roseperrone, Ruud Koot, Sakanarm, SchreiberBike, Shadowjams, Shmor, Shnownflake, Shyamal, Simoneau, Smurfix, Soumya92, Specs112, Sr3d, Stargazer7121, SteveJothen, Strainu, Svick, Taejo, Teles, Thomasda, Tháí Nho, Trappist the monk, Treyshonuff, Two Bananas, Volkan YAZICI, W3bb0, Wickethewok, Xyzzy n, Yekaixiong, 257 anonymous edits

Suurballe's algorithm for two shortest disjoint paths *Source:* <http://en.wikipedia.org/w/index.php?oldid=584648111> *Contributors:* David Eppstein, Dekart, Mcld, RobertoCfc1, Tedder, Vegaswikan, Vladhed, 9 anonymous edits

Bidirectional search *Source:* <http://en.wikipedia.org/w/index.php?oldid=598221479> *Contributors:* Chris the speller, Cobi, David Eppstein, Ddccc, Delirium, Jamelan, Jarble, JohnBlackburne, Mark viking, Mcld, Michael Devore, Mohammadali69, NathanHurst, Peatar, Quuxplusone, Qwertys, Regnaron, Xavier Combelle, Xbao, 25 anonymous edits

A* search algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=624151540> *Contributors:* 1ForTheMoney, 316 student, Abraham, B.S., Acdx, Aedieder, Ahoerstemeier, Albertzeyer, Ale jrb, Alejandro.isaza, Alex Krainov, Alex.mccarthy, AlexAlex, Alexander Shekhovtsov, AllanBz, Altenmann, Amcshane, Andrewrosenberg, Aninhumer, Antonbarkamsan, Arrandale, Arronax50, Auric, AxelBoldt, Axule, BACbKA, BIS Suma, Blesham, Beau, BenKovitz, Benstown, Bjorn Reese, Blogjack, Bojanestorovic, Boom1234567, Braphael, Brona, C. A. Russell, C7protal, CaroleHenson, Catskul, Cedar101, Cesarramsan, Chaos5023, Charles Matthews, Cherkash, Chire, CountingPine, Cycling-professor, DVDM, Damian Yerrick, Daveagg, David Chouinard, DavidHarkness, Dcoetze, Ddcce, Dionyzia, Disavian, Dmysternsbull, Docu, Dr.Koljan, Drb03ts, Dreske, Dysprosia, Electro, Excirial, Eyal0, Falsedef, Faure.thomas, FelipeVargasRigo, Fiam, Frasmog, Frecklefoot, Fredrik, Fresheneesz, Furykef, GPHemsley, Gaganbansal123, Geoffadams, George126, Ghodnsia, GiM, Gifflite, Glenstamp, Gmental, Grantstevens, GregorB, Gulliveig, Gökhán, HMSSolent, HXZBZSHDHDHED, Hadal, Hallows AG, HandsomeFella, Hart, Headbomb, HebrewHammerTime, Henke37, Hiihammuk, Hodja Nasreddin, Hornbydd, IMSoP, IanOsgood, JCcarriker, JLaTondre, JackSchmidt, Jacob Finn, Jamesfisher, JingguoYao, Jiri Pavlek, Jiyeyuran, JII, Joelp, JohnBlackburne, Johnjianfang, JonH, JoshReeves2, Joshua Issac, Josve05a, Jrouque, Julesed, Justin W Smith, JustinH, Kainino, Kaisal, KamuiShirou, Katiech5584, Keenan Pepper, Keithphw, Keo-san, Kesla, Kevin, Khanser, Kingpin13, Kku, Kndiaye, Korrawi, Kotniski, Kri, Laurens, Lee J Haywood, Leirbag.arc, Lotje, Lyn240690234, MIT Trekkie, Mac1421, Magioladitis, Malcolm, Manishi181192, Markulf, Martyr2566, Mat-C, MattGiua, Mav, Mcculley, Mcld, Mellum, Mhaward999, Michael Sloane, Millerl, Mintleaf, MithrandirAgain, Mohit05011992, Mortense, Mrwojo, Muhends, Nanobear, Naugtur, Neil, Neil, Nick Levine, Nohat, Orca456, Peatar, Phil Boswell, Piano non troppo, PierreAbbat, Pjrm, Pratyaya Ghosh, Quuxplusone, Qwertys, Raaghru03, Rankiri, Rdsmith4, Regnaron, Remko, Rich Farmbrough, Ripe, Ritchy, Rjwilmis, RoySmith, Rspeer, Rufous, Runtime, Ruud Koot, Salzahran, Samkass, Sander17, Sigmundur, Silnarm, Simeon, SiobhanHansa, Siroxo, SkyWalker, Sriharsh1234, Stephenb, SteveJothen, Subh83, Svick, Sylvrene, Syrhiss, Szabolcs Nagy, Tac-Tics, Taejo, Talldean, Tassedethe, Tekhnofied, TheGoblin, Theemathas, Thomas.W, Thsgrn, Timotei21, Timwi, Tobias Bergemann, Tomp, Trappist the monk, Treaster, TrentonLipscomb, Trisweb, Trudslev, Vanished user 04, VernoWhitney, Wavelength, Whosyourjudas, Wldr, WillUther, Xnk, Yintan, Yknott, Yuval Baror, ZeroOne, 458 anonymous edits

Longest path problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=602231139> *Contributors:* Andreas Kaufmann, C.Fred, David Eppstein, Dcoetze, Delaszk, Dfarrell07, Discospinster, Djbwiki, DragonLord, Enricorp, Eraserhead1, Gifflite, GregorB, Headbomb, Ivan Štambuk, Jitse Niesen, John Cline, Kyledalefrederick, LukeJacksonbn, Michael Hardy, Polyguo, Reyk,

RobinK, S.zahiri, Seaphoto, Sidgalt, Skysmith, Some jerk on the Internet, Tetha, Thore Husfeldt, 32 anonymous edits

Widest path problem Source: <http://en.wikipedia.org/w/index.php?oldid=607760350> Contributors: Baiyubin, Daveagp, David Eppstein, Giftlite, John of Reading, KConWiki, Michael Hardy, Ost316, R'n'B, Rjwilmsi, Volkan YAZICI, Woohookitty, 1 anonymous edits

Canadian traveller problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=607341374> *Contributors:* Algebraist, Andreas Kaufmann, C_lorenz, Cybercobra, Dominus, Draaglom, Erik Warmelink, Giftlite, Haeinous, Mastergreg82, Mathew5000, Michael Hardy, Reyk, Rjwilmsi, RobinK, Twri, Ukexpat, Vanish2, Welsh, Woohookitty, Zahy, 10 anonymous edits

K shortest path routing *Source:* <http://en.wikipedia.org/w/index.php?oldid=616259651> *Contributors:* Anna Frodesiak, Ccalmen, David Eppstein, Demiurge1000, Dewritech, Dreamyshade, Huon, Jflabourdette, Michael Hardy, Nathan2055, Navneet Singhvi, SchreiberBike, TBrandley, Toyopod100, 5 anonymous edits

Application: Centrality analysis of social networks Source: <http://en.wikipedia.org/w/index.php?oldid=624133593> Contributors: Aftersox, BAxelrod, Bkkbrad, Borgatts, Cherkash, Chmod644, ChristophE, Compsin, DarwinPeacock, David Eppstein, Dbabbitt, Douglas R. White, Dvoina13, Eassin, EbrushERB, Edchi, Ergotius, Fabercap, Fgnievinski, Flyingspuds, Frozen4322, Gitfile, GlennLawyer, Harthus, Hathawayc, Headbomb, Icarins, J.delaney, JJ Harrison, Jmagdanz, Jonesey95, Kazastankas, Kku, Loodog, Maddendallybrokaw, Marlon'n'marion, Mdvs, Mgwalker, Michael Hardy, Michael Rogers, Minimac, Mrocklin, Mstuomel, Nburden, Netzwerkerin, Orubt, Piotrus, Psorakis, RafG, RedHouse18, Reedy, Rjwilmsi, Robd75, Rocchini, Ruud Koot, Sandal bandit, Seanandjason, Sepreece, Sevenp, Spariggio82, Stochata, Sundirac, Tgr, Thegzk, Tore.opsahl, Trumpsternator, Upulcranga, Utopiah, Vanisaac, Vigna, Vynez Xnebara, WaddSpoiley, Wavelength, Yeda123, 83 anonymous edits

Application: Schulze voting system *Source:* <http://en.wikipedia.org/w/index.php?oldid=623997866> *Contributors:* Angus, Apostmodernist, Argyriou, ArielGlenn, Arno Nymus, Asirbu, AugustinMa, Avij, AxelBoldt, Bayberrylane, Baylink, Bdesham, Benjamin Mako Hill, BorgHunter, Borgx, BrainMarble, CRGreathouse, Cacycle, Carn, CatyTony, Cedar101, Chealer, ClementDavid, Daveagp, David J Wilson, Deathphoenix, Dismas, Dissident, DixonD, Diza, Drbug, DuncanHill, Eequor, Electionworld, EmSamulili, Esrogs, Fahrenheit451, Falvkinge, Frafl, Gamsbart, Geni, Giflite, Gioto, Grasyp, GregorB, Götz, H2g2bob, Hairy Dude, Hermitage, Hippopotamus Logie, Ilmari Karonen, Insertrealname, Iota, IronMan3000, Izno, James McStub, Jamesx12345, Jguk, Joy, Juju2000, Liangent, Mailer diablo, MarkusSchulze, McCart42, Meno25, Michael Hardy, Mitchumch, Müdigkeit, Nethgirb, Nishkid64, Obersachse, Physicistjedi, Pmythet, Pot, Purodrha, Qaanol, R'B, Ricardopoppo, RobLa, Robert Loring, Robert P Sheas, Robert Philip Ashcroft, RodCrosby, S.Orrvarr, S, SEWilco, Sppley, Sanbec, Scott Ritchie, Scs, SeL, SimonD, Sjwheel, Skyfaller, Smartse, Stiddmat, TWCarlson, Tchof, The ed17, ThurnerRupert, Tmh, Tomruen, Touriste, Tpbradbury, TreasuryTag, Twilsonb, VBGFscJUn3, Vyznev Xnebara, Wat 20, Wingedsubmariner, Wli, Xeroc, Xmath, Yayay2bhere, Yayay, 194 anonymous edits

Borúvka's algorithm Source: <http://en.wikipedia.org/w/index.php?oldid=606040497> Contributors: Adam Zivner, Agamir, Alieseraj, Altenmann, Angr, Auringore, Captain Segfault, Cedar101, Charles Matthews, Cobi, David Eppstein, Davitf, Dcoeteet, Dysprosia, Elanguescence, Electriccatfish2, Falcon95, Jaredwf, Jk2q3jrkse, Jogloran, Kaeso, Kendrick7, Mgreenbe, Michael Sloane, Nkojuharov, PierreAbbé, Qwertyus, Rich Farmbrough, Rjwilmsi, Rps, Sennitiel, Swfung8, Toncek, Zaslav, 27 anonymous edits

Kruskal's algorithm Source: <http://en.wikipedia.org/w/index.php?oldid=612381313> Contributors: 4v4l0n42, Abc45624, Abrech, Abu adam, Adamsandsberg, Ahmad.829, Ahyl1, Alcides, Aliokao, Altenmann, Andri Engels, Ap, Arithena, AysZ88, Bintu syam, BioTronic, Bkell, Booty443, CarpeCerevisi, Chipchap, Chmod07, CommonsDelinker, Cronholm144, DarrylNester, DavidEppstein, Dcoetzee, Deineka, Denisarona, Dixtosa, DoriSmith, Dysprosia, Dzenanz, EChamilakis, Ecb29, Eda eng, Gareth Jones, Gauravxpss, Giftlite, GregorB, HJ Mitchell, Hammadhaleem, Happyuk, HeadbomB, Hlbbz, Hikes395, IgushevEdward, Isnow, Jamelan, Jarble, Jaredwf, Jax Oxmen, Jeri Aulurtive, Jgarrett, JohnBlackburne, Jonsafari, Kenyon, Kruskal, Kvirkvan, LC, Levin, Lone boatman, M412K, MRFraga, Maciek_nowakowski, Magioladitis, Marekpetrik, MarkSweep, MathMartin, Mbarrenecheair, Mggreenbe, Michael Angelkovich, Michael Sloane, Michael_jaeger, Mikaya, MindAfterMath, MiqayelImaynas, Mislerou, Mitchelledduffy, Mlmtux, Monmohans, MrOllie, Msli210, Myisid, NFD9001, NawlinWiki, Oli Fith, Omargami, Orenburg1, Oskar Sigvardsson, Panarchy, Pinethicket, Poor Yoric, Pranay Varma, Qwertys, R3m0t, Sarcelles, Schullz, Shellreef, Shen, Shreyasjoshi, Sigurd120, Simeon, SiobhanHansa, Spangineer, Sumit210, THEN WHO WAS PHONE?, The Anome, Tokatato, Treyshonuff, Trunks175, Wakimakirolls, Wapcaplet, Wavelength, YahoKa, Ycl6, Zero0000, 190 anonymous edits

Prim's algorithm Source: <http://en.wikipedia.org/w/index.php?oldid=624238357> Contributors: 4v4l0n42, A5b, Abu adam, Adam Zivner, Ahy1, Alexandru.Olteanu, Alexey.kudinkin, Altenmann, Ammubhave, Arrandale, AxBalrod, Berteun, Bignonmacous, Carmitsp, Cesarsoorn, Charles Matthews, Connally, DallyNester, David Eppstein, Docetze, Dessertsheep, Dllu, Dmn, Drobilla, Dyslopista, EBusiness, Ecb29, Emil Sander Bak, Epochamo, Erpingham, Ertezid, Feynmantliang, Flowi, Fresheneesz, Gatoatigrado, Gauravpress, Giftlike, Hairy Dude, Hariva, Hwy12, Hike395, Igolstdt, IgushevEdward, Inquann, Isnow, Jamelan, Jaredwf, Jirkaf, Joe Schmedding, JohnBlackburne, Justin_W_Smith, K3rb, Karlhendrikse, Kingsindian, Kiril Simeonovski, Krivokon.dmitriy, Krp, LC, LOL, Lambiam, Lim_Wei_Quan, MER-C, Magioladitis, Matusz, Maurobio, McIntosh Natura, McKay, Michael Hardy, Mihirprehta, Mindbleach, Mitchellduffy, Miym, Moa18e, Moinik, Monmohans, N4nojohn, Ninjagecko, Obradovic Goran, Opian, Pit, Platypus222, Poor Yorick, Prasanth.moothedath, Qqliu, Qwertyus, R6144, Rami_R, Rbrewerer42, Rhughes, Romann, Ruud Koot, SchuminWeb, Sean01, Setheleb, Shen, ShashankHansa, Squizz, SuperSack56, Swfung8, Tameralkuly, Teimu.tm, Treyshonuff, Turketwh, UrsusArctos, Whosyourjudas, Wikimol, Wikizoli, Wtmitchell, Ycl6, Ysangkok, ZeroOne, 217 anonymous edits

Edmonds's algorithm for directed minimum spanning trees Source: <http://en.wikipedia.org/w/index.php?oldid=621475486> Contributors: A3 nm, Almit39, Andrewman327, Bender2k14, Cronholm144, Dricherby, Edaeda, Erik Sjölund, Eyesnore, Finity, Giflite, GregorB, Headbomb, John of Reading, Kallerdis, Kinglag, Lida Hayrapetyan, LilHelpa, MclD, Michael Hardy, Neuroalki, Omnipaediast, Pintoch, Senia, Silly rabbit, Yuide, 21 anonymous edits

Degree-constrained spanning tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=508890498> *Contributors:* David Eppstein, Dcoetzee, Hermel, Ixfd64, JaGa, Lisatwo, Michael Hardy, Mohitsinghr, RJFJR, Ruud Koot, SparsityProblem, Tom, Wladston, Yiyu Shi, 8 anonymous edits

Maximum-leaf spanning tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=622486254> *Contributors:* Darklawhf, David Eppstein, Lesser Cartographies, Mild Bill Hiccup, Miym, Ouuxplusone, Spunkylepton, Ylamp, 6 anonymous edits

K-minimum spanning tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=582644474> *Contributors:* A.A.Graff, Charles Matthews, Cronholm144, David Eppstein, Frap, Kilom691, R'studah, Silverfish, Timo Honkasalo, Tmigler, 5 anonymous edits

Capacitated minimum spanning tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=617417945> *Contributors:* Beeblebrox, DoctorKubla, Headbomb, IceCreamAntisocial, Lynxoid84, Riwiimci, Timrollnickering, 4 anonymous edits

Application: Single-linkage clustering Source: <http://en.wikipedia.org/w/index.php?oldid=613585768> Contributors: 3mta3, Ankit Rakha, Chire, Cindamuse, Cricetus, David Eppstein, Febrichtzmann, Gary, Goldden_Gin, Headhopper, Lyncoid84, Manuel_freire, Melcombe, Michael_Hardy, Pat_Bijweltmei, Prahbhu1224, Włodzimierz_Wesierski, XLgratz, Zeno_Gantner, 7 anonymous users

Application: Maze generation *Source:* <http://en.wikipedia.org/w/index.php?oldid=610634473> *Contributors:* 041744, A., B., Ap, Arichnad, Brandnewotter, Bryan Derksen, Charles Matthews, Chris Roy, Cruiser1, Cyp, Dabigkid, DerGraph, Dllu, Dysprosia, Ebaychatter0, Eequor, Ellenshuman, EvanBarr, Freependulum, Frizzil, Furrykef, GregorB, Grendelkhan, Haham hanuka, Hu12, JDougherty, Jahoe, Jawawizard, Karada, Keerthan, Kevin Saff, Melcombe, Mernen, Mfoltin, Michael Hardy, Mr2001, Nandhp, Nicolas.Rougier, Nskillen, Nyttend, Ofek Gila, Plugwash, Quetzive, RHJC, RJFJR, Radiant chains, Riking8, SchfiftyThree, SharkD, SimonP, Simplex, Staszek Lem, SveinMarvin, Technicat, The Earwig, Timwi, VeegaP, Wavelength, 81 anonymous users

Clique problem Source: <http://en.wikipedia.org/w/index.php?oldid=621053245> Contributors: Alanyst, Andreas Kaufmann, Andrewpmk, Ascánder, Bender2k14, Bkell, Charles Matthews, Chmod007, Chris the speller, David Eppstein, Dcoetzee, Dmitrey, Elias, Gambette, Gifflite, Gronk Oz, Headbomb, Hunterster, JidGom, Joannatrykowska, JoeKearney, Justin W Smith, Kimby, Kwertii, Leuko, Lilac Soul, Lokiclock, M1s1ontomars2k4, Mailer diablo, Mastergreg82, Materialscientist, MathMartin, Maximus Rex, Mellum, Michael Hardy, Miym, Neile, Nilesj, Obradovic Goran, Oddbd07, Ott2, PV=NR, Pcap, Phil Boswell, Poor Yoric, Quebec99, R'n'B, Rampion, Redrose64, Rentier, Rjwilmis, RobinK, Shanes, Silversmith, Singleheart, Taw, Thore Heufeld, Timo, Tomaž, Tomaz, Tuvi, Umarrow, Vassil, W. Wenzel, Whi1959, Xue, Yihui, Youzhilistar, 50 anonymous edits

Independent set problem Source: <http://en.wikipedia.org/w/index.php?oldid=620905676> Contributors: Adi.akbartauhidin, Alexmf, Aliabbas aa, Amelio Vázquez, Andris, Arjayay, AustinBuchanan, Bkeller, Bobanobahoba, Bramschoenmakers, Bsansouci, C_lorenz, Calle, ChrisGualtieri, Citrus538, Cjoa, CommonsDelinker, Davepape, David Eppstein, Docetzee, De0d0raran"Stain, Deuler, Dysprosia, Erel Segal, Flyer22, Frawr, Freeman0119, Giftlite, Helios, Hv, Jamelan, Jeffreywarrenor, Jittat, JoergenB, Justin W Smith, LandruBek, Life of Riley, Luciene, Mahue, Mailer diablo, Manuel Anastácio, Marozols, MathMartin, Maxal, Mellum, Michael Hardy, Mikaelq, Miym, Petter Strandmark, RobinK, Rocchini, Shafact, Spoon!, Syats, TheAnome, Thore Husfeldt, Twanvl, Twri, Tyri, Valentina.Anitnelav, Wavelength, Ymasrar, Ylloh, Zaslav, 38 anonymous edits

Maximal independent set *Source:* <http://en.wikipedia.org/w/index.php?oldid=621327758> *Contributors:* Austin512, Booyabazooka, CBM, Charles Matthews, David Eppstein, Hermel, Jmichel, Il, Jvimal, Kswenson, Leithp, Life of Riley, Miym, Phil Boswell, RDBury, Rjwilmsi, RobertBorgersen, RobinK, Silverfish, Twri, Yllohh, Zaslav, 10 anonymous edits

Graph coloring *Source:* <http://en.wikipedia.org/w/index.php?oldid=622809375> *Contributors:* -Midorihana-, Abelson, Adking80, Aeons, AgarwalSumeet, AgentPeppermint, Alex Dainiak, Alexs2008, Altenmann, Antonio Lopez, Arbor, Arpi Ter-Aragelyan, Atiliogomes, BB-Froggy, Bender2k14, Beta79, Bg9989, Booyabazooka, Briansdumb, CRGreathouse, Caesura, CanisRufus, Charles Matthews, Cheeser1, ChildofMidnight, Chris Capoccia, Chris Pressey, Claygate, Corti, Cybercobra, David Epstein, David Jordan, David.daileyatrusodutedo, Dcoetzee, Dd4932, Doradus, Drange nt, DuineSidhe, Dysprosia, El Charpi, Eubulides, Fawcett5, Flyingspuds, Fred Gandt, Fritjehs, Galaxy07, Galeru, Gazimoff, Geeeee, Gene Nygaard, Giffile, Glomerule, GregorB, Guybrush, Headbomb, Hermel, Hiiiiiiiiiiiiiiiiii, Hlg, Homei, Hv, JaGa, Jalal476, Jamelan, Jochen Burghardt, Josxyz, Jozefgajdos, Jérôme, Jórdan, Kapaina, Ke4roh, Kostmo, Kuszi, LiHelpa, Lunae, Lwr314, Lycaon, Lyondif02, Lyonsam, MathMartin, Mats.szx, Maxim, Mayooranathan, Mbell, McKay, Mellum, Michael Hardy, Michael Sloane, Miym, Mjaared, Nialsh, Nick twisper, Nina Cerutti, Nitishkorula, Oleg Alexandrov, Optim, Ott2, Peter Kwok, Phil Boswell, Ptrillian, Rjwilmsi, Robert Samal, RobinK, Roll-Morton, S.K., Saforest, Sam Hocevar, SausageLady, Shirifan, Silvrous, Suisi, Superdosh, TeleTeddy, Terryn3, Thesilverbail, Thore Husfeldt, Timdumol, Tonkawa68, VMS Mosaic, Vmohanaraj, Wierdcowman, Wikikoff, Wikirao, Xrchz, Zanetu, Zaslav, ZeroOne, ۱۴۱ anonymous edits

Bipartite graph Source: <http://en.wikipedia.org/w/index.php?oldid=623273277> Contributors: 149AFK, Altenmann, AndiPersti, AndreasWittenstein, BiT, Bkkbrad, Bobo192, Booyabazooka, Brighterorange, Burn, CRGreathouse, Catgofire, Charliecuri, Corti, David Eppstein, Delirium, DoostdarWKP, Eric119, Flamholz, Freiberg, Gassa, Gifflie, H@r@ld, Hoorayfortunes, HowardMcCay, JasonKlaus, Jdpipe, Jduard, JoergenB, JonAwbrey, Jpbrown, Jérôme, Kevimmon, Kozuch, Kri, Manojmp, MathMartin, Melchoir, MichaelDinolfo, MichaelHardy, MildBillHiccup, MistWiz, Miym, Nethgirb, Netpiлот43556, Nonenmac, Nullzero, Ojan, PaulAugust, PaulTanenbaum, Pgdx, Pinethicket, RobertIlles, RobinS, Shahab, Shmomuffin, SofjaKovalevskaja, TakuyaMurata, Tobo, Tomharrison, Twri, 103 anonymous edits

Greedy coloring *Source:* <http://en.wikipedia.org/w/index.php?oldid=545539905> *Contributors:* David Eppstein, Drpinkem, Hermel, Igorpak, Martynas Patasius, 2 anonymous edits

Application: Register allocation Source: <http://en.wikipedia.org/w/index.php?oldid=611889103> Contributors: Abdull, AndrewHowse, CanisRufus, Corti, Danfuzz, David Eppstein, DavidCary, Dcoetzee, Doradus, Ecb29, Fireice, FredGandi, FreplySpang, Galzigler, Girx, Guy Harris, Headbomb, Jamelan, Jonathan Watt, Jpo, Jterk, Kdcooper, Lecanard, Liao, Mdf, Mike Van Emmerik, Msiddalingala, Naasking, Nbarth, Ndanielm, Pintoch, Prari, Pronesto, Qartar, Radagast83, Robost, Sparshong, Speight, SteveJothen, StewieK, Taw, Tedickey, TomFitzhenry, Vukuncak, Walk&check, Wlievens, 63 anonymous edits

Vertex cover Source: <http://en.wikipedia.org/w/index.php?oldid=616077066> Contributors: Bardia90, Bender2k14, C_lorenz, David Eppstein, Dcoetzee, Dedevelop, Edemaine, Erel Segal, Ergor, Exol, Gdr, Giftlife, GorillaWarfare, Gasba figosha, IvanLanin, Jamarlan, Jayathrina, Jeremy_cowles, JoergenB, Justin_W_Smith, McMzMcBride, MathMartin, Mellum, Michael_Hardy, Michael_Slone, Mivym, Mycroft80, Nijahne, Phil_Boswell, Raphael_fuchs, Ricardo_Ferreira_de_Oliveira, Rjwilmsi, RobInK, Ross_Friar, Siddhant, Thore_Husfeldt, Ylloh, 32 anonymous edits

Feedback vertex set *Source:* <http://en.wikipedia.org/w/index.php?oldid=622110919> *Contributors:* A3 nm, AED, Afrozenator, Booyabazooka, ChrisGualtieri, David Eppstein, Dcoetzee, George

Zuparis Punk Studio, Gosha Rigozzi, Ruidi, Headbonito, Herme, Joshirosen, Jstickeinhusch, Mewmew, Michael Hardy, Mr. Ecity, Ottz, Paul August, Robinix, Senggasp, Stevey 7700, ThatFromThatShow!, Velella, Yixin.cao, Yllohh, 11 anonymous edits

Eulerian path Source: <http://en.wikipedia.org/w/index.php?oldid=620354295> Contributors: Altenmann, Anakata, Armed, Artem P. Melnitsyn, Arthena, Arthur Rubin, Arya84, At-par, Audiovideo, Bender235, Boemanneke, Booyabazooka, Brookie, Bueller 007, Bugthink, CRGreathouse, Cbarrick1, Charles Matthews, Chris the speller, CommonsDelinker, CtrlAltDel121, Headtbody, Hermell, Kelly Martin, LokoClock, Magioladitis, Mellum, Paul August, RobinK, That Guy, From That Show!, Twri, Vadimium, Yixin.ca, 20 anonymous edits

Danilo, Danlyus, David Eppstein, Deno.magnus, Dlskip, Doostash.WKI, Dorados, DwyerJ, Dysprosia, Echizen, El Kom, Lyman, Flangs, 1135khali, Frazz, Fyear, Frightmare, GTubio, Gelingvisor, Giftlife, Gogo Dodo, Hanru, Haterade111, Henning Makholm, Herbie, Hippophaë, Igorpak, Ilmari Karonen, Iosif, Jacobolus, Jason Quinn, John Sheu, Jonastf, Kilom691, Kinewma, Llaesrever, Logical Cowboy, Lunae, Magister Mathematicae, Martynas Patasius, MathMartin, Maxal, Maxtremus, McKay, Mh, MhyM, Michael Hardy, Modeha, Myasuda, Nguyen Thanh Quang, Obradovic Goran, Ohiostandard, Oliphant, Omnipediast, Peter Kwok, Pixeltoot, PonyToast, Powerpanda, Red Bulbs Fan, Rgdboer, Ricardo Ferreira de Oliveira, Ricardogpn, Rjwilmsi, Rompelstompe1777, Rschwieb, Taxman, Tide rolls, Tommy2010, Trevor Andersen, Twri, Uni4dfx, Vonkje, Yatin.mirakhus, Zero0000, 123 anonymous edits

Hamiltonian path Source: <http://en.wikipedia.org/w/index.php?oldid=621067399> Contributors: Alansohn, Albywind, Algebraist, Algent, Altenmann, Americanhero, Andrej.westermann, Anikings, Armanbuettner, ArnoldReinhold, Arzonak, Arthena, Arvindn, Austinmohr, Bigmonachus, Bkell, Bryan.burgers, Catgut, Cdunn2001, CentroBabbage, CharonX, Cliff, Da Joe, David Eppstein, Daviddaved, Deljr, Dcoetzee, Dimadima, DwyerJ, Dysprosia, Elommole, Erwinrat, Etaino, Ettrig, Ewlyahooocom, Giftlife, Gimsuloft, GodfriedToussaint, Haham hanuka, Hairy Dude, Headtbody, Henning Makholm, Henrygb, Henrylaxen, Igorpak, InverseHypercube, Ifsisk, Jamesx12345, Jic, JoergenB, John Vandenberg, Justin W Smith, Jwpat7, Kenb215, KirbyRider, Kope, Kri, Kurkku, Lambiam, Lemmio, Lesnai, Mgnum0n, Martynas Patasius, MathMartin, Matt Cook, McKay, Mcld, MhyM, Michael Hardy, Michael Sloane, Mickeysweatt, Miracle Pen, MiyM, Myasuda, Nafsahd, Nguyen Thanh Quang, Ninjagecko, Nr9, Nsantha2, Obradovic Goran, Ojigiri, Oleg Alexandrov, OneWeirdDude, Paul August, Paul bryner, Pcb21, Perray, Peruvianillama, Peter Kwok, Populus, RDBury, Radagast3, Rafaelmonteiro, Random8042, Reddi, Rjwilmsi, Robomaster, Rocchini, Rupert Clayton, SamAMac, Shell Kinney, Shreevatsa, Silas S. Brown, Sirommitt, SkyLined, SlumdogArasan, Smig, Status quo not acceptable, Stdazi, TakuyaMurata, Tanvir 09, Thore Husfeldt, Thue, Tweenk, Twri, Vinayak Pathak, Wavelength, Wcherowi, Widr, Yecri, Zapatu, 1, 145 anonymous edits

Hamiltonian path problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=615552060> *Contributors:* Alabarre, Altenmann, Andreas Kaufmann, Antidrugue, Araujoraphael, Aronzak, Arvindn, Contesteen, CryptoDerk, Davepape, David Eppstein, Dcoetzee, Dominus, Dwc89, Ecb29, Ewlyahoocom, Giftlite, Graph Theory page blanker, Henrylaxen, Hqb, Ifxld64, Jamelan, Jochgem, Joostik, Laurusnoobz, Luuva, MKoltown, Mariehuynh, MathPoetry, Matt Cook, Michael Sloane, Miracle Pen, Miym, Nr9, Obradovic Goran, Pahari Sahib, Rjwilmis, RehK9, Runpet Cleaton, Seabob2, Shlomo Shemtov, Silas S. Brown, Sprout122, Tawer, Thomas Heufeld, Werner Maaßen, Wijes, 46 anonymous edits

Travelling salesman problem Source: <http://en.wikipedia.org/w/index.php?oldid=623814314> Contributors: 130.233.251.xxx, 1exec1, 28421u2232nfencenc, 4ndyD, 62.202.117.xxx, ANONYMOUS COWARD0xCODE, Aaronbrick, Adammathias, Aftermath1983, Ahoerstemeier, Akokskis, Alan.ca, AlanUS, Aldie, Altenmann, Anaxial, Andreas Kaufmann, Andreas2d2, Andris, Andy19601, Angus Lepper, Apanag, ArglebargleIV, ArnoldReinhold, Aronisstav, Astral, AstroNomer, AustinBuchanan, Azotlichid, B4hand, Bathysphere, Beatkist, Bender2k14, Benhen1997, Bensin, Bernard Teo, BethNaught, Bjornson81, Bmearns11, Bo Jacoby, Bongwarrior, Boothinator, Brian Gunderson, Bruceved, Brw12, Bubba73, C. lorenz, CRGreathouse, Can't sleep, clown will eat me, Capricorn42, ChangChienFu, Chris-gore, Chris55, ChrisCork, ChrisGualtieri, Classicalecon, Cngoulimis, Coconut7594, Colonies Chris, Conversion script, CountingPine, DVdm, Daniel Karapetyan, David Eppstein, David.Mestel, David.Monniaux, David.hillschafer, DavidBiessack, Davidhorman, Dhfbirs, Doceteet, Devis, Dino, Disavian, Dishantpandya777, Donarreiskoffer, Doradus, Downtown dan seattle, DragonflySixtyseven, Dralea, DreamGuy, Dricherby, Duoduo, Dwdhwh, Dysprosia, Edward, El C, Ellywa, ErnestSDavis, EzequielBalmori, Fanis84, Ferris37, Fioravante Patronne, Flapirr, Fincown, Fmorstatter, Fredrik, Gaeddal, Galoubet, Gdssy, Gdr, Geoftech, Geomatique, Gerhardvalentin, Giftlite, Gnomz007, Gogo Dodo, Graham87, Greenmatter, GregorB, Gronk Oz, H, Haeinous, Hairy Dude, Hans Adler, Happyuk, Haterade111, Hawk777, Herbee, Hike395, Honmza, Horst-schlaemma, Hyperneural, Ilya.gazman, Ironholds, Irrevenant, Isac, IstvanWolf, IvR, Ixfhd64, J delaney, JackH. Jackbars, Jamesd9007, Iasmj2, Jasonb05, Jasonyo, Jeffhoy, Jim.Callahan,Orlando, John of Reading, Johnsgout185, Johnlreach, Jok2000, JonathanFreed, Jonesey95, Jonhkr, Jsamarziya, Juggander, Justin W Smith, KGV, Kane5187, Karada, Kenneth M Burke, Kenyon, Kf4bdy, Kiefer, Wolfowitz, Kjells, Klausikm, Kotasik, Kri, Ksana, Kvamsi82, Kyokpae, LFaraone, LOL, Laleppa, Lambiam, Lanthanum-138, Laudaka, Lesser Cartographies, Lingwanjae, Lone boatman, MSGJ, MagicMatt1021, Magicalbendi, Magioladitis, Male1979, Mantipula, MarSch, Marj Tiefert, Martynas Patasius, Materialscientist, MathMartin, McGeddon, Mcld, Mdd, Mellum, Melsmans, Mhahsler, Michael Hardy, Michael Stone, Mikaela, Mild Bill Hiccup, Miym,莫jorworker, Monstergurkan, MoraSique, Mornegil, Musiphil, Mzamora2, Naff89, Nethgirb, Nguyen Thanh Quang, Ninjagecko, Niraj Aher, Nobbie, Nr9, Obradovic Goran, Orfest, Ozziev, Paul Silverman, Pauli133, Pegasusbupt, PeterC, Petrus, Pgr94, Phcho8, Piano non troppo, PierreSelim, Pleasantville, Pmdboi, Proslilaes, Pschaus, Qaramazov, Qorilla, Quadell, Quebec99, R3m0t, RDBury, Random contributor, Ratfox, Raul654, Reconsider the static, RedLyons, Requestion, Rheun, Richmeister, Ricksmt, Rjwilmsi, RobinK, Rocarvaj, Romaldo, Ross Fraser, Rror, Ruakhi, Ruud Koot, Ryan Roos, STGM, Saeed.Veradi, Sahuagin, Samusoft1994, Sarkar112, Saarahb.harsh, Scravy, Seet82, Seraphimblade, Sergey539, Shadowjaws, Shafaet, Shakir28, Sharcho, ShelfSkewed, Shoujin, Shubharsankar, Siddhant, Simetrical, Sladen, Slartibartfastibast, Smurphy, Smremde, Smyth, Snow Blizzard, Some standardized rigour, Soupz, South Texas Waterboy, SpNeo, Sparshong, Spock of Vulcan, SpuriousQ, Stemonitis, Stevertigo, Stimp, Stochastix, StradivariusTV, Superm401, Superninja, Tamfang, Teamtheo, TedDunning, Tedder, That Guy, From That Show!, The Anome, The Thing That Should Not Be, The stuart, Theodore Kloba, Thisisbossi, Thore Husfeldt, Tigerqin, Timman, Tobias Bergemann, Tom Duff, Tom3118, Tongally, Tomhubbard, Tommy2010, Touriste, Trlkly, Tsplog, Twas Now, Vanished user wdjklasdjskla, Vasili, Vgy7ujm, Wavelength, WhatisFeelings?, Wizard191, Wumpus3000, Wwwwolf, XiaoJeng, Xnn, Yixin.cao, Ynhockey, Zaphraud, Zeno Gantner, ZeroOne, Zyuuoh, 650 anonymous edits

Bottleneck traveling salesman problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509363651> *Contributors:* Altenmann, Andreas Kaufmann, Ansell, David Eppstein, Dcoetzee, Dysprosia, Gdr, IRelayer, Oleg Alexandrov, Pkrecker, Silverfish, TRauMa, Vespristiano, 6 anonymous edits

Christofides' heuristic for the TSP *Source:* <http://en.wikipedia.org/w/index.php?oldid=613526457> *Contributors:* A3 nm, Andreas Kaufmann, Apanag, Cerebellum, CommonsDelinker, Daniel Dandranja, Edward Fkoenig, Helsaahn, Headbomb, Lanthanum-138, Lynxoid84, Magioladitis, Mastarh, Michael Hardy, Nobbie, Omnipaedista, Ruud Koot, Sh.pouriya, Vaclav.brozek, Zyqqh. תמונה: [ממוזער: חישובו של האלגוריתם של צריסטופידיס](#)

Route inspection problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=603984929> *Contributors:* AbcXyz, Arthur Rubin, Aviados, Bender235, Bkell, Bryan Derkens, C_lorenz, Da Joe, David Epstein, Dysprosia, ErikvanB, FritzAsbiner, Headbomb, Honzta, Justin W Smith, Kedwar5, Mangledorf, Michael Hardy, Mikhail Dvorkin, Mtwoetws, PvRnR, Petermire, PolyGnome, RobinK, Rockstone35, SDS, Sam Hoevar, Shuroo, Softtest123, Stern, Taesio, VictorAnaykin, MarieCrane, Wik, Zaslav, 39 anonymous edits

Matching *Source:* <http://en.wikipedia.org/w/index.php?oldid=621443505> *Contributors:* Adking80, Aednichols, Altenmann, Ankushshah89, Aranel, Arthur Rubin, Azer Red, Bender2k14, CRGreathouse, Calle, Cmansley, Cyhawk, Daqu, David Eppstein, Dcoetzee, Debamf, DonDiego, Edemaine, El C, Erel Segal, Excirial, Faisalsyn, Favonian, Fghtngthfght, Fyrael, Giftlite, Green Snake, Gutza, H@r@lD, Hermel, Itamarro, Jammydogger, Jdpipe, Kainaw, Kamyr1, Kerry Raymond, Kilom691, Kraymer, LOL, Lambiam, Mac, Madherro88, Mamling, Mangarah, MarSch, Markoid, Mastergreg82, MathMartin, Maxal, Mental Blank, Mgccel, Michael Hardy, Michael Slone, MickPurcell, Mintleaf, Miym, Mmarci1111, Movado73, Nils Grimsmo, Nono64, Oliphant, Omegatroy, Oudmalie, Pparys, Rahul38888, RamanTheGreat, Rentier, Rjwilmsi, RobertBorgersen, Ronz, Sabha ezami, SchreiberBike, Scottkwong, Shurakai, Smartnut007, Squizz, Surfingpete, TheEternalVortex, Tim32, Tomo, Wcherow, Ylloh, Zaslav, 104, יונתן לוי anonymous edits

Hopcroft–Karp algorithm for maximum matching in bipartite graphs Source: <http://en.wikipedia.org/w/index.php?oldid=593898016> Contributors: Altenmann, AmirOnWiki, Anks9b, David Epstein, Edward, Enobrev, Exile oI, Gnagnoyoi, GregorB, Headbomb, Justin W Smith, Kathmandu guy, Mange01, MarSukiasyan, Michael Veksler, Miym, Mogers, Nils Grimsmo, Nitavi, Oliphant, Pazardzo, Shkoorah, Siddhulant Goel, Thanabhat.io, Tim32, Twanly, WinerFresh, X7o, Zhaoob, 44 anonymous edits

Edmonds's algorithm for maximum matching in non-bipartite graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=620442100> *Contributors:* A3 nm, Aacombarro89, AlanTonisson, Bender2k14, Blkblck, Choeus, Daiyuda, Daveagp, Eddy Epstein, Edemaine, Favonian, Gifflite, Girlwithgreeneyes, Headbomb, Mangarah, Mark viking, Markoid, Michael Hardy, Miym, Phingo, Rund Koot, Simopratt, Wismon_25. רעד קודן קרדת anonymous edits

Assignment problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=614550084> *Contributors:* \$Mathe94\$, A.A.Graff, AZhnaZg, Adam.oberman, Altenmann, Andrewa, Anonymoues, Arthena, Beland, Benbest, Bender2k14, Charles Matthews, David Eppstein, Deltahedron, Duodoonduo, Evercat, Fnielsen, Garbodor, Infrogmation, Jklm, Justin W Smith, Karipuf, Klahnako, Lage, Lambiam, Lawsonsaw, Materialscientist, Michael Hardy, Michael Sloane, MichaelGensheimer, Miym, Nils Grimsimo, Oleg Alexandrov, Onebyone, Ophiccius, PAS, Paul Stanister, Pearle, RJFJR, Ripbi, Roger67, Scirium, SimonP, Teknic, The Anome, Tribaal, Vikram's jammal, WikId77, Wshun, 57 anonymous edits

Hungarian algorithm for the assignment problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=621641213> *Contributors:* \$Mathe94\$, Ahmabadolkader, Andreas Kaufmann, Aostrander, Arichnad, Balajivish, Bender2k14, Bkkbrad, Bob305, Bramdj, Buehrenn, Chmarkine, Culipan, Danyaljj, David Eppstein, Delaszk, Delta 51, Dewritech, Discospinster, DrJunge, Egnever, Fnielsen, Giftlike, Immunite, Incredible Shrinking Dandy, Jsoos, JokesFree4Me, Justin W Smith, Klahnako, Kope, LoveEncounterFlow, LutzL, Macrakis, Mahue, Mboverload, Mcld, Mekeor, Michael Hardy, Mike0001, Miskin, Miym, Mslusky, N5iln, Nils Grimsmo, NisansanaDdS, Nkarthiks, NoldorinElf, Nullzero, Pparent, Purple acid, R'n'B, Rich Farmbrough, Robert Nitsch, Roy hu, RoySmith, Ruud Koot, Shreevatsa, Singleheart, Slaunger, Spottedowl, Stiel, Tejas81, The Anome, Thegarybakery, Tomcatjerrymouse, Two Bananas, Vacio, Watson Ladd, Werdma, Zweije, زمچک ۱۱۷ anonymous edits

FKT algorithm for counting matchings in planar graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=609273031> *Contributors:* A3 nm, Bearcat, Bender2k14, David Eppstein, Dricherby, Extra999, Headbomb, Michael Hardy, R'n'B, Rjwilmsi, StAnselm, 3 anonymous edits

Stable marriage problem Source: <http://en.wikipedia.org/w/index.php?oldid=617965585>. Contributors: Alroth, Andrewaskew, Araqsy, Manukyan, Arthur Frayn, Augurar, BlckKnght, Brusegadi, Burn, Cachedio, Dalhamir, Danrah, David Eppstein, Dcoetzee, Dr. Universe, E7em, Ecb29, Evercat, Felagund, FinnishOverlord, Frietjes, Gabbe, Geoffrey, Giftlite, Headbomb, Henning Makholm, Hughdbrown, Jayamohan, Jojan, Josphans, Justin W Smith, Koczy, Lantonov, Lingust, MC10, Magioladitis, Mahahahaneapneap, MarketDesigner, Mcstrother, Michael Hardy, Miym, Nictich, Not-just-yet, Oleg Alexandrov, PV=nRT, PerryTachett, Rgdboer, Richwales, Rponamgi, Rtyho usa, Shreevatsa, Stuart Morrow, Teimu.tm, The Anome, Timtb, Undsoweter, User A1, Vrenator, Wshun, Zwar, 89 anonymous edits

Stable roommates problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=623665021> *Contributors:* David Eppstein, Gifflite, GregorB, Headbomb, JackSchmidt, Josve05a, Mcherm, MichaelExe, Mindspin311, Miym, Pip2andahalf, Pjcronje, Radomaj, RobWIrving, Skier Dude, Squeaky201, TakuyaMurata, Wcherowi, 18 anonymous edits

Permanent Source: <http://en.wikipedia.org/w/index.php?oldid=622215001> Contributors: AdjustShift, Alansohn, Altemann, Andris, Army1987, AxelBoldt, Backslash, Forwardslash, Bender2k14, Beyond My Ken, Borraj, BradHD, Bte99, Charles Matthews, Chris the speller, Creidieki, Cybercobra, Dcoetzee, Deltahedron, Draco flavus, El C, Emerson7, Esprit15d, EverettYou, Fuzzy, Galoubet, Garfl, Gdr, Giftlite, Gustavb, Headbomb, Irchans, Jim1138, Jmath666, Joachim Selke, Jérôme, Karmastan, Kilom691, Laudak, Lazylaces, Masnevets, Medical Historian III, Melaleus, Michael Hardy, Mikhail Dvorkin, Moe Epsilon, Mukaderat, Oleg Alexandrov, Pcap, Populus, Pschemp, Rjwilmsi, Sbyrnes321, Shreevatsa, Sodin, SoledadKabocha, Stemonitis, Thore Husfeldt, Twri, Vanish2, Wcherowi, Xev lexx, Xnn, Zaslav, 60 anonymous edits

Computing the permanent *Source:* <http://en.wikipedia.org/w/index.php?oldid=618259518> *Contributors:* Altenmann, Bender2k14, Circularargument, David Eppstein, Gifflite, Gregory Kogan, Headbomb, Hermel, Kilom691, Laudak, McKay, Michael Hardy, Mukadderat, Rjwilmsi, Ruziklan, Shreevatsa, Thore Husfeldt, Twri, Xezbeth, Ynaamad, Zaslav, 34 anonymous edits

Maximum flow problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=624218666> *Contributors:* Alansohn, Altenmann, Anescent, Arash.nouri, BenFrantzDale, Charles Matthews, Crystallina, Csyfpwiting, Cyhawk, Danaram91, Darkwind, Davapepe, David.Eppstein, Debamf, Deven7595, FelGru, Flyer22, Gareth Jones, Gifflite, GregorB, Headbomb, Huttar, Jackfork, Jackie Marie343, Jeremy112233, Jleedeve, Joao Meidanis, Joepln, LOL, Lambertch, Magioladitis, Majestic-chimp, Mandarinaclementina, Materialscientist, Mcld, Milad621, Minimac, Msm, Mtcv, Muhandes, Nealmich, Nick Number, Nils Grimsmo, Nawrat, Nilvaldrak, Parthasarathy.kr, Paul August, Polisher of Cobwebs, Pugget, Q-lio, Rizzolo, Rkleckner, Sgx, Skranthi, Svick, Tchashasopose, TempteCoeur, Tmfs10, Tommyboucher, Urod, Vegaswikian, Wikiklrs, WuTheFWasThat, X7q, ZeroOne, Zuphilip, 63 anonymous edits

Max-flow min-cut theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=621593740> *Contributors:* Aaron Schulz, Aashcrahin, Adam Zivner, Addshore, Aindeevar, Ali Esfandiari, AxelBoldt, Booyabazooka, C S, Daiyuda, Darren Strash, David Eppstein, Dcoetzee, Delask, Douglas R. White, Dyno8426, Dysprosia, ESkog, Eric119, Finfobia, Floridi, Gabn1, Gaius Cornelius, Geometry guy, Gifliffe, Guof, Huttar, Jamelan, Jokes Free4Me, Kawaa, LOL, Luc4, MathMartin, Meekohn, Mggreenbe, Mhym, Michael Hardy, Nils Grimsmo, Oleg Alexandrov, ParthaSarathy.kr, Paul August, Petter Strandmark, Q-llo, Qwertys, Ratfox, Rizzolo, Rshin, Ruu Koot, SchreiberBike, Sergio1, Sharanunumi1, Simon Lacoste-Julien, Snoyes, Spoon!, Srijanrshetty, Stdazi, SteveJothen, Tchashasoposse, Thore Husfeldt, Tomo, Ycl6, Yewang315, Zfeinst, 82, יובל סדר anonymous edits

Ford-Fulkerson algorithm for maximum flows *Source:* <http://en.wikipedia.org/w/index.php?oldid=608140765> *Contributors:* Afshinzairei, Alex Selby, Almi, Andreas Kaufmann, Araujo, B1993alram, Babbage, Benbread, Belet, Bobrayner, BrotherE, Cburrett, Cedar101, Colossus13, CryptoDerk, Cyhawk, DFRussia, Dcoetzee, Dekart, Dionyz, Dopehead9, Ecb29, EdGl, Gandalf61, Gareth Jones, Gerel, Giffile, Harrigan, Heineman, Hephaestos, Island Monkey, JanKuijpers, Jaredw, JordiGH, Knowledge-is-power, Leland McLInnes, Liorma, Mad728, Mamling, Mange61, Manuel Anastacio, Mark1421, Materialscientist, Me mosa, Michael Hardy, Myasuda, Nausher, Neo139, Nils Grimsmo, Petter Strandmark, Poor Yorick, Prara, Queeq, Quiark, Randywombat, Riedel, Rrusin, Shashwat986, SiobhanHansa, Svick, Tfri.didi, Toncek, Treyshonuff, Tuna027, User 38, Wfbergmann, Zero0000, gsls, 100 anonymous edits

Edmonds–Karp algorithm for maximum flows *Source:* <http://en.wikipedia.org/w/index.php?oldid=620277446> *Contributors:* Aaron Hurst, Amelio Vázquez, Balloonguy, Chburnett, Chopchopwhitey, Cosmi, Cquan, Darth Panda, Denisaroma, Gareth Jones, Giftlite, Glrx, Hashproduct, Headbomb, Htmnssn, Jamelan, Jfmantis, Jmgibson3312, John of Reading, Katieh5584, Kristjan Wager, Kubek15, Lxx, LiuZhaoliang, Magioladitis, Martani, Michael Hardy, Mihai Capotă, Nilofar piroozi, Nils Grimsmo, Nixdorf, Nkojuharov, Ohad trabelsi, Parodox, Pkirlin, Poor Yorick, Pt, Pugget, RJFJR, Sesse, Simonfl, TPReal, Zero0000, Zmwangx, قل زادگان 73 anonymous edits

Dinic's algorithm for maximum flows *Source:* <http://en.wikipedia.org/w/index.php?oldid=608833598> *Contributors:* Andreas Kaufmann, Bbi5291, Evergrey, Gawi, Giftlite, Headbomb, Magioladitis, Michael Hardy, Milicevic01, NuclearWarfare, Octahedron80, Omnipaedista, R'n'B, Rjwilmsi, Sun Creator, Teshasaposse, Urod, X7q, 20 anonymous edits

Push–relabel maximum flow algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=622687930> *Contributors:* Andreas Kaufmann, Babak.barati, Bgwhite, Damian Yerrick, Debamf, Dicklyon, Drrill, Dylan Thurston, Headbomb, JanKuijpers, Kcm1970, Kxx, Macofe, Nils Grimsmo, Nullzero, Rich Farmbrough, RussBlau, Shafaet, Sinha K, Slon02, Stefan Knauf, Sunderland06, Sverigeskillet, Svick, Wmayner, 55 anonymous edits

Closure problem Source: <http://en.wikipedia.org/w/index.php?oldid=606991998> Contributors: Ak1990, Andreas Kaufmann, Andrewman327, ChrisGualtieri, Cocoaghost, Connor mezza, David Eppstein, Erkan Yilmaz, Faridani, Headbomb, Juanpablolaj, LilHelpa, Malcolma, Michael Hardy, Nihola, RFJIR, Rjwilmsi, Wilhelmina Will, 9 anonymous edits

Minimum-cost flow problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=611724913> *Contributors:* Adamarthurryan, Alexandersaschawolff, Arash.nouri, Bwsullivan, David Eppstein, Ghostkeeper, Headbomb, Hrashi, Hs1771, John of Reading, Kovacsper84, MclD, Michael Hardy, Miym, Neo139, Nils Grimsmo, Rjwilmsi, Ruud Koot, Siddhant Goel, 23 anonymous edits

Planar graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=621454090> *Contributors:* A3 nm, ABCD, Afil, Ahoerstemeier, Alberto da Calvairate, AlexCornejo, Alexb@cut-the-knot.com, Altenmann, Arunachalam.t, AstroHurricane001, AxelBoldt, Bdesham, Bender2k14, Birutorul, Bobet, Bobo192, Booyabazooka, Brian0918, Burn, Cameron.walsh, Charles Matthews, Chris Pressey, Conversion script, Corti, Cutelyaware, DFRussia, DHN, Dainiak, David Eppstein, Dbenbenn, Dcoetzee, Debamf, Dilip rajeev, Disavian, Dominus, Don4of4, Doradus, Douglike, Dpv, Dtреббien, Dysprosia, Edemaine, El Roih, Eric119, Everyking, FreplySpang, Gabbe, Ged.R, Gifflite, Graph Theory page blanker, Guillaume Damiani, Hadaso, Hagman, Hat600, Headbomb, Heron, HorsePunchKid, Jamie King, Jaredwf, Jason Quinn, Jfraser, Jillbones, JohnBoyerPhd, Joriki, Jorunn, Joyous!, Justin W Smith, Koko90, Kv75, LC, Lenoxus, Libertyernie2, Life of Riley, Linas, Lyonsam, M4gnum0n, Magister Mathematicae, ManRabe, Marozols, Mastergreg82, MathMartin, May, McCart42, McKay, Mellum, Michael Hardy, NatusRoma, Nine Tail Fox, Nonenmac, Nsevs, Oleg Alexandrov, Omnipaedista, PassedTime, Paul August, Paul Pogonyshev, PhS, R'n'B, Rjwilmsi, Robin S, Romann, Rp, Scarfbay, Shellreef, Smyth, Snigbrook, Some jerk on the Internet, Sun Creator, Svdb, Tamfang, Taral, Taxipom, Taxman, Thewayforward, Timwi, Tobias Bergemann, Tomo, Trevor MacInnis, Ubersmekel, Urdutext, Vinayak Pathak, Wantnot, Wavelength, Yecril, Zanetu, Zero0000, یاری, 104 anonymous edits

Dual graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=616048549> *Contributors:* Altenmann, Andreas Kaufmann, Arthena, Bender2k14, Berland, Charles Matthews, David Eppstein, Doradus, Ewlyahocom, Futurebird, Gifflite, Hadaso, Headbomb, Jcarroll, Josephsangtjandra, Kirelagin, Kmote, Magister Mathematicae, Maproom, McKay, Michael Slone, MichaelMcGuffin, Mormegil, PEJO89, Paolo Lipparini, Pointillist, Prabhakar97, R'n'B, Rjwilmsi, Rp, Sangak, Sasquatch, Suffusion of Yellow, Taxipom, Tompw, Tomruen, Vadmium, Wangzhaoguang, Winnen209, 26 anonymous edits

Fáry's theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=614948355> *Contributors:* Altenmann, Andreas Kaufmann, Brad7777, David Eppstein, El Roih, Gene Nygaard, Gifflite, Headbomb, Jsdondow, Kolja21, Mhym, Michael Hardy, Miym, OdedSchramm, Oleg Alexandrov, Qcstudent1223, RDBury, RobinK, SaschaWolff, Taxipom, VladimirReshetnikov, 3 anonymous edits

Steinitz's theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=607020950> *Contributors:* Agaffs, Akriasas, Andreschulz, Brad7777, David Eppstein, E-Kartoffel, Gifflite, Headbomb, Kbuchin, Kiefer.Wolfowitz, Michael Hardy, Phil Boswell, Rjwilmsi, Terrek, Twri, 3 anonymous edits

Planarity testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=616186999> *Contributors:* A5b, DFRussia, David Eppstein, Dcoetzee, Edward, Frizzil, Gabbe, HairyFot, Michael Hardy, R'n'B, Rjwilmsi, RobinK, SeniorInt, Svick, Taxipom, 15 anonymous edits

Fraysseix–Rosenstiehl planarity criterion *Source:* <http://en.wikipedia.org/w/index.php?oldid=617382885> *Contributors:* Charles Matthews, Davepape, David Eppstein, Gifflite, Michael Slone, Salix alba, Taxipom, 1 anonymous edits

Graph drawing *Source:* <http://en.wikipedia.org/w/index.php?oldid=624211962> *Contributors:* A.bit, Aaronzat, Almit39, Altenmann, AnakngAraw, Andreas Kaufmann, Anonymous Dissident, Anrusna, ArthurDenture, Babcat64, Charles Matthews, Cubicbine119, Crecy99, Dardasa, DavRosen, David Eppstein, Dbenbenn, Dysprosia, Euphilos, Experiment123, FirefoxRocks, Foo123, Fred Bradstadt, Fredrik, Gbrosse85, Gifflite, Gregbard, HaeB, Harrigan, Headbomb, Hintss, JMP EAX, Jaredwf, Jby Yeah, Jldwig, Jon Awbrey, JonHarder, Justin W Smith, Karada, Ketiltrout, Kuru, Liberatus, MForster, Manuscript, Martarius, Mdd, Mgreenbe, MichaelMcGuffin, Michele.dallachiesa, Narayanese, Osric, Pintman, Pointillist, Readyheavygo, Rholton, Rjwilmsi, Roland wiese, Rp, Sbwoodside, ShelfSkewed, Some jerk on the Internet, Stephan Leeds, StephenFerg, Sverdrup, Taxipom, Thoughtpuzzle, Tomo, Vonkje, Wavelength, ZweiOhren, 100 anonymous edits

Force-directed graph drawing *Source:* <http://en.wikipedia.org/w/index.php?oldid=590915251> *Contributors:* Adam Schloss, Altenmann, Andreas Kaufmann, AnnGabrieli, Anrusna, ArthurDenture, Bkell, Boneill, Charles Matthews, Classicaelecon, Craig Baker, Dannaf, David Eppstein, Dcoetzee, Denkkar, Drphilharmonic, Dtunkelang, Efficacious, Fried-peach, G. Moore, Gak, Gareth McCaughan, Gifflite, Headbomb, Hypotroph, Ivan Kuckir, Ixfd64, Jldwig, Jojalozzo, JonHarder, Justin W Smith, Klu, Kostmo, Mahak library, McGeddon, Michael Hardy, MrBlueSky, MrOllie, Nickaschenbach, Olive, Oleg Alexandrov, Parakkum, Resprinter123, RodrigoAiEs, Roenbaeck, Ronz, Rswarbrick, Ruud Koot, Samwass, Scftn, SteveCoast, Stevecooperorg, Tgdwyer, Thorwald, Thue, Yifanhу, 62 anonymous edits

Layered graph drawing *Source:* <http://en.wikipedia.org/w/index.php?oldid=615613486> *Contributors:* David Eppstein

Upward planar drawing *Source:* <http://en.wikipedia.org/w/index.php?oldid=614938074> *Contributors:* A3 nm, David Eppstein

Graph embedding *Source:* <http://en.wikipedia.org/w/index.php?oldid=615613430> *Contributors:* A.A.Graff, Altenmann, Bender2k14, Bentsm, Birutorul, CBM, CRGreathouse, Chris the speller, DFRussia, David Eppstein, Decora, Deltaway, Eassin, Ferritecore, Flammifer, Genusfour, Gifflite, Headbomb, JaGa, Peskoj, PieMan.EXE, Quebec99, Taxipom, Tomo, Twri, Zaslav, 17 anonymous edits

Application: Sociograms *Source:* <http://en.wikipedia.org/w/index.php?oldid=582928340> *Contributors:* Alansohn, AndreniW, Bendistraw, David Eppstein, Doczilla, Egmontaz, HaeB, Isarra, J.delanoy, Joekoz451, Marc Bernardin, Mdd, Mlaboria, My name is not dave, Mysid, Plrk, Riley Huntley, Tgr, TheGroble, Wykis, 36 anonymous edits

Application: Concept maps *Source:* <http://en.wikipedia.org/w/index.php?oldid=623271270> *Contributors:* A. B., AP Shinobi, Alan Pascoe, Alexander.stohr, Alfredotifi, Allan McInnes, Altenmann, Andyjsmith, AntiVan, Argey, Arthena, AubreyEllenShomo, AugPi, B9 hummingbird hovering, BenFrantzDale, Bender235, Bertoli, Blbowen108, Bobo192, Brian.m.moon, Bschwendimann, Bstpierre, Buddhipriya, Carbo1200, Chan Bok, Chance74, Charlottehamilton, ChemGardener, Chowbok, Closedmouth, Cmdrjameson, Conradwiki, Cornellrockey, Crusoe2, D.Right, DVdm, DanMS, David Eppstein, Deflagro, Derek Andrews, Dezignr, Digitalis3, Disavian, Discopinst, Dmcrcray, DrDooBig, Dstringham, EBlack, EbenezerLePage, El C, Eliazar, Enzo Aquarius, Epbr123, ErikCincinnati, Escape Orbit, Estevaoei, Excirial, FatalDragonIsBeast, Fleminia, Floorsheim, Fran҃ois Robere, FreplySpang, Friviere, Gaius Cornelius, Gdm, Gecko, George100, Gibbia, Gregbard, Ground, Hallenrm, Hbent, Headbomb, Henk Daalder, Heron, Helen, IceCreamAnotocial, Iridescent, J.delanoy, Jackfork, Jamelan, Jengsb, Jm34harvey, Jmabel, John Cline, Jojalozzo, Jtnie1, Juspert, KY Park, Karl-Henner, Khalid hassani, Kku, Klimon, KnightRider, Kollision, Konetidy, Kuru, LarryQ, Leiferz, Lheuer, Ligulem, Lindsay658, MBSaddoris, Majorjy, Marybjat, Mdd, Meena74, Mel Etitis, Metcalm, Michael Hardy, Michig, Micru, Mlevitt1, Modify, Mrwojo, Mwanner, NRaja, Nagarjunag, Nagy, Nesbit, NextD Journal, Nigholith, Nikevich, Oceans and oceans, Ohnoitsjamie, Orionus, Orphan Wiki, Oxymoron83, Pashilkar, Pavel Vozemilek, Perigean, Pharaoh of the Wizards, Piano non troppo, Picksg, Pinethicket, Ps07swt, Pwilkins, Quercusrobur, Quiddity, RJBurkhart, Ray Oaks, Rcaraval, Reinyday, RichardF, Ronz, Rursus, SEWilco, Sam Sailor, Saxifrage, Shwoodside, SchreiberBike, Schvan, Sepreece, Sherbrooke, Smahr2013, Spdegabrielle, Spookfish, Spot2112, Staszek Lem, Stephan Leeds, Stephen0928, Tedickey, Tennismenace88, Tentinator, The Anome, The ed17, The wub, Tide rolls, Tobias Bergemann, Tomicmap, Tommy2010, Tomo, Truman Burbank, Umer992, Viewood40, Vladimir B., Vrenator, Wid, William Avery, WindOwl, Wxidea, 314 anonymous edits

Interval graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=622809974> *Contributors:* Arvindn, BD2412, David Eppstein, Dysprosia, Headbomb, Hnriddler, Jaredwf, Justin W Smith, Kiefer.Wolfowitz, Lyonsam, M11101, Makyen, Mastergreg82, Mgreenbe, Miym, Odwl, Ott2, PaulTanenbaum, Polymedes, Rich Farmbrough, Rjwilmsi, Ruvald, Silvrous, Smyle 6, Terrykel, Tgr, Tonkawa68, TonyW, Twri, Yixin.cao, 22 anonymous edits

Chordal graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=612863107> *Contributors:* A3 nm, Altenmann, Bkell, CountMacula, Danielx, David Eppstein, Domination989, Ged.R, Headbomb, JP.Martin-Flatin, Jochgem, Kilom691, Landon1980, LokiClock, Lyonsam, Martygo, Michael Slone, Michaell, Oliphant, Peter Kwok, Rich Farmbrough, Someguy1221, Tentinator, Tizio, Zaslav, 44 anonymous edits

Perfect graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=622267732> *Contributors:* Adking80, After Midnight, Awdteemo, BeteNoir, CXCV, Chaney, Charles Matthews, Claygate, Colorajo, David Eppstein, Dcoetzee, FREL98, Freiert, Gifflite, Columbic, Headbomb, Igorpak, Jamie King, Jaredwf, Michael Hardy, Mon4, Odwl, Peter Kwok, PhS, Qutezuce, Sundar, Svick, Taxipom, Teorth, Tizio, Vince Vatter, Xnn, 34 anonymous edits

Intersection graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=605040775> *Contributors:* Altenmann, Bender2k14, Bgwhite, David Eppstein, Dcoetzee, Erel Segal, Gandalf61, Gato ocioso, Justin W Smith, Kope, Mcoury, Michael Hardy, OdedSchramm, PaulTanenbaum, Quebec99, Rocchini, Salix alba, Tangi-tamma, Twri, Zaslav, 12 anonymous edits

Unit disk graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=622652175> *Contributors:* Akriasas, Altenmann, Avb, Davepape, David Eppstein, Igorpak, Ixfd64, Lesser Cartographies, Lyonsam, Mgcl, Pbranda, Rb82, Rjwilmsi, Tales Heimfarth, Twri, 5 anonymous edits

Line graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=623182782> *Contributors:* 2D, Abdull, Ahruman, Ale jr, Allens, Altenmann, Anbu121, Anrusna, Archdukefranz, Arman Cagle, At-par, Avraham, Bender2k14, Booyabazooka, Braindrain0000, CBM, Calle, Capricorn42, Confession0791, DVD R W, Danski14, David Eppstein, DavidCary, Dbenbenn, Dcoetzee, Deadbeef, Discopinst, Dominus, DropZone, Dysprosia, El Roih, Epr123, Excirial, Gelingvistoj, Gifflite, Graham5571, Gscshoyru, Guardian of Light, Hayhay0147, Headbomb, Hydrogen Iodide, Iain99, Ilmari Karonen, J.delanoy, Jaredwf, Jim1138, John Cline, Josve05a, Kitresaiba, LilHelpa, Lostinforest, Lyonsam, MathMartin, Meaghan, Melaan, Merlin444, Miym, Nbarth, Oatmealcookiemon, Oleg Alexandrov, OllieFury, Omnipaedista, Onecanadasquarebisshopgate, PMajer, Paolo Lipparini, Paul August, Peter Kwok, PhS, Qineticket, Playerwithgraphs, Qef, RA0808, Reyk, Rjwilmsi, Rutebega, SJP, Seba5618, Tangi-tamma, TedPavlic, Twri, Vary, Wiki13, YouWillBeAssimilated, Zaslav, 133 anonymous edits

Claw-free graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=604957977> *Contributors:* Adking80, Cdills, David Eppstein, Drange net, GregorB, Headbomb, John of Reading, Michael Hardy, Miym, Paisa, PaulTanenbaum, PigFlu Oink, RDBury, Rich Farmbrough, Robert Samal, Twri, Vanish2, 6 anonymous edits

Median graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=624181607> *Contributors:* BD2412, Brewcrew, CharlotteWebb, David Eppstein, FghIJkIm, Fratrep, Gambette, Gifflite, Howard McCay, Igorpak, Jochen Burghardt, Justin W Smith, Kilom691, Libcub, Magioladitis, Philippe Giabbanelli, Rjwilmsi, 5 anonymous edits

Graph isomorphism *Source:* <http://en.wikipedia.org/w/index.php?oldid=611203702> *Contributors:* AdamAtlas, Aeons, Aerosprite, Altenmann, Anrmusna, Ansatz, Arthur Rubin, AxelBoldt, Bkell, Blaisorblade, Blake-, Booyabazooka, CBM, Citrus538, Daniele.tampieri, Davepape, David Eppstein, Dcoetze, El Roih, EmilJ, Gifflite, Headbomb, Igor Yalovecky, Iotatau, Itub, J. Finkelstein, Jamelan, Jason Quinn, Jitse Niesen, Jludwig, Justin W Smith, KSmrq, Kingfishr, Kozuch, Linas, Loopology, MathMartin, Maxal, McKay, Michael Hardy, Michael Slone, MoraSique, Nemnkm, Nibbio84, Norlesh, Oliphant, PV=nRT, Paul August, PierreCA22, Puzne, Ricardo Ferreira de Oliveira, Rich Farmbrough, Rjwilmsi, Robert Illes, Shreevatsa, Sleepinj, SuperMidget, Thomasp1985, Tim32, Trjumpet, Twri, Vegasprof, Verbal, 57 anonymous edits

Graph isomorphism problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=621047343> *Contributors:* A.A.Graff, Akiezun, BarroColorado, Bender2k14, Benja, Birutorul, Charles Matthews, David Eppstein, Dcoetze, Dekart, Delirium, Dylan Thurston, EmilJ, EsfirK, Headbomb, Iotatau, J. Finkelstein, Jamelan, Janm67, JoergenB, Joriki, Justin W Smith, Laurusnobilis, LouScheffer, Mart071, Maxal, Melchoir, Miym, Michael Hardy, Miym, Neile, Ott2, Pascal.Tesson, Rjwilmsi, RobinK, Root4(one), Rsarge, Ruud Koot, Shreevatsa, Slawekb, Stux, Tim32, Tomo, Twri, Verbal, Kaneyoku, 24 anonymous edits

Graph canonization *Source:* <http://en.wikipedia.org/w/index.php?oldid=622954225> *Contributors:* Altenmann, BenFrantzDale, Blaisorblade, David Eppstein, Gifflite, GrEp, Headbomb, Michael Hardy, Qudit, QuarantinedVessel, Tim32, Ynaamad, 5 anonymous edits

Subgraph isomorphism problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=621625475> *Contributors:* A3 nm, APerson, Andreas Kaufmann, Brian0918, BrotherE, Centrx, Citrus538, Cowsandmilk, D6, DaveTheRed, David Eppstein, Dcoetze, Decay7, Dysprosia, EmilJ, Fabior1984, Gbruin, Graph Theory page blanker, Graue, HanielBarbosa, Hasanjamil, Hdc1112, JMP EAX, Joerg Kurt Wegner, Justin W Smith, MathMartin, Mazi, Mellum, Orkybash, PigFlu Oink, R'n'B, Rjwilmsi, RobinK, That Guy, From That Show!, Thv, Tim Goodwyn, 27 anonymous edits

Color-coding *Source:* <http://en.wikipedia.org/w/index.php?oldid=617918022> *Contributors:* DangerousPanda, David Eppstein, Dhwuang, FairFare, Frap, Hermel, Jenny Lam, Khazar2, Michael Hardy, Pullpull123, R'n'B, SwisterTwister, Swpb, Widefox, 7 anonymous edits

Induced subgraph isomorphism problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=617355234> *Contributors:* Alabarre, CBM, Charles Matthews, Citrus538, David Eppstein, Miym, Momotaro, PsyberS, RobinK, 1 anonymous edits

Maximum common subgraph isomorphism problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=581433031> *Contributors:* A3 nm, Andreas Kaufmann, DVanDyck, David Eppstein, Joerg Kurt Wegner, Jugander, Michael Hardy, Modify, R'n'B, RobinK, Smr, 8 anonymous edits

Graph partition *Source:* <http://en.wikipedia.org/w/index.php?oldid=620522428> *Contributors:* Almeria.raul, Aydin.Khatamnejad, Cebichot, ChrisGualtieri, Christian.schulz, Concertmusic, David Eppstein, Dicklyon, Eboman, EncMstr, Headbomb, Jheiv, Jitse Niesen, JonDePlume, JosephGrimaldi, LouScheffer, Mark_viking, Mastergreg82, Michael Hardy, Miym, Nanobear, Poochy, Riyad parvez, Rjwilmsi, RobinK, Ronz, Ruud Koot, SchreiberBike, Skatalites, Sohini6685, Staszek Lem, Thine Antique Pen, Wavelength, Zeno Gantner, Zombieanalytics, 29 anonymous edits

Kernighan–Lin algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=620239585> *Contributors:* CBM, CarlFeynman, Headbomb, Helios, Johlo, Karanbudhraja, Luoq, Magioladitis, Marmale, NameIsRon, Nanobear, Poochy, Rjwilmsi, RobinK, Ruud Koot, 18 anonymous edits

Tree decomposition *Source:* <http://en.wikipedia.org/w/index.php?oldid=592364676> *Contributors:* Alexdow, Andreas Kaufmann, Bender2k14, Bodlaender, Brutaldeluxe, Dagh, David Eppstein, Gambette, Gosha figosha, Hermel, Herve1729, Kedwar5, MapsMan, Max Librecht, Michael Slone, Oleg Alexandrov, Oliphant, R'n'B, Rjwilmsi, RobinK, SanderEvers, Silverfish, Tarquin, Taxipom, Tizio, Toomim, Treepleks, Twri, Wrp103, Wzhao553, Xnn, Zaslav, III, 24 anonymous edits

Branch-decomposition *Source:* <http://en.wikipedia.org/w/index.php?oldid=607158175> *Contributors:* Andreas Kaufmann, Bender2k14, David Eppstein, Hiiiiiiiiiiiiiiiiiiiiii, Jitse Niesen, LeadSongDog, Marsupilcoalt, Mdd, Peskoj, RobinK, Schmitz, WereSpielChequers, 3 anonymous edits

Path decomposition *Source:* <http://en.wikipedia.org/w/index.php?oldid=620477032> *Contributors:* Bgwhite, David Eppstein, Eran.a, Headbomb, Hermel, Jitse Niesen, Jonesey95, Justin W Smith, Nicole Wein, R'n'B, Rjwilmsi, 8 anonymous edits

Planar separator theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=610327949> *Contributors:* A3 nm, Andreas Kaufmann, AntoniosAntoniadis, Bazonka, David Eppstein, EoGuy, Erel Segal, Fhcy, Gifflite, Headbomb, Hermel, Jeff Erickson, Justin W Smith, Miym, Michael Hardy, Mr. Granger, Oleg Alexandrov, R'n'B, SarielHP, Sodin, Tilieu, 7 anonymous edits

Graph minors *Source:* <http://en.wikipedia.org/w/index.php?oldid=615613527> *Contributors:* A1kmm, Adoarns, AxelBoldt, Bethnim, Cmcq, David Eppstein, Dbenbenn, Dcoetze, Dkostic, Dr.Chuckster, Edemaine, El T, Flamholz, Genusfour, Gifflite, GoingBatty, Hairy Dude, Headbomb, Ilmari_Karonen, JFromm, JHunterJ, Karlhendrikse, Khazar2, Kilom691, Lambiam, Luis Goddyn, MathMartin, Michael Hardy, Msperdock, Myasuda, OldMacDonalds, Phil Boswell, Qutezuce, R'n'B, Rjwilmsi, RobinK, Shahab, Sniffnoy, Taxipom, Tizio, Twri, Wdflake, 26 anonymous edits

Courcelle's theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=617415961> *Contributors:* David Eppstein, Deltahedron, Gifflite, Jochen Burghardt, VanishedUserABC

Robertson–Seymour theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=624198721> *Contributors:* A3 nm, Alex Dainiaik, Altenmann, At-par, AxelBoldt, BeteNoir, Charles Matthews, Cyan, David Eppstein, Dbenbenn, Dcoetze, Dfledmann, Dominus, EmilJ, FvdP, Gifflite, Glasser, Headbomb, Jon Awbrey, Justin W Smith, MSGJ, Mandarax, Melchoir, Michael Hardy, PhS, Populus, Psychonaut, Quuxplusone, R.e.b., R.e.s., RDBury, RobinK, Sfztang, Svick, Tizio, Twri, Udufruduhu, Unzerlegbarkeit, Wzhao553, Zaslav, 18 anonymous edits

Bidimensionality *Source:* <http://en.wikipedia.org/w/index.php?oldid=620655119> *Contributors:* David Eppstein, Diwas, Gosha figosha, Headbomb, Jan Winnicki, Joaquin008, Ruud Koot, Ylloh, 3 anonymous edits

Image Sources, Licenses and Contributors

Image:6n-graf.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:6n-graf.svg> *License:* Public Domain *Contributors:* User:AzaToth

File:Wikipedia multilingual network graph July 2013.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Wikipedia_multilingual_network_graph_July_2013.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Computermagyver

Image:Konigsberg bridges.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Konigsberg_bridges.png *License:* GNU Free Documentation License *Contributors:* Bogdan Giușcă

Image:Wiktionary-logo-en.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Wiktionary-logo-en.svg> *License:* Public Domain *Contributors:* Vectorized by , based on original logo tossed together by Brion Vibber

File:Multi-pseudograph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Multi-pseudograph.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* 0x24a537r9

Image:Directed cycle.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Directed_cycle.svg *License:* Public domain *Contributors:* en:User:Dcoetzee, User:Stannered

Image:Tree graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_graph.svg *License:* Public Domain *Contributors:* Dcoetzee, Tacspacs, Tizio

Image:Complete graph K5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Complete_graph_K5.svg *License:* Public Domain *Contributors:* David Benbennick wrote this file.

File:4-critical graph.png *Source:* http://en.wikipedia.org/w/index.php?title=File:4-critical_graph.png *License:* Creative Commons Zero *Contributors:* Jmerm

File:6n-graf.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:6n-graf.svg> *License:* Public Domain *Contributors:* User:AzaToth

file:Directed.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Directed.svg> *License:* Public Domain *Contributors:* Grafite, Jcb, Josette, 2 anonymous edits

file:Undirected.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Undirected.svg> *License:* Public Domain *Contributors:* JMCC1, Josette, Kilom691, 2 anonymous edits

File:Complete graph K5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Complete_graph_K5.svg *License:* Public Domain *Contributors:* David Benbennick wrote this file.

File:Directed.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Directed.svg> *License:* Public Domain *Contributors:* Grafite, Jcb, Josette, 2 anonymous edits

File:DirectedDegrees.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:DirectedDegrees.svg> *License:* Creative Commons Attribution-Share Alike *Contributors:* Melchoir

File:Directed acyclic graph 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Directed_acyclic_graph_2.svg *License:* Public Domain *Contributors:* Johannes Rössel (talk)

File:4-tournament.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:4-tournament.svg> *License:* Public Domain *Contributors:* Booyabazooka

File:Directed acyclic graph 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Directed_acyclic_graph_3.svg *License:* Public Domain *Contributors:* Directed_acyclic_graph_2.svg: Johannes Rössel (talk) derivative work: Maat (talk)

file:speakerlink-new.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Speakerlink-new.svg> *License:* Creative Commons Zero *Contributors:* User:Kelvinsong

Image:Hasse diagram of powerset of 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hasse_diagram_of_powerset_of_3.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* KSMrq

Image:Simple cycle graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Simple_cycle_graph.svg *License:* Public Domain *Contributors:* Booyabazooka at en.wikipedia

Image:On-graph2.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:6n-graph2.svg> *License:* Public Domain *Contributors:* Booyabazooka, Dcoetzee, 1 anonymous edits

File:Symmetric group 4; Cayley graph 1,5,21 (Nauru Petersen); numbers.svg *Source:* [http://en.wikipedia.org/w/index.php?title=File:Symmetric_group_4;_Cayley_graph_1,5,21_\(Nauru_Petersen\);_numbers.svg](http://en.wikipedia.org/w/index.php?title=File:Symmetric_group_4;_Cayley_graph_1,5,21_(Nauru_Petersen);_numbers.svg) *License:* Public Domain *Contributors:* Lipedia

File:Symmetric group 4; Cayley graph 1,5,21 (adjacency matrix).svg *Source:* [http://en.wikipedia.org/w/index.php?title=File:Symmetric_group_4;_Cayley_graph_1,5,21_\(adjacency_matrix\).svg](http://en.wikipedia.org/w/index.php?title=File:Symmetric_group_4;_Cayley_graph_1,5,21_(adjacency_matrix).svg) *License:* Public Domain *Contributors:* Lipedia

File:Symmetric group 4; Cayley graph 4,9; numbers.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Symmetric_group_4;_Cayley_graph_4,9;_numbers.svg *License:* Public Domain *Contributors:* GrapheCayley-S4-Plan.svg: Fool (talk) derivative work: Lipedia (talk)

File:Symmetric group 4; Cayley graph 4,9 (adjacency matrix).svg *Source:* [http://en.wikipedia.org/w/index.php?title=File:Symmetric_group_4;_Cayley_graph_4,9_\(adjacency_matrix\).svg](http://en.wikipedia.org/w/index.php?title=File:Symmetric_group_4;_Cayley_graph_4,9_(adjacency_matrix).svg) *License:* Public Domain *Contributors:* Lipedia

Image:Commons-logo.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Commons-logo.svg> *License:* logo *Contributors:* Anomie

Image:Depth-first-tree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Depth-first-tree.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Alexander Drichel

Image:graph.traversal.example.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Graph.traversal.example.svg> *License:* GNU Free Documentation License *Contributors:* Miles

Image:Tree edges.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_edges.svg *License:* Public domain *Contributors:* Geraki

Image:If-then-else-control-flow-graph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:If-then-else-control-flow-graph.svg> *License:* Public domain *Contributors:* Geraki, Jcb

File:MAZE 30x20 DFS.ogv *Source:* http://en.wikipedia.org/w/index.php?title=File:MAZE_30x20_DFS.ogv *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Purpy Puppe

Image:Breadth-first-tree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Breadth-first-tree.svg> *License:* Creative Commons Attribution 3.0 *Contributors:* Alexander Drichel

Image:Animated BFS.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Animated_BFS.gif *License:* GNU Free Documentation License *Contributors:* Blake Matheny. Original uploader was Bmatheny at en.wikipedia

Image:MapGermanyGraph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:MapGermanyGraph.svg> *License:* Public Domain *Contributors:* AndreasPraefcke, Mapmarks, MistWiz, Regnaron

Image:GermanyBFS.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:GermanyBFS.svg> *License:* Public Domain *Contributors:* Regnaron

Image:Directed acyclic graph.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Directed_acyclic_graph.png *License:* Public Domain *Contributors:* Anarkman, Dcoetzee, Ddxc, EugenZelenko, Fæ, Joey-das-WBF

File:Dependencygraph.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Dependencygraph.png> *License:* Public Domain *Contributors:* Laurensmast

Image:Pseudoforest.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Pseudoforest.svg> *License:* Public Domain *Contributors:* David Eppstein

File:Ear decomposition.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Ear_decomposition.png *License:* Public Domain *Contributors:* Tmigler. Original uploader was Tmigler at en.wikipedia

Image:Graph cut edges.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_cut_edges.svg *License:* GNU Free Documentation License *Contributors:* User:Booyabazooka

Image:Undirected.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Undirected.svg> *License:* Public Domain *Contributors:* JMCC1, Josette, Kilom691, 2 anonymous edits

Image:Graph-Biconnected-Components.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Graph-Biconnected-Components.svg> *License:* Creative Commons Attribution 3.0 *Contributors:* Original uploader was Zyqqh at en.wikipedia

File:SPQR tree 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:SPQR_tree_2.svg *License:* Creative Commons Zero *Contributors:* User:David Eppstein

File:Min cut example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Min_cut_example.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Kilom691

File:Edge contraction in a multigraph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edge_contraction_in_a_multigraph.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt

File:Single run of Karger's Mincut algorithm.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Single_run_of_Karger's_Mincut_algorithm.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt

File:Spanning tree interpretation of Karger's algorithm.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Spanning_tree_interpretation_of_Karger's_algorithm.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt

File:10 repetitions of Karger's contraction procedure.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:10_repetitions_of_Karger's_contraction_procedure.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt

Image:Scc.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Scc.png> *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* User:Maksim, User:Quackor

File:Tarjan's Algorithm Animation.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Tarjan's_Algorithm_Animation.gif *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Digital Organism

File:Graph suitable for Kameda's method.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_suitable_for_Kameda's_method.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Gbhnt

File:Kameda's algorithm run.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kameda's_algorithm_run.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Gbint

File:Transitive-closure.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Transitive-closure.svg> *License:* Creative Commons Attribution-Share Alike *Contributors:* Anish Bramhandkar

Image:tred-G.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Tred-G.svg> *License:* Public Domain *Contributors:* Dmitry Dzhus, Grafite, Jochen Burghardt, Ksd5

Image:tred-Gprime.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Tred-Gprime.svg> *License:* Creative Commons Zero *Contributors:* User:A3 nm

File:Implication graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Implication_graph.svg *License:* Public Domain *Contributors:* David Eppstein

File:Paint by numbers Animation.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Paint_by_numbers_Animation.gif *License:* Creative Commons Attribution-ShareAlike 3.0 *Contributors:* Unported *Contributors:* Juraj Simlovic

File:2SAT median graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:2SAT_median_graph.svg *License:* Public Domain *Contributors:* David Eppstein

Image:Shortest path with direct weights.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Shortest_path_with_direct_weights.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Artyom Kalinin

Image:Dijkstra Animation.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Dijkstra_Animation.gif *License:* Public Domain *Contributors:* Ibmua

Image:Dijkstras progress animation.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Dijkstras_progress_animation.gif *License:* Creative Commons Attribution 3.0 *Contributors:* Subh83

File:Bellman-Ford worst-case example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bellman-Ford_worst-case_example.svg *License:* Creative Commons Zero *Contributors:* User:Dcoetzee

File:Johnson's algorithm.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Johnson's_algorithm.svg *License:* Public Domain *Contributors:* David Eppstein

File:Floyd-Warshall example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Floyd-Warshall_example.svg *License:* Creative Commons Zero *Contributors:* User:Dcoetzee

File:First_graph.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:First_graph.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Robertoocf1

File:Speaker Icon.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Speaker_Icon.svg *License:* Public Domain *Contributors:* Blast, G.Hagedorn, Jianhui67, Mobius, Tehdog, 3 anonymous edits

Image:Astar progress animation.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Astar_progress_animation.gif *License:* Creative Commons Attribution 3.0 *Contributors:* Subh83

File:AstarExampleEn.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:AstarExampleEn.gif> *License:* Creative Commons Zero *Contributors:* User:CountingPine

Image:Weighted A star with eps 5.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Weighted_A_star_with_eps_5.gif *License:* Creative Commons Attribution 3.0 *Contributors:* Subh83

File:CPT-Graphs-undirected-weighted.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:CPT-Graphs-undirected-weighted.svg> *License:* Creative Commons Zero *Contributors:* Pluke

File:15-node network containing a combination of bi-directional and uni-directional links.png *Source:* http://en.wikipedia.org/w/index.php?title=File:15-node_networkContaining_a_combination_of_bi-directional_and_uni-directional_links.png *License:* Creative Commons Attribution-Share Alike *Contributors:* Ccalmen

File:Centrality.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Centrality.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Rocchini

Image:Graph betweenness.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_betweenness.svg *License:* Creative Commons Attribution 2.5 *Contributors:* Claudio Rocchini

Image:Preferential ballot.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Preferential_ballot.svg *License:* GNU Free Documentation License *Contributors:* Original uploader was Rspeer at en.wikipedia Later version(s) were uploaded by Mark at en.wikipedia

Image:Schulze method example1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 AB.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_AB.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 AC.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_AC.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 AD.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_AD.svg *License:* Creative Commons Attribution-Share Alike *Contributors:* Markus Schulze

Image:Schulze method example1 AE.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_AE.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 BA.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_BA.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 BC.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_BC.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 BD.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_BD.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 BE.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_BE.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 CA.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_CA.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 CB.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_CB.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 CD.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_CD.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 CE.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_CE.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 DA.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_DA.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 DB.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_DB.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 DC.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_DC.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 DE.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_DE.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 EA.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_EA.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 EB.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_EB.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 EC.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_EC.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Schulze method example1 ED.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Schulze_method_example1_ED.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Markus Schulze

Image:Voting2.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Voting2.png> *License:* GNU General Public License *Contributors:* Ral315 and authors of software.

File:Minimum spanning tree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Minimum_spanning_tree.svg *License:* Public Domain *Contributors:* User:Dcoetzee

File:Multiple minimum spanning trees.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Multiple_minimum_spanning_trees.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Ftiercel, Sarang, Wereldburger758

File:Msp-the-cut-correct.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Msp-the-cut-correct.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Kingrhoton

File:Boruvka's algorithm (Sollin's algorithm) Anim.gif *Source:* [http://en.wikipedia.org/w/index.php?title=File:Boruvka%27s_algorithm_\(Sollin%27s_algorithm\)_Anim.gif](http://en.wikipedia.org/w/index.php?title=File:Boruvka%27s_algorithm_(Sollin%27s_algorithm)_Anim.gif) *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Alieseraj

File:Borůvka Algorithm 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bor%C4%9Bvka_Algorithm_1.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Dcoetzee, User:Maksim, User:Alexander Drichel

File:Borůvka Algorithm 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bor%C4%9Bvka_Algorithm_2.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Dcoetzee, User:Maksim, User:Alexander Drichel

File:Borůvka Algorithm 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bor%C4%9Bvka_Algorithm_3.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Dcoetzee, User:Maksim, User:Alexander Drichel

File:MST_kruskal_en.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:MST_kruskal_en.gif *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Schulllz

Image:Kruskal Algorithm 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_1.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_2.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_3.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_4.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_5.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 6.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_6.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

File:Prim's algorithm.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim%27s_algorithm.svg *License:* Creative Commons Zero *Contributors:* User:Dcoetzee

File:MAZE 30x20 Prim.ogv *Source:* http://en.wikipedia.org/w/index.php?title=File:MAZE_30x20_Prim.ogv *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* PurpyPuppe

File:Prim's algorithm_proof.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim%27s_algorithm_proof.svg *License:* Creative Commons Zero *Contributors:* User:Dcoetzee

Image:K-mst 1.pdf *Source:* http://en.wikipedia.org/w/index.php?title=File:K-mst_1.pdf *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Tmigler

Image:K-mst 2.pdf *Source:* http://en.wikipedia.org/w/index.php?title=File:K-mst_2.pdf *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Tmigler (talk) (Uploads)

Image:K-mst 3.pdf *Source:* http://en.wikipedia.org/w/index.php?title=File:K-mst_3.pdf *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Tmigler (talk) (Uploads)

Image:Prim Maze.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Maze.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Nandhp

File:Depth-First_Search_Animation.ogv *Source:* http://en.wikipedia.org/w/index.php?title=File:Depth-First_Search_Animation.ogv *License:* GNU Free Documentation License *Contributors:* Ofek Gila

File:Horizontally_Influenced_Depth-First_Search_Generated_Maze.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Horizontally_Influenced_Depth-First_Search_Generated_Maze.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Ofek Gila

File:KruskalGeneratedMaze.webm *Source:* <http://en.wikipedia.org/w/index.php?title=File:KruskalGeneratedMaze.webm> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Veega

Image:Chamber.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Chamber.png> *License:* Public domain *Contributors:* Simplex (talk) (Uploads)

Image:Chamber division.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_division.png *License:* Public domain *Contributors:* Simplex (talk) (Uploads)

Image:Chamber divided.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_divided.png *License:* Public domain *Contributors:* Simplex (talk) (Uploads)

Image:Chamber subdivision.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_subdivision.png *License:* Public domain *Contributors:* Simplex (talk) (Uploads)

Image:Chamber finished.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_finished.png *License:* Public domain *Contributors:* Simplex (talk) (Uploads)

File:Prim Maze 3D.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Maze_3D.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Prim_Maze.svg; Nandhp derivative work: SharkD Talk

File:Brute force Clique algorithm.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Brute_force_Clique_algorithm.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt (talk)

File:6n-graf-clique.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:6n-graf-clique.svg> *License:* Public Domain *Contributors:* Dcoetzee, Erin Silversmith, GeorgHH, 1 anonymous edits

File:Permutation graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Permutation_graph.svg *License:* Public domain *Contributors:* Lyonsam

File:Sat reduced to Clique from Sipser.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Sat_reduced_to_Clique_from_Sipser.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt (talk)

File:Monotone circuit for 3-clique.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Monotone_circuit_for_3-clique.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt (talk)

File:Decision tree for 3-clique no arrowheads.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Decision_tree_for_3-clique_no_arrowheads.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt (talk)

File:Cube-face-intersection-graph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Cube-face-intersection-graph.svg> *License:* Public Domain *Contributors:* David Eppstein

File:Independent set graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Independent_set_graph.svg *License:* GNU Free Documentation License *Contributors:* Life of Riley

File:Cube-maximal-independence.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Cube-maximal-independence.svg> *License:* Public Domain *Contributors:* David Eppstein

Image:Petersen graph 3-coloring.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Petersen_graph_3-coloring.svg *License:* Public Domain *Contributors:* Booyabazooka, Dcoetzee, Mate2code

File:Graph with all three-colourings 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_with_all_three-colourings_2.svg *License:* GNU Free Documentation License *Contributors:* (PNG file), (corrections + SVG conversion)

Image:Chromatic polynomial of all 3-vertex graphs.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chromatic_polynomial_of_all_3-vertex_graphs.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt (talk)

Image:3-coloringEx.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:3-coloringEx.svg> *License:* GNU Free Documentation License *Contributors:* Booyabazooka, Dcoetzee

Image:Greedy colourings.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Greedy_colourings.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Thore Husfeldt

Image:Simple-bipartite-graph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Simple-bipartite-graph.svg> *License:* Public Domain *Contributors:* MistWiz

File:Biclique K 3 5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Biclique_K_3_5.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Koko90

File:Odd Cycle Transversal of size 2.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Odd_Cycle_Transversal_of_size_2.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Pgdx

File:Vertex-cover.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Vertex-cover.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:Minimum-vertex-cover.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Minimum-vertex-cover.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:Vertex-cover-from-maximal-matching.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Vertex-cover-from-maximal-matching.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:Dominating-set.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Dominating-set.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:Dominating-set-2.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Dominating-set-2.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:Dominating-set-reduction.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Dominating-set-reduction.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:konigsburg graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Konigsburg_graph.svg *License:* GNU Free Documentation License *Contributors:* AnonMoos, McZusatz, Piotrus, Riojajar, Squizzz

File:Labelled Eulergraph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Labelled_Eulergraph.svg *License:* Public Domain *Contributors:* S Sepp

Image:Hamiltonian path.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hamiltonian_path.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* Christoph Sommer

Image:Herschel graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Herschel_graph.svg *License:* Public Domain *Contributors:* David Eppstein

Image:William Rowan Hamilton painting.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:William_Rowan_Hamilton_painting.jpg *License:* Public Domain *Contributors:* Quibik, 1 anonymous edits

Image:Weighted K4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Weighted_K4.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Sdo

Image:Aco TSP.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Aco_TSP.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* User:Nojhan, User:Mastarh

File:AntColony.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:AntColony.gif> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Saurabh.harsh

File:Metrischer Graph mit 5 Knoten.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Metrischer_Graph_mit_5_Knoten.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:Christofides MST.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Christofides_MST.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:V'.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:V'.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:G V'.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:G_V'.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:Christofides Matching.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Christofides_Matching.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:TuM.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:TuM.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:Eulertour.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Eulertour.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:Eulertour bereinigt.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Eulertour_bereinigt.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Mastarh

File:Maximal-matching.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Maximal-matching.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:Maximum-matching-labels.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Maximum-matching-labels.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Miym

File:Edmonds augmenting path.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds_augmenting_path.svg *License:* Creative Commons Zero *Contributors:* User:A3 nm

File:Edmonds blossom.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds_blossom.svg *License:* Creative Commons Zero *Contributors:* User:A3 nm

File:Edmonds lifting path.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds_lifting_path.svg *License:* Creative Commons Zero *Contributors:* User:A3 nm

File:Edmonds lifting end point.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds_lifting_end_point.svg *License:* Creative Commons Zero *Contributors:* User:A3 nm

File:forest expansion.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Forest_expansion.png *License:* Public Domain *Contributors:* Original uploader was Markoid at en.wikipedia

File:blossom contraction.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Blossom_contraction.png *License:* Public Domain *Contributors:* Original uploader was Markoid at en.wikipedia

File:path detection.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Path_detection.png *License:* Public Domain *Contributors:* Original uploader was Markoid at en.wikipedia

File:path lifting.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Path_lifting.png *License:* Public Domain *Contributors:* Original uploader was Markoid at en.wikipedia

File:Pfaffian orientation via FKT algorithm example.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Pfaffian_orientation_via_FKT_algorithm_example.gif *License:* Creative Commons Attribution 3.0 *Contributors:* Original uploader was Bender2k14 at en.wikipedia

File:Gale-Shapley.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Gale-Shapley.gif> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User A1

File:Max flow.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Max_flow.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* Min_cut.png; Maksim derivative work: Cyhawk (talk)

File:MFP1.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:MFP1.jpg> *License:* Creative Commons Attribution 2.0 *Contributors:* Csfpypwaiting

File:Multi-source multi-sink flow problem.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Multi-source_multi-sink_flow_problem.svg *License:* Public domain *Contributors:* Chin Ho Lee

File:Maximum bipartite matching to max flow.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Maximum_bipartite_matching_to_max_flow.svg *License:* Public domain *Contributors:* Chin Ho Lee

File:Node splitting.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Node_splitting.svg *License:* Public domain *Contributors:* Chin Ho Lee

File:Baseball Elimination Problem.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Baseball_Elimination_Problem.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Arash.nouri

File:max-flow min-cut example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Max-flow_min-cut_example.svg *License:* Public Domain *Contributors:* Chin Ho Lee

File:Max-flow min-cut project-selection.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Max-flow_min-cut_project-selection.svg *License:* Public Domain *Contributors:* Chin Ho Lee

File:Image segmentation.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Image_segmentation.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Sharunumil

Image:Ford-Fulkerson example 0.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ford-Fulkerson_example_0.svg *License:* GNU Free Documentation License *Contributors:* en:User:Cburnett

Image:Ford-Fulkerson example 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ford-Fulkerson_example_1.svg *License:* GNU Free Documentation License *Contributors:* en:User:Cburnett

Image:Ford-Fulkerson example 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ford-Fulkerson_example_2.svg *License:* GNU Free Documentation License *Contributors:* en:User:Cburnett

Image:Ford-Fulkerson example final.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ford-Fulkerson_example_final.svg *License:* GNU Free Documentation License *Contributors:* en:User:Cburnett

File:Ford-Fulkerson forever.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ford-Fulkerson_forever.svg *License:* Creative Commons Attribution 3.0 *Contributors:* Svick

file:Commons-logo.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Commons-logo.svg> *License:* logo *Contributors:* Anomie

Image:Wikibooks-logo-en-noslogan.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Wikibooks-logo-en-noslogan.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Bastique, User:Ramac et al.

Image:Edmonds-Karp flow example 0.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds-Karp_flow_example_0.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* en:User:Cburnett

Image:Edmonds-Karp flow example 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds-Karp_flow_example_1.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* en:User:Cburnett

Image:Edmonds-Karp flow example 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds-Karp_flow_example_2.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* en:User:Cburnett

Image:Edmonds-Karp flow example 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds-Karp_flow_example_3.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* en:User:Cburnett

Image:Edmonds-Karp flow example 4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Edmonds-Karp_flow_example_4.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* en:User:Cburnett

File:Dinic algorithm G1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_G1.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm Gf1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_Gf1.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm GL1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_GL1.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm G2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_G2.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm Gf2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_Gf2.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm GL2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_GL2.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm G3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_G3.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm Gf3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_Gf3.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

File:Dinic algorithm GL3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dinic_algorithm_GL3.svg *License:* Public Domain *Contributors:* Chin Ho Lee. Original uploader was Teshasapose at en.wikipedia

Image:closure.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Closure.png> *License:* Creative Commons Attribution 3.0 *Contributors:* Faridani

File:Minimum weight bipartite matching.pdf *Source:* http://en.wikipedia.org/w/index.php?title=File:Minimum_weight_bipartite_matching.pdf *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Arash.nouri

Image:Butterfly graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Butterfly_graph.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Koko90

Image:CGK4PLN.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:CGK4PLN.svg> *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* User:Yecril71pl, User:Yecril71pl

Image:Biclique K 3 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Biclique_K_3_3.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Koko90

Image:Nonplanar no subgraph K 3 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Nonplanar_no_subgraph_K_3_3.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Daniel M. Short

File:Kuratowski.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Kuratowski.gif> *License:* Creative Commons Attribution 3.0 *Contributors:* Pablo Angulo using Sage software

File:Dodecahedron schlegel diagram.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Dodecahedron_schlegel_diagram.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* David Eppstein, Stannic

File:Circle packing theorem K5 minus edge example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circle_packing_theorem_K5_minus_edge_example.svg *License:* Creative Commons Zero *Contributors:* Dcoetzee

File:Goldner-Harary graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Goldner-Harary_graph.svg *License:* Public Domain *Contributors:* David Eppstein

Image:dual graphs.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Dual_graphs.svg *License:* Public Domain *Contributors:* Original uploader was Booyabazooka at en.wikipedia

File:Duals graphs.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Duals_graphs.svg *License:* Creative Commons Zero *Contributors:* Self

Image:Noniso dual graphs.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Noniso_dual_graphs.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Kirelagin

File:Fary-induction.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Fary-induction.svg> *License:* Public Domain *Contributors:* Original uploader was David Eppstein at en.wikipedia

File:Schegel diagram as shadow.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Schegel_diagram_as_shadow.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* David Eppstein

Image:leftright1.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Leftright1.png> *License:* GNU Free Documentation License *Contributors:* Dcoetzee, Maksim

Image:leftright2.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Leftright2.png> *License:* GNU Free Documentation License *Contributors:* Dcoetzee, Maksim

Image:leftright3.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Leftright3.png> *License:* GNU Free Documentation License *Contributors:* Dcoetzee, Maksim

File:WorldWideWebAroundWikipedia.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:WorldWideWebAroundWikipedia.png> *License:* GNU Free Documentation License *Contributors:* Bawolff, Chris 73, Flappiefl, Fried-peach, Guillom, Kozuch, LX, Mattes, Mdd, Mschlindwein, Plugwash, SheeEttin, Tacspacis, Timeshifter, Ustas, Zarex, 1 anonymous edits

File:4node-digraph-natural.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:4node-digraph-natural.svg> *License:* Creative Commons Zero *Contributors:* User:Pointillist

File:4node-digraph-embed.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:4node-digraph-embed.svg> *License:* Creative Commons Zero *Contributors:* User:Pointillist

File:Goldner-Harary-linear.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Goldner-Harary-linear.svg> *License:* Creative Commons Zero *Contributors:* User:David Eppstein

Image:Visualization of wiki structure using prefuse visualization package.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Visualization_of_wiki_structure_using_prefuse_visualization_package.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Chris Davis at en.wikipedia

File:DC++_derivatives.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:DC++_derivatives.svg *License:* Creative Commons Zero *Contributors:* Tehnick

File:Upward planar drawing.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Upward_planar_drawing.svg *License:* Creative Commons Zero *Contributors:* User:David Eppstein

File:Not upward planar.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Not_upward_planar.svg *License:* Creative Commons Zero *Contributors:* User:David Eppstein

Image:Social-network.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Social-network.svg> *License:* Public Domain *Contributors:* User:Wykis

Image:Screen_Shot_2012-07-19_at_5.56.57_PM.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Screen_Shot_2012-07-19_at_5.56.57_PM.png *License:* Public Domain *Contributors:* Wxidea

Image:Conceptmap.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Conceptmap.gif> *License:* GNU Free Documentation License *Contributors:* Vicwood40

Image:Wikiversity-logo.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Wikiversity-logo.svg> *License:* logo *Contributors:* Snorky (optimized and cleaned up by verdy_-p)

Image:Interval graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Interval_graph.svg *License:* Public Domain *Contributors:* David Eppstein

Image:Chordal-graph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Chordal-graph.svg> *License:* Public Domain *Contributors:* Dcoetzee, Grafite, Tizio

Image:Tree decomposition.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_decomposition.svg *License:* Public Domain *Contributors:* David Eppstein

Image:Paley9-perfect.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Paley9-perfect.svg> *License:* Public Domain *Contributors:* Original uploader was David Eppstein at en.wikipedia

File:7-hole and antihole.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:7-hole_and_antihole.svg *License:* Creative Commons Zero *Contributors:* User:David Eppstein

Image:Intersection graph.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Intersection_graph.gif *License:* Creative Commons Attribution 2.5 *Contributors:* Claudio Rocchini

Image:Unit disk graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Unit_disk_graph.svg *License:* Public Domain *Contributors:* David Eppstein at en.wikipedia

File:Line graph construction 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Line_graph_construction_1.svg *License:* GNU Free Documentation License *Contributors:* User:Booyabazooka

File:Line graph construction 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Line_graph_construction_2.svg *License:* GNU Free Documentation License *Contributors:* User:Booyabazooka

File:Line graph construction 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Line_graph_construction_3.svg *License:* GNU Free Documentation License *Contributors:* User:Booyabazooka

File:Line graph construction 4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Line_graph_construction_4.svg *License:* GNU Free Documentation License *Contributors:* User:Booyabazooka

File:Diamond graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Diamond_graph.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Koko90

File:Line perfect graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Line_perfect_graph.svg *License:* Creative Commons Zero *Contributors:* User:David Eppstein

File:Line graph clique partition.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Line_graph_clique_partition.svg *License:* Creative Commons Zero *Contributors:* User:David Eppstein

File:LineGraphExampleA.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:LineGraphExampleA.svg> *License:* Public Domain *Contributors:* Ilmari Karonen ()

File:Forbidden line subgraphs.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Forbidden_line_subgraphs.svg *License:* Public Domain *Contributors:* Originally uploaded as a png image by David Eppstein at en.wikipedia. Redrawn as svg by Brainbrain0000 at en.wikipedia

File:DeBruijn-as-line-digraph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:DeBruijn-as-line-digraph.svg> *License:* Public Domain *Contributors:* David Eppstein

Image:K_3,1 Claw.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:K_3,1_Claw.svg *License:* Public Domain *Contributors:* drange (talk)

Image:Complete bipartite graph K3,1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Complete_bipartite_graph_K3,1.svg *License:* Public Domain *Contributors:* User:Dbenbenn

File:Icosahedron.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Icosahedron.svg> *License:* GNU Free Documentation License *Contributors:* User:DTR

File:Sumner claw-free matching.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Sumner_claw-free_matching.svg *License:* Public Domain *Contributors:* David Eppstein

File:Claw-free augmenting path.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Claw-free_augmenting_path.svg *License:* Public Domain *Contributors:* David Eppstein

File:Median graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Median_graph.svg *License:* Public Domain *Contributors:* David Eppstein

File:Tree median.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_median.svg *License:* Public Domain *Contributors:* David Eppstein

File:Squaregraph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Squaregraph.svg> *License:* Public Domain *Contributors:* David Eppstein

File:Distributive lattice example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Distributive_lattice_example.svg *License:* Public Domain *Contributors:* David Eppstein

File:Cube retraction.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Cube_retraction.svg *License:* Public Domain *Contributors:* David Eppstein

File:Median from triangle-free.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Median_from_triangle-free.svg *License:* Public Domain *Contributors:* David Eppstein

File:Buneman graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Buneman_graph.svg *License:* Creative Commons Attribution-Share Alike *Contributors:* David Eppstein

File:Graph-Cartesian-product.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Graph-Cartesian-product.svg> *License:* Public Domain *Contributors:* Original uploader was David Eppstein at en.wikipedia

Image:Graph isomorphism a.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_isomorphism_a.svg *License:* GNU Free Documentation License *Contributors:* User:Booyabazooka

Image:Graph isomorphism b.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_isomorphism_b.svg *License:* GNU Free Documentation License *Contributors:* User:Booyabazooka

Image:Whitneys theorem exception.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Whitneys_theorem_exception.svg *License:* Public Domain *Contributors:* User:Dcoetzee *derivative work: Dcoetzee (talk) Complete_graph_K3.svg: Dbenbenn Complete_bipartite_graph_K3,1.svg: Dbenbenn

File:Bisected network.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bisected_network.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Sohini6685

File:Connected graph.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Connected_graph.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Sohini6685

File:Graph comparison.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_comparison.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Sohini6685

File:Branch-decomposition.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Branch-decomposition.svg> *License:* Public Domain *Contributors:* David Eppstein

File:Branchwidth 3-forbidden minors.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Branchwidth_3-forbidden_minors.svg *License:* Public Domain *Contributors:* David Eppstein

File:Interval graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Interval_graph.svg *License:* Public Domain *Contributors:* David Eppstein

File:Caterpillar tree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Caterpillar_tree.svg *License:* Public Domain *Contributors:* David Eppstein

File:Pathwidth-1 obstructions.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Pathwidth-1_obstructions.svg *License:* Public Domain *Contributors:* David Eppstein

File:Grid separator.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Grid_separator.svg *License:* Public Domain *Contributors:* David Eppstein

File:Geode10.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Geode10.png> *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Original uploader was Theon at fr.wikipedia

File:Unit disk graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Unit_disk_graph.svg *License:* Public Domain *Contributors:* David Eppstein at en.wikipedia

Image:GraphMinorExampleA.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:GraphMinorExampleA.png> *License:* Public Domain *Contributors:* User:Ilmari Karonen, User:Maksim

Image:GraphMinorExampleB.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:GraphMinorExampleB.svg> *License:* Public Domain *Contributors:* El T

Image:GraphMinorExampleC.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:GraphMinorExampleC.svg> *License:* GNU Free Documentation License *Contributors:* Grafite, Kilom691, Marnanel

File:Petersen family.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Petersen_family.svg *License:* Public Domain *Contributors:* David Eppstein

Image:Gamma graph.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Gamma_graph.jpg *License:* Public Domain *Contributors:* Gosha figosha

License

Creative Commons Attribution-Share Alike 3.0
[//creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)