

Graph Algorithms

Contents

Articles

Introduction	1
Graph theory	1
Glossary of graph theory	8
Undirected graphs	19
Directed graphs	26
Directed acyclic graphs	28
Computer representations of graphs	32
Adjacency list	35
Adjacency matrix	37
Implicit graph	40
Graph exploration and vertex ordering	44
Depth-first search	44
Breadth-first search	49
Lexicographic breadth-first search	52
Iterative deepening depth-first search	54
Topological sorting	57
Application: Dependency graphs	60
Connectivity of undirected graphs	62
Connected components	62
Edge connectivity	64
Vertex connectivity	65
Menger's theorems on edge and vertex connectivity	66
Ear decomposition	67
Algorithms for 2-edge-connected components	70
Algorithms for 2-vertex-connected components	72
Algorithms for 3-vertex-connected components	73
Karger's algorithm for general vertex connectivity	76
Connectivity of directed graphs	82
Strongly connected components	82
Tarjan's strongly connected components algorithm	83
Path-based strong component algorithm	86

Kosaraju's strongly connected components algorithm	87
Application: 2-satisfiability	88
Shortest paths	101
Shortest path problem	101
Dijkstra's algorithm for single-source shortest paths with positive edge lengths	106
Bellman–Ford algorithm for single-source shortest paths allowing negative edge lengths	112
Johnson's algorithm for all-pairs shortest paths in sparse graphs	115
Floyd–Warshall algorithm for all-pairs shortest paths in dense graphs	118
Suurballe's algorithm for two shortest disjoint paths	122
Bidirectional search	125
A* search algorithm	127
Longest path problem	134
Widest path problem	137
Canadian traveller problem	141
Application: Centrality analysis of social networks	144
Application: Schulze voting system	149
Minimum spanning trees	164
Minimum spanning tree	164
Borůvka's algorithm	170
Kruskal's algorithm	171
Prim's algorithm	174
Edmonds's algorithm for directed minimum spanning trees	177
Degree-constrained spanning tree	180
Maximum-leaf spanning tree	181
K-minimum spanning tree	182
Capacitated minimum spanning tree	184
Application: Single-linkage clustering	185
Application: Maze generation	187
Cliques, independent sets, and coloring	193
Clique problem	193
Bron–Kerbosch algorithm for listing all maximal cliques	206
Independent set problem	210
Maximal independent set	214
Graph coloring	218
Bipartite graph	232

Greedy coloring	237
Application: Register allocation	239
Covering and domination	242
Vertex cover	242
Dominating set	247
Feedback vertex set	251
Feedback arc set	254
Tours	256
Eulerian path	256
Hamiltonian path	260
Hamiltonian path problem	263
Travelling salesman problem	265
Bottleneck traveling salesman problem	278
Christofides' heuristic for the TSP	278
Route inspection problem	279
Matching	281
Matching	281
Hopcroft–Karp algorithm for maximum matching in bipartite graphs	285
Edmonds's algorithm for maximum matching in non-bipartite graphs	289
Assignment problem	296
Hungarian algorithm for the assignment problem	298
FKT algorithm for counting matchings in planar graphs	303
Stable marriage problem	306
Stable roommates problem	309
Permanent	311
Computing the permanent	314
Network flow	318
Maximum flow problem	318
Max-flow min-cut theorem	322
Ford–Fulkerson algorithm for maximum flows	325
Edmonds–Karp algorithm for maximum flows	330
Dinic's algorithm for maximum flows	333
Push-relabel maximum flow algorithm	335
Closure problem	342
Minimum cost flow problem	345

Graph drawing and planar graphs	347
Planar graph	347
Dual graph	353
Fáry's theorem	355
Steinitz's theorem	357
Planarity testing	360
Fraysseix–Rosenstiehl's planarity criterion	362
Graph drawing	363
Force-based graph drawing algorithms	368
Graph embedding	372
Application: Sociograms	374
Application: Concept maps	375
Special classes of graphs	378
Interval graph	378
Chordal graph	381
Perfect graph	384
Intersection graph	388
Unit disk graph	390
Line graph	392
Claw-free graph	397
Median graph	403
Graph isomorphism	414
Graph isomorphism	414
Graph isomorphism problem	416
Graph canonization	423
Subgraph isomorphism problem	424
Color-coding	426
Induced subgraph isomorphism problem	429
Maximum common subgraph isomorphism problem	430
Graph decomposition and graph minors	431
Graph partition	431
Kernighan–Lin algorithm	435
Tree decomposition	436
Branch-decomposition	441
Path decomposition	445

Planar separator theorem	457
Graph minors	469
Courcelle's theorem	474
Robertson–Seymour theorem	475
Bidimensionality	479

References

Article Sources and Contributors	482
Image Sources, Licenses and Contributors	489

Article Licenses

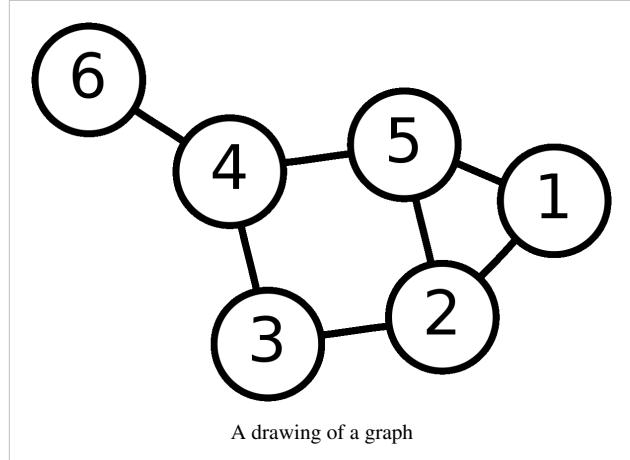
License	494
---------	-----

Introduction

Graph theory

Further information: Graph (mathematics) and Glossary of graph theory

In mathematics and computer science, **graph theory** is the study of *graphs*, which are mathematical structures used to model pairwise relations between objects from a certain collection. A "graph" in this context is a collection of "vertices" or "nodes" and a collection of *edges* that connect pairs of vertices. 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.



The graphs studied in graph theory should not be confused with the graphs of functions or other kinds of graphs.

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

Applications

Graphs are among the most ubiquitous models of both natural and human-made structures. They can be used to model many types of relations and process dynamics in physical, biological^[1] and social systems. Many problems of practical interest 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. One practical example: The link structure of a website could be represented by a directed graph. The vertices are the web pages available at the website and a directed edge from page *A* to page *B* exists if and only if *A* contains a link to *B*. 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. There, the transformation of graphs is often formalized and represented by graph rewrite systems. They are either directly used or properties of the rewrite systems (e.g. confluence) are studied. Complementary to graph transformation systems focussing 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 (HPSG) model syntactic constructions via the unification of 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^[2], 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. For example, Franzblau's shortest-path (SP) rings. 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.

Graph theory is also widely used in sociology as a way, for example, to measure actors' prestige or to explore diffusion mechanisms, notably through the use of social network analysis software.

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, e.g. 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.

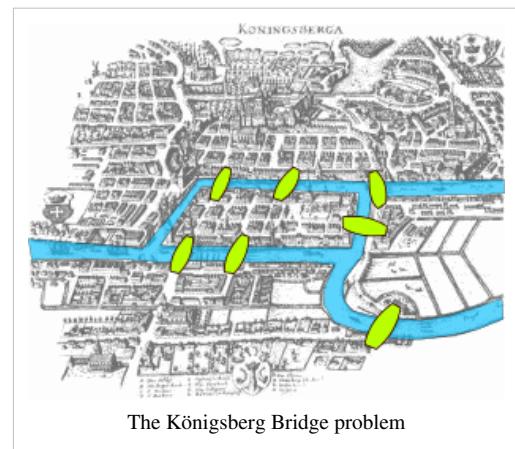
A digraph with weighted edges in the context of graph theory is called a network. Network analysis have many practical applications, for example, to model and analyze traffic networks. Applications of network analysis split broadly into three categories:

1. First, analysis to determine structural properties of a network, such as the distribution of vertex degrees and the diameter of the graph. A vast number of graph measures exist, and the production of useful ones for various domains remains an active area of research.
2. Second, analysis to find a measurable quantity within the network, for example, for a transportation network, the level of vehicular flow within any portion of it.
3. Third, analysis of dynamical properties of networks.

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.^[3] 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^[4] and L'Huillier,^[5] 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.^[6] 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:^[7]

"[...] 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.^[8] A later textbook by Frank Harary, published in 1969, was enormously popular, 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.^[9]

One of the most famous and productive problems of 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.^[10] A computer-aided proof produced in 1976 by Kenneth Appel and Wolfgang Haken makes fundamental use of the notion of "discharging" developed by Heesch.^{[11][12]} 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.^[13]

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

Graphs are represented graphically 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

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

Incidence list

The edges are represented by an array containing pairs (tuples if directed) of vertices (that the edge connects) and possibly weight and other data. Vertices connected by an edge are said to be *adjacent*.

Adjacency list

Much like the incidence list, each vertex has a list of which vertices it is adjacent to. This causes redundancy in an undirected graph: for example, if vertices A and B are adjacent, A's adjacency list contains B, while B's list contains A. Adjacency queries are faster, at the cost of extra storage space.

Matrix structures

Incidence matrix

The graph is represented by a matrix of size $|V|$ (number of vertices) by $|E|$ (number of edges) where the entry [vertex, edge] contains the edge's endpoint data (simplest case: 1 - incident, 0 - not incident).

Adjacency matrix

This is an n by n matrix A , where n is the number of vertices in the graph. If there is an edge from a vertex x to a vertex y , then the element $a_{x,y}$ is 1 (or in general the number of xy edges), otherwise it is 0. In computing, this matrix makes it easy to find subgraphs, and to reverse a directed graph.

Laplacian matrix or "Kirchhoff matrix" or "Admittance matrix"

This is defined as $D - A$, where D is the diagonal degree matrix. It explicitly contains both adjacency information and degree information. (However, there are other, similar matrices that are also called "Laplacian matrices" of a graph.)

Distance matrix

A symmetric n by n matrix D , where n is the number of vertices in the graph. The element $d_{x,y}$ is the length of a shortest path between x and y ; if there is no such path $d_{x,y} = \infty$. It can be derived from powers of A

$$d_{x,y} = \min\{n \mid A^n[x, y] \neq 0\}.$$

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 (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 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

Graph classes

Many problems involve characterizing the members of various classes of graphs. Overlapping significantly with other types in this list, this type of problem includes, for instance:

- 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] Mashaghi, A.; et al. (2004). "Investigation of a protein complex network". *European Physical Journal B* **41** (1): 113–121. doi:10.1140/epjb/e2004-00301-0.
- [2] <http://www.textgraphs.org>
- [3] Biggs, N.; Lloyd, E. and Wilson, R. (1986), *Graph Theory, 1736–1936*, Oxford University Press
- [4] Cauchy, A.L. (1813), "Recherche sur les polyèdres - premier mémoire", *Journal de l'École Polytechnique* **9** (Cahier 16): 66–86.
- [5] L'Huillier, S.-A.-J. (1861), "Mémoire sur la polyèdrométrie", *Annales de Mathématiques* **3**: 169–189.
- [6] Cayley, A. (1875), "Ueber die Analytischen Figuren, welche in der Mathematik Bäume genannt werden und ihre Anwendung auf die Theorie chemischer Verbindungen", *Berichte der deutschen Chemischen Gesellschaft* **8** (2): 1056–1059, doi:10.1002/cber.18750080252.
- [7] John Joseph Sylvester (1878), *Chemistry and Algebra*. Nature, volume 17, page 284. doi:10.1038/017284a0. Online version (<http://www.archive.org/stream/nature15unkngoog#page/n312/mode/1up>). Retrieved 2009-12-30.
- [8] Tutte, W.T. (2001), *Graph Theory* (<http://books.google.com/books?id=uTGhooU37h4C&pg=PA30>), Cambridge University Press, p. 30, ISBN 978-0-521-79489-3 .
- [9] Society for Industrial and Applied Mathematics (2002), "The George Polya Prize" (<http://www.siam.org/about/more/siam50.pdf>), *Looking Back, Looking Ahead: A SIAM History*, p. 26, .
- [10] Heinrich Heesch: Untersuchungen zum Vierfarbenproblem. Mannheim: Bibliographisches Institut 1969.
- [11] Appel, K. and Haken, W. (1977), "Every planar map is four colorable. Part I. Discharging", *Illinois J. Math.* **21**: 429–490.
- [12] Appel, K. and Haken, W. (1977), "Every planar map is four colorable. Part II. Reducibility", *Illinois J. Math.* **21**: 491–567.
- [13] Robertson, N.; Sanders, D.; Seymour, P. and Thomas, R. (1997), "The four color theorem", *Journal of Combinatorial Theory Series B* **70**: 2–44, doi:10.1006/jctb.1997.1750.

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

- Graph Theory with Applications (<http://www.math.jussieu.fr/~jabondy/books/gtwa/gtwa.html>) (1976) by Bondy and Murty
- 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 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 (<http://web.archive.org/web/20060206155001/http://www.nd.edu/~networks/gallery.htm>)
- 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

Glossary of graph theory

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 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.

The **size** of a graph is the number of its edges, i.e. $|E(G)|$.^[1]

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.)

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 .

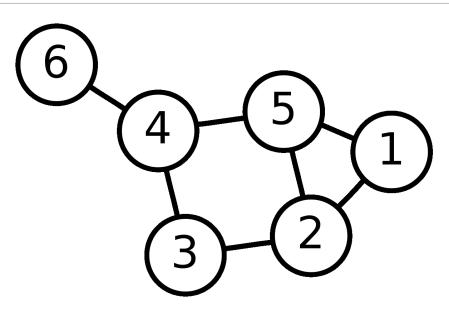
Occasionally the term **cotriangle** or **anti-triangle** is used for a set of three vertices none of which are connected.

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 .



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

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 that does *not* contain H as an induced subgraph is said to be **H -free**.

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.

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 $P(H)$.

Many important classes of graphs can be defined by sets of forbidden subgraphs, the minimal graphs that are not in the class.

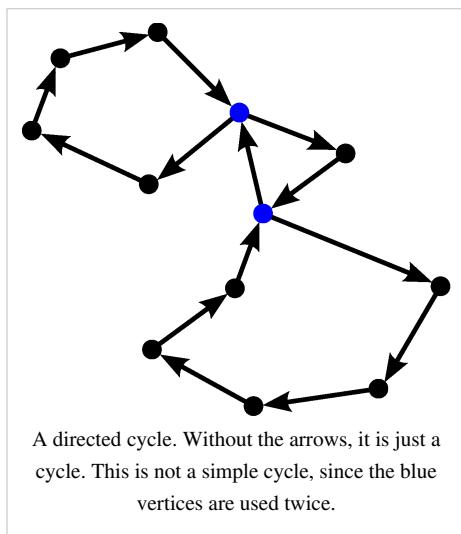
Walks

A **walk** is an alternating sequence of vertices and edges, beginning and ending with a vertex, where each vertex is incident to both the edge that precedes it and the edge that follows it in the sequence, and where the vertices that precede and follow an edge are the 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).

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. 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.)

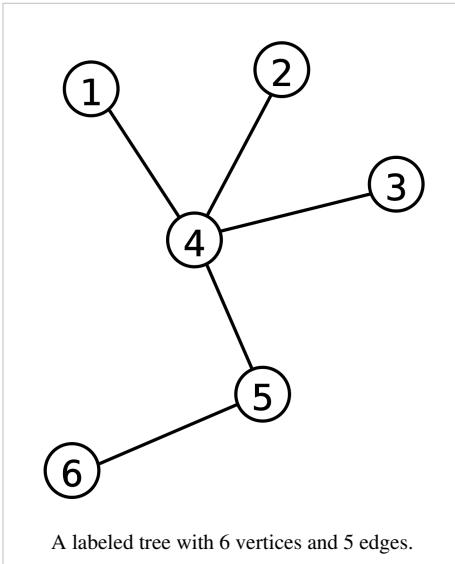
A **subforest** of the forest F is a subgraph of F .

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**.



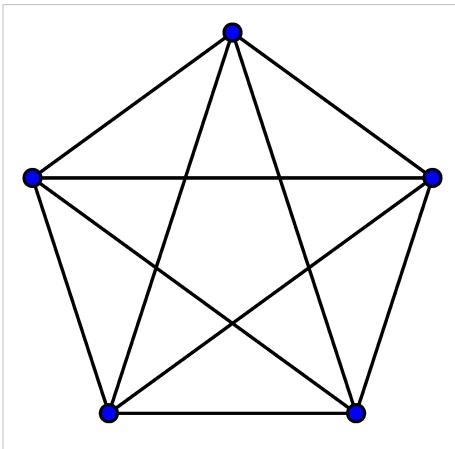
A labeled tree with 6 vertices and 5 edges.

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 .



K_5 , a complete graph. If a subgraph looks like this, the vertices in that subgraph form a clique of size 5.

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.

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 (disjoint from all other such paths) 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 equal to two times the number of edges, loops included. This means that for a graph with 3 vertices with each vertex having a degree of two (i.e. a triangle) the total degree would be six (e.g. $3 \times 2 = 6$). The general formula for this is **total degree = $2n$** where **n = number of edges**.

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 *coclique*, 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 above, vertices 1, 3, and 6 form an independent set; and 3, 5, and 6 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.^[2]

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 **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 *cut vertex* is an analogous vertex (see above).

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** $e_G(v)$ 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

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 graph G is an independent set S so that $(G \setminus S)$ is an out-dominating set. 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

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

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.

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] Harris, John M. (2000). *Combinatorics and Graph Theory* ([http://www.springer.com/new+&+forthcoming+titles+\(default\)/book/978-0-387-79710-6](http://www.springer.com/new+&+forthcoming+titles+(default)/book/978-0-387-79710-6)). New York: Springer-Verlag. pp. 5. ISBN 0-387-98736-3..
- [2] Neel, David L. (2006), "The linear complexity of a graph" (http://www.emis.ams.org/journals/EJC/Volume_13/PDF/v13i1r9.pdf), *The electronic journal of combinatorics*,
- [3] Béla Bollobás, *Modern Graph theory*, p. 298
- Graph Triangulation, Jayson Rome, October 14, 2002 (http://prl.cs.gc.cuny.edu/web/LabWebsite/Rome/jrome/Slides_Tutorials/GraphTriangulation.pdf)
- Bollobás, Béla (1998). *Modern Graph Theory*. New York: Springer-Verlag. ISBN 0-387-98488-7. [Packed with advanced topics followed by a historical overview at the end of each chapter.]
- Diestel, Reinhard (2005). *Graph Theory* (<http://www.math.uni-hamburg.de/home/diestel/books/graph-theory/>) (3rd edition ed.). Graduate Texts in Mathematics, vol. 173, Springer-Verlag. ISBN 3-540-26182-6. [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. Free electronic version is available.]
- 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>)" from 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/>

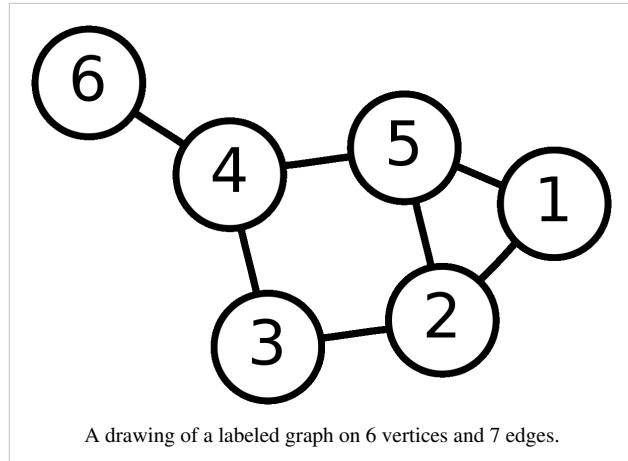
Undirected graphs

Further information: Graph theory

In mathematics, a **graph** is an abstract representation of a set of objects where some pairs of the 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*.^[1] 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 (asymmetric) or undirected (symmetric). 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. On the other hand, if the vertices represent people at a party, and 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 *lines* or *arcs*. Graphs are the basic subject studied by graph theory. The word "graph" was first used in this sense by J.J. Sylvester in 1878.^[2]



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

Definitions

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

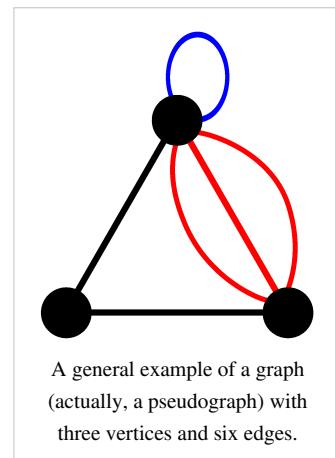
Graph

In the most common sense of the term,^[3] 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 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,^[4] 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.



A general example of a graph (actually, a pseudograph) with three vertices and six edges.

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 .

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 $u \sim v$.

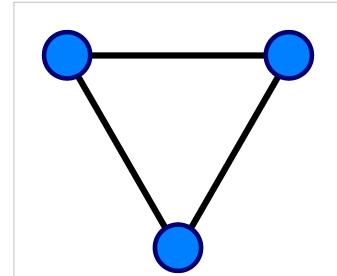
Types of graphs

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).

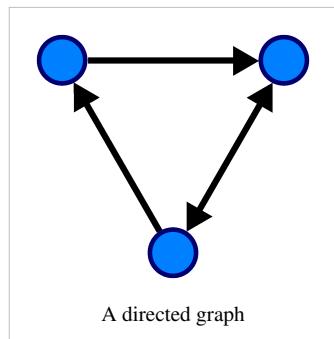
Undirected graph

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.



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

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 .

A variation on this definition is the **oriented graph**, in which not more than one of (x, y) and (y, x) may be arcs.

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,^[5] 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,^[6] although many use the term "pseudograph" for this meaning.^[7]

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.

Simple graph

As opposed to a multigraph, a simple graph is an undirected graph that has no loops 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.^[8] 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.^[9]

Half-edges, loose edges

In exceptional situations it is even necessary to have 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

A regular graph is a graph where each vertex has the same number of neighbors, 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

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.

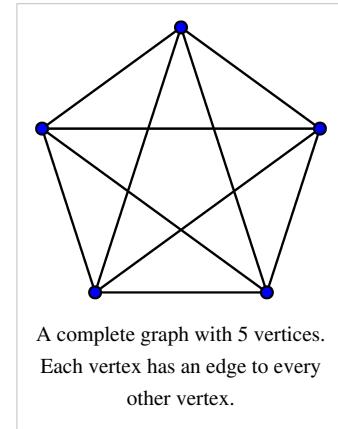
Most commonly in graph theory it is implied that the graphs discussed are finite. If the graphs are infinite, that is usually specifically stated.

Graph classes in terms of connectivity

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 .



Properties of graphs

Two edges of a graph are called **adjacent** (sometimes **coincident**) 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 nock (notch end) of the second one. Similarly, two vertices are called **adjacent** if they share a common edge (**consecutive** if they are at the notch 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.)

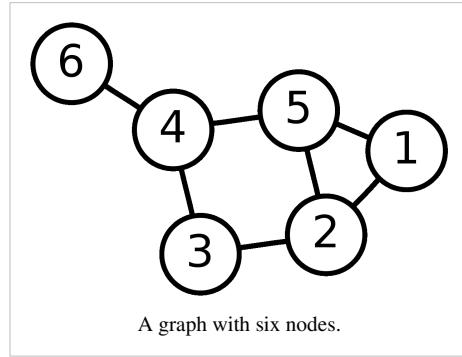
Examples

- The diagram at right is a graphic representation of the following graph:

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = .$$

- 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 .



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

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] Trudeau, Richard J. (1993). *Introduction to Graph Theory* (<http://store.doverpublications.com/0486678709.html>) (Corrected, enlarged republication. ed.). New York: Dover Pub.. pp. 19. ISBN 978-0-486-67870-2. . Retrieved 8 August 2012. "A graph is an object consisting of two sets called its **vertex set** and its **edge set**."
- [2] Gross, Jonathan L.; Yellen, Jay (2004). *Handbook of graph theory* (http://books.google.com/?id=mKkIGIea_BkC). CRC Press. p. 35 (http://books.google.com/books?id=mKkIGIea_BkC&pg=PA35&lpg=PA35). ISBN 978-1-58488-090-5.
- [3] See, for instance, Iyanaga and Kawada, **69 J**, p. 234 or Biggs, p. 4.
- [4] See, for instance, Graham et al., p. 5.
- [5] For example, see Balakrishnan, p. 1, Gross (2003), p. 4, and Zwillinger, p. 220.
- [6] For example, see Bollobás, p. 7 and Diestel, p. 25.
- [7] Gross (1998), p. 3, Gross (2003), p. 205, Harary, p.10, and Zwillinger, p. 220.
- [8] Fletcher, Peter; Hoyle, Hughes; Patty, C. Wayne (1991). *Foundations of Discrete Mathematics* (International student ed. ed.). Boston: PWS-KENT Pub. Co.. pp. 463. ISBN 0-53492-373-9. "A **weighted graph** is a graph in which a number $w(e)$, called its **weight**, is assigned to each edge e ."
- [9] Strang, Gilbert (2005), *Linear Algebra and Its Applications* (<http://books.google.com/books?vid=ISBN0030105676>) (4th ed.), Brooks Cole, ISBN 0-03-010567-6,

References

- Balakrishnan, V. K. (1997-02-01). *Graph Theory* (1st ed.). McGraw-Hill. ISBN 0-07-005489-4.
- Berge, Claude (1958) (in French). *Théorie des graphes et ses applications*. Dunod, Paris: Collection Universitaire de Mathématiques, 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.
- Bollobás, Béla (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., Grötschel, M., and Lovász, 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, Shôkichi; 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

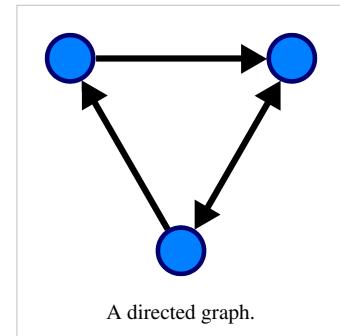
- A searchable database of small connected graphs (<http://www.gfredericks.com/main/sandbox/graphs>)
- VisualComplexity.com (<http://www.visualcomplexity.com>) — A visual exploration on mapping complex networks
- PulseView Data-To-Graph Program (<http://www.quarktet.com/PulseView.html>) — An efficient data plotting and analysis program.
- Weisstein, Eric W., " Graph (<http://mathworld.wolfram.com/Graph.html>)" from MathWorld.
- Intelligent Graph Visualizer (<https://sourceforge.net/projects/igv-intelligent/>) — IGV create and edit graph, automatically places graph, search shortest path (+coloring vertices), center, degree, eccentricity, etc.
- Visual Graph Editor 2 (<http://code.google.com/p/vge2/>) — VGE2 designed for quick and easy creation, editing and saving of graphs and analysis of problems connected with graphs.
- GraphsJ (<http://gianlucacosta.info/software/graphsj/>) — GraphsJ is an open source didactic Java software which features an easy-to-use GUI and interactively solves step-by-step many graph problems. Extensible via its Java SDK.
- GraphClasses (<http://graphclasses.org>) — Information System on Graph Classes and their Inclusions.

Directed graphs

In mathematics, 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.



Sometimes a digraph is called a *simple digraph* to distinguish it from a *directed multigraph*, in which the arcs constitute a multiset, rather than a set, of ordered pairs of vertices. Also, in a simple digraph loops are disallowed. (A loop is an arc that pairs a vertex to itself.) On the other hand, some texts allow loops, multiple arcs, or both in a digraph.

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*.

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

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 graph if and only if it has neither self-loops nor 2-cycles.^[2]

A *weighted digraph* is a digraph with weights assigned for its arcs, similarly to the weighted graph. A digraph with weighted edges in the context of graph theory 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 digraph 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 for a digraph is unique up to the permutations of rows and columns.

Another matrix representation for a digraph is its incidence matrix.

See Glossary of graph theory#Direction for more definitions.

Indegree and outdegree

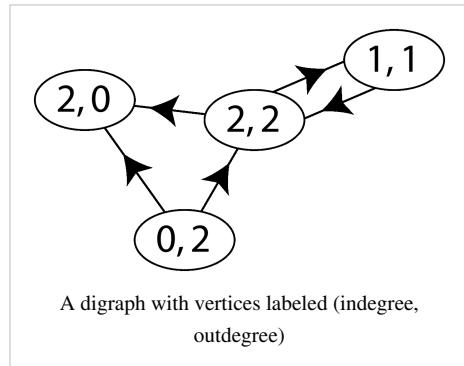
For a node, the number of head endpoints adjacent to a node is called the *indegree* of the node and the number of tail endpoints is its *outdegree* (called "branching factor" in trees).

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]

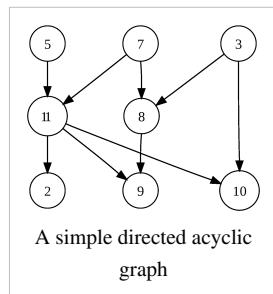


Digraph connectivity

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.

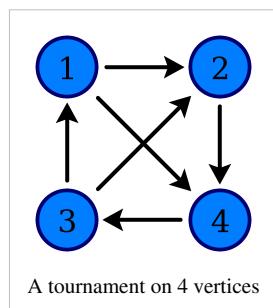
Classes of digraphs

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



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 $\mathbf{FinVet}_K^{F(Q)}$ where $F(Q)$ is the free category on Q consisting of paths in Q and \mathbf{FinVet}_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.



Notes

- [1] Bang-Jensen & Gutin (2000). Diestel (2005), Section 1.10. Bondy & Murty (1976), Section 10.
- [2] Diestel (2005), Section 1.10.
- [3] Satyanarayana, Bhavanari; Prasad, Kuncham Syam, *Discrete Mathematics and Graph Theory*, PHI Learning Pvt. Ltd., p. 460, ISBN 978-81-203-3842-5; Brualdi, Richard A. (2006), *Combinatorial matrix classes*, Encyclopedia of mathematics and its applications, **108**, Cambridge University Press, p. 51, ISBN 978-0-521-86565-4.
- [4] Bang-Jensen & Gutin (2000) p. 19 in the 2007 edition; p. 20 in the 2nd edition (2009).

References

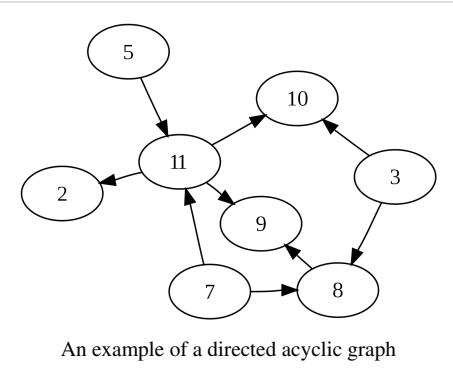
- Bang-Jensen, Jørgen; Gutin, Gregory (2000), *Digraphs: Theory, Algorithms and Applications* (<http://www.cs.rhul.ac.uk/books/dbook/>), Springer, ISBN 1-85233-268-9
(the corrected 1st edition of 2007 is now freely available on the authors' site; the 2nd edition appeared in 2009 ISBN 1-84800-997-6).
- 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.^{[1][2][3]}

DAGs may be used to model several different kinds of structure in mathematics and computer science. 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. DAGs may also be used to model processes in which information flows in a consistent direction through a network of processors. The reachability relation in a DAG forms a partial order, and any finite partial order may be represented by a DAG using reachability. Additionally, DAGs may be used as a space-efficient representation of a collection of sequences with overlapping subsequences.

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

Partial orders and topological ordering

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.

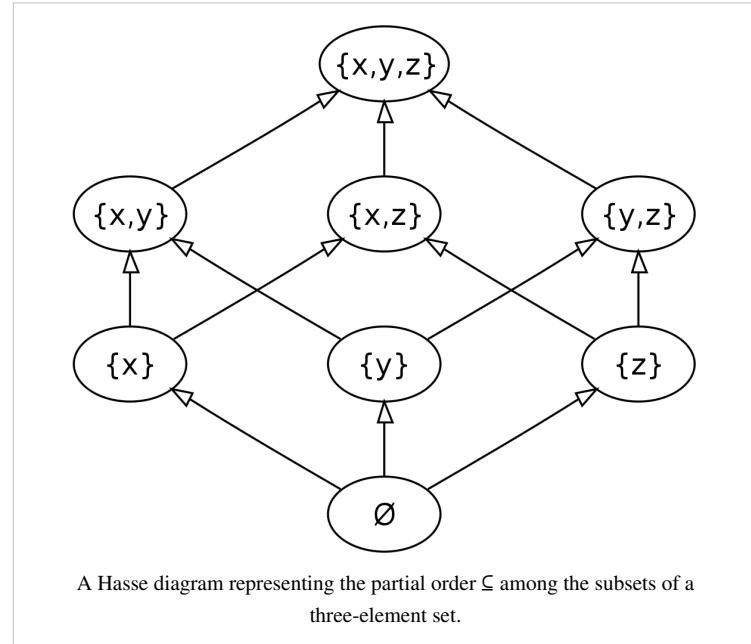
The transitive closure of G has an edge $u \rightarrow v$ for every related pair $u \leq v$ of distinct elements in the reachability relation of G ,

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.

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. Topological sorting is the algorithmic problem of finding topological orderings; it can be solved in linear time. It is also possible to check whether a given directed graph is a DAG in linear time, by attempting to find a topological ordering and then testing whether the resulting ordering is valid.

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. In contrast, for arbitrary graphs the shortest path may require slower algorithms such as Dijkstra's algorithm or the Bellman-Ford algorithm, and longest paths in arbitrary graphs are NP-hard to find.

DAG representations of partial orderings have many applications in scheduling problems for systems of tasks with ordering constraints. 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



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

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 order to be used to schedule the recalculations of cell values when the spreadsheet is changed.

Causal and temporal structures

Graphs that have vertices representing events, and edges representing causal relations between events, are often acyclic. 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.

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, but the time ordering of births (a parent's birthday is always prior to their child's birthday) causes these graphs to be 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.

Paths with shared structure

A third type of application of directed acyclic graphs arises in representing 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,^{[4][5]} 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.

In many randomized algorithms in computational geometry, the algorithm maintains a *history DAG* representing features of some geometric construction that have been replaced by later finer-scale features; point location queries may be answered, as for the above two data structures, by following paths in this DAG.

Relation to other kinds 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 ambiguous 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.

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.^[6]

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. 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.^[7] When the graph is already acyclic, its smallest feedback vertex sets and feedback arc sets are empty, and its condensation is the graph itself.

Enumeration

The graph enumeration problem of counting directed acyclic graphs was studied by Robinson (1973).^[8] The number of DAGs on n labeled nodes, for $n = 1, 2, 3, \dots$, is

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

These numbers may be computed by the recurrence relation

$$a_n = \sum_{k=1}^n (-1)^{k-1} \binom{n}{k} 2^{k(n-k)} a_{n-k}. \quad [8]$$

Eric W. Weisstein conjectured,^[9] and McKay et al. (2004) proved,^[10] 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 the eigenvalues of the (0,1) matrix $A + I$ are positive, where I denotes the identity matrix.

References

- [1] Christofides, Nicos (1975), *Graph theory: an algorithmic approach*, Academic Press, pp. 170–174.
- [2] Thulasiraman, K.; Swamy, M. N. S. (1992), "5.7 Acyclic Directed Graphs", *Graphs: Theory and Algorithms*, John Wiley and Son, p. 118, ISBN 978-0-471-51356-8.
- [3] Bang-Jensen, Jørgen (2008), "2.1 Acyclic Digraphs", *Digraphs: Theory, Algorithms and Applications*, Springer Monographs in Mathematics (2nd ed.), Springer-Verlag, pp. 32–34, ISBN 978-1-84800-997-4.
- [4] Lee, C. Y. (1959), "Representation of switching circuits by binary-decision programs", *Bell Systems Technical Journal* **38**: 985–999.
- [5] Akers, Sheldon B. (1978), "Binary decision diagrams", *IEEE Transactions on Computers* **C-27** (6): 509–516, doi:10.1109/TC.1978.1675141.
- [6] Stanley, Richard P. (1973), "Acyclic orientations of graphs", *Discrete Mathematics* **5** (2): 171–178, doi:10.1016/0012-365X(73)90108-8.
- [7] Harary, Frank; Norman, Robert Z.; Cartwright, Dorwin (1965), *Structural Models: An Introduction to the Theory of Directed Graphs*, John Wiley & Sons, p. 63.
- [8] Robinson, R. W. (1973), "Counting labeled acyclic digraphs", in Harary, F., *New Directions in the Theory of Graphs*, Academic Press, pp. 239–273. See also Harary, Frank; Palmer, Edgar M. (1973), *Graphical Enumeration*, Academic Press, p. 19, ISBN 0-12-324245-2.
- [9] Weisstein, Eric W., "Weisstein's Conjecture" (<http://mathworld.wolfram.com/WeissteinsConjecture.html>) from MathWorld.
- [10] McKay, B. D.; Royle, G. F.; Wanless, I. M.; Oggier, F. E.; Sloane, N. J. A.; Wilf, H. (2004), "Acyclic digraphs and eigenvalues of (0,1)-matrices" (<http://www.cs.uwaterloo.ca/journals/JIS/VOL7/Sloane/sloane15.html>), *Journal of Integer Sequences* **7**, Article 04.3.3.

External links

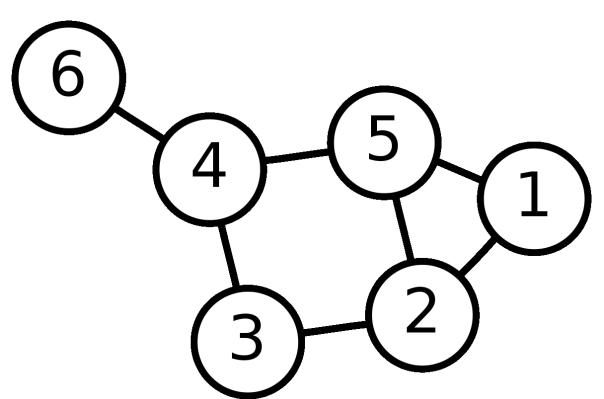
- Weisstein, Eric W., "Acyclic Digraph" (<http://mathworld.wolfram.com/AcyclicDigraph.html>) from MathWorld.

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.

A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford–Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph.

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.

Incidence list

Vertices and edges are stored as records or objects. Each vertex stores its incident edges, and each edge stores its incident vertices. This data structure allows the storage of additional data on vertices and edges.

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. 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	Incidence list	Adjacency matrix	Incidence matrix
Storage	$O(V + E)$	$O(V + E)$	$O(V ^2)$	$O(V \cdot E)$
Add vertex	$O(1)$	$O(1)$	$O(V ^2)$	$O(V \cdot E)$
Add edge	$O(1)$	$O(1)$	$O(1)$	$O(V \cdot E)$
Remove vertex	$O(E)$	$O(E)$	$O(V ^2)$	$O(V \cdot E)$
Remove edge	$O(E)$	$O(E)$	$O(1)$	$O(V \cdot E)$
Query: are vertices u, v adjacent? (Assuming that the storage positions for u, v are known)	$O(V)$	$O(E)$	$O(1)$	$O(E)$
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]

Types

- Skip graphs

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

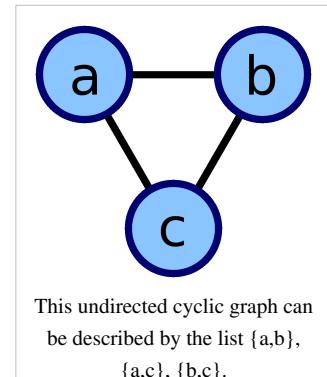
- Boost Graph Library: a powerful C++ graph library (<http://www.boost.org/libs/graph>)
- PulseView Graphing: an efficient Data-To-Graph program (<http://www.quarktet.com/PulseView.html>)

Adjacency list

In graph theory, an **adjacency list** is the representation of all edges or arcs in a graph as a list.

If the graph is undirected, every entry is a set (or multiset) of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc.

Typically, adjacency lists are unordered.



Application in computer science

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

In computer science, an adjacency list is a data structure for representing graphs. In an adjacency list representation, we keep, for each vertex in the graph, a list of all other vertices which it has an edge to (that vertex's "adjacency list"). For instance, the representation suggested by van Rossum, in which a hash table is used to associate each vertex with an array of adjacent vertices, can be seen as an example of this type of representation. Another example is the representation in Cormen et al. in which an array indexed by vertex numbers points to a singly linked list of the neighbors of each vertex.

One difficulty with the adjacency list structure is that it has no obvious place to store data associated with the edges of a graph, such as the lengths or costs of the edges. To remedy this, some texts, such as that of Goodrich and Tamassia, advocate a more object oriented variant of the adjacency list structure, sometimes called an incidence list, which stores for each vertex a list of objects representing the edges incident to that vertex. To complete the structure, each edge must point back to the two vertices forming its endpoints. The extra edge objects in this version of the adjacency list cause it to use more memory than the version in which adjacent vertices are listed directly, but these extra edges are also a convenient location to store additional information about each edge (e.g. their length).

Trade-offs

The main alternative to the adjacency list is the adjacency matrix. For a graph with a sparse adjacency matrix an adjacency list representation of the graph occupies less space, because it does not use any space to represent edges that are *not* present. Using a naive array implementation of adjacency lists 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: each edge gives rise to entries in the two adjacency lists and uses four bytes in each.

On the other hand, because each entry in an adjacency matrix requires only one bit, they 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 just avoiding wasted space, this compactness encourages locality of reference.

Noting that a graph can have at most n^2 edges (allowing loops) we can let $d = e/n^2$ denote the *density* of the graph. Then, if $8e > n^2/8$, the adjacency list representation occupies more space, which is true when $d > 1/64$. Thus a graph must be sparse for an adjacency list representation to be more memory efficient than an adjacency matrix. However, this analysis is valid only when the representation is intended to store the connectivity structure of the graph without any numerical information about its edges.

Besides the space trade-off, the different data structures also facilitate different operations. It is easy to find all vertices adjacent to a given vertex in an adjacency list representation; you simply read its adjacency list. With an adjacency matrix you must instead scan over an entire row, taking $O(n)$ time. If you, instead, want to perform a neighbor test on two vertices (i.e., determine if they have an edge between them), an adjacency matrix provides this at once. However, this neighbor test in an adjacency list requires time proportional to the number of edges associated with the two vertices.

References

- Joe Celko (2004). *Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann. excerpt from Chapter 2: "Adjacency List Model" [1]. ISBN 1-55860-920-2.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2001). *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill. pp. 527–529 of section 22.1: Representations of graphs. ISBN 0-262-03293-7.
- David Eppstein (1996). "ICS 161 Lecture Notes: Graph Algorithms" [2].
- Michael T. Goodrich and Roberto Tamassia (2002). *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons. ISBN 0-471-38365-1.
- Guido van Rossum (1998). "Python Patterns — Implementing Graphs" [3].

External links

- The Boost Graph Library implements an efficient adjacency list [4]
- Open Data Structures - Section 12.2 - AdjacencyList: A Graph as a Collection of Lists [5]

References

- [1] <http://www.SQLSummit.com/AdjacencyList.htm>
- [2] <http://www.ics.uci.edu/~eppstein/161/960201.html>
- [3] <http://www.python.org/doc/essays/graphs/>
- [4] http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/index.html
- [5] 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
	$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ <p>Coordinates are 1-6.</p>
<p>The Nauru graph</p>	<p>Coordinates are 0-23. White fields are zeros, colored fields are ones.</p>
<p>Directed Cayley graph of S_4</p>	<p>As the graph is directed, the matrix is not symmetric.</p>

- The adjacency matrix of a complete graph is all 1's except for 0's on the diagonal.
- 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

$$A = \begin{pmatrix} O & B \\ B^T & O \end{pmatrix},$$

where B is an $r \times s$ matrix and O is an all-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.^[2]

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, if the graph is sparse, 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
- [2] Seidel, J. J. (1968). "Strongly Regular Graphs with $(-1,1,0)$ Adjacency Matrix Having Eigenvalue 3". *Lin. Alg. Appl.* **1** (2): 281–298.
doi:10.1016/0024-3795(68)90008-6.

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

- 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://cafemath.kegtux.org/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.^[1] 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.^[2] 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.^[3] PLS, another complexity class, captures the complexity of finding local optima in an implicit graph.^[4]

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.^[5]

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.^[6]

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.^{[7][8]}
- **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.^{[7][9]} 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.^[10]
- **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.^{[11][12]}

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.^{[7][9]} 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.^{[6][9]} 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.^{[7][9]}

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.^[7] 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.^[13]

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;^[14] 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^[15]—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,^[16]

References

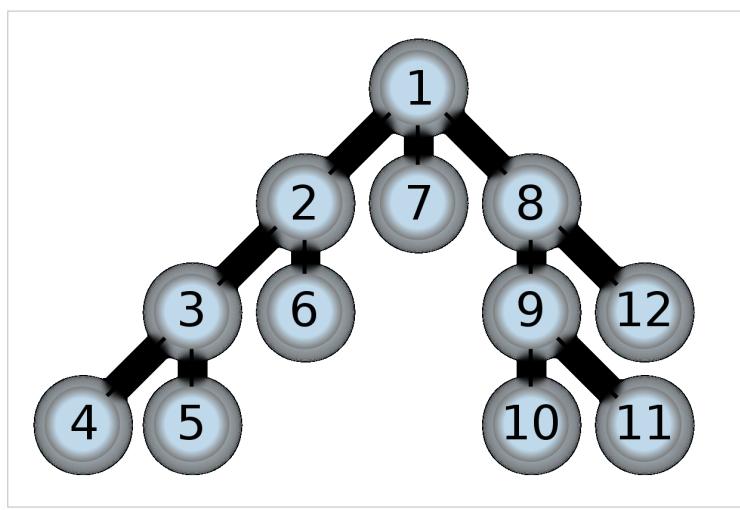
- [1] Korf, Richard E. (2008), "Linear-time disk-based implicit graph search", *Journal of the ACM* **55** (6): Article 26, 40pp, doi:10.1145/1455248.1455250, MR2477486.
- [2] Papadimitriou, Christos (1994), "On the complexity of the parity argument and other inefficient proofs of existence" (<http://www.cs.berkeley.edu/~christos/papers/On the Complexity.pdf>), *Journal of Computer and System Sciences* **48** (3): 498–532, doi:10.1016/S0022-0000(05)80063-7,
- [3] Immerman, Neil (1999), "Exercise 3.7 (Everything is a Graph)" (<http://books.google.com/books?id=kWSZ0OWnupkC&pg=PA48>), *Descriptive Complexity*, Graduate Texts in Computer Science, Springer-Verlag, p. 48, ISBN 978-0-387-98600-5, .
- [4] Yannakakis, Mihalis (2009), "Equilibria, fixed points, and complexity classes", *Computer Science Review* **3** (2): 71–85.
- [5] Childs, Andrew M.; Cleve, Richard; Deotto, Enrico; Farhi, Edward; Gutmann, Sam; Spielman, Daniel A. (2003), "Exponential algorithmic speedup by a quantum walk", *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, New York: ACM, pp. 59–68, doi:10.1145/780542.780552, MR2121062.
- [6] Muller, John Harold (1988), *Local structure in graph classes*, Ph.D. thesis, Georgia Institute of Technology.
- [7] Kannan, Sampath; Naor, Moni; Rudich, Steven (1992), "Implicit representation of graphs", *SIAM Journal on Discrete Mathematics* **5** (4): 596–603, doi:10.1137/0405049, MR1186827.
- [8] Chrobak, Marek; Eppstein, David (1991), "Planar orientations with low out-degree and compaction of adjacency matrices" (<http://www.ics.uci.edu/~eppstein/pubs/ChrEpp-TCS-91.pdf>), *Theoretical Computer Science* **86** (2): 243–266, doi:10.1016/0304-3975(91)90020-3, .
- [9] Spinrad, Jeremy P. (2003), "2. Implicit graph representation" (<http://books.google.com/books?id=RrtXSKMAmWgC&pg=PA17>), *Efficient Graph Representations*, pp. 17–30, ISBN 0-8218-2815-0, .
- [10] Kang, Ross J.; Müller, Tobias (2011), *Sphere and dot product representations of graphs* (<http://homepages.cwi.nl/~mueller/Papers/SphericityDotproduct.pdf>), .
- [11] Ma, Tze Heng; Spinrad, Jeremy P. (1991), "Cycle-free partial orders and chordal comparability graphs", *Order* **8** (1): 49–61, doi:10.1007/BF00385814, MR1129614.
- [12] Curtis, Andrew R.; Izurieta, Clemente; Joeris, Benson; Lundberg, Scott; McConnell, Ross M. (2010), "An implicit representation of chordal comparability graphs in linear time", *Discrete Applied Mathematics* **158** (8): 869–875, doi:10.1016/j.dam.2010.01.005, MR2602811.
- [13] Alstrup, Stephen; Rauhe, Theis (2002), "Small induced-universal graphs and compact implicit graph representations" (<http://www.it-c.dk/research/algorithms/Kurser/AD/2002E/Uge7/parent.pdf>), *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*: 53–62, doi:10.1109/SFCS.2002.1181882, .
- [14] Rivest, Ronald L.; Vuillemin, Jean (1975), "A generalization and proof of the Aanderaa–Rosenberg conjecture", *Proc. 7th ACM Symposium on Theory of Computing*, Albuquerque, New Mexico, United States, pp. 6–11, doi:10.1145/800116.803747.

- [15] Kahn, Jeff; Saks, Michael; Sturtevant, Dean (1983), "A topological approach to evasiveness", *Symposium on Foundations of Computer Science*, Los Alamitos, CA, USA: IEEE Computer Society, pp. 31–33, doi:10.1109/SFCS.1983.4.
- [16] Bender, Michael A.; Ron, Dana (2000), "Testing acyclicity of directed graphs in sublinear time", *Automata, languages and programming (Geneva, 2000)*, Lecture Notes in Comput. Sci., **1853**, Berlin: Springer, pp. 809–820, doi:10.1007/3-540-45022-X_68, MR1795937.

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	$O(V + E)$ 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, $O(\text{longest path length searched})$ for implicit graphs without elimination of duplicate nodes

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) 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^[1] as a strategy for solving mazes.^{[2][3]}

Formal definition

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

Properties

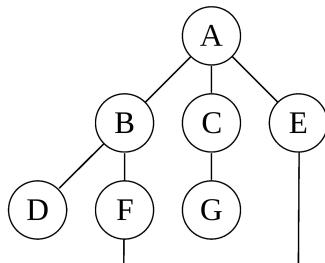
The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time $O(|V| + |E|)$, linear in the size of the graph. In these applications it also uses space $O(|V|)$ in the worst case to store 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 to search problems in artificial intelligence, however, the graph to be searched is often either too large to visit in its entirety or even infinite, and DFS may suffer from non-termination when the length of a path in the search tree is infinite. Therefore, the search is only performed to a limited depth, and due to limited memory availability one typically does not use data structures that keep track of the set of all previously visited vertices. In this case, the time is still linear in 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, 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 of 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 be also 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:



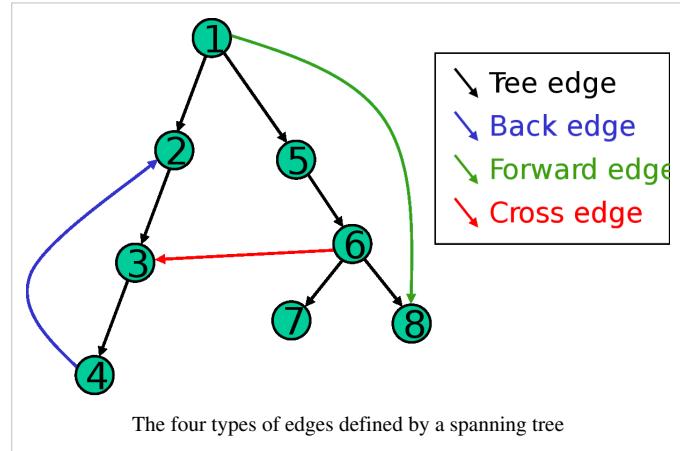
a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges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

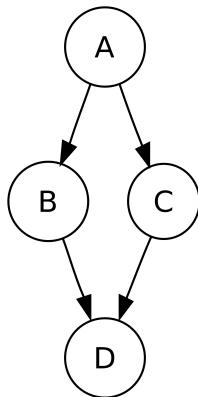
A convenient description of a depth first search of a graph is in terms of a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: **forward edges**, which point from a node of the tree to one of its descendants, **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



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 (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 flow. 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: A labeling of the edges in the connected component of v as discovery edges and back edges

```

1  procedure DFS( $G, v$ ) :
2      label  $v$  as explored
3      for all edges  $e$  in  $G.\text{incidentEdges}(v)$  do
4          if edge  $e$  is unexplored then
5               $w \leftarrow G.\text{opposite}(v, e)$ 
6              if vertex  $w$  is unexplored then
7                  label  $e$  as a discovery edge
8                  recursively call DFS( $G, w$ )
9              else
10                 label  $e$  as a back edge

```

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.
- Finding strongly connected components.
- Planarity testing^{[4][5]}
- 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.)
- Maze generation may use a randomized depth-first search.
- Finding biconnectivity in graphs.



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

Notes

- [1] 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)
- [2] Even, Shimon (2011), *Graph Algorithms* (<http://books.google.com/books?id=m3QTSMYm5rkC&pg=PA46>) (2nd ed.), Cambridge University Press, pp. 46–48, ISBN 978-0-521-73653-4, .
- [3] Sedgewick, Robert (2002), *Algorithms in C++: Graph Algorithms* (3rd ed.), Pearson Education, ISBN 978-0-201-36118-6.
- [4] Hopcroft, John; Tarjan, Robert E. (1974), "Efficient planarity testing", *Journal of the Association for Computing Machinery* **21** (4): 549–568, doi:10.1145/321850.321852.
- [5] de Fraysseix, H.; Ossona de Mendez, P.; Rosenstiehl, P. (2006), "Trémaux Trees and Planarity", *International Journal of Foundations of Computer Science* **17** (5): 1017–1030, doi:10.1142/S0129054106004248.

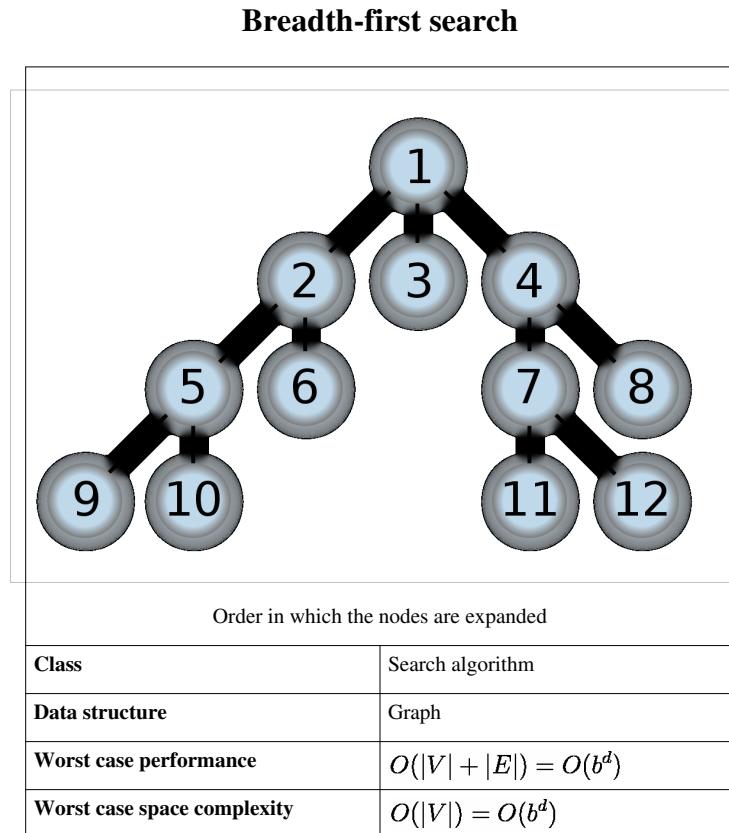
References

- 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.
- 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
- Goodrich, Michael T. (2001), *Algorithm Design: Foundations, Analysis, and Internet Examples*, Wiley, ISBN 0-471-38365-1

External links

- 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.algolist.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

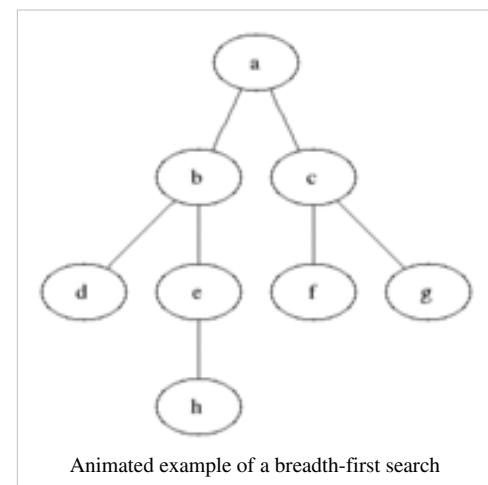


In graph theory, **breadth-first search (BFS)** is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspect 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 it with the depth-first search.

How it works

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

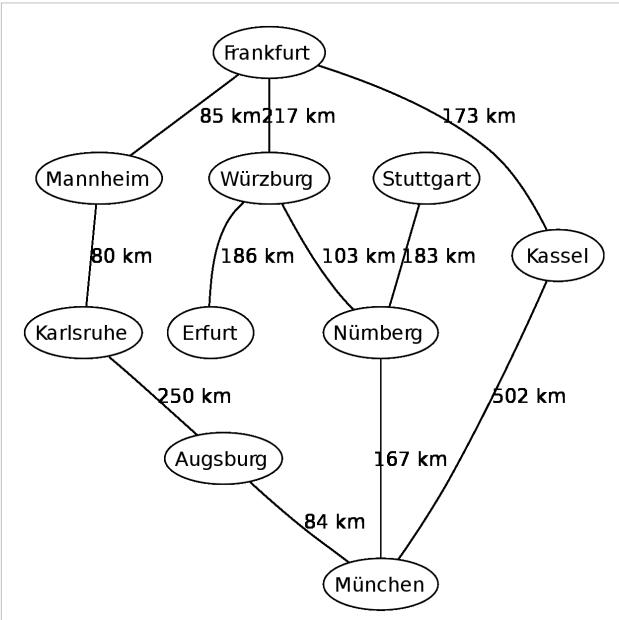


Algorithm

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

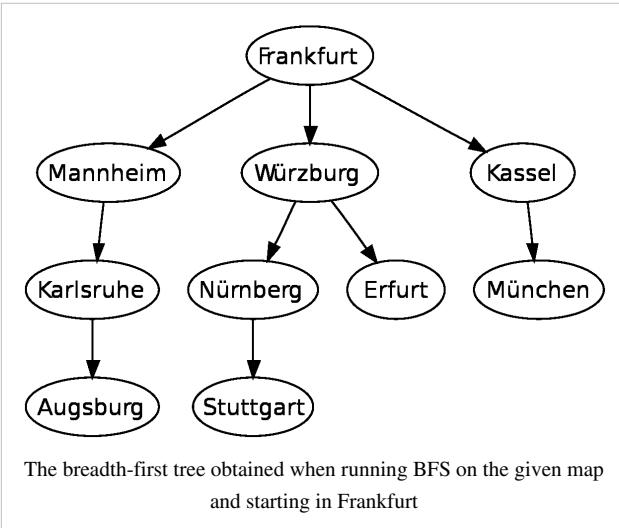
1. Enqueue the root node
2. Dequeue a node and examine it
 - If the element sought is found in this node, quit the search and return a result.
 - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined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

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.



Pseudocode

Input: A graph G and a root v of G



```

1  procedure BFS( $G, v$ ) :
2      create a queue  $Q$ 
3      enqueue  $v$  onto  $Q$ 
4      mark  $v$ 
5      while  $Q$  is not empty:
6           $t \leftarrow Q.\text{dequeue}()$ 
7          if  $t$  is what we are looking for:
8              return  $t$ 
9          for all edges  $e$  in  $G.\text{incidentEdges}(t)$  do
10              $o \leftarrow G.\text{opposite}(t, e)$ 
11             if  $o$  is not marked:
12                 mark  $o$ 
13                 enqueue  $o$  onto  $Q$ 

```

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.

Time complexity

The time complexity can be expressed as $O(|E| + |V|)$ since every vertex and every edge will be explored in the worst case. Note: $O(|E| + |V|)$ may vary between $O(|V|)$ and $O(|V|^2)$, depending on known estimates of the number of graph edges.

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.

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

- 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>)

Lexicographic breadth-first search

In computer science, **lexicographic breadth-first search** or Lex-BFS is a linear time algorithm for ordering the vertices of a graph, that is used as part of other graph algorithms such as the recognition of chordal graphs and optimal coloring of distance-hereditary graphs. The algorithm for constructing Lex-BFS orderings is different from a standard breadth-first search; every Lex-BFS ordering is a possible BFS ordering but not vice versa. 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).

The 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 the sequence.

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 lexicographic 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.

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, as Rose, Tarjan, and Lueker show, 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.

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.^[1]

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] Brandstädt, Le & Spinrad (1999), 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.

- Corneil, Derek G. (2004), "Lexicographic breadth first search – a survey" (<http://www.springerlink.com/content/nwddhwbtbgafm6jaal/>), *Graph-Theoretic Methods in Computer Science*, Lecture Notes in Computer Science, **3353**, Springer-Verlag, pp. 1–19.
- 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.
- 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.

Iterative deepening depth-first search

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.^[1]

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.^[1]

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 is not possible with a traditional depth-first search.

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.^[1] So the total number of expansions in an iterative deepening search is

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

$$\sum_{i=0}^d (d+1-i)b^i$$

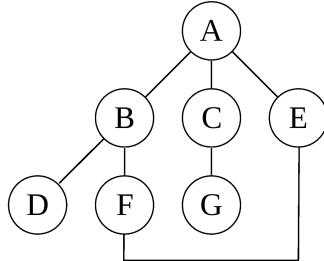
For $b = 10$ and $d = 5$ the number is

$$6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$$

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.^[1]

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)
{
    depth = 0
    repeat
    {
        result = DLS(root, goal, depth)
        if result == goal
            return result
        if result == failure
            return failure
        depth = depth + 1
    }
}
  
```

```

    if (result is a solution)
        return result
    depth = depth + 1
}
}

```

Depth-limited search can be implemented recursively as follows. Note that it need only check for goal nodes when `depth == 0`, because when `depth > 0`, DLS is expanding nodes that it has seen in a previous iteration of IDDFS.

```

DLS(node, goal, depth)
{
    if (depth >= 0 and node == goal)
        return node
    else if (depth > 0)
        for each child in expand(node)
            DLS(child, goal, depth-1)
    else
        return no-solution
}

```

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 make it less useful than iterative deepening.

Notes

- [1] Russell, Stuart J.; Norvig, Peter (2003), *Artificial Intelligence: A Modern Approach* (<http://aima.cs.berkeley.edu/>) (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2,

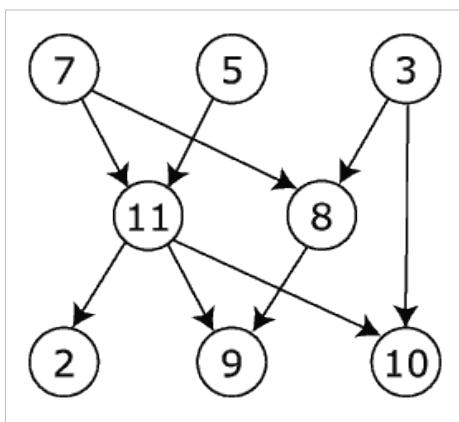
Topological sorting

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 edge uv , 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, serialization, and resolving symbol dependencies in linkers.



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)
- 3, 7, 8, 5, 11, 10, 2, 9
- 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

Algorithms

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

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
  insert n into 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.

Note that, 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. For this algorithm, the edges are examined in the opposite direction as the previous one (it looks for nodes with edges pointing *to* a given node instead of *from* it, which might set different requirements for the data structure used to represent the graph; and it starts with the set of nodes with no *outgoing* edges). 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
S ← Set of all nodes with no outgoing edges
for each node n in S do
    visit(n)
function visit(node n)
    if n has not been visited yet then
        mark n as visited
        for each node m with an edge from m to n do
            visit(m)
        add n to L

```

Note that each node *n* gets added to the output list L only after considering all other nodes on which *n* depends (all ancestor 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. Note that the simple pseudocode above cannot detect the error case where the input graph contains cycles. The algorithm can be refined to detect cycles by watching for nodes which are visited more than once during any nested sequence of recursive calls to visit() (e.g., by passing a list down as an extra argument to visit(), indicating which nodes have already been visited in the current call stack). 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).

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 polynomial 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 ($x = x$), 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

- 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.
- 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.
- Tarjan, Robert E. (1976), "Edge-disjoint spanning trees and depth-first search", *Acta Informatica* **6** (2): 171–185, doi:10.1007/BF00268499.
- 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.

External links

- NIST Dictionary of Algorithms and Data Structures: topological sort ^[1]
- Weisstein, Eric W., "TopologicalSort" ^[2]" from MathWorld.

References

[1] <http://www.nist.gov/dads/HTML/topologicalSort.html>

[2] <http://mathworld.wolfram.com/TopologicalSort.html>

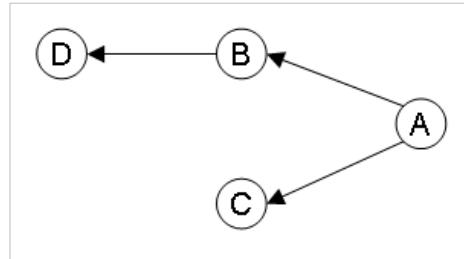
Application: Dependency graphs

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 $R = S \times S$ 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, 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 $n : S \rightarrow N$ 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 the Unix Make system 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] wcre, 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

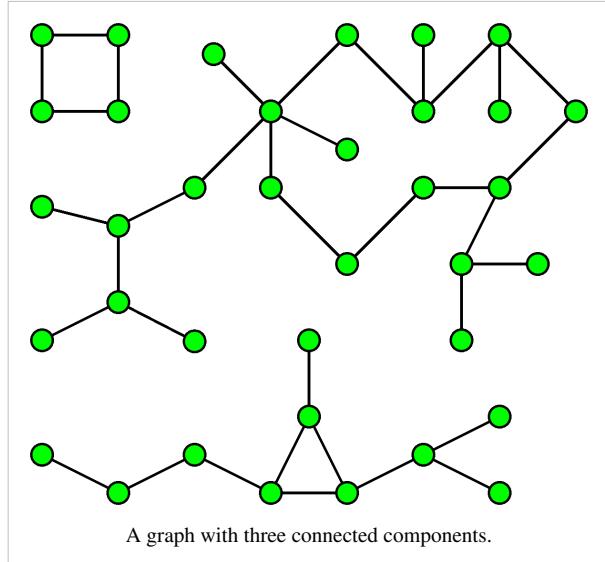
In graph theory, a **connected component** of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. For example, the graph shown in the illustration on the right has three connected components. 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)^[1] 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.^[2]

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] Hopcroft, J.; Tarjan, R. (1973). "Efficient algorithms for graph manipulation". *Communications of the ACM* **16** (6): 372–378. doi:10.1145/362248.362272.
- [2] 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.
- Reingold, Omer (2008), "Undirected connectivity in log-space", *Journal of the ACM* **55** (4): Article 17, 24 pages, doi:10.1145/1391289.1391291.

External links

- 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.

Formal definition

Let $G = (E, V)$ be an arbitrary graph. If $G' = (E \setminus X, V)$ is connected for all $X \subseteq E$ where $|X| < k$, then G is k -edge-connected. Trivially, a graph that is k -edge-connected is also $(k-1)$ -edge-connected.

Relation to minimum vertex degree

Minimum vertex degree gives a trivial upper bound on edge-connectivity. That is, if a graph $G = (E, V)$ 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 V is the number of vertices in the graph, this simple algorithm would perform $O(V^2)$ iterations of the Maximum flow problem, which can be solved in $O(V^3)$ time. Hence the complexity of the simple algorithm described above is $O(V^5)$ 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 $O(V^4)$ and is sound since, if a cut of capacity less than k exists, it is bound to separate u from some other vertex.

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 $k \geq 2$.^[1]

References

[1] 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 with vertex set $V(G)$ is said to be **k -vertex-connected** (or **k -connected**) if the graph remains connected when you delete fewer than k vertices from the graph. Alternatively, a graph is **k -connected** if k is the size of the smallest subset of vertices such that the graph becomes disconnected if you delete them.^[1]

An equivalent definition for graphs that are not complete is that a graph is k -connected if any two of its vertices can be joined by k independent paths; see Menger's theorem (Diestel 2005, p. 55). However, for complete graphs the two definitions differ: the n -vertex complete graph has unbounded connectivity according to the definition based on deleting vertices, but connectivity $n - 1$ according to the definition based on independent paths, and some authors use alternative definitions according to which its connectivity is n .^[1].

A 1-vertex-connected graph is called connected, while a 2-vertex-connected graph is said to be biconnected.

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

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.

Notes

[1] Schrijver, *Combinatorial Optimization*, Springer

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.
- 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 basic result about connectivity in finite undirected graphs. 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.

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.

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 it. This is equivalent to Menger's theorem for finite graphs and is a deep recent result of Ron Aharoni and Eli Berger for infinite graphs (originally a conjecture proposed by Paul Erdős):

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.
- 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, MR0335355.

External links

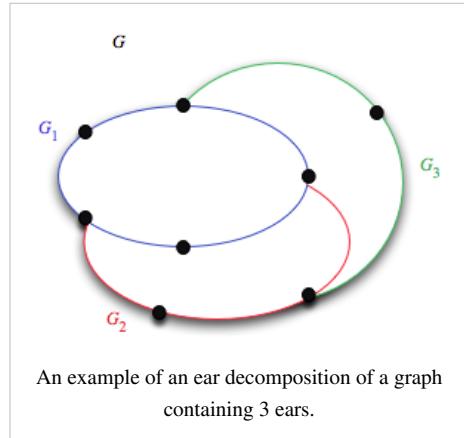
- A Proof of Menger's Theorem ^[2]
- Menger's Theorems and Max-Flow-Min-Cut ^[3]
- Network flow ^[4]
- Max-Flow-Min-Cut ^[5]

References

- [1] <http://www.springerlink.com/content/267k231365284lr6/?p=ddccdd0319b24e53958e286488757ca7&pi=0>
- [2] http://www.math.unm.edu/~loring/links/graph_s05/Menger.pdf
- [3] <http://www.math.fau.edu/locke/Menger.htm>
- [4] <http://gepard.bioinformatik.uni-saarland.de/teaching/ws-2008-09/bioinformatik-3/lectures/V12-NetworkFlow.pdf>
- [5] <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.

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 & Yellin 2006).

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-edge-connected graphs may be found by a greedy algorithm that finds each ear one at a 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).

References

- Eppstein, D. (1992), "Parallel recognition of series-parallel graphs", *Information & Computation* **98** (1): 41–55, doi:10.1016/0890-5401(92)90041-D, MR1161075.
- Frank, András (1993), "Conservative weightings and ear-decompositions of graphs", *Combinatorica* **13** (1): 65–81, doi:10.1007/BF01202790, MR1221177.
- Gross, Jonathan L.; Yellen, Jay (2006), "Characterization of strongly orientable graphs" ^[1], *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, MR2181153.
- Khuller, Samir (1989), "Ear decompositions" ^[2], *SIGACT News* **20** (1): 128.
- Lovász, László (1972), "A note on factor-critical graphs", *Studia Sci. Math. Hung.* **7**: 279–280, MR0335371.
- 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.
- 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, MR0882357.
- 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", *The American Mathematical Monthly* **46**: 281–283.
- 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, MR2246153.
- Whitney, H. (1932), "Non-separable and planar graphs", *Transactions of the American Mathematical Society* **34**: 339–362.

References

- [1] http://books.google.com/books?id=unEloQ_sYmkC&pg=PA498
[2] <http://portalparts.acm.org/70000/65780/bm/backmatter.pdf>

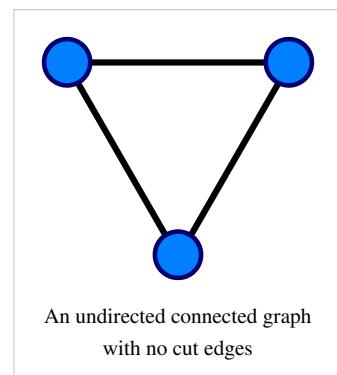
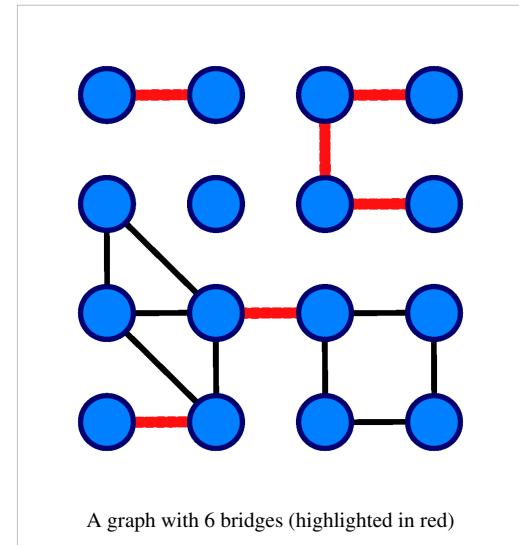
Algorithms for 2-edge-connected components

In graph theory, a **bridge** (also known as a **cut-edge** or **cut arc** or an **isthmus**) is an edge whose deletion increases the number of connected components.^[1] Equivalently, an edge is a bridge if and only if it is not contained in any cycle. A graph is said to be **bridgeless** if it contains no bridges.

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.^[2]



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 necessarily articulation vertices, 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,^[3] that each connected component is 2-edge-connected, or (by Robbins' theorem) that every connected component has a strong orientation.^[3]

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.^[4]

Bridge-finding algorithm

A linear time algorithm for finding the bridges in a graph was described by Robert Tarjan in 1974.^[5] 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.

Notes

- [1] Bollobás, Béla (1998), *Modern Graph Theory* (<http://books.google.com/books?id=SbZKSZ-1qrwC&pg=PA6>), Graduate Texts in Mathematics, **184**, New York: Springer-Verlag, p. 6, doi:10.1007/978-1-4612-0619-4, ISBN 0-387-98488-7, MR1633290, .
- [2] Westbrook, Jeffery; Tarjan, Robert E. (1992), "Maintaining bridge-connected and biconnected components on-line", *Algorithmica* **7** (5-6): 433–464, doi:10.1007/BF01758773, MR1154584.
- [3] Robbins, H. E. (1939), "A theorem on graphs, with an application to a problem of traffic control", *The American Mathematical Monthly* **46**: 281–283.
- [4] Jaeger, F. (1985), "A survey of the cycle double cover conjecture", *Annals of Discrete Mathematics 27 – Cycles in Graphs*, North-Holland Mathematics Studies, **27**, pp. 1–12, doi:10.1016/S0304-0208(08)72993-1.
- [5] Tarjan, R. Endre, "A note on finding the bridges of a graph", *Information Processing Letters* **2** (6): 160–161, doi:10.1016/0020-0190(74)90003-9, MR0349483.

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 that when removed increases the number of connected components.

Algorithm

The classic sequential algorithm for computing biconnected components in a connected undirected graph due to John Hopcroft and Robert Tarjan (1973)^[1] 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.

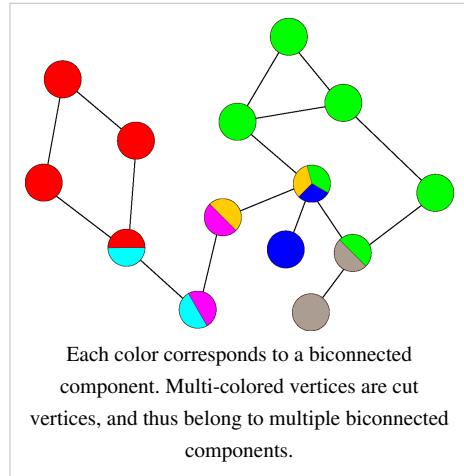
The key fact is that a nonroot vertex v is a cut vertex (or articulation point) separating two biconnected components if and only if there is a child y of v such that $\text{lowpoint}(y) \geq \text{depth}(v)$. This property can be tested once the depth-first search returned from every child of v (i.e., 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).

Other algorithms

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)^[2] 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)^[3] 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)^[4] 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)^[5]



Notes

- [1] Hopcroft, J.; Tarjan, R. (1973). "Efficient algorithms for graph manipulation". *Communications of the ACM* **16** (6): 372–378. doi:10.1145/362248.362272.
- [2] Westbrook, J.; Tarjan, R. E. (1992). "Maintaining bridge-connected and biconnected components on-line". *Algorithmica* **7**: 433–464. doi:10.1007/BF01758773.
- [3] Tarjan, R.; Vishkin, U. (1985). "An Efficient Parallel Biconnectivity Algorithm". *SIAM Journal on Computing* **14** (4): 862–000. doi:10.1137/0214061.
- [4] Guojing Cong and David A. Bader, (2005). "An Experimental Study of Parallel Biconnected Components Algorithms on Symmetric Multiprocessors (SMPs)". *Proceedings of the 19th IEEE International Conference on Parallel and Distributed Processing Symposium*. pp. 45b. doi:10.1109/IPDPS.2005.100.
- [5] Edwards, J. A.; Vishkin, U. (2012). "Better speedups using simpler parallel programming for graph connectivity and biconnectivity". *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '12*. pp. 103. doi:10.1145/2141702.2141714. ISBN 9781450312110.

References

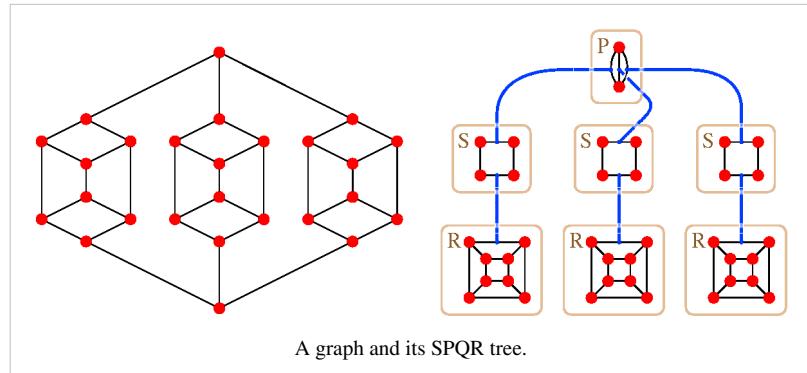
- Eugene C. Freuder (1985). "A Sufficient Condition for Backtrack-Bounded Search". *Journal of the Association for Computing Machinery* **32** (4): 755–761. doi:10.1145/4221.4225.

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".^[3]
- 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".^[3]
- In a Q node, the associated graph has a single edge. This trivial case is necessary to handle the graph that has only one edge, but does not appear in the SPQR trees of more complex graphs.
- 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.^[3]

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 .

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.

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.

Notes

- [1] Hopcroft & Tarjan (1973); Gutwenger & Mutzel (2001).
- [2] E.g., Hopcroft & Tarjan (1973) and Bienstock & Monma (1988), both of which are cited as precedents by Di Battista and Tamassia.
- [3] Di Battista & Tamassia (1989).

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.
- 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.
- 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.
- 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.
- 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.
- Hopcroft, John; Tarjan, Robert (1973), "Dividing a graph into triconnected components", *SIAM Journal on Computing* **2** (3): 135–158, doi: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.

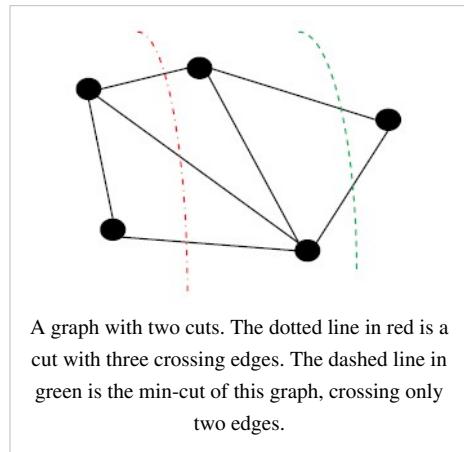
External links

- SQPR 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 Monte Carlo method to compute the minimum cut of a connected graph. It was invented by David Karger and first published in 1993.^[1]

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 the min-cut of this graph, crossing only two edges.

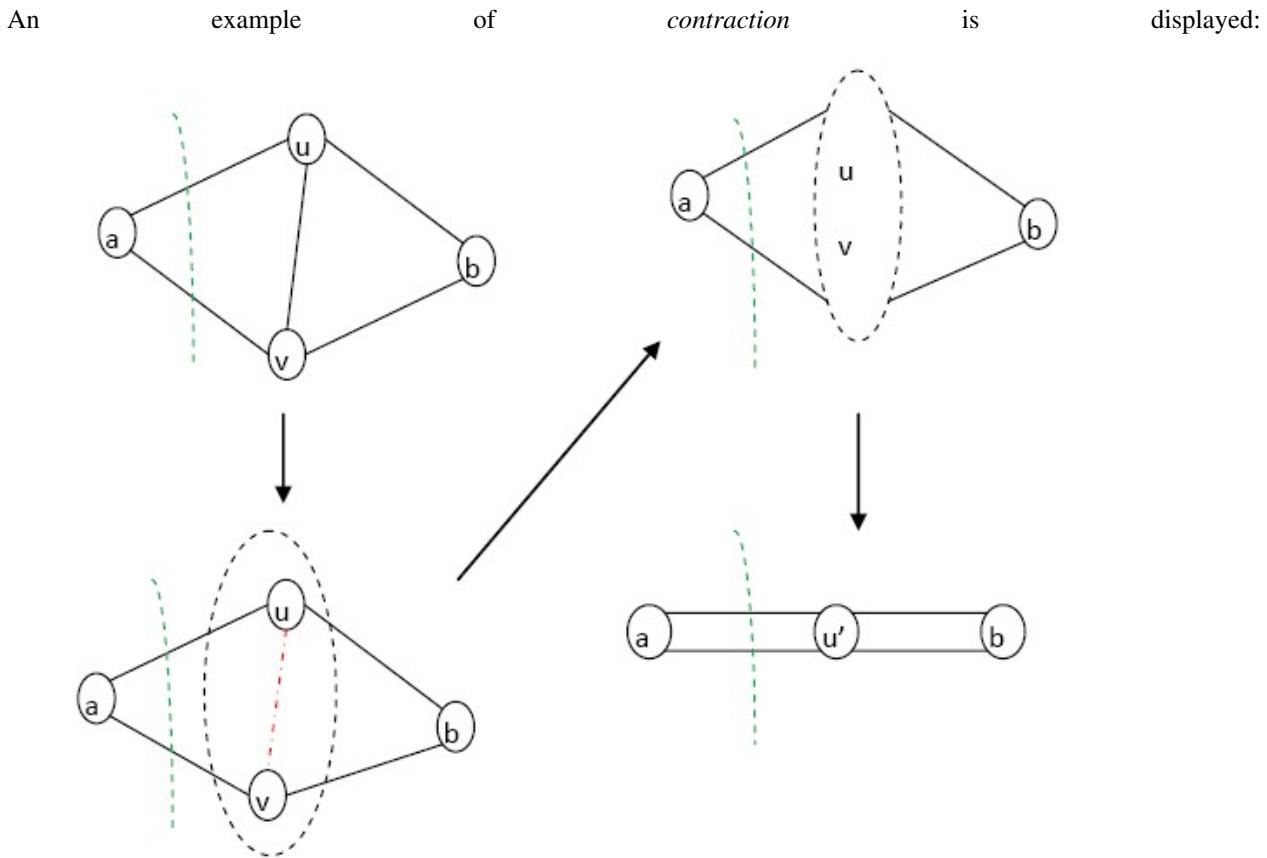
The general min-cut problem

This problem is that given an undirected graph $G = (V, E)$, trying to find a partition that divides V into a pair of non-empty, disjoint node sets $S \cup T = V$ that minimizes the number of crossed edges between S and T :

$$\text{minimize } \sum_{(u,v) \in E} \mathbf{1}_{S \times T}(u, v).$$

Algorithm description

Before describing the algorithm, we have to define the *contraction* of two nodes, which is to combine two different nodes u and v , in a graph, as a new node u' with edges that were connected with either u or v , except for the edge(s) connecting u and v .



This algorithm is shown as below:^[1]

```

INPUT: An undirected graph  $G = (V, E)$ 

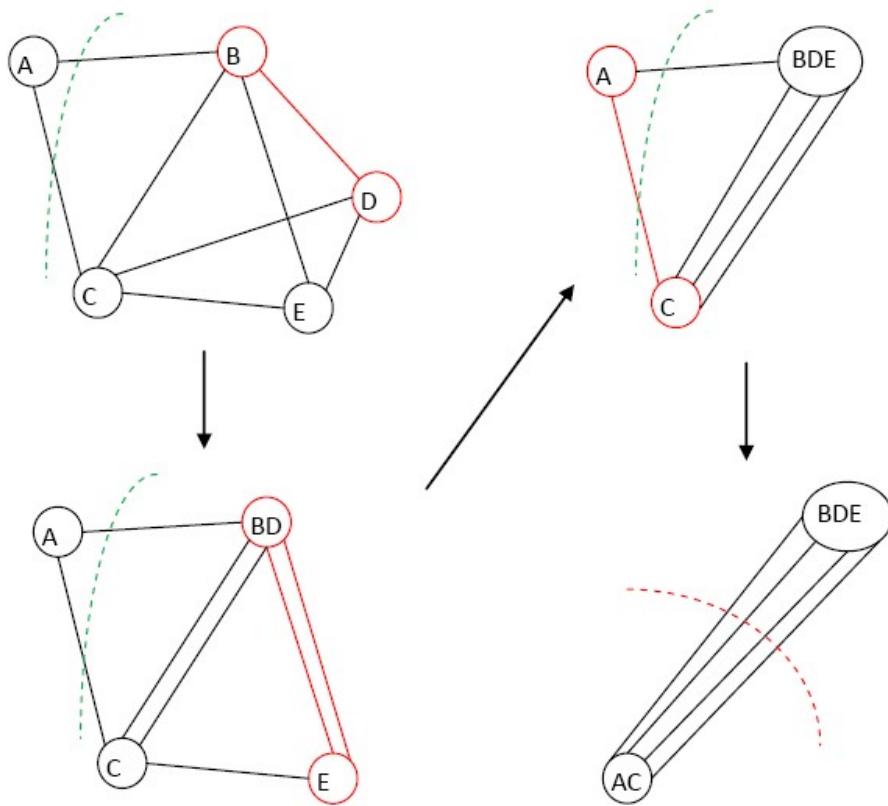
OUTPUT: The min-cut of  $G$ 

function Karger's Min-Cut( $G$ ) :
    1, Let  $T$  equal  $\theta(n^2) = cn(n - 1)/2$ ;
    2, repeat  $T$  times
    3,     while (there are more than 2 nodes left in the graph)
    4,         randomly pick an edge in  $G$ ,  $(u, v) \in E$ ;
    5,         replace  $u$  and  $v$  with the contraction of those two
    nodes,  $u'$ ;
    6,         Keep record of the number of edges between those two super
    nodes, if the new result is smaller than the previous one, replace the
    previous result with the new one;
    7, return the result that is the smallest number of edges between
    two super nodes.
end function

```

Example

This is an example of executing the inner while loop of Karger's Basic algorithm once. There are 5 nodes and 7 edges in the graph G . The min-cut of G is 2, while after one execution of the inner while loop, the cut is 4.



Analysis

Proof of correctness

Lemma 1: let k denote the actual result of the min-cut of G , every node in the graph has a degree that is equal or larger than k .

Proof: If there exists a node N in G , whose degree is smaller than k , then select N as on one side and the other nodes on the other side. As a result we get a min-cut which is smaller than k . This is a contradiction. \square

Theorem: With high probability we can find the min-cut of given graph G by executing Karger's algorithm.

Proof: Let C denote the edge set of minimum k -cut. At each stage, we have $n - i$ node and by Lemma 1 there are at least $(n - i)k/2$ edges. As such, the probability of selecting an edge in C to make a contraction is

$$\Pr[(u, v) \in C] \leq \frac{k}{(n - i)k/2} = \frac{2}{n - i}$$

In this manner, we have to run $n - 2$ iterations to reduce a graph of n nodes to a graph of only two nodes with i from 0 to $n - 3$. Thus, the probability of C surviving after $n - 2$ iterations is

$$\begin{aligned} \Pr[C] &\geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right) \\ &= \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\left(1 - \frac{2}{n-2}\right) \cdots \left(\frac{1}{3}\right) \\ &= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \\ &= \frac{2}{n(n-1)} \end{aligned}$$

Therefore, we have at least $\Omega(1/n^2)$ chances to find the min-cut C for executing the inner while loop of Karger's Basic Algorithm once. Suppose $\text{Pr}[\text{Failure}]$ indicates that the probability of failing finding the min-cut via executing the inner while loop once. As such if we execute the inner while loop $T = cn(n - 1)/2 = O(n^2)$ times, the probability of successfully returning C is

$$\begin{aligned}\text{Pr}[\text{Final Success}] &\geq 1 - \text{Pr}[\text{Failure}]^T \\ &= 1 - \left(1 - \frac{2}{n(n-1)}\right)^T \\ &= 1 - \left(1 - \frac{2}{n(n-1)}\right)^{\frac{cn(n-1)}{2}} \\ &= 1 - \frac{1}{e^c} > 99\% \text{ when } c = 10.\end{aligned}$$

Running time

Karger's algorithm is the fastest known minimum cut algorithm, is randomized, and computes a minimum cut with high probability in time $O(|V|^2 \log^3 |V|)$. To prove this authors at first mention that contraction of G to $\lceil n/\sqrt{2} + 1 \rceil$ vertices can be implemented in $O(n^2)$ time. And the running time is $T(n) = 2(n^2 + T(\lceil n/\sqrt{2} + 1 \rceil))$. This recurrence is solved by $O(n^2 \log(n))$. One run of an algorithm finds a particular minimum cut with probability $\Omega(1/\log(n))$. Running algorithm $\log^2(n)$ times reduce the chance of missing any particular minimum cut to $O(1/n^2)$.

Lemma 2: For each execution of the inner while loop (Line 3–5) of Karger's Basic Algorithm, it takes $O(n^2)$ running time.

Proof: We can represent this graph G by maintaining an adjacency list, which means all edges incident to node v are in a linked list. What is more, we have to construct pointers between the two copies the same edge (u, w) and (w, u) . When v and w are contracted, we traverse the adjacency list of v , and for each edge (v, u) find the corresponding edge (u, v) by the pointer and rename it to (u, w) . As such, each contraction costs $O(n)$ time and we have to contract $n - 2$ times, which in total is $O(n^2)$. \square

According to the algorithm, we have to repeat the inner while loop of Karger's Basic Algorithm $O(n^2)$ times. Hence, with the correctness of Lemma 2, the overall running time is $O(n^4)$.

The Karger–Stein's algorithm

This algorithm is developed by David Karger and Clifford Stein. And it is an order of magnitude improvement of the running time compared with Karger's Basic Algorithm.^[2]

Before we move forwards to the detailed description, we notice that when we do the contractions until i vertices are left, the probability that C survives is

$$\begin{aligned}\text{Pr}[C] &\geq \prod_{j=0}^{n-i} \left(1 - \frac{2}{n-j}\right) \\ &= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{n-i}{n-i+2} \cdot \frac{n-i-1}{n-i+1} \cdot \frac{n-i-2}{n-i} \\ &= \frac{(n-i-1)(n-i-2)}{n(n-1)} \\ &\in O\left(\frac{i^2}{n^2}\right)\end{aligned}$$

Algorithm description

Inspired by the formula above, we can find that the less edges that left in this graph, the less chance that C survives. We, therefore, improve the basic algorithm that

```
INPUT: An undirected graph  $G = (V, E)$ 

OUTPUT: find the min-cut of  $G$ 
function Karger-Stein's Min-Cut ( $G$ ) :
    1, run the Karger's Basic Algorithm to  $i = n^{1/2}$  twice getting  $G_1$  and  $G_2$ ,
    2, recursively run the Karger-Stein's Min-Cut Algorithm on  $G_1$  and  $G_2$ 
end function
```

Running time

Then we can get the probability of min cut survive $P(n)$ (n is the number of vertices) meets this recurrence relation:

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

Also, we can prove that $P(n) = O\left(\frac{1}{\ln n}\right)$. and the recurrence relation of running time is

$$T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2) \text{ and } T(n) = O(1) \text{ when } n \leq 2.$$

According to the Master Theorem, $T(n) = O(n^2 \ln n)$. Therefore the overall running time is $T(n) \cdot \frac{1}{P(n)} = O(n^2 \ln^2 n)$, which is an order of magnitude improvement.

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. 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

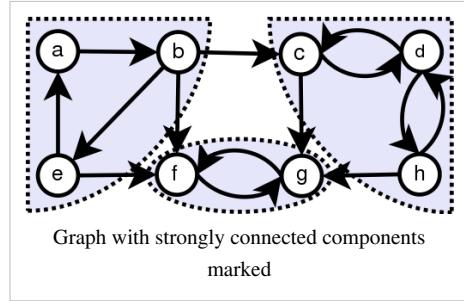
- [1] Karger, David (1993). "Global Min-cuts in RNC and Other Ramifications of a Simple Mincut Algorithm" (<http://people.csail.mit.edu/karger/Papers/mincut.ps>). *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms*. .
- [2] Karger, David; Clifford Stein (1996). "A New Approach to the Minimum Cut Problem" (<http://people.csail.mit.edu/karger/Papers/contract.ps>). *Journal of the ACM* **43** (4): 601–640. .

Connectivity of directed graphs

Strongly connected components

A directed graph is called *strongly connected* if there is a path from each vertex in the graph to every other vertex. In particular, this means paths in each direction; a path from a to b and also a path from b to a .

The **strongly connected components** of a directed graph G are its maximal strongly connected subgraphs. 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.



Kosaraju's algorithm, Tarjan's algorithm and the path-based strong component algorithm all efficiently compute the strongly connected components of a directed graph, but Tarjan's and the path-based algorithm are favoured in practice since they require only one depth-first search rather than two.

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.

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.

References

- Aspvall, Bengt; Plass, Michael F.; Tarjan, Robert E. (1979), "A linear-time algorithm for testing the truth of certain quantified boolean formulas", *Information Processing Letters* **8** (3): 121–123, doi:10.1016/0020-0190(79)90002-4.
- 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.

External links

- Java implementation for computation of strongly connected components^[1] in the jBPT library (see StronglyConnectedComponents class).

References

[1] <http://code.google.com/p/jbpt/>

Tarjan's strongly connected components algorithm

Tarjan's Algorithm (named for its discoverer, Robert Tarjan^[1]) 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. Every vertex of the graph appears in a single strongly connected component, even if it means a vertex appears in a strongly connected component by itself (as is the case with tree-like parts of the graph, as well as any vertex with no successor or no predecessor).

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). The search does not explore any node that has already been explored. The strongly connected components form the subtrees of the search tree, the roots of which are the roots of the strongly connected components.

The nodes are placed on a stack in the order in which they are visited. When the search returns from a subtree, the nodes are taken from the stack and it is determined whether each node is the root of a strongly connected component. If a node is the root of a strongly connected component, then it and all of the nodes taken off before it form that strongly connected component.

The root property

The crux of the algorithm comes in determining whether a node is the root of a strongly connected component. The concept of the "root" applies only to this algorithm (outside of the algorithm, a strongly connected component has no single "root" node). The root node is simply the first node of the strongly connected component which is encountered during the depth-first traversal. When a node is identified as the root node, once recursion on its successors has finished, all nodes on the stack from the root upwards form a complete strongly connected component.

To find the root, each node is given a depth search index `v.index`, which numbers the nodes consecutively in the order in which they are discovered. In addition, each node is assigned a value `v.lowlink` that is equal to the smallest index of some node reachable from `v`, and is always less than or equal to `v.index` if no other node is reachable from `v`. Therefore `v` is the root of a strongly connected component if and only if `v.lowlink == v.index`. The value `v.lowlink` is computed during the depth first search such that it is always known when needed.

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 ( $v.index$  is undefined) then
      strongconnect ( $v$ )

```

```

end if
repeat

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

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

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

1. 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 in G, i.e. $O(|V| + |E|)$.
2. The test for whether v' 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.
3. 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.^[2]

References

- [1] Tarjan, R. E. (1972), "Depth-first search and linear graph algorithms", *SIAM Journal on Computing* **1** (2): 146–160, doi:10.1137/0201010
[2] Harrison, Paul. "Robust topological sorting and Tarjan's algorithm in Python" (<http://www.logarithmic.net/pfh/blog/01208083168>)..
Retrieved 9 February 2011.

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 Java (http://algowiki.net/wiki/index.php?title=Tarjan's_algorithm)
- Implementation of Tarjan's Algorithm in Python (<http://www.logarithmic.net/pfh/blog/01208083168>)
- 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>)

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.^[1] 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.^[2]

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] Sedgewick (2004).
- [2] 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

- Cherian, J.; Mehlhorn, K. (1996), "Algorithms for dense graphs and networks on the random access computer", *Algorithmica* **15**: 521–549, doi: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, MR1761551.
- Munro, Ian (1971), "Efficient determination of the transitive closure of a directed graph", *Information Processing Letters* **1**: 56–58.
- Purdom, P., Jr. (1970), "A transitive closure algorithm", *BIT* **10**: 76–94.
- 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 an 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 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.

Kosaraju's algorithm is simple and 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 . 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 0-262-03384-8 .
- 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]
- Java implementation at AlgoWiki.net ^[3]

References

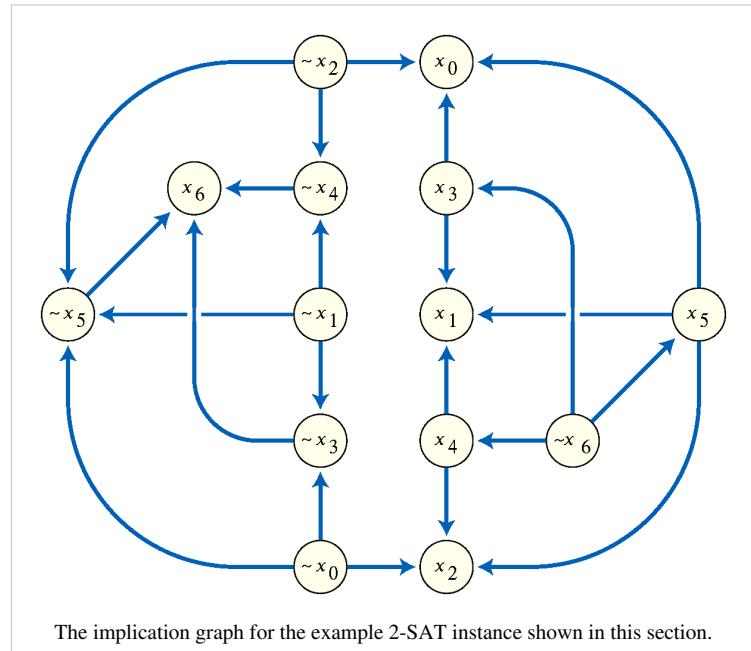
- [1] <http://lcm.csa.iisc.ernet.in/dsa/node171.html>
[2] http://scienceblogs.com/goodmath/2007/10/computing_strongly_connected_c.php
[3] http://algowiki.net/wiki/index.php?title=Kosaraju%27s_algorithm

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:



$$(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge (x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge (x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6).$$

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.^[1]

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.^[2]

Algorithms

Several algorithms are known for solving the 2-satisfiability problem; the most efficient of them take linear time.^{[1][2][3]}

Resolution and transitive closure

Krom (1967) described the following polynomial time decision procedure for solving 2-satisfiability instances.^[1]

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 $(\neg x \vee \neg x)$.^[1] 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.^[4]

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.^[3]

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.^[3]

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.^[3]

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.^[3]

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.^[2]

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^[5] and Gabow's algorithm^[6] each perform a single depth first search. Kosaraju's algorithm performs two depth first searches, but is very simple: the first search is used only to order the vertices, in the reverse of a postorder depth-first traversal. Then, the second pass of the algorithm loops through the vertices in this order, and for each vertex that has not already been assigned to a component, it performs a depth-first search of the transpose graph starting from that vertex, backtracking when the search reaches a previously assigned vertex; the unassigned vertices reached by this search form a new component.

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.^[2]

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; the order in which the components are generated by Kosaraju's algorithm is automatically a topological ordering.
- 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.^[2]

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.^[2]

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.^[7] 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.^[8]

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.^{[7][9]} 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.^[10]

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, 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.^[11] 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.^[12]

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.^[13]

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.^[14] 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.^[15]

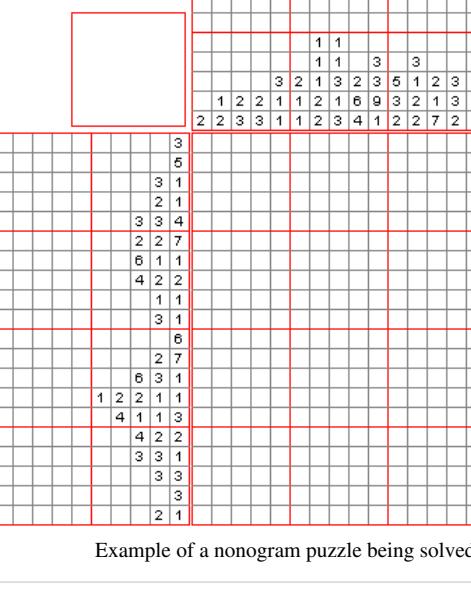
Scheduling

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.^[3]

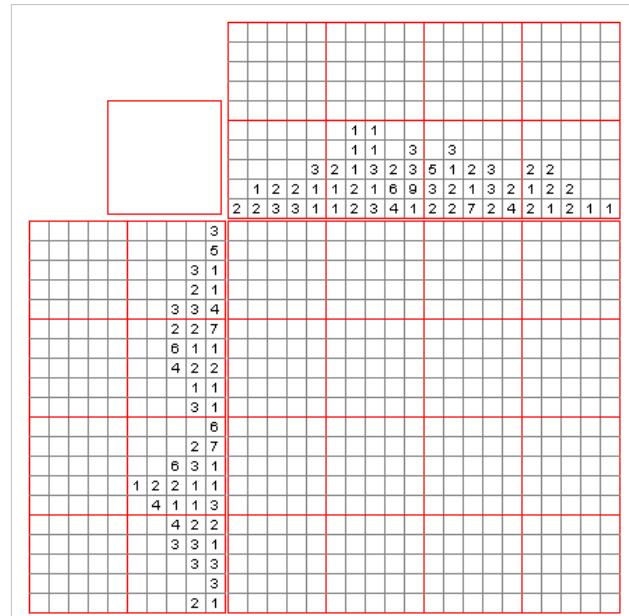
Miyashiro & Matsui (2005) apply 2-satisfiability to a problem of sports scheduling, in which the pairings of a round-robin tournament have already been chosen and the games must be assigned to the teams' stadiums. In this problem, it is desirable to alternate home and away games to the extent possible, 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).^[16]

Digital tomography

Tomography is the process of recovering shapes from their cross-sections. In digital 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 bitmap 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.



Example of a nonogram puzzle being solved.

Although there exist polynomial time algorithms to find a matrix having given row and column sums,^[17] 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.^[18] 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.^[19] 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.^[20]

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)^[3] are significantly more effective than the dynamic programming and flow heuristics without 2-SAT when applied to more difficult randomly generated nonograms.^[21]

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,^[22] inferring business relationships among autonomous subsystems of the internet,^[23] and reconstruction of evolutionary trees.^[24]

Complexity and extensions

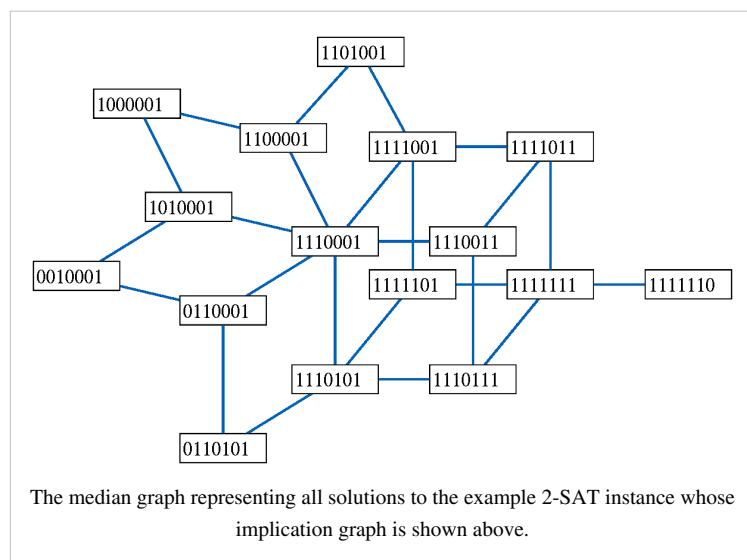
NL-completeness

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,^[25] 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.^[26] Similarly, the implicative normal form can be expressed in first order logic with the addition of an operator for transitive closure.^[26]

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.^[27]

Feder (1994) describes an algorithm for efficiently listing all solutions to a given 2-satisfiability instance, and for solving several related problems.^[28] There also exist algorithms for finding two satisfying assignments that have the maximal Hamming distance from each other.^[29]

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,^[30] 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.^[31]

The fastest known algorithm for computing the exact number of satisfying assignments to a 2SAT formula runs in time $O(1.246^n)$.^{[32] [33]}

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 $\alpha = 1$.^[34]

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.^[35] 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.^[36] Under the weaker assumption that P \neq NP, the problem is only known to be inapproximable within a constant better than $21/22 = 0.95454\dots$ ^[37]

Various authors have also explored exponential worst-case time bounds for exact solution of MAX-2-SAT instances.^[38]

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,^[39] 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.^[40]

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.^{[1][2]}

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.^[41]

References

- [1] Krom, Melven R. (1967), "The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary", *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* **13**: 15–20, doi:10.1002/malq.19670130104.
- [2] Aspvall, Bengt; Plass, Michael F.; Tarjan, Robert E. (1979), "A linear-time algorithm for testing the truth of certain quantified boolean formulas" (http://www.math.ucsd.edu/~sbuss/CourseWeb/Math268_2007WS/2SAT.pdf), *Information Processing Letters* **8** (3): 121–123, doi:10.1016/0020-0190(79)90002-4. .
- [3] Even, S.; Itai, A.; Shamir, A. (1976), "On the complexity of time table and multi-commodity flow problems", *SIAM Journal on Computing* **5** (4): 691–703, doi:10.1137/0205048.
- [4] Cook, Stephen A. (1971), "The complexity of theorem-proving procedures", *Proc. 3rd ACM Symp. Theory of Computing (STOC)*, pp. 151–158, doi:10.1145/800157.805047.
- [5] Tarjan, Robert E. (1972), "Depth-first search and linear graph algorithms", *SIAM Journal on Computing* **1** (2): 146–160, doi:10.1137/0201010.
- [6] First published by Cherian, J.; Mehlhorn, K. (1996), "Algorithms for dense graphs and networks on the random access computer", *Algorithmica* **15** (6): 521–549, doi:10.1007/BF01940880. Rediscovered in 1999 by Harold N. Gabow, and published in Gabow, Harold N. (2003), "Searching (Ch 10.1)", in Gross, J. L.; Yellen, J., *Discrete Math. and its Applications: Handbook of Graph Theory*, **25**, CRC Press, pp. 953–984.
- [7] Formann, M.; Wagner, F. (1991), "A packing problem with applications to lettering of maps", *Proc. 7th ACM Symposium on Computational Geometry*, pp. 281–288, doi:10.1145/109648.109680.
- [8] Poon, Chung Keung; Zhu, Binhai; Chin, Francis (1998), "A polynomial time solution for labeling a rectilinear map", *Information Processing Letters* **65** (4): 201–207, doi:10.1016/S0020-0190(98)00002-7.
- [9] Wagner, Frank; Wolff, Alexander (1997), "A practical map labeling algorithm", *Computational Geometry: Theory and Applications* **7** (5–6): 387–404, doi:10.1016/S0925-7721(96)00007-7.
- [10] Doddi, Srinivas; Marathe, Madhav V.; Mirzaian, Andy; Moret, Bernard M. E.; Zhu, Binhai (1997), "Map labeling and its generalizations" (<http://portal.acm.org/citation.cfm?id=314250>), *Proc. 8th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pp. 148–157, .
- [11] Efrat, Alon; Erten, Cesim; Kobourov, Stephen G. (2007), "Fixed-location circular arc drawing of planar graphs" (<http://jgaa.info/accepted/2007/EfratErtenKobourov2007.11.1.pdf>), *Journal of Graph Algorithms and Applications* **11** (1): 145–164, .
- [12] Raghavan, Raghunath; Cohoon, James; Sahni, Sartaj (1986), "Single bend wiring", *Journal of Algorithms* **7** (2): 232–237, doi:10.1016/0196-6774(86)90006-4.
- [13] Boros, Endre; Hammer, Peter L.; Minoux, Michel; Rader, David J., Jr. (1999), "Optimal cell flipping to minimize channel density in VLSI design and pseudo-Boolean optimization", *Discrete Applied Mathematics* **90** (1–3): 69–88, doi:10.1016/S0166-218X(98)00114-0.
- [14] Hansen, P.; Jaumard, B. (1987), "Minimum sum of diameters clustering", *Journal of Classification* **4** (2): 215–226, doi:10.1007/BF01896987.
- [15] Ramnath, Sarnath (2004), "Dynamic digraph connectivity hastens minimum sum-of-diameters clustering", *SIAM Journal on Discrete Mathematics* **18** (2): 272–286, doi:10.1137/S0895480102396099.

- [16] Miyashiro, Ryuhei; Matsui, Tomomi (2005), "A polynomial-time algorithm to find an equitable home-away assignment", *Operations Research Letters* **33** (3): 235–241, doi:10.1016/j.orl.2004.06.004.
- [17] Brualdi, R. A. (1980), "Matrices of zeros and ones with fixed row and column sum vectors", *Linear Algebra Appl.* **33**: 159–231, doi:10.1016/0024-3795(80)90105-6.
- [18] Woeginger, G. J. (1996), *The reconstruction of polyominoes from their orthogonal projections*, Technical Report SFB-65, Graz, Austria: TU Graz.
- [19] Chrobak, Marek; Dürr, Christoph (1999), "Reconstructing hv-convex polyominoes from orthogonal projections", *Information Processing Letters* **69** (6): 283–289, doi:10.1016/S0020-0190(99)00025-3.
- [20] Kuba, Attila; Balogh, Emese (2002), "Reconstruction of convex 2D discrete sets in polynomial time", *Theoretical Computer Science* **283** (1): 223–242, doi:10.1016/S0304-3975(01)00080-9; Brunetti, Sara; Daurat, Alain (2003), "An algorithm reconstructing convex lattice sets", *Theoretical computer science* **304** (1–3): 35–57, doi:10.1016/S0304-3975(03)00050-1.
- [21] Batenburg, K. Joost; Kosters, Walter A. (2008), "A reasoning framework for solving Nonograms", *Combinatorial Image Analysis, 12th International Workshop, IWCIA 2008, Buffalo, NY, USA, April 7–9, 2008, Proceedings*, Lecture Notes in Computer Science, **4958**, Springer-Verlag, pp. 372–383, doi:10.1007/978-3-540-78275-9_33; Batenburg, K. Joost; Kosters, Walter A. (2009), "Solving Nonograms by combining relaxations", *Pattern Recognition* **42** (8): 1672–1683, doi:10.1016/j.patcog.2008.12.003.
- [22] Brandstädt, Andreas; Hammer, Peter L.; Le, Van Bang; Lozin, Vadim V. (2005), "Bisplit graphs", *Discrete Mathematics* **299** (1–3): 11–32, doi:10.1016/j.disc.2004.08.046.
- [23] Wang, Hao; Xie, Haiyong; Yang, Yang Richard; Silberschatz, Avi; Li, Li Erran; Liu, Yanbin (2005), "Stable egress route selection for interdomain traffic engineering: model and analysis", *13th IEEE Conf. Network Protocols (ICNP)*, pp. 16–29, doi:10.1109/ICNP.2005.39.
- [24] Eskin, Eleazar; Halperin, Eran; Karp, Richard M. (2003), "Efficient reconstruction of haplotype structure via perfect phylogeny" (<http://www.worldscinet.com/cgi-bin/details.cgi?id=pii:S0219720003000174&type=html>), *Journal of Bioinformatics and Computational Biology* **1** (1): 1–20, doi:10.1142/S0219720003000174, PMID 15290779, .
- [25] Papadimitriou, Christos H. (1994), *Computational Complexity*, Addison-Wesley, pp. chapter 4.2, ISBN 0-201-53082-1., Thm. 16.3.
- [26] Cook, Stephen; Kolokolova, Antonina (2004), "A Second-Order Theory for NL", *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pp. 398–407, doi:10.1109/LICS.2004.1319634.
- [27] Bandelt, Hans-Jürgen; Chepoi, Victor (2008), "Metric graph theory and geometry: a survey" (http://www.lif-sud.univ-mrs.fr/~chepoi/survey_cm_bis.pdf), *Contemporary Mathematics*, , to appear. 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, . Feder, T. (1995), *Stable Networks and Product Graphs*, Memoirs of the American Mathematical Society, **555**.
- [28] Feder, Tomás (1994), "Network flow and 2-satisfiability", *Algorithmica* **11** (3): 291–319, doi:10.1007/BF01240738.
- [29] Angelmark, Ola; Thapper, Johan (2005), "Algorithms for the maximum Hamming distance problem", *Recent Advances in Constraints*, Lecture Notes in Computer Science, **3419**, Springer-Verlag, pp. 128–141, doi:10.1007/11402763_10.
- [30] Valiant, Leslie G. (1979), "The complexity of enumeration and reliability problems", *SIAM Journal on Computing* **8** (3): 410–421, doi:10.1137/0208032
- [31] Welsh, Dominic; Gale, Amy (2001), "The complexity of counting problems", *Aspects of complexity: minicourses in algorithmics, complexity and computational algebra: mathematics workshop, Kaikoura, January 7–15, 2000*: 115ff, Theorem 57.
- [32] Dahllöf, Vilhelm; Jonsson, Peter; Wahlström, Magnus (2005), "Counting models for 2SAT and 3SAT formulae", *Theoretical Computer Science* **332** (1–3): 265–291, doi:10.1016/j.tcs.2004.10.037
- [33] Fürer, Martin; Kasiviswanathan, Shiva Prasad (2007), "Algorithms for counting 2-SAT solutions and colorings with applications", *Algorithmic Aspects in Information and Management*, Lecture Notes in Computer Science, **4508**, Springer-Verlag, pp. 47–57, doi:10.1007/978-3-540-72870-2_5.
- [34] Bollobás, Béla; Borgs, Christian; Chayes, Jennifer T.; Kim, Jeong Han; Wilson, David B. (2001), "The scaling window of the 2-SAT transition", *Random Structures and Algorithms* **18** (3): 201–256, doi:10.1002/rsa.1006; Chvátal, V.; Reed, B. (1992), "Mick gets some (the odds are on his side)", *Proc. 33rd IEEE Symp. Foundations of Computer Science (FOCS)*, pp. 620–627, doi:10.1109/SFCS.1992.267789; Goerdt, A. (1996), "A threshold for unsatisfiability", *Journal of Computer and System Sciences* **53** (3): 469–486, doi:10.1006/jcss.1996.0081.
- [35] Lewin, Michael; Livnar, Dror; Zwick, Uri (2002), "Improved Rounding Techniques for the MAX 2-SAT and MAX DI-CUT Problems", *Proceedings of the 9th International IPCO Conference on Integer Programming and Combinatorial Optimization* (Springer-Verlag): 67–82, ISBN 3-540-43676-6
- [36] Khot, Subhash; Kindler, Guy; Mossel, Elchanan; O'Donnell, Ryan (2004), "Optimal Inapproximability Results for MAX-CUT and Other 2-Variable CSPs?", *FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (IEEE)*: 146–154, doi:10.1109/FOCS.2004.49, ISBN 0-7695-2228-9
- [37] Hästads, Johan (2001), "Some optimal inapproximability results", *Journal of the Association for Computing Machinery* **48** (4): 798–859, doi:10.1145/502090.502098.
- [38] Bansal, N.; Raman, V. (1999), "Upper bounds for MaxSat: further improved", in Aggarwal, A.; Pandu Rangan, C., *Proc. 10th Conf. Algorithms and Computation, ISAAC'99*, Lecture Notes in Computer Science, **1741**, Springer-Verlag, pp. 247–258; Gramm, Jens; Hirsch, Edward A.; Niedermeier, Rolf; Rossmanith, Peter (2003), "Worst-case upper bounds for MAX-2-SAT with an application to MAX-CUT", *Discrete Applied Mathematics* **130** (2): 139–155, doi:10.1016/S0166-218X(02)00402-X; Kojevnikov, Arist; Kulikov, Alexander S. (2006), "A new approach to proving upper bounds for MAX-2-SAT", *Proc. 17th ACM-SIAM Symp. Discrete Algorithms*, pp. 11–17, doi:10.1145/1109557.1109559

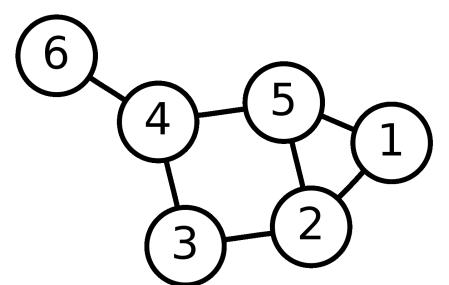
- [39] Flum, Jörg; Grohe, Martin (2006), *Parameterized Complexity Theory* (<http://www.springer.com/east/home/generic/search/results?SGWID=5-40109-22-141358322-0>), Springer, ISBN 978-3-540-29952-3,
- [40] Chen, Jianer; Huang, Xiuzhen; Kanj, Iyad A.; Xia, Ge (2006), "Strong computational lower bounds via parameterized complexity", *J. Comput. Syst. Sci.* **72** (8): 1346–1367, doi:10.1016/j.jcss.2006.04.007
- [41] Hähnle, Reiner (2001), "Advanced many-valued logics" (http://books.google.com/books?id=_ol81ow-1s4C&pg=PA373), in Gabbay, Dov M.; Günthner, Franz, *Handbook of philosophical logic*, 2 (2nd ed.), Springer, p. 373, ISBN 978-0-7923-7126-7, ; Hähnle, Reiner (2003), "Complexity of Many-valued Logics", in Fitting, Melvin; Orlowska, Ewa, *Beyond two: theory and applications of multiple-valued logic*, Springer, ISBN 978-3-7908-1541-2

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.



(6, 4, 5, 1) and (6, 4, 3, 2, 1) are both paths between vertices 6 and 1.

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} . 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 the graph is unweighted or

$f : E \rightarrow \{c\}$, $c \in \mathbb{R}^+$, 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 problems.
- 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.
- Perturbation theory finds (at worst) the locally shortest path.

Additional algorithms and associated evaluations may be found in Cherkassky et al.^[1]

Roadnetworks

A roadnetwork 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 (i.e. highways). This property has been formalized using the notion of highway dimension. (research.microsoft.com/pubs/115272/soda10.pdf^[2]) 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. (research.microsoft.com/pubs/142356/HL-TR.pdf). Other techniques that have been used are:

- ALT
- Arc Flags
- Contraction Hierarchies (<http://www.springerlink.com/index/j062316602803057.pdf>)
- Transit Node Routing
- Reach based Pruning
- Labeling

Single-source shortest paths

Directed graphs with nonnegative weights

Algorithm	Time complexity	Author
	$O(V^4)$	Shimbel 1955
	$O(V^2 EL)$	Ford 1956
Bellman–Ford algorithm	$O(VE)$	Bellman 1958, Moore 1959
	$O(V^2 \log V)$	Dantzig 1958, Dantzig 1960, Minty (cf. Pollack & Wiebenson 1960), Whiting & Hillier 1960
Dijkstra's algorithm	$O(V^2)$	Leyzorek et al. 1957, Dijkstra 1959
...
Dijkstra's algorithm with Fibonacci heaps	$O(E + V \log V)$	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.

All-pairs shortest paths

Floyd–Warshall algorithm

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.^[3]

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 include "operations research, plant and facility layout, robotics, transportation, and VLSI design".^[4]

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 (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 [5] is a representation of the primitive path network within the framework of Reptation theory.

The problems of recalculation of shortest paths arises if some graph transformations (e.g., shrinkage of nodes) are made with a graph.^[6]

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 trivial 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 arc (i, j) in A , consider the program with variables x_{ij}

$$\text{minimize } \sum_{ij \in A} w_{ij} x_{ij} \text{ subject to } x \geq 0 \text{ and for all } i, \sum_j x_{ij} - \sum_j x_{ji} = \begin{cases} 1, & \text{if } i = s; \\ -1, & \text{if } i = t; \\ 0, & \text{otherwise.} \end{cases}$$

This LP, which is common fodder for operations research courses, 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.^[7] 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.

References

- [1] Cherkassky, Boris V.; Goldberg, Andrew V.; Radzik, Tomasz (1996). "Shortest paths algorithms: theory and experimental evaluation" (<http://ftp.cs.stanford.edu/cs/theory/pub/goldberg/sp-alg.ps.Z>). *Mathematical Programming*. Ser. A **73** (2): 129–174.
doi:10.1016/0025-5610(95)00021-6. MR1392160. <!-- Bot inserted parameter. Either remove it; or change its value to "." for the cite to end in a ".">>.
- [2] <http://research.microsoft.com/pubs/115272/soda10.pdf>
- [3] Sanders, Peter (March 23, 2009). *Fast route planning* (<http://www.youtube.com/watch?v=-0ErpE8tQbw>). Google Tech Talk. .
- [4] Chen, Danny Z. (December 1996). "Developing algorithms and software for geometric path planning problems". *ACM Computing Surveys* **28** (4es): 18. doi:10.1145/242224.242246.
- [5] Kroger, Martin (2005). "Shortest multiple disconnected path for the analysis of entanglements in two- and three-dimensional polymeric systems". *Computer Physics Communications* **168** (168): 209–232. doi:10.1016/j.cpc.2005.01.020.
- [6] Ladyzhensky Y., Popoff Y. Algorithm to define the shortest paths between all nodes in a graph after compressing of two nodes. Proceedings of Donetsk national technical university, Computing and automation. Vol.107. Donetsk, 2006, pp. 68–75.
- [7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin (1993). *Network Flows: Theory, Algorithms and Applications*. Prentice Hall. ISBN 0-13-617549-X.
- 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.
- 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.

Further reading

- 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.
- 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): 338–346. doi: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.
- 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.
- 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.

Dijkstra's algorithm for single-source shortest paths with positive edge lengths

Dijkstra's algorithm

<p>Dijkstra's algorithm runtime</p>	
Class	Search algorithm
Data structure	Graph
Worst case performance	$O(E + V \log V)$

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959,^{[1][2]} is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative 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 first 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 $O(|V|^2)$. 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 $O(|E| + |V| \log |V|)$ is due to (Fredman & Tarjan 1984). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded nonnegative weights.

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 except the initial node.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. For example, if the current node A is marked with a tentative 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 this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as *visited* at this time, and it remains in the *unvisited set*.
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; its distance recorded now is final and minimal.
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), then stop. The algorithm has finished.
6. Set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3.

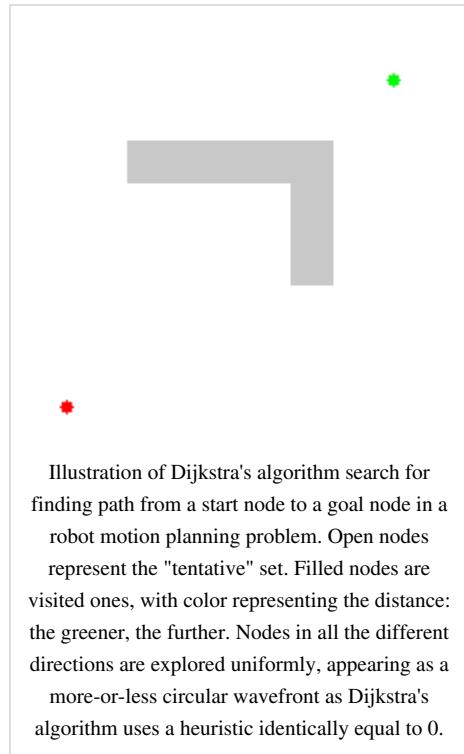


Illustration of Dijkstra's algorithm search for finding path from a start node to a goal node in a robot motion planning problem. Open nodes represent the "tentative" set. Filled nodes are visited ones, with color representing the distance: the greener, the further. Nodes in all the different directions 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, formally these terms are **vertex**, **edge** and **graph**, respectively.

Suppose you want 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 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 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. In some sense, this algorithm "expands outward" from the starting point, iteratively 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 smallest dist[]`, searches for the vertex `u` in the vertex set `Q` that has the least `dist[u]` value. That vertex is removed from the set `Q` and returned to the user. `dist_between(u, v)` calculates the length between the two neighbor-nodes `u` and `v`. The variable `alt` on lines 20 & 22 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      for each vertex v in Graph:                                // Initializations
3          dist[v] := infinity ;                                 // Unknown distance function from
4
4          previous[v] := undefined ;                            // Previous node in optimal path
5
5          previous[v] := undefined ;                            // from source
6
6
7
8          dist[source] := 0 ;                                  // Distance from source to source
9
9          Q := the set of all nodes in Graph ;                // All nodes in the graph are
10
10
11         while Q is not empty:                                // unoptimized - thus are in Q
12             u := vertex in Q with smallest distance in dist[] ; // The main loop
13             remove u from Q ;                               // Start node in first case
14
14             if dist[u] = infinity:
15                 break ;                                    // all remaining vertices are
16
16
17
18             for each neighbor v of u:                      // where v has not yet been
19
19
20                 alt := dist[u] + dist_between(u, v) ;
21
21                 if alt < dist[v]:                           // Relax (u,v,a)
22                     dist[v] := alt ;
23                     previous[v] := u ;
24
24
25             return dist;

```

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 iteration:

```

1  S := empty sequence
2  u := target
3  while previous[u] is defined:
4      insert u at the beginning of S
5      u := previous[u]
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 short paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search.

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 binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. With a 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.^[3]

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.)

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 nonnegative 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.

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.^{[4][5][6]}

In fact, Dijkstra's explanation of the logic behind the algorithm,^[7] 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] Dijkstra, Edsger; Thomas J. Misa, Editor (2010-08). "An Interview with Edsger W. Dijkstra". *Communications of the ACM* **53** (8): 41–47. doi:10.1145/1787234.1787249. "What is the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path which I designed in about 20 minutes. One morning I was shopping with my young fianceé, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path."
- [2] Dijkstra 1959
- [3] http://www.boost.org/doc/libs/1_44_0/libs/graph/doc/dag_shortest_paths.html
- [4] Sniedovich, M. (2006). "Dijkstra's algorithm revisited: the dynamic programming connexion" (<http://matwbn.icm.edu.pl/ksiazki/cc/cc35/cc3536.pdf>) (PDF). *Journal of Control and Cybernetics* **35** (3): 599–620. . Online version of the paper with interactive computational modules. (http://www.ifors.ms.unimelb.edu.au/tutorial/dijkstra_new/index.html)
- [5] Denardo, E.V. (2003). *Dynamic Programming: Models and Applications*. Mineola, NY: Dover Publications. ISBN 978-0-486-42810-9.
- [6] Sniedovich, M. (2010). *Dynamic Programming: Foundations and Principles*. Francis & Taylor. ISBN 978-0-8247-4099-3.
- [7] Dijkstra 1959, p. 270

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.
- 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" (<http://www.computer.org/portal/web/csdl/doi/10.1109/SFCS.1984.715934>). *25th Annual Symposium on Foundations of Computer Science* (IEEE): 338–346. doi: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.
- 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.
- 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.

External links

- Oral history interview with Edsger W. Dijkstra (<http://purl.umn.edu/107247>), Charles Babbage Institute University of Minnesota, Minneapolis.
- 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>)
- Applet by Carla Laffra of Pace University (<http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html>)
- Animation of Dijkstra's algorithm (<http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html>)
- Visualization of Dijkstra's Algorithm (http://students.ceid.upatras.gr/~papagel/english/java_docs/minDijk.htm)
- Shortest Path Problem: Dijkstra's Algorithm (<http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml>)
- Dijkstra's Algorithm Applet (<http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm>)
- Open Source Java Graph package with implementation of Dijkstra's Algorithm (<http://code.google.com/p/annas/>)
- Haskell implementation of Dijkstra's Algorithm (<http://bonsaicode.wordpress.com/2011/01/04/programming-praxis-dijkstra's-algorithm/>) on Bonsai code
- Java Implementation of Dijkstra's Algorithm (http://algowiki.net/wiki/index.php?title=Dijkstra's_algorithm) on AlgoWiki
- QuickGraph, Graph Data Structures and Algorithms for .NET (<http://quickgraph.codeplex.com/>)
- Implementation in Boost C++ library (http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/dijkstra_shortest_paths.html)
- Implementation in T-SQL (<http://hansolav.net/sql/graphs.html>)
- A Java library for path finding with Dijkstra's Algorithm and example applet (<http://www.stackframe.com/software/PathFinder>)

- A MATLAB program for Dijkstra's algorithm (<http://www.mathworks.com/matlabcentral/fileexchange/20025-advanced-dijkstras-minimum-path-algorithm>)
- A basic C++ implementation of Dijkstra's algorithm (<http://www.technical-recipes.com/2011/implementing-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	$O(V)$

The **Bellman–Ford algorithm** computes single-source shortest paths in a weighted digraph.^[1] For graphs with only non-negative edge weights, the faster Dijkstra's algorithm also solves the problem. Thus, Bellman–Ford is used primarily for graphs with negative edge weights. The algorithm is named after its developers, Richard Bellman and Lester Ford, Jr.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.^[2] However, if a graph contains a "negative cycle", i.e., a cycle whose edges sum to a negative value, then walks of arbitrarily low weight can be constructed by repeatedly following the cycle, so there may not be a *shortest* path. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence, but it cannot produce a correct "shortest path" answer if a negative cycle is reachable from the source.^{[3][1]}

Algorithm

Bellman–Ford is based on dynamic programming approach. In its basic structure it is similar to Dijkstra's Algorithm, but instead of greedily selecting the minimum-weight node not yet processed to relax, it simply relaxes *all* the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. The repetitions allow minimum distances to propagate accurately throughout the graph, since, in the absence of negative cycles, the shortest path can visit each node at most only once. Unlike the greedy approach, which depends on certain structural assumptions derived from positive weights, this straightforward approach extends to the general case.

Bellman–Ford runs in $O(|V|\cdot|E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively.

```

procedure BellmanFord(list vertices, list edges, vertex source)
    // This implementation takes in a graph, represented as lists of vertices
    // and edges, and modifies the vertices so that their distance and
    // predecessor attributes store the shortest paths.

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then v.distance := 0
        else v.distance := infinity
        v.predecessor := null

```

```

// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge uv in edges: // uv is the edge from u to v
        u := uv.source
        v := uv.destination
        if u.distance + uv.weight < v.distance:
            v.distance := u.distance + uv.weight
            v.predecessor := u

// Step 3: check for negative-weight cycles
for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if u.distance + uv.weight < v.distance:
        error "Graph contains a negative-weight cycle"

```

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 $u.\text{distance} + uv.\text{weight}$ 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, $v.\text{distance}$ after $i-1$ cycles is at most the length of this path. Therefore, $uv.\text{weight} + v.\text{distance}$ is at most the length of the path from s to u . In the i^{th} cycle, $u.\text{distance}$ gets compared with $uv.\text{weight} + v.\text{distance}$, and is set equal to it if $uv.\text{weight} + v.\text{distance}$ was smaller. Therefore, after i cycles, $u.\text{distance}$ 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 $v[0], \dots, v[k-1]$,

$v[i].\text{distance} \leq v[(i-1) \bmod k].\text{distance} + v[(i-1) \bmod k]v[i].\text{weight}$

Summing around the cycle, the $v[i].\text{distance}$ terms and the $v[i-1 \bmod k]v[i].\text{weight}$ terms cancel, leaving

$0 \leq \sum_{i=1}^k v[i-1 \bmod k]v[i].\text{weight}$

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.^[1]

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).

Yen's improvement

Yen (1970) described an improvement to the Bellman–Ford algorithm for a graph without negative-weight cycles. Yen's improvement first assigns some arbitrary linear order on all vertices and then partitions the set of all edges into one of 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 . Yen's improvement effectively halves the number of "passes" required for the single-source-shortest-path solution.

Notes

- [1] Bang-Jensen, Jørgen; Gregory Gutin. "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..
- [2] Sedgewick, Robert. "Section 21.7: Negative Edge Weights" (<http://safari.oreilly.com/0201361213/ch21lev1sec7>). *Algorithms in Java* (Third ed.). ISBN 0-201-36121-3..
- [3] Kleinberg, Jon; Tardos, Éva (2006), Algorithm Design, New York: Pearson Education, Inc..

References

- Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* **16**: 87–90. MR0102435..
- Ford, L. R., Jr.; Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press..
- 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.

- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. *Introduction to Algorithms*. MIT Press and McGraw-Hill., 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.
- 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. MR0253822..

External links

- C code example (<http://snippets.dzone.com/posts/show/4828>)
- Open Source Java Graph package with Bellman–Ford Algorithms (<http://code.google.com/p/annas/>)
- C++ implementation of Yen's Algorithm (Microsoft Visual Studio project) (<http://www.technical-recipes.com/2012/the-k-shortest-paths-algorithm-in-c/>)

Johnson's algorithm for all-pairs shortest paths in sparse graphs

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse 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. 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 (1974) for finding two disjoint paths of minimum total length between the same two vertices in a graph with non-negative edge weights.

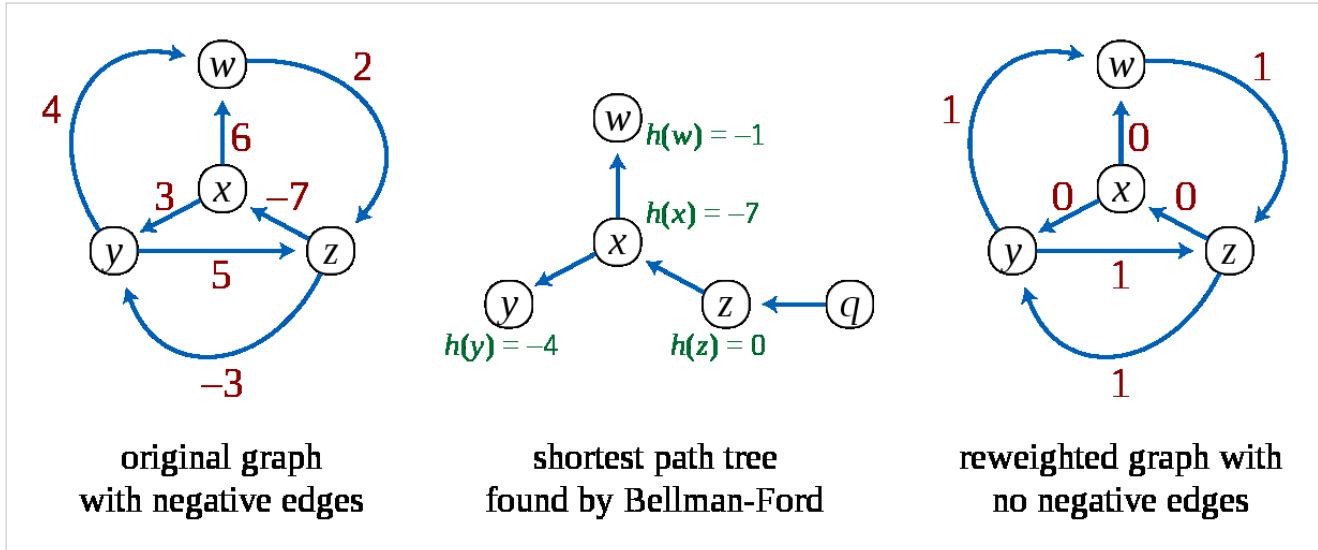
Algorithm description

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:

$$(w(s,p_1)+h(s)-h(p_1))+(w(p_1,p_2)+h(p_1)-h(p_2))+\dots+(w(p_n,t)+h(p_n)-h(t)).$$

Notice that 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 :

$$(w(s,p_1) + w(p_1,p_2) + \dots + w(p_n,t)) + h(s) - h(t)$$

Notice that 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

- Black, Paul E. (2004), "Johnson's Algorithm" ^[1], *Dictionary of Algorithms and Data Structures*, National Institute of Standards and Technology.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), *Introduction to Algorithms*, MIT Press and McGraw-Hill, ISBN 978-0-262-03293-3. Section 25.3, "Johnson's algorithm for sparse graphs", pp. 636–640.
- Johnson, Donald B. (1977), "Efficient algorithms for shortest paths in sparse networks", *Journal of the ACM* **24** (1): 1–13, doi:10.1145/321992.321993.
- Suurballe, J. W. (1974), "Disjoint paths in a network", *Networks* **14** (2): 125–145, doi:10.1002/net.3230040204.

External links

- Boost: All Pairs Shortest Paths ^[2]

References

[1] <http://www.nist.gov/dads/HTML/johnsonsAlgorithm.html>

[2] 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–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	$\Theta(V ^2)$

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) 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 algorithm is an example of dynamic programming. It 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.^[1] The modern formulation of Warshall's algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962.

Algorithm

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with only $\Theta(|V|^3)$ 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 , each 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 $\text{shortestPath}(i, j, k)$ in terms of the following recursive formula: the base case is

$$\text{shortestPath}(i, j, 0) = w(i, j)$$

and the recursive case is

$$\text{shortestPath}(i, j, k) = \min(\text{shortestPath}(i, j, k-1), \text{shortestPath}(i, k, k-1) + \text{shortestPath}(k, j, k-1))$$

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

Conveniently, when calculating the k th case, one can overwrite the information saved from the computation of $k - 1$. This means the algorithm uses quadratic memory. Be careful to note the initialization conditions:

```

1 /* Assume a function edgeCost(i, j) which returns the cost of the edge from i to j
2     (infinity if there is none).
3     Also assume that n is the number of vertices and edgeCost(i, i) = 0
4 */
5
6 int path[][][];
7 /* A 2-dimensional matrix. At each step in the algorithm, path[i][j] is the shortest path
8    from i to j using intermediate vertices (1..k-1). Each path[i][j] is initialized to
9    edgeCost(i, j).
10 */
11
12 procedure FloydWarshall ()
13     for k := 1 to n
14         for i := 1 to n
15             for j := 1 to n
16                 path[i][j] = min ( path[i][j], path[i][k]+path[k][j] );

```

Behavior with negative cycles

A negative cycle is a cycle whose edges sum to a negative value. Between any pair of vertices which form part of a negative cycle, the shortest path is not well-defined because the path can be arbitrarily 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.^[2] Obviously, in an undirected graph a negative edge creates a negative cycle (i.e., a closed walk) involving its incident vertices.

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. For each vertex, one need only store the information about the highest index intermediate vertex one must pass through if one wishes to arrive at any given vertex. Therefore, information to reconstruct all paths can be stored in a single $N \times N$ matrix $next$ where $next[i][j]$ represents the highest index vertex one must travel through if one intends to take the shortest path from i to j . Implementing such a scheme

is trivial; when a new shortest path is found between two vertices, the matrix containing the paths is updated. The *next* matrix is updated along with the *path* matrix, so at completion both tables are complete and accurate, and any entries which are infinite in the *path* table will be null in the *next* table. The path from i to j is the path from i to $\text{next}[i][j]$, followed by the path from $\text{next}[i][j]$ to j . These two shorter paths are determined recursively. This modified algorithm runs with the same time and space complexity as the unmodified algorithm.

```

1 procedure FloydWarshallWithPathReconstruction ()
2     for k := 1 to n
3         for i := 1 to n
4             for j := 1 to n
5                 if path[i][k] + path[k][j] < path[i][j] then {
6                     path[i][j] := path[i][k]+path[k][j];
7                     next[i][j] := k; }
8
9 function GetPath (i,j)
10    if path[i][j] equals infinity then
11        return "no path";
12    int intermediate := next[i][j];
13    if intermediate equals 'null' then
14        return " "; /* there is an edge from i to j, with no vertices between */
15    else
16        return GetPath(i,intermediate) + intermediate + GetPath(intermediate,j);

```

Analysis

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)
- Inversion of real matrices (Gauss–Jordan algorithm)^[3]
- 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.
- Testing whether an undirected graph is bipartite.
- Fast computation of Pathfinder networks.
- Widest paths/Maximum bandwidth paths

Implementations

Implementations are available for many programming languages.

- For C++, in the boost::graph^[4] library
- For C#, at QuickGraph^[5]
- For Clojure, at paste.lisp.org^[6]
- For Java, in the Apache commons graph^[7] library, or at Algowiki^[8]
- For MATLAB, in the Matlab_bgl^[9] package
- For Perl, in the Graph^[10] module
- For PHP, on page^[11] and PL/pgSQL, on page^[12] at Microshell
- For Python, in the NetworkX library
- For R, in package e1017^[13]
- For Ruby, in script^[14]

References

- [1] Weisstein, Eric. "Floyd-Warshall Algorithm" (<http://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html>). *Wolfram MathWorld*. . Retrieved 13 November 2009.
- [2] "Lecture 12: Shortest paths (continued)" (<http://www.ier.berkeley.edu/~ier266/Lecture12.pdf>) (PDF). *Network Flows and Graphs*. Department of Industrial Engineering and Operations Research, University of California, Berkeley. 7 October 2008. .
- [3] Penaloza, Rafael. "Algebraic Structures for Transitive Closure" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.7650>). .
- [4] <http://www.boost.org/libs/graph/doc/>
- [5] <http://www.codeplex.com/quickgraph>
- [6] <http://paste.lisp.org/display/95370>
- [7] <http://svn.apache.org/repos/asf/commons/dormant/graph2/branches/jakarta/src/java/org/apache/commons/graph/impl/AllPaths.java>
- [8] http://algowiki.net/wiki/index.php?title=Floyd-Warshall%27s_algorithm
- [9] <http://www.mathworks.com/matlabcentral/fileexchange/10922>
- [10] <http://search.cpan.org/search?query=Graph&mode=all>
- [11] <http://www.microshell.com/programming/computing-degrees-of-separation-in-social-networking/2/>
- [12] <http://www.microshell.com/programming/floyd-warshall-algorithm-in-postgresql-plpgsql/3/>
- [13] <http://cran.r-project.org/web/packages/e1071/index.html>
- [14] <https://github.com/choillier/ruby-floyd>
- 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.
- Ingerman, Peter Z. (November 1962). *Algorithm 141: Path Matrix*. **5**. p. 556. doi: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.
- Kenneth H. Rosen (2003). *Discrete Mathematics and Its Applications*, 5th Edition. Addison Wesley. ISBN 0-07-119881-4 (ISE).
- 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/shorest_path/shorest_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/~papagel/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.^{[1][2][3]} 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.

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 .

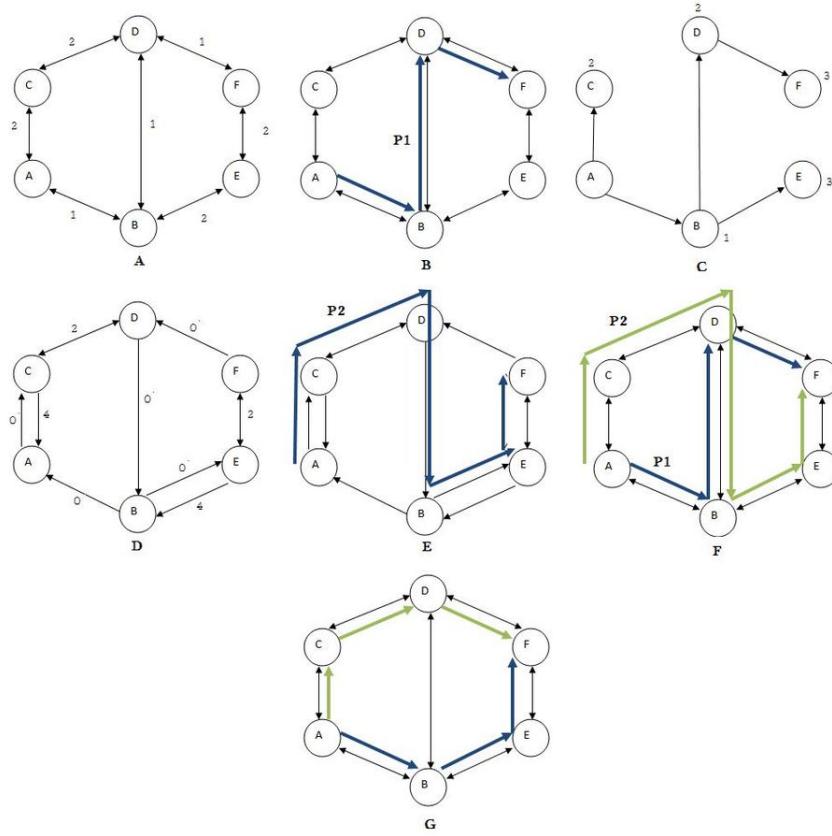


Figure **A** illustrates a weighted graph G .

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

- [1] Suurballe, J. W. (1974), "Disjoint paths in a network", *Networks* **14**: 125–145, doi:10.1002/net.3230040204.
- [2] Suurballe, J. W.; Tarjan, R. E. (1984), "A quick method for finding shortest pairs of disjoint paths" (http://www.cse.yorku.ca/course_archive/2007-08/F/6590/Notes/surballe_alg.pdf), *Networks* **14**: 325–336, doi:10.1002/net.3230140209, .
- [3] Bhandari, Ramesh (1999), "Suurballe's disjoint pair algorithms", *Survivable Networks: Algorithms for Diverse Routing*, Springer-Verlag, pp. 86–91, ISBN 978-0-7923-8381-9.

Bidirectional search

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 was the first one to design and implement a bi-directional heuristic search algorithm. Andrew Goldberg and others explain how the correct termination for the bidirectional Dijkstra's Algorithm has to be.^[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>)
- 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.
 - de Champeaux, Dennis (1983), "Bidirectional heuristic search again", *Journal of the ACM* **30** (1): 22–32, doi: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

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.

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first described the algorithm in 1968.^[1] It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance (with respect to time) 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 *known* heuristic cost, keeping a sorted priority queue of alternate path segments along the way.

It uses a distance-plus-cost heuristic function of node x (usually denoted $f(x)$) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions:

- the path-cost function, which is the cost from the starting node to the current node x (usually denoted $g(x)$)
- 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 $d'(x, y) := d(x, y) - h(x) + h(y)$.

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. They thus named the new algorithm in Kleene star syntax to be the algorithm that starts with A and includes all possible version numbers or A^* .

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*. 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 h 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. If the actual shortest path is desired, the algorithm may also update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal 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.

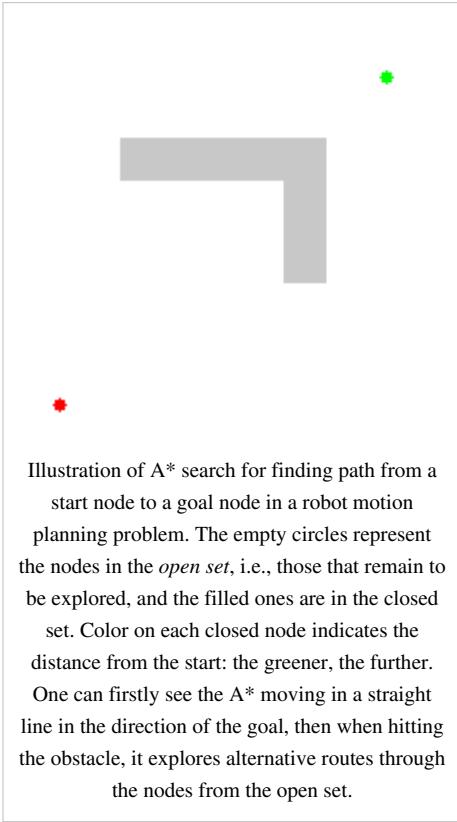


Illustration of A* search for finding path from a start node to a goal node in a robot motion planning problem. The empty circles represent the nodes in the *open set*, i.e., those that remain to be explored, and the filled ones are in the *closed set*. Color on each closed node indicates the distance from the start: the greener, the further. One can firstly see the A* moving in a straight line in the direction of the goal, then when hitting the obstacle, it explores alternative routes through the nodes from the open set.

Pseudocode

The following pseudocode describes the algorithm:

```

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]
        if neighbor not in openset
            add neighbor to openset
            came_from[neighbor] := current
            g_score[neighbor] := tentative_g_score
            f_score[neighbor] := g_score[neighbor] +
heuristic_cost_estimate(neighbor, goal)

    return failure

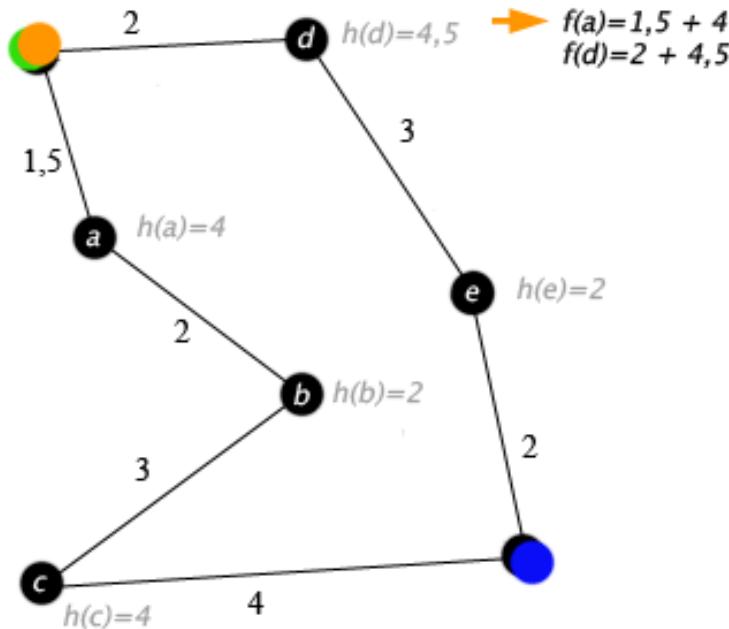
function reconstruct_path(came_from, current_node)
    if came_from[current_node] is set
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node

```

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

Note: This example uses a comma as the decimal separator.

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 $L(\cdot)$ 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 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.

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.

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. See (Generalized best-first search strategies and the optimality of A*, Rina Dechter and Judea Pearl, 1985^[2])
- 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. See (Incremental heuristic search in artificial intelligence, Sven Koenig, Maxim Likhachev, Yixin Liu and David Furcy, 2004^[3])

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 $(1 + \epsilon)$ times the optimal solution path. This new guarantee is referred to as ϵ -admissible.

There are a number of ϵ -admissible algorithms:

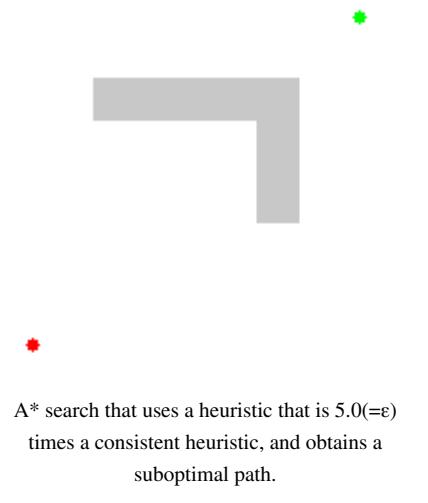
- Weighted A*. If $h_a(n)$ is an admissible heuristic function, in the weighted version of the A* search one uses $h_w(n) = \epsilon h_a(n)$, $\epsilon > 1$ as the heuristic function, and perform the A* search as usual (which eventually happens faster than using h_a since fewer nodes are expanded). The 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.^[4]
- Static Weighting^[5] uses the cost function $f(n) = g(n) + (1 + \epsilon)h(n)$.
- Dynamic Weighting^[6] uses the cost function $f(n) = g(n) + (1 + \epsilon w(n))h(n)$, 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.^[7] uses sampling of nodes to better estimate and debias the heuristic error.
- A_{ϵ}^* .^[8] 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} .^[9] selects nodes with the function $Af(n) + Bh_F(n)$, where A and B are constants. If no nodes can be selected, the algorithm will backtrack with the function $Cf(n) + Dh_F(n)$, where C and D are constants.
- AlphA*.^[10] 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 (see Pearl 1984^[11] and also Russell and Norvig 2003, p. 101^[12])



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*)
- Theta*
- A* can be adapted to a bidirectional search algorithm

References

- [1] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics SSC-4* (2): 100–107. doi:10.1109/TSSC.1968.300136.
- [2] Dechter, Rina; Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*" (<http://portal.acm.org/citation.cfm?id=3830&coll=portal&dl=ACM>). *Journal of the ACM* **32** (3): 505–536. doi:10.1145/3828.3830..
- [3] Koenig, Sven; Maxim Likhachev, Yixin Liu, David Furcy (2004). "Incremental heuristic search in AI" (<http://portal.acm.org/citation.cfm?id=1017140>). *AI Magazine* **25** (2): 99–112. .
- [4] Pearl, Judea (1984). *Heuristics: intelligent search strategies for computer problem solving* (<http://portal.acm.org/citation.cfm?id=525>). Addison-Wesley Longman Publishing Co., Inc.. ISBN 0-201-05594-5. .
- [5] Pohl, Ira (1970). "First results on the effect of error in heuristic search". *Machine Intelligence* **5**: 219-236.
- [6] Pohl, Ira (August, 1973). "The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving". *Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI-73)*. **3**. California, USA. pp. 11-17.
- [7] Köll, Andreas; Hermann Kaindl (August, 1992). "A new approach to dynamic weighting". *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*. Vienna, Austria. pp. 16-17.
- [8] Pearl, Judea; Jin H. Kim (1982). "Studies in semi-admissible heuristics". *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* **4** (4): 392-399.
- [9] "A_ε - an efficient near admissible heuristic search algorithm". *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*. **2**. Karlsruhe, Germany. August, 1983. pp. 789-791.
- [10] Reese, Bjørn (1999). *AlphA*: An ε -admissible heuristic search algorithm*.
- [11] Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley. ISBN 0-201-05594-5.
- [12] Russell, S. J.; Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, N.J.: Prentice Hall. pp. 97–104. ISBN 0-13-790395-2.

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.
- Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto, California: Tioga Publishing Company. ISBN 0-935382-01-1.
- Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley. ISBN 0-201-05594-5.

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>)

Longest path problem

In the mathematical discipline of graph theory, 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. Unlike the shortest path problem, which asks for the shortest path between two vertices and can be solved in polynomial time in graphs without negative-weight cycles, the decision version of this problem is NP-complete, which means that the optimal solution cannot be found in polynomial time unless P = NP.

The standard decision version of this problem asks whether the graph contains a simple path of length greater than or equal to k , where the length of a path is defined to be the number of edges along the path.

NP-completeness

The NP-completeness of the decision problem can be shown using a reduction from the Hamiltonian path problem. Clearly, if a certain general graph has a Hamiltonian path, this Hamiltonian path is the longest path possible, as it traverses all possible vertices. To solve the Hamiltonian path problem using an algorithm for the longest path problem, we use the algorithm for the longest path problem on the same input graph and set $k=|V|-1$, where $|V|$ is the number of vertices in the graph.

If there is a Hamiltonian path in the graph, then the algorithm will return *yes*, since the Hamiltonian path has length equal to $|V|-1$. Conversely, if the algorithm outputs *yes*, there is a simple path of length $|V|-1$ in the graph. Since it is a simple path of length $|V|-1$, it is a path that visits all the vertices of the graph without repetition, and this is a Hamiltonian path by definition.

Since the Hamiltonian path problem is NP-complete, this reduction shows that this problem is NP-hard. To show that it is NP-complete, we also have to show that it is in NP. This is easy to see, however, since the certificate for the *yes*-instance is a description of the path of length k .

Relation to the shortest path problem

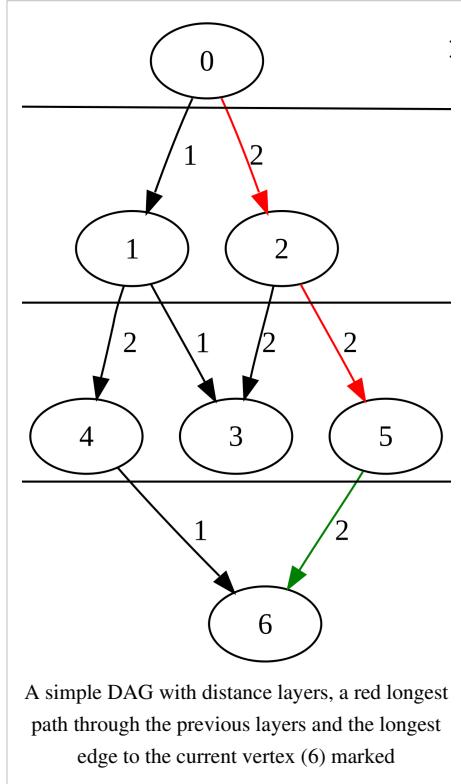
The longest path problem can be reduced to the shortest path problem (although the graph may have negative-weight cycles), by exploiting the duality of optimizations (maximizing a positive value is the same as minimizing a negative value). If the input graph to the longest path problem is G , the shortest simple path on the graph H , which is exactly the same as G but with edge weights negated, is the longest simple path on G . However, any positive-weight cycles in the original graph G lead to negative-weight cycles in H . Finding the shortest simple path on a graph with negative-weight cycles is therefore also NP-complete. If G contains no cycles, then H will have no negative-weight cycles, and any shortest-path finding algorithm can now be run on H to solve the original problem in polynomial time. Thus the longest path problem is easy on acyclic graphs.

Algorithms for acyclic graphs

As mentioned above, the longest path problem on acyclic graphs can be solved in polynomial time by negating the edge weights and running a shortest-path finding algorithm. The algorithm described below does not use this reduction and achieves a better time complexity.

Weighted directed acyclic graphs

If G is a directed acyclic graph, the longest path problem on G can be solved in linear time using dynamic programming. Assume that $\text{topOrder}(G)$ outputs a sequence of vertices in topological order (This can be computed via a topological sort and this step requires that the input graph is a directed acyclic graph). Furthermore, let $V(G)$ be the set of vertices of G and $E(G)$ be the set of edges in G and, if weights are defined, let $\text{weight}(G, e)$ be the weight of an edge e in the graph G (if the graph is unweighted, insert an arbitrary constant other than zero for any occurrence of $\text{weight}(G, e)$). Then the following algorithm computes the length of the longest path:



```

algorithm dag-longest-path is
    input:
        Directed acyclic graph G
    output:
        Length of the longest path

    length_to = array with  $|V(G)|$  elements of type int with default value 0

    for each vertex v in topOrder(G) do
        for each edge (v, w) in E(G) do
            if length_to[w] <= length_to[v] + weight(G, (v, w)) then
                length_to[w] = length_to[v] + weight(G, (v, w))

    return max(length_to[v] for v in V(G))

```

Correctness can be checked as follows: The topological order traverses the given graph in layers of ascending distance from the source vertices of the graph (at first, all vertex with distance 0 from the source vertices, then all vertices with distance 1 from the source vertices, ...). For each vertex, it clearly holds that a path of maximum length consists of a path of maximum distance through all the layers which are closer to the source than this vertex and also

an edge of maximum length connecting the longest path up to this node with this node. This is exactly $\text{length_to}(w) = \max(\text{weight}(G, (v, w)) + \text{length_to}(v))$ for all edges (v, w) in G , just written in prose. Considering that the topological order traverses all possible vertices v on the previous layer before traversing w shows that the implementation computes exactly this maximum for each vertex and thus, the maximum of all lengths to a vertex is the correct length.

The worst-case running time of this algorithm is $O(T + |V| + |E| + |V|)$, if T is the time required by the topological order. Assuming a standard algorithm for computing the topological order, this simplifies into $O(|V| + |E| + |V| + |E| + |V|)$, which simplifies further into $O(|V| + |E|)$.

Furthermore, extending this algorithm to compute the actual path is straightforward. In order to do this, it is necessary to introduce a predecessor (or successor)-array which is updated whenever the length_to is modified. Given this, one can try to reconstruct the path from the sources and sinks of the directed acyclic graph and once there is a path of maximum length reconstructed, this is one of the longest paths.

Related problems

- Travelling salesman problem
- Gallai–Hasse–Roy–Vitaver theorem

External links

- NP-Completeness ^[1]
- Source of the algorithm ^[2]
- "Find the Longest Path ^[3]", song by Dan Barrett

References

- [1] <http://www.ics.uci.edu/~eppstein/161/960312.html>
- [2] <http://www.cs.princeton.edu/courses/archive/spr04/cos226/lectures/digraph.4up.pdf>
- [3] <http://valis.cs.uiuc.edu/~sariel/misc/funny/longestpath.mp3>

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 directed 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.^[1] 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,^[2] and has been applied to digital compositing,^[3] metabolic analysis,^[4] and the computation of maximum flows.^[5] 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.^[6] However, in many cases even faster algorithms are possible.

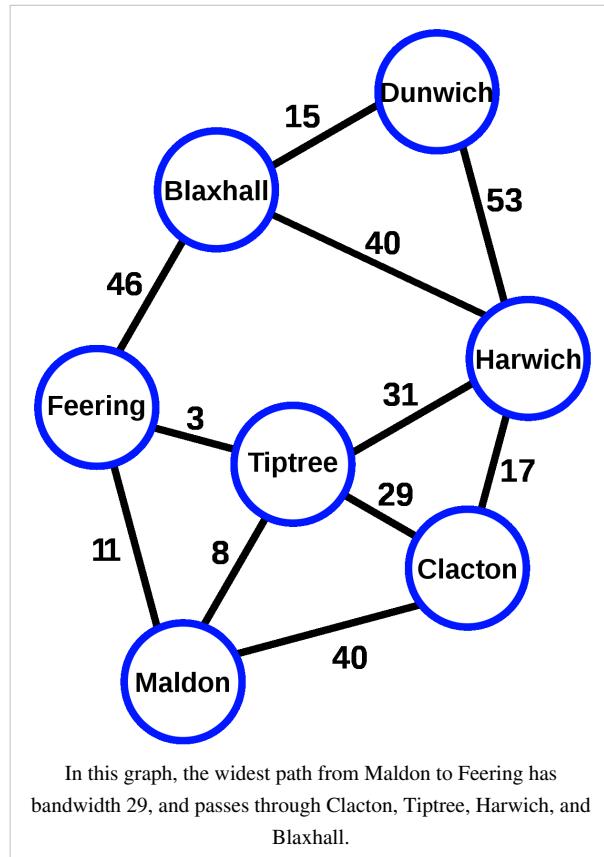
A closely related problem, the **minimax path problem**, asks for the path that minimizes the maximum weight of any of its edges. It has applications that include transportation planning.^[7] Any algorithm for the widest path problem can be transformed into an algorithm for the minimax path problem, 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.^{[8][9][10]}

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.^{[9][11][12]}

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.^[3]

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.^[13] 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.^[14]

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.^[2] 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.^[15] The Schulze method has been used by several organizations including the Wikimedia Foundation.^[16]

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 $O(n^{2.688})$.^[17] Instead, the reference implementation for the Schulze method uses a modified version of the simpler Floyd–Warshall algorithm, which takes $O(n^3)$ time.^[2] 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.^[12]

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.^[7] 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.^[4]

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 $O(mn \log U)$.^[5]

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.^{[12][18]} 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 $O(m \log^* n)$.^[18] 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.^[19]

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.^[20]

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 ?^[21]

References

- [1] Shacham, N. (1992), "Multicast routing of hierarchical data", *IEEE International Conference on Communications (ICC '92)*, **3**, pp. 1217–1221, doi:10.1109/ICC.1992.268047; Wang, Zheng; Crowcroft, J. (1995), "Bandwidth-delay based routing algorithms", *IEEE Global Telecommunications Conference (GLOBECOM '95)*, **3**, pp. 2129–2133, doi:10.1109/GLOCOM.1995.502780
- [2] Schulze, Markus (2011), "A new monotonic, clone-independent, reversal symmetric, and Condorcet-consistent single-winner election method", *Social Choice and Welfare* **36** (2): 267–303, doi:10.1007/s00355-010-0475-4
- [3] Fernandez, Elena; Garfinkel, Robert; Arbiol, Roman (1998), "Mosaicking of aerial photographic maps via seams defined by bottleneck shortest paths", *Operations Research* **46** (3): 293–304, JSTOR 222823
- [4] Ullah, E.; Lee, Kyongbum; Hassoun, S. (2009), "An algorithm for identifying dominant-edge metabolic pathways" (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5361299), *IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2009)*, pp. 144–150,
- [5] Ahuja, Ravindra K.; Magnanti, Thomas L.; Orlin, James B. (1993), "7.3 Capacity Scaling Algorithm", *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, pp. 210–212, ISBN 0-13-617549-X
- [6] Pollack, Maurice (1960), "The maximum capacity through a network", *Operations Research* **8** (5): 733–736, JSTOR 167387
- [7] Berman, Oded; Handler, Gabriel Y. (1987), "Optimal Minimax Path of a Single Service Unit on a Network to Nonservice Destinations", *Transportation Science* **21** (2): 115–122, doi:10.1287/trsc.21.2.115
- [8] Hu, T. C. (1961), "The maximum capacity route problem", *Operations Research* **9** (6): 898–900, doi:10.1287/opre.9.6.898, JSTOR 167055
- [9] Punnen, Abraham P. (1991), "A linear time algorithm for the maximum capacity path problem", *European Journal of Operational Research* **53** (3): 402–404, doi:10.1016/0377-2217(91)90073-5
- [10] Malpani, Navneet; Chen, Jianer (2002), "A note on practical construction of maximum bandwidth paths", *Information Processing Letters* **83** (3): 175–180, doi:10.1016/S0020-0190(01)00323-4, MR1904226
- [11] Camerini, P. M. (1978), "The min-max spanning tree problem and some extensions", *Information Processing Letters* **7** (1): 10–14, doi:10.1016/0020-0190(78)90030-3
- [12] Kaibel, Volker; Peinhardt, Matthias A. F. (2006), *On the bottleneck shortest path problem* (<http://www.zib.de/Publications/Reports/ZR-06-22.pdf>), ZIB-Report 06-22, Konrad-Zuse-Zentrum für Informationstechnik Berlin,
- [13] Leclerc, Bruno (1981), "Description combinatoire des ultramétriques" (in French), *Centre de Mathématique Sociale. École Pratique des Hautes Études. Mathématiques et Sciences Humaines* (73): 5–37, 127, MR623034
- [14] Demaine, Erik D.; Landau, Gad M.; Weimann, Oren (2009), "On Cartesian trees and range minimum queries", *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5–12, 2009, Lecture Notes in Computer Science*, **5555**, pp. 341–353, doi:10.1007/978-3-642-02927-1_29
- [15] 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.
- [16] 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.
- [17] Duan, Ran; Pettie, Seth (2009), "Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths" (<http://portal.acm.org/citation.cfm?id=1496813>), *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*, pp. 384–391, . For an earlier algorithm that also used fast matrix multiplication to speed up all pairs widest paths, see Vassilevska, Virginia; Williams, Ryan; Yuster, Raphael (2007), "All-pairs bottleneck paths for general graphs in truly sub-cubic time", *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC '07)*, New York: ACM, pp. 585–589, doi:10.1145/1250790.1250876, MR2402484 and Chapter 5 of Vassilevska, Virginia (2008), *Efficient Algorithms for Path Problems in Weighted Graphs* (<http://www.cs.cmu.edu/afs/cs/Web/People/virgi/thesis.pdf>), Ph.D. thesis, Report CMU-CS-08-147, Carnegie Mellon University School of Computer Science,
- [18] Gabow, Harold N.; Tarjan, Robert E. (1988), "Algorithms for two bottleneck optimization problems", *Journal of Algorithms* **9** (3): 411–417, doi:10.1016/0196-6774(88)90031-4, MR955149
- [19] Han, Yijie; Thorup, M. (2002), "Integer sorting in $O(n/\log \log n)$ expected time and linear space", *Proc. 43rd Annual Symposium on Foundations of Computer Science (FOCS 2002)*, pp. 135–144, doi:10.1109/SFCS.2002.1181890.

- [20] Bose, Prosenjit; Maheshwari, Anil; Narasimhan, Giri; Smid, Michiel; Zeh, Norbert (2004), "Approximating geometric bottleneck shortest paths", *Computational Geometry. Theory and Applications* **29** (3): 233–249, doi:10.1016/j.comgeo.2004.04.003, MR2095376
- [21] Gethner, Ellen; Wagon, Stan; Wick, Brian (1998), "A stroll through the Gaussian primes", *American Mathematical Monthly* **105** (4): 327–337, doi:10.2307/2589708, MR1614871.

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 adjacent 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 E , 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 adjacent 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 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.^[2]

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?^[3]

Notes

[1] Papadimitriou and Yannakakis, 1982, p. 148

[2] Amy J. Briggs; "Carrick Detweiler, Daniel Scharstein (2004). "Expected shortest paths for landmark-based robot navigation". *"International Journal of Robotics Research"* **23** (7–8): 717–718. doi:10.1177/0278364904045467.

[3] Karger and Nikolova, 2008, p. 1

References

- C.H. Papadimitriou; M. Yannakakis (1989). "Shortest paths without a map". *Lecture notes in computer science.* **372**. Proc. 16th ICALP. 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*.

Application: Centrality analysis of social networks

Within the scope of graph theory and network analysis, there are various types of measures of the **centrality** of a vertex within a graph that determine the relative importance of a vertex within the graph (i.e. how influential a person is within a social network, or, in the theory of space syntax, how important a room is within a building or how well-used a road is within an urban network). Many of the centrality concepts were first developed in social network analysis, and many of the terms used to measure centrality reflect their sociological origin.^[1]

There are four measures of centrality that are widely used in network analysis: degree centrality, betweenness, closeness, and eigenvector centrality. For a review as well as generalizations to weighted networks, see Opsahl et al. (2010).^[2]

Degree centrality

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 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.^[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 fast it will take to spread information from s to all other nodes sequentially.^[4]

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.^[5]

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 length of paths ending at a vertex \mathbf{i} , which is smaller if \mathbf{i} has many short paths connecting it to other vertices.^[6]

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 calculated more easily and can be also applied to graphs which lack connectivity^[7]:

$$C_C(v) = \sum_{t \in V \setminus v} 2^{-d_G(v,t)}.$$

Another extension to networks with disconnected components has been proposed by Opsahl (2010).^[8]

Betweenness centrality

Betweenness is a centrality measure of a vertex within a graph (there is also edge betweenness, which is not discussed here). 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.^[9] In his conception, vertices that have a high probability to occur on a randomly chosen shortest path between two randomly chosen nodes have a high betweenness.

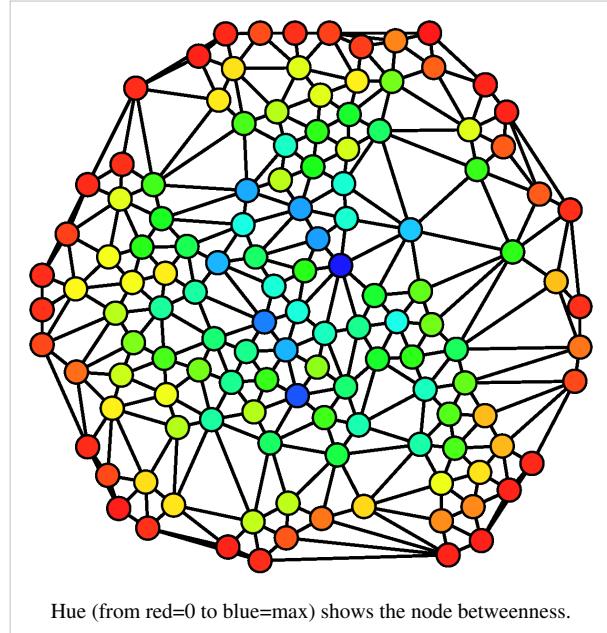
The betweenness of a vertex v in a graph $G := (V, E)$ with V vertices is computed as follows:

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^[10]:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

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 unweighted graph, a center vertex (which is contained in every possible shortest path) would have a betweenness of $(n - 1)(n - 2)/2$ (1, if normalized), 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^[10] 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, oftentimes 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.^[10]

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.^[11] 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.^[12] 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.^[11] 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

Katz centrality^[13] 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 contribution of a distant node is penalized by an attenuation factor $\alpha \in (0, 1)$. Mathematically, it is defined as

$$x_i = \sum_{k=1}^{\infty} \sum_{j=1}^N \alpha^k (A^k)_{ij}.$$

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^[14] 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).^[15]

Definition and characterization of centrality indices

Next to the above named classic centrality indices, there are dozens of other more specialized centrality indices. Despite its intuitive notion there is not yet a definition or characterization of centrality indices which captures all of them.^[16] A very loose definition of a centrality index is the following:

A centrality index is a real-valued function on the nodes of a graph. It is a structural index, i.e., if G and H are two isomorphic graphs and Φ is the mapping from the vertex set $V(G)$ of G to $V(H)$, then the centrality of a vertex v of G must be the same as the centrality of $\Phi(v)$ in H . Conventionally, the higher the centrality index of a node, the higher its perceived centrality in the graph.^[17] This definition comprises all classic centrality measures but not all measures that fulfill this definition would be accepted as centrality indices.

Borgatti and Everett summarize that centrality indices measure the position of a node along a predefined set of walks. They characterize centrality indices along four dimensions: the set of walks, whether the length or the number of these walks is considered, the position of the node on the walks (at the start=radial; in the middle=medial), and how the numbers assigned to the paths are summarized in the measure (average, median, weighted sum, ...).^[16] This leads to a characterization by the way a centrality index is calculated. In a different characterization, Borgatti differentiates the centrality indices by what type of paths they consider and which type of network flow they imply.^[18] The latter characterizes the centrality indices by the quality with which they predict which node is most central for a given network flow process. This characterization thus provides guidance on when to use which centrality index.

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.^[19] The general definition of centralization for non-weighted networks was proposed by Linton Freeman (1979). 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 degree.^[19] 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 $\max \sum_{j=1}^N C_x(p_*) - C_x(p_i)$ 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:^[19]

$$C_x = \frac{\sum_{j=1}^N C_x(p_*) - C_x(p_i)}{\max \sum_{j=1}^N C_x(p_*) - C_x(p_i)}$$

Notes and references

- [1] Newman, M.E.J. 2010. *Networks: An Introduction*. Oxford, UK: Oxford University Press.
- [2] Opsahl, Tore; Agneessens, Filip; Skvoretz, John (2010). "Node centrality in weighted networks: Generalizing degree and shortest paths" (<http://toreopsahl.com/2010/04/21/article-node-centrality-in-weighted-networks-generalizing-degree-and-shortest-paths/>). *Social Networks* **32** (3): 245. doi:10.1016/j.socnet.2010.03.006.
- [3] Sabidussi, G. (1966) The centrality index of a graph. *Psychometrika* **31**, 581–603.
- [4] M.E.J. Newman (2005), *A measure of betweenness centrality based on random walks*, **27**, pp. 39-54, arXiv:cond-mat/0309045, doi:10.1016/j.socnet.2004.11.009 Papercore summary (<http://papercore.org/Newman2005>).
- [5] J. D. Noh and H. Rieger, Phys. Rev. Lett. 92, 118701 (2004).
- [6] Stephenson, K. A. and Zelen, M., 1989. Rethinking centrality: Methods and examples. *Social Networks* **11**, 1–37.
- [7] Dangalchev Ch., Residual Closeness in Networks, *Phisica A* **365**, 556 (2006).
- [8] Tore Opsahl. *Closeness centrality in networks with disconnected components* (<http://toreopsahl.com/2010/03/20/closeness-centrality-in-networks-with-disconnected-components/>). .
- [9] Freeman, Linton (1977). "A set of measures of centrality based upon betweenness". *Sociometry* **40**: 35–41.
- [10] Brandes, Ulrik (2001). "A faster algorithm for betweenness centrality" (<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.2024>) (PDF). *Journal of Mathematical Sociology* **25**: 163–177. . Retrieved 10.11.2011. Papercore summary (<http://papercore.org/Brandes2001>)
- [11] <http://www.ams.org/samplings/feature-column/fcarc-pagerank>
- [12] M. E. J. Newman (PDF). *The mathematics of networks* (<http://www-personal.umich.edu/~mejn/papers/palgrave.pdf>). . Retrieved 2006-11-09.
- [13] Katz, L. 1953. A New Status Index Derived from Sociometric Index. *Psychometrika*, 39-43.
- [14] Bonacich, P., 1991. Simultaneous group and individual centralities. *Social Networks* **13**, 155–168.
- [15] How does Google rank webpages? (http://scenic.princeton.edu/network20q/lectures/Q3_notes.pdf) 20Q: About Networked Life
- [16] Borgatti, Stephen P.; Everett, Martin G. (2005). "A Graph-Theoretic Perspective on Centrality". *Social Networks* (Elsevier) **28**: 466–484. doi:10.1016/j.socnet.2005.11.005.
- [17] Koschützki, Dirk; Katharina A. Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, Oliver Zlotowski (2005). "Centrality Indices". In Ulrik Brandes, Thomas Erlebach. *Network Analysis – Methodological Foundations*. LNCS **3418**. Springer Verlag, Heidelberg, Germany. pp. 16–60. ISBN 978-3-540-24979-5.
- [18] Stephen P. Borgatti (2005). "Centrality and Network Flow". *Social Networks* (Elsevier) **27**: 55–71.
- [19] Freeman, L. C. (1979). Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3), 215-239.

Further reading

- Freeman, L. C. (1979). Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3), 215-239.
- Sabidussi, G. (1966). The centrality index of a graph. *Psychometrika*, 31 (4), 581-603.
- Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry* **40**, 35-41.
- 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.
- Bonacich, P. (1987). Power and Centrality: A Family of Measures, *The American Journal of Sociology*, 92 (5), pp 1170–1182.

External links

- https://networkx.lanl.gov/trac/attachment/ticket/119/page_rank.py
- http://www.faculty.ucr.edu/~hanneman/nettext/C10_Centrality.html

Application: Schulze voting system

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.

Currently, the Schulze method is the most widespread Condorcet method (list). The Schulze method is used by several organizations including Wikimedia, Debian, Ubuntu, Gentoo, and Software in the Public Interest.

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.

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.

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="text"/>	Joe Smith
1	John Citizen
3	Jane Doe
<input type="text"/>	Fred Rubble
2	Mary Hill

Schulze Method

Let $d[V,W]$ 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 $i = 1, \dots, (n-1)$: $d[C(i), C(i+1)] \geq p$.

$p[A,B]$, 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 $p[D,E] \geq p[E,D]$ for every other candidate E.

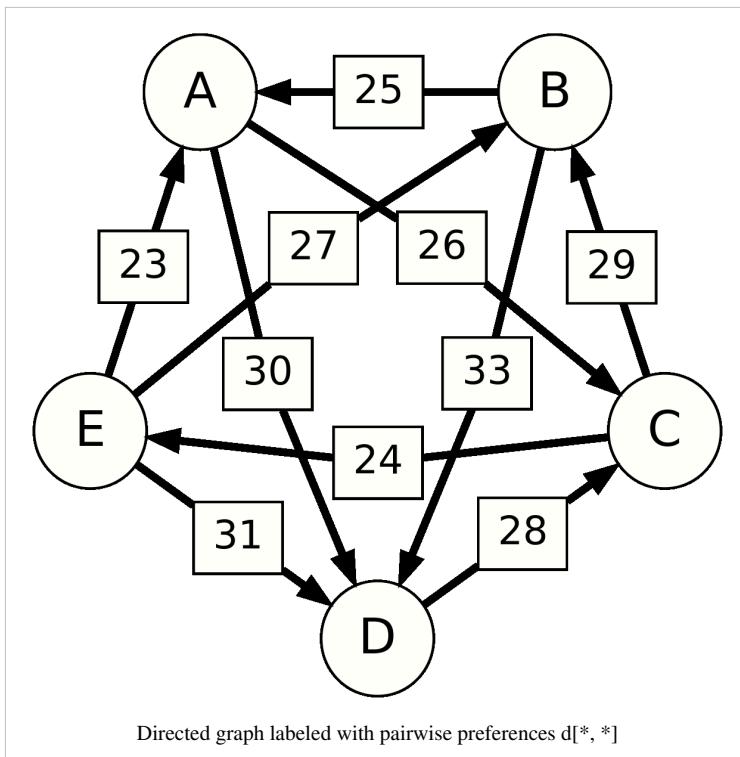
It can be proven that $p[X,Y] > p[Y,X]$ and $p[Y,Z] > p[Z,Y]$ together imply $p[X,Z] > p[Z,X]$.^{[1]:\\$4.1} Therefore, it is guaranteed (1) that the above definition of "better" really defines a transitive relation and (2) that there is always at least one candidate D with $p[D,E] \geq p[E,D]$ for every other candidate E.

Example

Consider the following example, in which 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

First, we compute the pairwise preferences. For example, in 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	

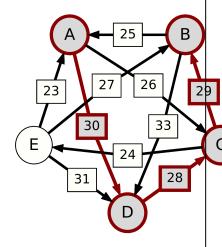
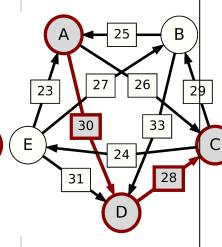
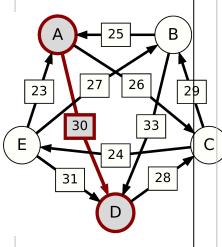
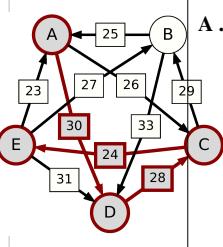
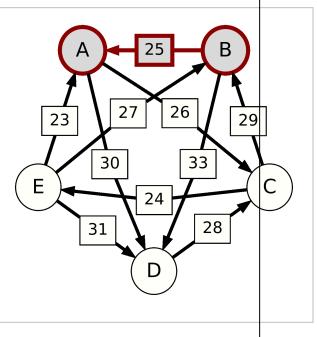
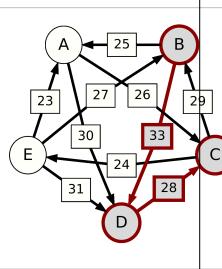
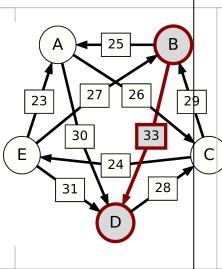
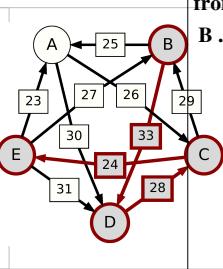
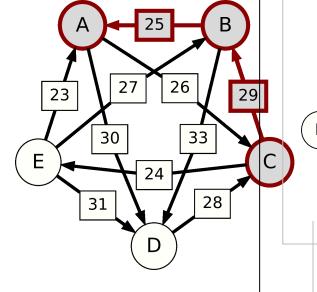
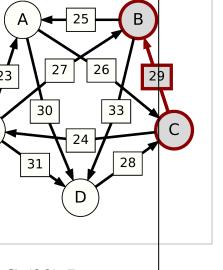
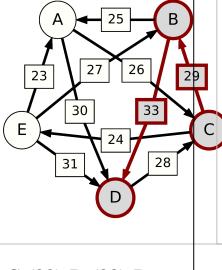
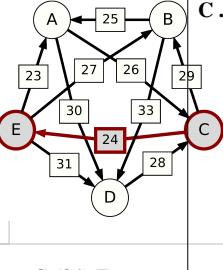
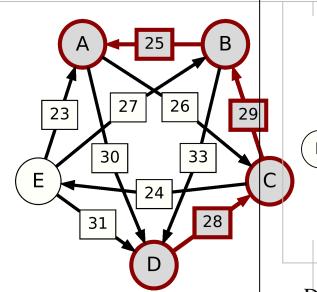
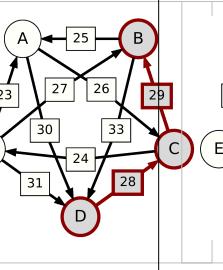
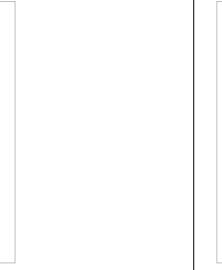
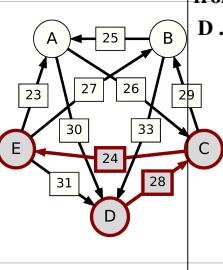
The cells for $d[X, Y]$ have a light green background iff $d[X, Y] > d[Y, X]$, otherwise the background is light red. Note, that there is no undisputed winner by only looking at the pairwise differences.

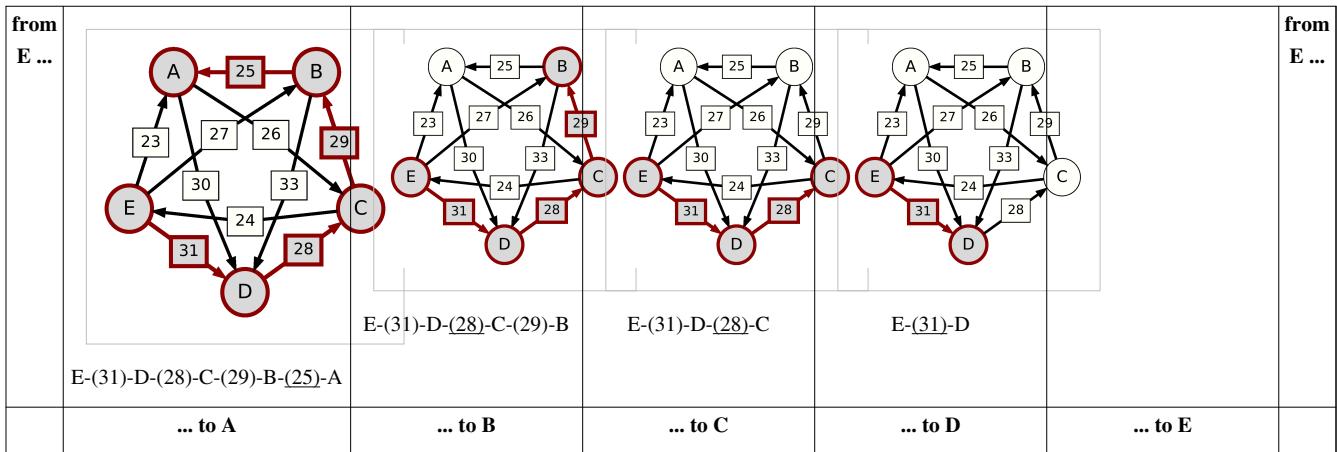
Now the strongest paths have to be identified. To help visualize the strongest paths, the set of pairwise preferences is depicted in the diagram on the right in the form of a directed graph. An arrow from the node representing a candidate X to the one representing a candidate Y is labelled with $d[X, Y]$. To avoid cluttering the diagram, we have only drawn an arrow 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. For contrast, let us also compute $p[A, C]$. 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 $\min(30, 28) = 28$. Recall that 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 ...	
from D ...					from D ...	
	A-(30)-D-(28)-C-(29)-B	A-(30)-D-(28)-C	A-(30)-D	A-(30)-D-(28)-C-(24)-E		
	B-(25)-A	B-(33)-D-(28)-C	B-(33)-D	B-(33)-D-(28)-C-(24)-E		
	C-(29)-B	C-(29)-B-(33)-D	C-(29)	C-(24)-E		
	D-(28)-C-(29)-B-(25)-A	D-(28)-C-(29)-B	D-(28)-C	D-(28)-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 we can determine the output of the Schulze method. Comparing A and B for example, 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 we get 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 we allow users to have ties in their preferences, the outcome of the Schulze method naturally depends on how we interpret these ties 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.^[1]

Although ties in the Schulze ranking are unlikely,^[2] they are possible. Schulze's original paper^[1] 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^{[1]:§4.3}
- Monotonicity criterion^{[1]:§4.5}
- Majority criterion
- Majority loser criterion
- Condorcet criterion
- Condorcet loser criterion
- Schwartz criterion
- Smith criterion^{[1]:§4.7}
- Independence of Smith-dominated alternatives^{[1]:§4.7}
- Mutual majority criterion
- Independence of clones^{[1]:§4.6}
- Reversal symmetry^{[1]:§4.4}
- Mono-append^[3]
- Mono-add-plump^[3]
- Resolvability criterion^{[1]:§4.2}
- Polynomial runtime^{[1]:§2.3"}
- prudence^{[1]:§4.9"}

- MinMax sets^{[1]:§4.8"}
 - Woodall's plurality criterion if winning votes are used for $d[X, Y]$
 - Symmetric-completion^[3] 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^{[1]:\$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:

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 \mathbf{X} of candidates is the strength of the strongest pairwise win of a candidate $A \notin \mathbf{X}$ against a candidate $B \in \mathbf{X}$. Then the Schulze method, but not Ranked Pairs, guarantees that the winner is always a candidate of the set with minimum MinMax score.^{[1]:§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 of the Schulze method

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]

Use of the Schulze method

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:

- Annodex Association^[18]
- Blitzed^{[19] [20]}
- BoardGameGeek^[21]
- Cassandra^[22]
- Codex Alpe Adria^{[23] [24]}
- Collective Agency^{[25] [26]}
- College of Marine Science^[27]
- Computer Science Departmental Society for York University (HackSoc)^[28]
- County Highpointers^{[29] [30]}
- Debian^[8]
- Demokratische Bildung Berlin^{[31] [32]}
- EuroBillTracker^[33]
- European Democratic Education Conference (EUDEC)^[34]
- Fair Trade Northwest^{[35] [36]}
- FFmpeg^[37]
- Flemish Student Society of Leuven^{[38] [39]}

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

- Free Geek [40]
- Free Hardware Foundation of Italy [41] [42]
- Free Software Foundation Europe (FSFE) [13]
- Gentoo Foundation [9]
- GNU Privacy Guard (GnuPG) [43]
- Gothenburg Hacker Space (GHS) [44] [45]
- Graduate Student Organization at the State University of New York: Computer Science (GSOCs) [46] [47]
- Haskell [48]
- Kanawha Valley Scrabble Club [49] [50]
- KDE e.V. [12]
- Kingman Hall [51]
- Knight Foundation [52]
- Kumoricon [53]
- League of Professional System Administrators (LOPSA) [54]
- Libre-Entreprise [55] [56]
- Lumiera/Cinelerra [57]
- Mathematical Knowledge Management Interest Group (MKM-IG) [58] [59]
- Metalab [60]
- Music Television (MTV) [61]
- Neo [62]
- Noisebridge [63]
- North Shore Cyclists (NSC) [64] [65]
- OpenEmbedded [66]
- OpenStack [67]
- Park Alumni Society (PAS) [68] [69]
- Pirate Party of Australia [70]
- Pirate Party of Austria [71]
- Pirate Party of Belgium [72]
- Pirate Party of Brazil
- Pirate Party of France [73]
- Pirate Party of Germany [15]
- Pirate Party of Italy [74]
- Pirate Party of New Zealand [75]
- Pirate Party of Sweden [14]
- Pirate Party of Switzerland [76]
- Pirate Party of the United States [77]
- Pitcher Plant of the Month [78]
- Pittsburgh Ultimate [79] [80]
- RLLMUK [81]
- RPMrepo [82] [83]
- Sender Policy Framework (SPF) [84]
- Software in the Public Interest (SPI) [7]
- Squeak [85]
- Students for Free Culture [86]
- Sugar Labs [87]
- Sverok [88]
- TestPAC [89]

- TopCoder^[10]
- Ubuntu^[90]
- University of British Columbia Math Club^{[91] [92]}
- Wikimedia Foundation^[11]
- Wikipedia in French,^[16] Hebrew,^[93] Hungarian,^[94] and Russian.^[95]

Notes

- [1] Markus Schulze, A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method (<http://www.springerlink.com/content/y5451n4908227162/?p=78c7a3edd3f64751ac9c2afc8aa6fad2&pi=0>), Social Choice and Welfare, volume 36, number 2, page 267–303, 2011. Preliminary version in *Voting Matters*, 17:9-19, 2003.
- [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%C3%B6lln/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] 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

[19] <http://blitzed.org/>

[20] Condorcet method for admin voting (http://wiki.blitzed.org/Condorcet_method_for_admin_voting), January 2005

[21] 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

[22] Project Logo (<http://article.gmane.org/gmane.comp.db.cassandra.devel/424/match=condorcet+schwartz+sequential+dropping+beatpath>), October 2009

[23] <http://0xAA.org>

[24] "Codex Alpe Adria Competitions" (<http://0xAA.org/competitions/>). 0xaa.org. 2010-01-24. . Retrieved 2010-05-08.

[25] <http://collectiveagency.co/>

[26] Civics Meeting Minutes (<http://collectiveagency.co/2012/03/21/civics-meeting-minutes-32012/>), March 2012

[27] "Fellowship Guidelines" (<http://www.marine.usf.edu/fellowships/Guidelines-and-Application-2011-2012.pdf>) (PDF). . Retrieved 2011-06-01.

[28] Report on HackSoc Elections (<http://www.hacksoc.org/HackSocElections.pdf>), December 2008

[29] <http://www.cohp.org/>

[30] Adam Helman, Family Affair Voting Scheme - Schulze Method (http://www.cohp.org/records/votes/family_affair_voting_scheme.html)

- [31] <http://www.demokratische-schule-x.de/>
 - [32] appendix 1 of the constitution (http://www.demokratische-schule-x.de/media/DBB_Satzung_Stand_August_2010.pdf)
 - [33] 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
 - [34] "Guidance Document" (<http://www.eudec.org/Guidance Document>). Eudec.org. 2009-11-15. . Retrieved 2010-05-08.
 - [35] <http://fairtradenorthwest.org/>
 - [36] article XI section 2 of the bylaws (http://fairtradenorthwest.org/FTNW_Bylaws.pdf)
 - [37] Democratic election of the server admins (<http://article.gmane.org/gmane.comp.video.ffmpeg-devel/113026/match=%22schulze method%22+%22Cloneproof schwartz sequential droping%22+Condorcet>), July 2010
 - [38] <http://www.vtk.be/>
 - [39] article 51 of the statutory rules (http://www.vtk.be/vtk/statuten/huishoudelijk_reglement.pdf)
 - [40] Voters Guide (http://wiki.freegeek.org/images/7/7a/Voters_guide.pdf), September 2011
 - [41] <http://fhh.it/>
 - [42] See:
 - Eletto il nuovo Consiglio nella Free Hardware Foundation (<http://fhh.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
 - [43] GnuPG Logo Vote (<http://logo-contest.gnupg.org/results.html>), November 2006
 - [44] <http://gbg.hackerspace.se/site/>
 - [45] §14 of the bylaws (<http://gbg.hackerspace.se/site/om-ghs/stadgar/>)
 - [46] http://gso.cs.binghamton.edu/index.php/GSOCS_Home
 - [47] "User Voting Instructions" (<http://gso.cs.binghamton.edu/index.php/Voting>). Gso.cs.binghamton.edu. . Retrieved 2010-05-08.
 - [48] 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
 - [49] <http://www.wvscrabble.com/>
 - [50] A club by any other name ... (<http://wvscrabble.blogspot.com/2009/04/club-by-any-other-name.html>), April 2009
 - [51] 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
 - [52] 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
 - [53] 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
 - [54] article 8.3 of the bylaws (http://governance.lopsa.org/LOPSA_Bylaws)
 - [55] <http://www.libre-entreprise.org/>
 - [56] 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
 - [57] Lumiera Logo Contest (http://www.cs.cornell.edu/w8/~andru/cgi-perl/civs/results.pl?id=E_7df51370797b45d6), January 2009
 - [58] <http://www.mkm-ig.org/>
 - [59] 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 (<http://www.mkm-ig.org/charter.html>)
 - Michael Kohlhase, MKM-IG Trustees Election Details & Ballot (<http://lists.jacobs-university.de/pipermail/projects-mkm-ig/2004-November/000041.html>), November 2004

[95] 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

General

- Schulze method website (<http://m-schulze.webhop.net/>) by Markus Schulze

Tutorials

- Schulze-Methode (<http://blog.cgiesel.de/a/schulze-methode>) (German) by Christoph Giesel
- 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
- Schulze-Methode (http://m-schulze.webhop.net/serie3_9-10.pdf) (German) by the University of Stuttgart

Advocacy

- Descriptions of ranked-ballot voting methods (<http://www.cs.wustl.edu/~legrand/rbvote/desc.html>) by Rob LeGrand
- Accurate Democracy (http://accuratedemocracy.com/voting_rules.htm) by Rob Loring
- Election Methods and Criteria (<http://nodesiege.tripod.com/elections/>) by Kevin Venzke
- The Debian Voting System (<http://www.seehuhn.de/pages/vote>) by Jochen Voss
- election-methods: a mailing list containing technical discussions about election methods (<http://lists.electorama.com/pipermail/election-methods-electorama.com/>)

Books

- 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
- Saul Stahl and Paul E. Johnson (2006), *Understanding Modern Mathematics* (<http://books.google.com/books?id=CMLL9sVGLb8C&pg=PA119>), Sudbury: Jones and Bartlett Publishers, ISBN 0-7637-3401-2
- 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

Software

- Modern Ballots (<https://modernballots.com/>) and Python Vote Core (<https://github.com/bradbeattie/python-vote-core>) by Brad Beattie
- Voting Software Project (<http://vote.sourceforge.net/>) by Blake Cretney
- Condorcet with Dual Dropping Perl Scripts (<http://condorcet-dd.sourceforge.net/>) by Mathew Goldstein
- Condorcet Voting Calculator (<http://condorcet.ericgorr.net/>) by Eric Gorr
- Selectricity (<http://selectricity.org/>) and RubyVote (<http://rubylvote.rubyforge.org/>) by Benjamin Mako Hill (<http://web.mit.edu/newsoffice/2008/voting-tt0312.html>) (<http://labcast.media.mit.edu/?p=56>)
- Schulze voting for DokuWiki (<http://www.cosmocode.de/en/blog/lang/2010-07/05-schulze-voting-for-dokuwiki>) by Adrian Lang
- Electowidget (<http://wiki.electorama.com/wiki/Electowidget>) by Rob Lanphier

- Online ranked-ballot voting calculator (<http://www.cs.wustl.edu/~legrand/rbvote/index.html>) by Rob LeGrand
- Condorcet Internet Voting Service (CIVS) (<http://www.cs.cornell.edu/andru/civs.html>) by Andrew Myers
- BetterPolls.com (<http://betterpolls.com/>) by Brian Olson
- OpenSTV (<http://www.openstv.org/>) by Jeffrey O'Neill
- LiquidFeedback
- preftools (<http://www.public-software-group.org/preftools>) by the Public Software Group
- Voting Excel Template and Add-In (<http://geekspeakdecoded.com/2011/03/07/voting-excel-template-and-add-in-borda-counting-and-schulze-method/>)

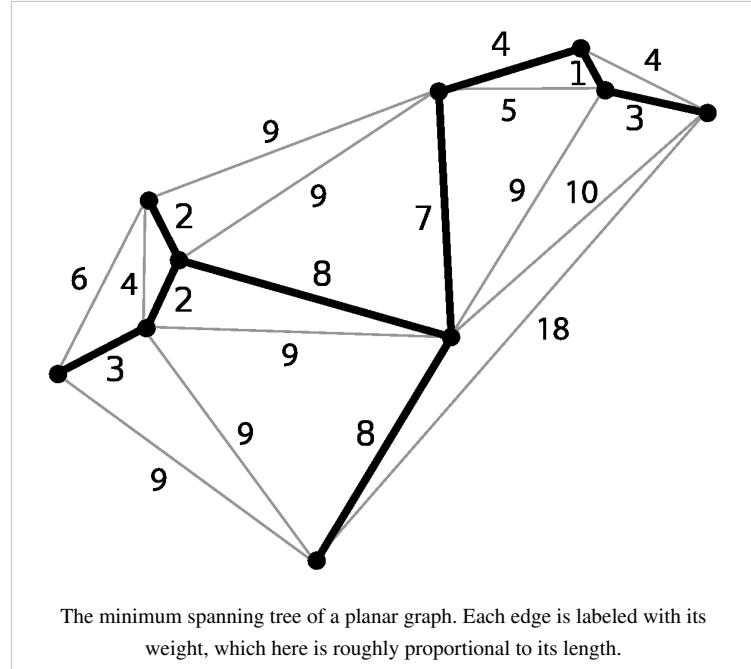
Legislative projects

- Arizonans for Condorcet Ranked Voting (<http://www.azsos.gov/election/2008/general/ballotmeasuretext/I-21-2008.pdf>) ([http://ballotpedia.org/wiki/index.php?title=Arizona_Competitive_Elections_Reform_Act_\(2008\)](http://ballotpedia.org/wiki/index.php?title=Arizona_Competitive_Elections_Reform_Act_(2008))) (<http://www.azcentral.com/members/Blog/PoliticalInsider/22368>) (<http://www.ballot-access.org/2008/04/29/arizona-high-school-student-files-paperwork-for-initiatives-for-irv-and-easier-ballot-access/>)

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.



One example would be a cable TV company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, 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. 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.

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 can be proved by induction or contradiction. This is true in many realistic situations, such as the cable TV 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.^[2]

1. Say we have an algorithm that finds an MST (which we will call **A**) based on the structure of the graph and the order of the edges when ordered by weight. (Such algorithms do exist, see below.)
2. Assume MST **A** is not unique.
3. There is another spanning tree with equal weight, say MST **B**.
4. Let **e1** be an edge that is in **A** but not in **B**.
5. As **B** is a MST, $\{e1\} \cup B$ must contain a cycle **C**.
6. Then **B** should include at least one edge **e2** that is not in **A** and lies on **C**.
7. Assume the weight of **e1** is less than that of **e2**.
8. Replace **e2** with **e1** in **B** yields the spanning tree $\{e1\} \cup B - \{e2\}$ which has a smaller weight compared to **B**.
9. Contradiction. As we assumed **B** is a MST but it is not.

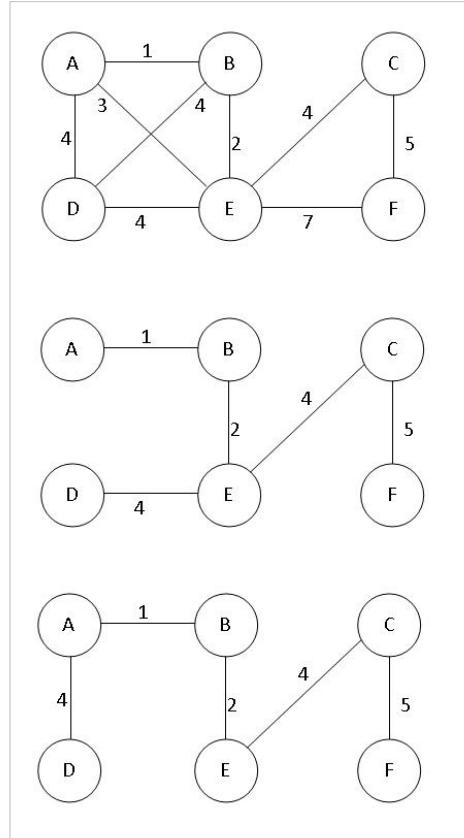
If the weight of **e1** is larger than that of **e2**, a similar argument involving tree $\{e2\} \cup A - \{e1\}$ also leads to a contradiction. Thus, we conclude that the assumption that there can be a second MST was false.

Minimum-cost subgraph

If the weights are *positive*, then a minimum spanning tree is in fact the minimum-cost subgraph connecting all vertices, since subgraphs containing cycles necessarily have more total weight.

Cycle property

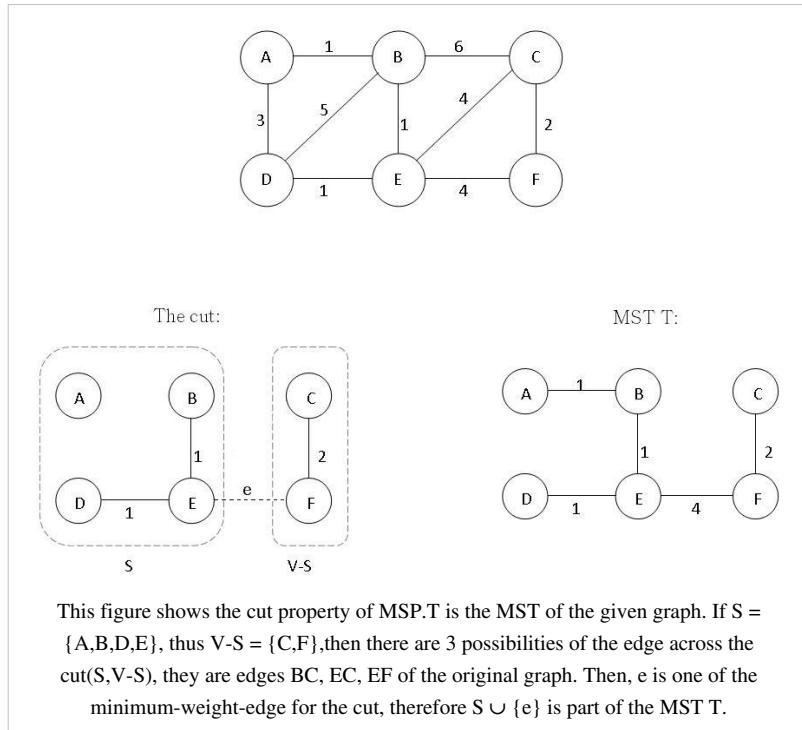
For any cycle C in the graph, if the weight of an edge e of C is larger than the weights of other edges of C, then this edge cannot belong to an MST. Assuming the contrary, i.e. that e belongs to an MST T1, then deleting e will break T1 into two subtrees with the two ends of e in different subtrees. 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 T2 with weight less than that of T1, 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 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 left, 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.



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.^[3] 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.^{[4][5]} 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.^{[6][7]} Its running time is $O(m \alpha(m,n))$, where m is the number of edges, n is the number of vertices and 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.^[8]

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.^{[9][10]} Bader & Cong (2003) demonstrate an algorithm that can compute MSTs 5 times faster on 8 processors than an optimized sequential algorithm.^[11] Later, Nobari *et al.* in^[12] propose a novel, scalable, parallel Minimum Spanning Forest (MSF) algorithm for undirected

weighted graphs. This algorithm leverages Prim's algorithm in a parallel fashion, concurrently expanding several subsets of the computed MSF. PMA minimizes the communication among different processors by not constraining the local growth of a processor's computed subtree. In effect, PMA achieves a scalability that previous approaches lacked. PMA, in practice, outperforms the previous state-of-the-art GPU-based MSF algorithm, while being several order of magnitude faster than sequential CPU-based algorithms.

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.,^[13] 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. Under the additional assumption of finite variance, Frieze also proved convergence in probability. Subsequently, J. Michael Steele showed that the variance assumption could be dropped.

In later work, 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.^[14]

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

Related problems

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.^{[15][16][17]} (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. Mathematical definition of the problem is the same but has different approaches for 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 requires exponential time, 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.^[18] Maximum spanning trees find applications in parsing algorithms for natural languages^[19] 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.^{[20][21]}

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.

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] Gallager, R. G.; Humblet, P. A.; Spira, P. M. (January 1983), "A distributed algorithm for minimum-weight spanning trees", *ACM Transactions on Programming Languages and Systems* **5** (1): 66–77, doi:10.1145/357195.357200.
- [3] Fredman, M. L.; Willard, D. E. (1994), "Trans-dichotomous algorithms for minimum spanning trees and shortest paths", *Journal of Computer and System Sciences* **48** (3): 533–551, doi:10.1016/S0022-0000(05)80064-9, MR1279413.
- [4] Karger, David R.; Klein, Philip N.; Tarjan, Robert E. (1995), "A randomized linear-time algorithm to find minimum spanning trees", *Journal of the Association for Computing Machinery* **42** (2): 321–328, doi:10.1145/201019.201022, MR1409738.
- [5] Pettie, Seth; Ramachandran, Vijaya (2002), "Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms" (<http://portal.acm.org/citation.cfm?id=545477>), *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, San Francisco, California, pp. 713–722, .
- [6] Chazelle, Bernard (2000), "A minimum spanning tree algorithm with inverse-Ackermann type complexity", *Journal of the Association for Computing Machinery* **47** (6): 1028–1047, doi:10.1145/355541.355562, MR1866456.
- [7] Chazelle, Bernard (2000), "The soft heap: an approximate priority queue with optimal error rate", *Journal of the Association for Computing Machinery* **47** (6): 1012–1027, doi:10.1145/355541.355554, MR1866455.
- [8] Pettie, Seth; Ramachandran, Vijaya (2002), "An optimal minimum spanning tree algorithm", *Journal of the Association for Computing Machinery* **49** (1): 16–34, doi:10.1145/505241.505243, MR2148431.
- [9] Chong, Ka Wong; Han, Yijie; Lam, Tak Wah (2001), "Concurrent threads and optimal parallel minimum spanning trees algorithm", *Journal of the Association for Computing Machinery* **48** (2): 297–323, doi:10.1145/375827.375847, MR1868718.
- [10] Pettie, Seth; Ramachandran, Vijaya (2002), "A randomized time-work optimal parallel algorithm for finding a minimum spanning forest", *SIAM Journal on Computing* **31** (6): 1879–1895, doi:10.1137/S0097539700371065, MR1954882.

- [11] Bader, David A.; Cong, Guojing (2006), "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs", *Journal of Parallel and Distributed Computing* **66** (11): 1366–1378, doi:10.1016/j.jpdc.2006.06.001.
- [12] Nobari, Sadegh; Cao, Thanh-Tung; Karras, Panagiotis; Bressan, Stéphane (2012), "Scalable parallel minimum spanning forest computation" (<http://www.comp.nus.edu.sg/~snobari/PMA/>), *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA: ACM) (11): 205–214, doi:10.1145/2145816.2145842, ISBN 978-1-4503-1160-1, .
- [13] Dementiev, Roman; Sanders, Peter; Schultes, Dominik; Sibeyn, Jop F. (2004), "Engineering an external memory minimum spanning tree algorithm" (<http://algo2.iti.kit.edu/dementiev/files/emst.pdf>), *Proc. IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004)*, pp. 195–208, .
- [14] Steele, J. Michael (2002), "Minimal spanning trees for graphs with random edge lengths", *Mathematics and computer science, II (Versailles, 2002)*, Trends Math., Basel: Birkhäuser, pp. 223–245, MR1940139
- [15] Gabow, Harold N. (1977), "Two algorithms for generating weighted spanning trees in order", *SIAM Journal on Computing* **6** (1): 139–150, MR0441784.
- [16] Eppstein, David (1992), "Finding the k smallest spanning trees", *BIT* **32** (2): 237–248, doi:10.1007/BF01994879, MR1172188.
- [17] Frederickson, Greg N. (1997), "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees", *SIAM Journal on Computing* **26** (2): 484–538, doi:10.1137/S0097539792226825, MR1438526.
- [18] Hu, T. C. (1961), "The maximum capacity route problem", *Operations Research* **9** (6): 898–900, doi:10.1287/opre.9.6.898, JSTOR 167055.
- [19] McDonald, Ryan; Pereira, Fernando; Ribarov, Kiril; Hajic, Jan (2005). "Non-projective dependency parsing using spanning tree algorithms" (<http://www.seas.upenn.edu/~strctlrn/bib/PDF/nonprojectiveHLT-EMNLP2005.pdf>). *Proc. HLT/EMNLP*. .
- [20] Spira, P. M.; Pan, A. (1975), "On finding and updating spanning trees and shortest paths", *SIAM Journal on Computing* **4** (3): 375–380, MR0378466.
- [21] Holm, Jacob; de Lichtenberg, Kristian; Thorup, Mikkel (2001), "Poly-logarithmic deterministic fully dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity", *Journal of the Association for Computing Machinery* **48** (4): 723–760, doi:10.1145/502090.502095, MR2144928.

Additional reading

- Graham, R. L.; Hell, Pavol (1985), "On the history of the minimum spanning tree problem", *Annals of the History of Computing* **7** (1): 43–57, doi:10.1109/MAHC.1985.10011, MR783327.
- Otakar Boruvka on Minimum Spanning Tree Problem (translation of the both 1926 papers, comments, history) (2000) (<http://citeseer.ist.psu.edu/nesetril00otakar.html>) Jaroslav Nešetřil, Eva Milková, Helena Nešetřilová. (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/index.html#ms97>). 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

- Jeff Erickson's MST lecture notes (<http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/12-mst.pdf>)
- 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

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.^{[1][2][3]} The algorithm was rediscovered by Choquet in 1938;^[4] again by Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki;^[5] in 1951; and again by Sollin^[6] 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
1 Begin with an empty set of edges  $T$ 
2 While the vertices of  $G$  connected by  $T$  are disjoint:
3   Begin with an empty set of edges  $E$ 
4   For each component of  $T$ :
5     Begin with an empty set of edges  $S$ 
6     For each vertex  $v$  in the component:
7       Add the cheapest edge from  $v$  to a vertex in a different component of  $T$  to  $S$ 
8     Add the cheapest edge in  $S$  to  $E$ 
9   Add the resulting set of edges  $E$  to  $T$ .
10 The resulting set of edges  $T$  is the minimum spanning tree of  $G$ .
```

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.^[7]

Other algorithms for this problem include Prim's algorithm (actually discovered by Vojtěch Jarník) and Kruskal's algorithm. Faster 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] Borůvka, Otakar (1926). "O jistém problému minimálním (About a certain minimal problem)" (in Czech, German summary). *Práce mor. přírodověd. spol. v Brně III* 3: 37–58.
- [2] Borůvka, Otakar (1926). "Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí (Contribution to the solution of a problem of economical construction of electrical networks)" (in Czech). *Elektronický Obzor* 15: 153–154.
- [3] Nešetřil, Jaroslav; Milková, Eva; Nešetřilová, Helena (2001). "Otakar Borůvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history". *Discrete Mathematics* 233 (1–3): 3–36. doi:10.1016/S0012-365X(00)00224-7. MR1825599.
- [4] Choquet, Gustave (1938). "Étude de certains réseaux de routes" (in French). *Comptes-rendus de l'Académie des Sciences* 206: 310–313.
- [5] Florek, Kazimierz (1951). "Sur la liaison et la division des points d'un ensemble fini" (in French). *Colloquium Mathematicum* 2 (1951): 282–285.
- [6] Sollin, M. (1965). "Le tracé de canalisation" (in French). *Programming, Games, and Transportation Networks*.
- [7] Eppstein, David (1999). "Spanning trees and spanners". In Sack, J.-R.; Urrutia, J.. *Handbook of Computational Geometry*. Elsevier. pp. 425–461; Mareš, Martin (2004). "Two linear time algorithms for MST on minor closed graph classes" (<http://www.emis.de/journals/AM/04-3/am1139.pdf>). *Archivum mathematicum* 40 (3): 315–320. .

Kruskal's algorithm

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
 - otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Performance

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 $O(\log V)$.
- If we ignore isolated vertices, which will each be their own component of the minimum spanning forest, $V \leq E+1$, 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, since the last edge added to that cycle would have 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

- Joseph. B. Kruskal: *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem* [2]. In: *Proceedings of the American Mathematical Society*, Vol 7, No. 1 (Feb, 1956), pp. 48–50
- 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

- Download the example minimum spanning tree data. [1]
- Animation of Kruskal's algorithm (Requires Java plugin) [3]
- C# Implementation [4]
- C++ Implementation [5]

References

- [1] <http://www.carlschroedl.com/blog/comp/kruskals-minimum-spanning-tree-algorithm/2012/05/14/>
- [2] [http://links.jstor.org/sici?siici=0002-9939\(195602\)7%3A1%3C48%3AOTSSO%3E2.0.CO%3B2-M](http://links.jstor.org/sici?siici=0002-9939(195602)7%3A1%3C48%3AOTSSO%3E2.0.CO%3B2-M)
- [3] <http://students.ceid.upatras.gr/~papagei/project/kruskal.htm>
- [4] http://www.codeproject.com/KB/recipes/Kruskal_Algorithm.aspx
- [5] <http://www.technical-recipes.com/2011/finding-minimal-spanning-trees-using-kruskals-algorithm/>

Prim's algorithm

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. However, these other algorithms can also find minimum spanning forests of disconnected graphs, while Prim's algorithm requires the graph to be connected.

Description

Informal

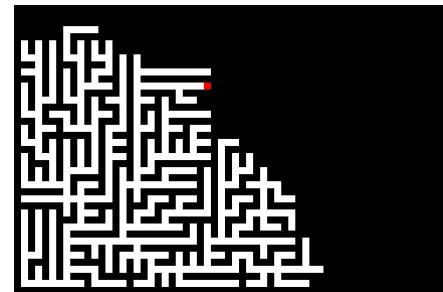
- create a tree containing a single vertex, chosen arbitrarily from the graph
- create a set containing all the edges in the graph
- loop until every edge in the set connects two vertices in the tree
 - remove from the set an edge with minimum weight that connects a vertex in the tree with a vertex not in the tree
 - add that edge to the tree

Technical

An empty graph cannot have a spanning tree, so we begin by assuming that the graph is non-empty.

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, 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



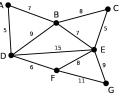
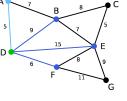
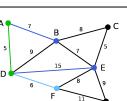
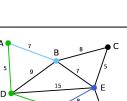
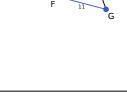
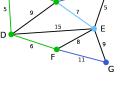
Prim's algorithm has many applications, such as in the generation of this maze, which applies Prim's algorithm to a randomly weighted grid graph.

Time complexity

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 using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can 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 $\Omega(V)$.

Example run

Image	U	Edge(u,v)	V \ U	Description
	{}		{A,B,C,D,E,F,G}	This is our original weighted graph. The numbers near the edges indicate their weight.
	{D}	(D,A) = 5 V (D,B) = 9 (D,E) = 15 (D,F) = 6	{A,B,C,E,F,G}	Vertex D has been arbitrarily chosen as a starting point. Vertices A , B , E and F are connected to D through a single edge. A is the vertex nearest to D and will be chosen as the second vertex along with the edge AD .
	{A,D}	(D,B) = 9 (D,E) = 15 (D,F) = 6 V (A,B) = 7	{B,C,E,F,G}	The next vertex chosen is the vertex nearest to <i>either</i> D or A . B is 9 away from D and 7 away from A , E is 15, and F is 6. F is the smallest distance away, so we highlight the vertex F and the edge DF .
	{A,D,F}	(D,B) = 9 (D,E) = 15 (A,B) = 7 V (F,E) = 8 (F,G) = 11	{B,C,E,G}	The algorithm carries on as above. Vertex B , which is 7 away from A , is highlighted.
	{A,B,D,F}	(B,C) = 8 (B,E) = 7 V (D,B) = 9 cycle (D,E) = 15 (F,E) = 8 (F,G) = 11	{C,E,G}	In this case, we can choose between C , E , and G . C is 8 away from B , E is 7 away from B , and G is 11 away from F . E is nearest, so we highlight the vertex E and the edge BE .
	{A,B,D,E,F}	(B,C) = 8 (D,B) = 9 cycle (D,E) = 15 cycle (E,C) = 5 V (E,G) = 9 (F,E) = 8 cycle (F,G) = 11	{C,G}	Here, the only vertices available are C and G . C is 5 away from E , and G is 9 away from E . C is chosen, so it is highlighted along with the edge EC .

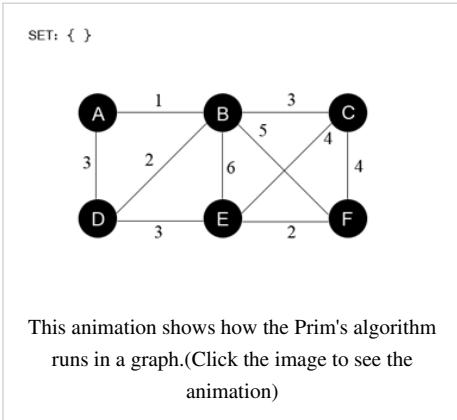
	{A,B,C,D,E,F}	(B,C) = 8 cycle (D,B) = 9 cycle (D,E) = 15 cycle (E,G) = 9 (F,E) = 8 cycle (F,G) = 11 cycle	{G}	Vertex G is the only remaining vertex. It is 11 away from F , and 9 away from E . E is nearer, so we highlight G and the edge EG .
	{A,B,C,D,E,F,G}	(B,C) = 8 cycle (D,B) = 9 cycle (D,E) = 15 cycle (F,E) = 8 cycle (F,G) = 11 cycle	{ }	Now all the vertices have been selected and the minimum spanning tree is shown in green. In this case, it has weight 39.

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 (we know we encountered the opportunity to take "f" before "e" because "f" is connected to V , and we visited every vertex of V before the vertex to which we connected "e" ["e" is connected to the last vertex we visited in V]). 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.



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

- Animated example of Prim's algorithm (<http://students.ceid.upatras.gr/~papagel/project/prim.htm>)
- Prim's Algorithm (Java Applet) (<http://www.mincel.com/java/prim.html>)
- 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/>)

Edmonds's algorithm for directed minimum spanning trees

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.

Order

The order of this algorithm is $O(EV)$. There is a faster implementation of the algorithm by Robert Tarjan. The order is $O(E \log V)$ for a sparse graph and $O(V^2)$ for a dense graph. This is as fast as Prim's algorithm for an undirected minimum spanning tree. In 1986, Gabow, Galil, Spencer, and Tarjan made a faster implementation, and its order is $O(E + V \log V)$.

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 $u \notin C$, include the edge e described below, in D' .

If $v \in C$, $e = (u, v_C)$, and $w(e) = w(u, v) - w(\pi(v), v)$. Otherwise, if $v \notin C$, let $e = (u, v)$, and $w(e) = w(u, v)$.

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 in D' . Suppose in this SRT, 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) . 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 ← BE ← ∅
i=0

A:
if BV = Vi then goto B
for some vertex v ∉ BV and v ∈ Vi {
    BV ← BV ∪ {v}
    find an edge e = (x, v) such that w(e) = max{ w(y, v) | (y, v) ∈ Ei }
    if w(e) ≤ 0 then goto A
}
if BE ∪ {e} contains a circuit {
    i=i+1
    construct Gi by shrinking Ci to ui
    modify BE, BV and some edge weights
}
BE ← BE ∪ e
goto A

B:
while i ≠ 0 {
    reconstruct Gi-1 and rename some edges in BE
    if ui was a root of an out-tree in BE {
        BE ← BE ∪ {e | e ∈ Ci and e ≠ e0i}
    }
}
```

```

}else{
    BE ← BE ∪ {e|e ∈ Ci and e ≠ ēi}
}
i=i-1
}
Maximum branching weight =  $\sum_{e \in BE} w(e)$ 

```

References

- Y. J. Chu and T. H. Liu, "On the Shortest Arborescence of a Directed Graph", *Science Sinica*, vol. 14, 1965, pp. 1396–1400.
- J. Edmonds, "Optimum Branchings", *J. Res. Nat. Bur. Standards*, vol. 71B, 1967, pp. 233–240.
- R. E. Tarjan, "Finding Optimum Branchings", *Networks*, v.7, 1977, pp. 25–35.
- P.M. Camerini, L. Fratta, and F. Maffioli, "A note on finding optimum branchings", *Networks*, v.9, 1979, pp. 309–312.
- Alan Gibbons *Algorithmic Graph Theory*, Cambridge University press, 1985 ISBN 0-521-28881-9
- H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica* 6 (1986), 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.
- AlgoWiki – Edmonds's algorithm [3] - A public-domain implementation of Edmonds's algorithm written in Java.

References

- [1] <http://www.ce.rit.edu/~sjyee/dmst.html>
[2] <http://edmonds-alg.sourceforge.net/>
[3] http://algowiki.net/wiki/index.php?title=Edmonds%27s_algorithm

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.

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.^[1]

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.^[2]

Complementarity

If d is the connected domination number of an n -vertex graph G , and l is its max leaf number, then the three quantities d , l , and n obey the simple equation

$$n = d + l. \text{^[3]}$$

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 .^[4] The maximum leaf spanning tree problem is MAX-SNP hard, implying that no polynomial time approximation scheme is likely.^[5] However, it can be approximated to within a factor of 2 in polynomial time.^[6]

Applications

Connected dominating set 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.^[7]

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.^[2]

References

- [1] Sampathkumar, E.; Walikar, HB (1979), "The connected domination number of a graph", *J. Math. Phys. Sci* **13** (6): 607–613.
- [2] Fellows, Michael; Lokshtanov, Daniel; Misra, Neeldhara; Mnich, Matthias; Rosamond, Frances; Saurabh, Saket (2009), "The complexity ecology of parameters: an illustration using bounded max leaf number", *Theory of Computing Systems* **45** (4): 822–848, doi:10.1007/s00224-009-9167-9.
- [3] Douglas, Robert J. (1992), "NP-completeness and degree restricted spanning trees", *Discrete Mathematics* **105** (1–3): 41–47, doi:10.1016/0012-365X(92)90130-8.
- [4] Guha, S.; Khuller, S. (1998), "Approximation algorithms for connected dominating sets", *Algorithmica* **20** (4): 374–387, doi:10.1007/PL00009201.
- [5] Galbiati, G.; Maffioli, F.; Morzenti, A. (1994), "A short note on the approximability of the maximum leaves spanning tree problem", *Information Processing Letters* **52** (1): 45–49, doi:10.1016/0020-0190(94)90139-2.
- [6] Solis-Oba, Roberto (1998), "2-approximation algorithm for finding a spanning tree with maximum number of leaves", *Proc. 6th European Symposium on Algorithms (ESA'98)*, Lecture Notes in Computer Science, **1461**, Springer-Verlag, pp. 441–452, doi:10.1007/3-540-68530-8_37.
- [7] Wu, J.; Li, H. (1999), "On calculating connected dominating set for efficient routing in ad hoc wireless networks", *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, ACM, pp. 7–14, doi:10.1145/313239.313261.

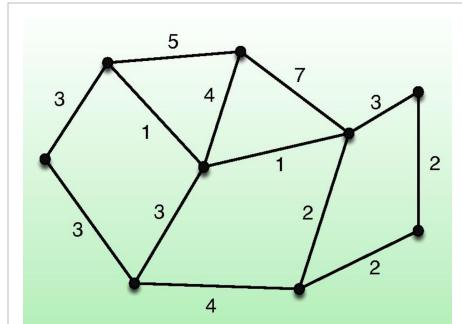
K-minimum spanning tree

In mathematics, given an undirected graph $G = (V, E)$ with non-negative edge costs and an integer k , the **k -minimum spanning tree**, or **k -MST**, of G is a tree of minimum cost that spans exactly k vertices of G . A k -MST does not have to be a subgraph of the minimum spanning tree (MST) of G . This problem is also known as **Edge-Weighted k -Cardinality Tree** (KCT).

The k -MST problem is shown to be NP-Hard by reducing the Steiner tree problem to the k -MST problem. There are many constant factor approximations for this problem. The current best approximation is a 2-approximation due to Garg. This approximation relies heavily on the primal-dual schema of Goemans and Williamson.

When the k -MST problem is restricted to the Euclidean plane, there exists a PTAS due to Arora.

Refer to KCTLIB^[1] for more information.



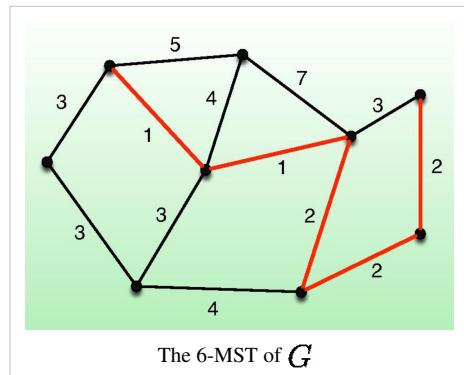
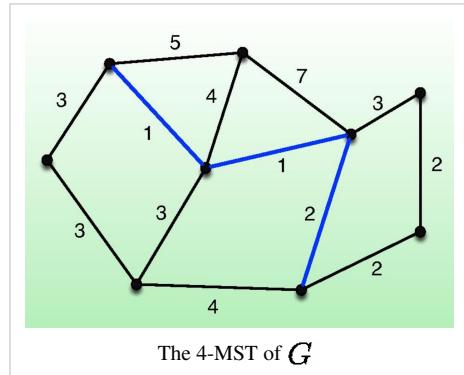
An example of an undirected graph G with edge costs

References

- Arora, S. (1998), "Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems.", *J. ACM*.
- Garg, N. (2005), "Saving an epsilon: a 2-approximation for the k-MST problem in graphs.", *STOC*.
- Ravi, R.; Sundaram, R.; Marathe, M.; Rosenkrantz, D.; Ravi, S. (1996), "Spanning trees short or small", *SIAM Journal on Discrete Mathematics*.
- Goemans, M.; Williamson, P. (1992), "A general approximation technique for constrained forest problems", *SIAM Journal on Computing*.

External links

- Minimum k-spanning tree in "A compendium of NP optimization problems" [2]



References

- [1] <http://iridia.ulb.ac.be/~cblum/kctlib/>
[2] <http://www.nada.kth.se/~viggo/wwwcompendium/node71.html>

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*. To find the optimal solution, one has to go through all the possible spanning tree configurations for a given graph and pick the one with the lowest cost; such search requires an exponential number of computations.

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^[1]

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  $a_i$ 
             $a_i = \text{closest node in a different subtree}$ 
             $t(a_i) = g_i - c_{ij}$ 
             $t_{\max} = \max(t(a_i))$ 
             $k = i \text{ such that } t(a_k) = t_{\max}$ 
            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 [2].

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

- [1] Esau, L.R.; Williams, K.C. (1966). "On teleprocessing network design: Part II. A method for approximating the optimal network.". *IBM Systems Journal* 5 (3): 142–147. doi:10.1147/sj.53.0142.
- [2] Sharma, R.L.; El-Bardai, M.T. (1977). "Suboptimal communications network synthesis". In *Proc. of International Conference on Communications*: 19.11–19.16.

Application: Single-linkage clustering

In cluster analysis, **single linkage**, **nearest neighbour** or **shortest distance** is a method of calculating distances between clusters in hierarchical clustering. In single linkage, the distance between two clusters is computed as the distance between the two closest elements in the two clusters.

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*: clusters may be forced together due to single elements being close to each other, even though many of the elements in each cluster may be very distant to each other.

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, ..., $(n - 1)$ 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.^[1]

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.

External links

- Single linkage clustering algorithm implementation in Ruby (AI4R)^[2]
- Linkages used in Matlab^[3]

References

- [1] R. Sibson (1973). "SLINK: an optimally efficient algorithm for the single-link cluster method" (http://www.cs.gsu.edu/~wkim/index_files/papers/sibson.pdf). *The Computer Journal* (British Computer Society) **16** (1): 30–34. .
 - [2] <http://ai4r.rubyforge.org>
 - [3] <http://www.mathworks.com/help/toolbox/stats/linkage.html>
-

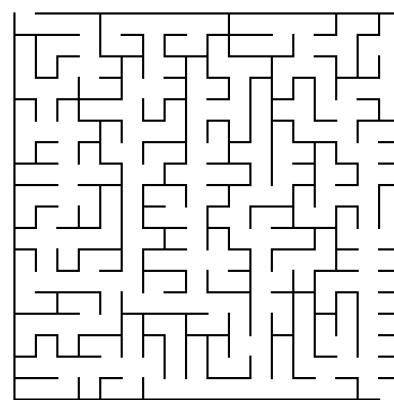
Application: Maze generation

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.



An animation of generating a 30 by 20 maze using depth-first search.

As stated, the algorithm is very simple and does not produce overly-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, which makes depth-first a good algorithm for generating mazes in video games. Randomly removing a number of walls after creating a DFS-maze can make its corridors less narrow, which can be suitable in situations where the difficulty of solving the maze is not of importance. This too can be favorable in video games.

In mazes generated by that algorithm, it 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.

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
 1. Pop a cell from the stack
 2. Make it the current cell

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.

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.

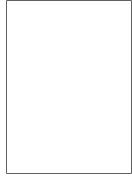
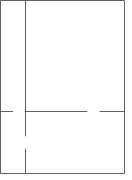
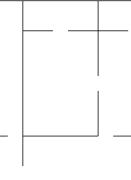
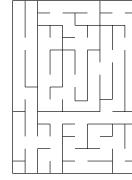
Note that simply running classical Prim's on a graph with random weights would create mazes stylistically identical to Kruskal's, because they are both minimal spanning tree algorithms. Instead, this algorithm introduces stylistic variation because the edges closer to the starting point have a lower effective weight.

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.

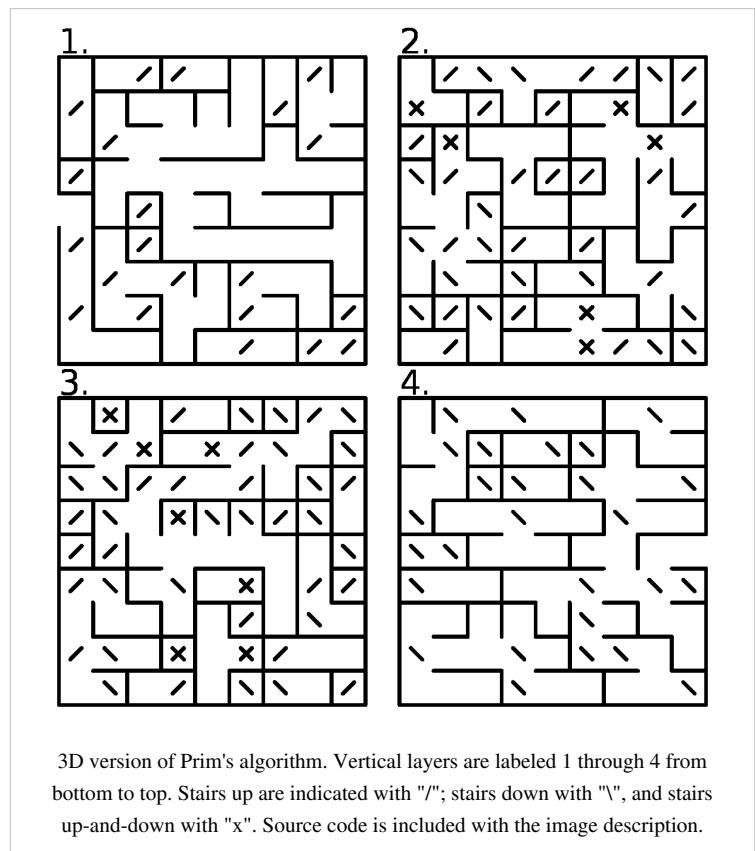
For example, in a rectangular maze, build at random points two walls that are perpendicular to each other. These two walls divide the large chamber into four smaller chambers separated by four walls. Choose three of the four walls at

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 automata algorithms

Certain types of cellular automata can be used to generate mazes.^[1] Two well-known such cellular automata, Maze and Mazectric, have rulestrings B3/S12345 and B3/S1234.^[1] 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.^[1] However, for large patterns, it behaves very differently.^[1]

For a random starting pattern, these maze-generating cellular automata will evolve into complex mazes with well-defined walls outlining corridors. Mazectric, which has the rule B3/S1234 has a tendency to generate longer and straighter corridors compared with Maze, with the rule B3/S12345.^[1] 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 as np
from numpy.random import random_integers as rnd
import matplotlib.pyplot as plt

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 = np.zeros(shape, dtype=bool)
    # Fill borders
    Z[0,:] = Z[-1,:] = 1
    Z[:,0] = Z[:,-1] = 1
    # Make isles
    for i in range(density):
        x, y = rnd(0,shape[1]//2)*2, rnd(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[rnd(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

plt.figure(figsize=(10,5))
plt.imshow(maze(80,40),cmap=plt.cm.binary,interpolation='nearest')
plt.xticks([]),plt.yticks([])
plt.show()

```

References

- [1] Nathaniel Johnston (<http://www.conwaylife.com/wiki/index.php?title=User:Nathaniel>) *et al* (21 August 2010). "Maze - LifeWiki" (<http://www.conwaylife.com/wiki/index.php?title=Maze>). LifeWiki. . Retrieved 1 March 2011.

External links

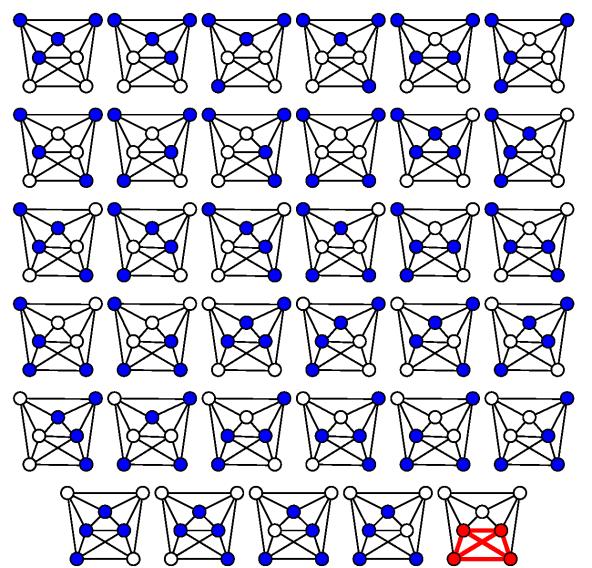
- Think Labyrinth: Maze algorithms (<http://www.astrolog.org/labyrnth/algrithm.htm#perfect>) (details on these and other maze generation algorithms)
- Explanation of an Obfuscated C maze algorithm (<http://homepages.cwi.nl/~tromp/maze.html>) (a program to generate mazes line-by-line, obfuscated in a single physical line of code)
- Maze generation and solving Java applet (<http://www.mazeworks.com/mazegen/>)
- Maze generating Java applets with source code. (<http://www.ccs.neu.edu/home/snuffy/maze/>)
- 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)
- Create Your Own Mazes (<http://www.pedagonet.com/Labyrynth/mazes.htm>)
- Collection of maze generation code (<http://rosettacode.org/wiki/Maze>) in different languages in Rosetta Code
- Maze Generation Using JavaScript (<http://www.hightechdreams.com/weaver.php?topic=mazegeneration>)
- Maze generation script for the Unity game engine (<http://unifycommunity.com/wiki/index.php?title=MazeGenerator>)

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.^[1] Along with its applications in social networks, the clique problem also has many applications in bioinformatics and computational chemistry.^[2]



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,^[3] 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),^[1] 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.^[1] 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,^[4] showed that it is not even possible to approximate the problem accurately and efficiently.

Definitions

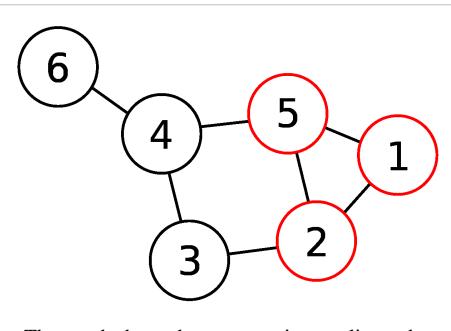
An undirected graph is formed by a finite set of vertices and a set of unordered pairs of vertices, which are called edges. By convention, in algorithm analysis, the number of vertices in the graph is denoted by n and the number of edges is denoted by m . A clique in a graph G is a complete subgraph of G ; that is, it is a subset S of the vertices such that every two vertices in S form 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 .^[1]

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 one.
- 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^[5] 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.^[6]

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.^[7] 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.^[5] 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 .^[8]

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 $O(3^{n/3})$.^[9] 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.^[10]

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.^[11] 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.^{[5][12]}

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,^[13] local search,^[14] greedy algorithms,^[15] and constraint programming.^[16] Non-standard computing methodologies for finding cliques include DNA computing^[17] and adiabatic quantum computation.^[18] The maximum clique problem was the subject of an implementation challenge sponsored by DIMACS in 1992–1993,^[19] and a collection of graphs used as benchmarks for the challenge is publicly available.^[20]

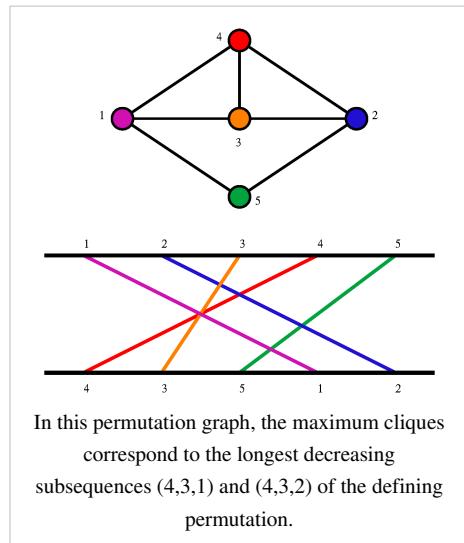
Special classes of graphs

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.^[5] 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.^[21] However, this method is complex and non-combinatorial, and specialized clique-finding algorithms have been developed for many subclasses of perfect graphs.^[22] 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.^[23] 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.^[24] 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.^[25]

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^[26] 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^[27] or semidefinite programming.^[28]



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.^[29]

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.

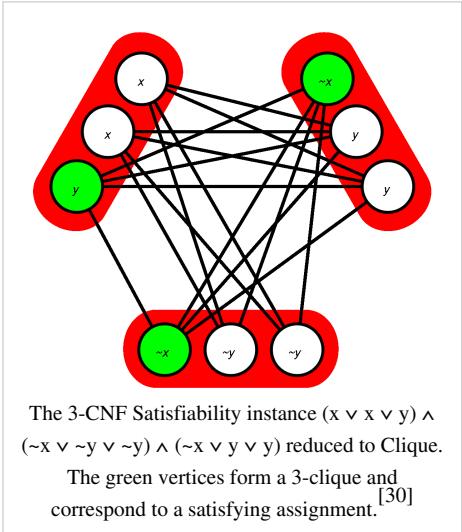
Lower bounds

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.

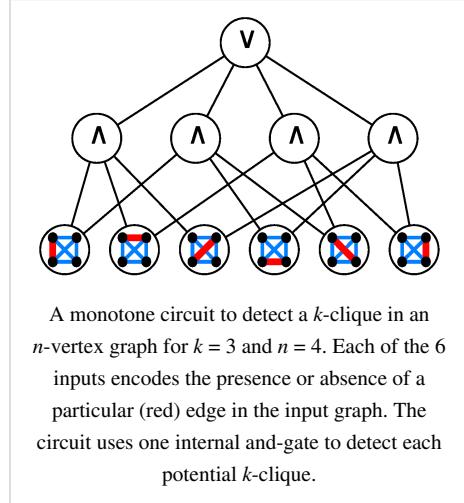
Karp's NP-completeness proof is a many-one reduction from the Boolean satisfiability problem for formulas in conjunctive normal form, which was proved NP-complete in the Cook–Levin theorem.^[31] From a given CNF formula, Karp forms a graph that has a vertex for every pair (v,c) , where v is a variable 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 others' 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 .^[32] 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.^[33] Even if a small number of NOT gates are allowed, the complexity remains superpolynomial.^[34] 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.^[35]

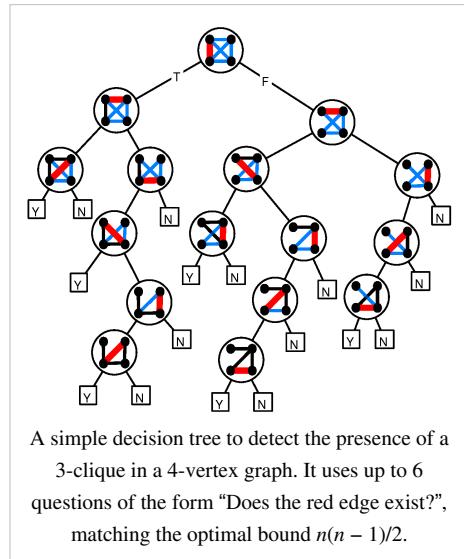


Decision tree complexity

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.^[36]

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 $\Theta(n^2)$.^[37] For quantum decision trees, the best known lower bound is $\Omega(n)$, but no matching algorithm is known for the case of $k \geq 3$.^[38]



Fixed-parameter intractability

Parameterized complexity^[39] 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.

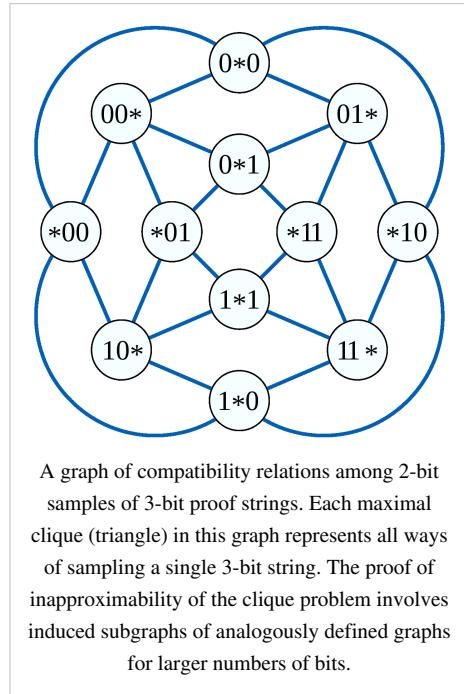
Chen et al. (2006) showed that the clique problem cannot be solved in time $n^{o(k)}$ unless the exponential time hypothesis fails.

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.^[12]

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.^{[4]–[40]} 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 $\varepsilon > 0$.^[41]

The rough idea of these inapproximability results^[42] 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.

Notes

- [1] For surveys of these algorithms, and basic definitions used in this article, see Bomze et al. (1999) and Gutin (2004).
- [2] For more details and references, see clique (graph theory).
- [3] Complete subgraphs make an early appearance in the mathematical literature in the graph-theoretic reformulation of Ramsey theory by Erdős & Szekeres (1935).
- [4] Kolata, Gina (June 26, 1990), "In a Frenzy, Math Enters Age of Electronic Mail" (<http://www.nytimes.com/1990/06/26/science/in-a-frenzy-math-enters-age-of-electronic-mail.html>), *New York Times*, .
- [5] Chiba & Nishizeki (1985).
- [6] E.g., see Downey & Fellows (1995).
- [7] Itai & Rodeh (1978) provide an algorithm with $O(m^{3/2})$ running time that finds a triangle if one exists but does not list all triangles; Chiba & Nishizeki (1985) list all triangles in time $O(m^{3/2})$.
- [8] Eisenbrand & Grandoni (2004); Kloks, Kratsch & Müller (2000); Nešetřil & Poljak (1985); Vassilevska & Williams (2009); Yuster (2006).
- [9] Tomita, Tanaka & Takahashi (2006).
- [10] Cazals & Karande (2008); Eppstein & Strash (2011).
- [11] Rosgen & Stewart (2007).
- [12] Eppstein, Löffler & Strash (2010).
- [13] Balas & Yu (1986); Carraghan & Pardalos (1990); Pardalos & Rogers (1992); Östergård (2002); Fahle (2002); Tomita & Seki (2003); Tomita & Kameda (2007); Konc & Janežič (2007).
- [14] Battiti & Protasi (2001); Katayama, Hamamoto & Narihisa (2005).
- [15] Abello, Pardalos & Resende (1999); Grossi, Locatelli & Della Croce (2004).
- [16] Régin (2003).
- [17] Ouyang et al. (1997). Although the title refers to maximal cliques, the problem this paper solves is actually the maximum clique problem.
- [18] Childs et al. (2002).
- [19] Johnson & Trick (1996).
- [20] DIMACS challenge graphs for the clique problem (<ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/>), accessed 2009-12-17.
- [21] Grötschel, Lovász & Schrijver (1988).
- [22] Golumbic (1980).
- [23] Golumbic (1980), p. 159. Even, Pnueli & Lempel (1972) 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.
- [24] Gavril (1973); Golumbic (1980), p. 247.
- [25] Clark, Colbourn & Johnson (1990).
- [26] Jerrum (1992).
- [27] Alon, Krivelevich & Sudakov (1998).
- [28] Feige & Krauthgamer (2000).
- [29] Boppana & Halldórsson (1992); Feige (2004); Halldórsson (2000).
- [30] Adapted from Sipser (1996)
- [31] Cook (1971) gives essentially the same reduction, from 3-SAT instead of Satisfiability, to show that subgraph isomorphism is NP-complete.
- [32] Lipton & Tarjan (1980).
- [33] Alon & Boppana (1987). For earlier and weaker bounds on monotone circuits for the clique problem, see Valiant (1983) and Razborov (1985).
- [34] Amano & Maruoka (1998).
- [35] Goldmann & Hästad (1992) used communication complexity to prove this result.

- [36] Wegener (1988).
- [37] For instance, this follows from Gröger (1992).
- [38] Childs & Eisenberg (2005); Magniez, Santha & Szegedy (2007).
- [39] Downey & Fellows (1999).
- [40] Feige et al. (1991); Arora & Safra (1998); Arora et al. (1998).
- [41] Håstad (1999) showed inapproximability for this ratio using a stronger complexity theoretic assumption, the inequality of NP and ZPP; Khot (2001) described more precisely the inapproximation ratio, and Zuckerman (2006) derandomized the construction weakening its assumption to $P \neq NP$.
- [42] This reduction is originally due to Feige et al. (1991) 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.

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.
- 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.
- 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/m64ju7clmqhqm9g/>), *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, ECCC TR98-008. Originally presented at the 1992 Symposium on Foundations of Computer Science, doi: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. Originally presented at the 1992 Symposium on Foundations of Computer Science, doi: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.
- Battiti, R.; Protasi, M. (2001), "Reactive local search for the maximum clique problem", *Algorithmica* **29** (4): 610–637, doi: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, 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.
- 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.
- 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.
- 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.
 - 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
 - Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM Journal on Computing* **14** (1): 210–223, doi: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.
 - Childs, A. M.; Eisenberg, J. M. (2005), "Quantum algorithms for subset finding", *Quantum Information and computation* **5** (7): 593–604, arXiv: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
 - 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.
 - 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.
 - 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.
 - 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, doi: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.
 - 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.
 - 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, 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.
 - 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, 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.
 - 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.
 - 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.

- 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.
- 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.
- 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.
- 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, JSTOR 2785673, MR0110590.
- Håstad, J. (1999), "Clique is hard to approximate within $n^{1 - \varepsilon}$ ", *Acta Mathematica* **182** (1): 105–142, doi: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.
- Itai, A.; Rodeh, M. (1978), "Finding a minimum circuit in a graph", *SIAM Journal on Computing* **7** (4): 413–423, doi: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.
- 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, 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.
- 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.
- 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, 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.
- 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.
- Luce, R. Duncan; Perry, Albert D. (1949), "A method of matrix analysis of group structure", *Psychometrika* **14** (2): 95–116, doi:10.1007/BF02289146, PMID 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, doi: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, MR0182577.
- 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.
- 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, PMID 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.
- Razborov, A. A. (1985), "Lower bounds for the monotone complexity of some Boolean functions" (in Russian), *Proceedings of the USSR Academy of Sciences* **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.
- 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.
- 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.
- 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, 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.
- 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.
- 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, 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, 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.
- 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.
- 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, ISBN 1-59593-134-1, ECCC TR05-100.

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 a description of it 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.^[1] It is well-known and widely used in application areas of graph algorithms such as computational chemistry.^[2]

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.^[3]

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 . Within the recursive calls to the algorithm, P and X are restricted to vertices that form cliques when added to R , as these are the only vertices that can be used as part of the output or to prevent some clique from being reported as output.

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 neighbors of v , $N(v)$, which finds and reports all clique extensions of R that contain v . Then, it moves v from P to X 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,^[4] 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.^[5]

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.^{[6][7]}

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 ( $R \cup \{v\}$ ,  $P \cap N(v)$ ,  $X \cap N(v)$ )
 $P := P \setminus \{v\}$ 
 $X := X \cup \{v\}$ 

```

This variant of the algorithm can be proven to be efficient for graphs of small degeneracy,^[6] and experiments show that it also works well in practice for large sparse social networks and other real-world graphs.^[7]

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 .

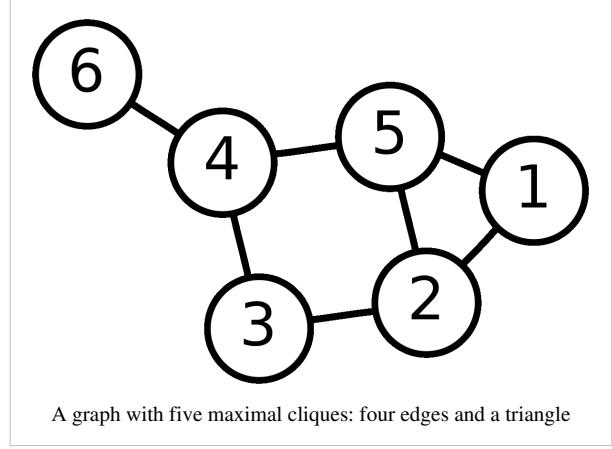
In the third and final iteration of the inner loop of the algorithm, for $v = 6$, there is a recursive call to the algorithm with $R = \{6\}$, $P = \emptyset$, and $X = \{4\}$. Because this recursive call has P empty and X non-empty, it immediately backtracks without reporting any more cliques, as there can be no maximal clique that includes vertex 6 and excludes vertex 4.

The call tree for the algorithm, therefore, looks like:

```

BronKerbosch2 ( $\emptyset$ ,  $\{1,2,3,4,5,6\}$ ,  $\emptyset$ )
BronKerbosch2 ( $\{2\}$ ,  $\{1,3,5\}$ ,  $\emptyset$ )
    BronKerbosch2 ( $\{2,3\}$ ,  $\emptyset$ ,  $\emptyset$ ): output  $\{2, 3\}$ 
    BronKerbosch2 ( $\{2,5\}$ ,  $\{1\}$ ,  $\emptyset$ )
        BronKerbosch2 ( $\{1,2,5\}$ ,  $\emptyset$ ,  $\emptyset$ ): output  $\{1,2,5\}$ 
BronKerbosch2 ( $\{4\}$ ,  $\{3,5,6\}$ ,  $\emptyset$ )
    BronKerbosch2 ( $\{3,4\}$ ,  $\emptyset$ ,  $\emptyset$ ): output  $\{3,4\}$ 
    BronKerbosch2 ( $\{4,5\}$ ,  $\emptyset$ ,  $\emptyset$ ): output  $\{4,5\}$ 
    BronKerbosch2 ( $\{4,6\}$ ,  $\emptyset$ ,  $\emptyset$ ): output  $\{4,6\}$ 
BronKerbosch2 ( $\{6\}$ ,  $\emptyset$ ,  $\{4\}$ ): 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.^[8]

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.^[6]

Notes

- [1] Cazals & Karande (2008).
- [2] Chen (2004).
- [3] Johnston (1976).
- [4] Tomita, Tanaka & Takahashi (2006); Cazals & Karande (2008).
- [5] Johnston (1976); Koch (2001); Cazals & Karande (2008).
- [6] Eppstein, Löffler & Strash (2010).
- [7] Eppstein & Strash (2011).
- [8] Tomita, Tanaka & Takahashi (2006).

References

- Akkoyunlu, E. A. (1973), "The enumeration of maximal cliques of large graphs", *SIAM Journal on Computing* **2**: 1–6, doi: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.
- 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.
- 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, doi: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.
- 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.
- 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.
- Moon, J. W.; Moser, L. (1965), "On cliques in graphs", *Israel J. Math.* **3**: 23–28, doi:10.1007/BF02760024, MR0182577.
- 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.

External links

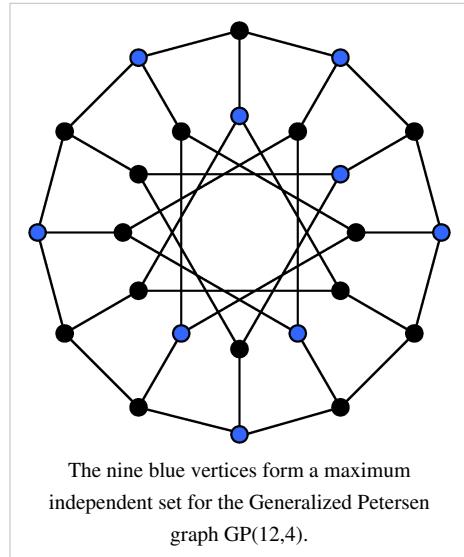
- Bron-Kerbosch algorithm implementation in Python (<http://www.kuchaev.com/files/graph.py>)
- Finding all cliques of an undirected graph (http://www.dfki.de/~neumann/ie-seminar/presentations/finding_cliques.pdf). Seminar notes by Michaela Regneri, January 11, 2007.

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.

A maximal independent set is an independent set such that adding any other vertex to the set forces the set to contain an edge.

A **maximum independent set** is a largest independent set for a given graph G and its size is 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.



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. The sum of $\alpha(G)$ and the size of a minimum vertex cover $\beta(G)$ is the number of vertices in the graph.

In a bipartite graph, 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

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,^[2] 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.^[3] 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.
- 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.

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 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.

Finding maximum independent sets

Further information: Clique problem#Finding maximum cliques in arbitrary graphs

The problem of finding the maximum independent set is sometimes referred to as "**vertex packing**". The maximum independent set problem and the maximum clique problem are polynomially equivalent: one can find a maximum independent set in a graph by finding a maximum clique in its complement graph, so many authors do not carefully distinguish between the two problems. Both problems are NP-complete, so it is unlikely that they can be solved in polynomial time. Nevertheless the maximum independent set problem 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 $O(2^{0.276n})$; see also Fomin, Grandoni & Kratsch (2009).

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;^[4] 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.^[5] However, effective approximation algorithms are known with approximation ratios that are worse than this threshold; 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 .^[6]

In some classes of graphs, including claw-free graphs and perfect graphs, the maximum independent set may be found in polynomial time.^[7] In planar graphs, the maximum independent set remains NP-complete to find exactly but 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.^[8]

Finding maximal independent sets

The problem of finding a maximal independent set can be solved in polynomial time by a trivial greedy algorithm.^[9] All maximal independent sets can be found in time $O(3^{n/3})$.

Software for searching maximum independent set

Name	License	API language	Brief info
igraph ^[10]	GPL	C, Python, R, Ruby	exact solution
NetworkX	BSD	Python	approximate solution, see the routine <code>maximum_independent_set</code> ^[11]
OpenOpt	BSD	Python	exact and approximate solutions, see STAB ^[12] class

Software for searching maximal independent set

Name	License	API language	Brief info
NetworkX	BSD	Python	see the routine maximal_independent_set [13]

Notes

- [1] Godsil & Royle (2001), p. 3.
- [2] Moon & Moser (1965).
- [3] Füredi (1987).
- [4] Chiba & Nishizeki (1985).
- [5] Berman & Fujito (1995).
- [6] Halldórsson & Radhakrishnan (1997).
- [7] For claw-free graphs, see Sbihi (1980). For perfect graphs, see Grötschel, Lovász & Schrijver (1988).
- [8] Baker (1994); Grohe (2003).
- [9] Luby (1985).
- [10] <http://igraph.sourceforge.net>
- [11] http://networkx.lanl.gov/reference/generated/networkx.algorithms.approximation.independent_set.maximum_independent_set.html
- [12] <http://openopt.org/STAB>
- [13] http://networkx.lanl.gov/reference/generated/networkx.algorithms.mis.maximal_independent_set.html#networkx.algorithms.mis.maximal_independent_set

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.
- 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.
- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM Journal on Computing* **14** (1): 210–223, doi:10.1137/0214017.
- 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, 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.
- 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.
- 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.
- Luby, M. (1985), "A simple parallel algorithm for the maximal independent set problem" (<http://www.cs.rpi.edu/~buschc/courses/distributed/spring2007/papers/mis.pdf>), *Proc. 17th Symposium on Theory of Computing*, Association for Computing Machinery, pp. 1–10, doi:10.1145/22145.22146.
- Moon, J. W.; Moser, Leo (1965), "On cliques in graphs", *Israel Journal of Mathematics* **3** (1): 23–28, doi:10.1007/BF02760024, MR0182577.
- Robson, J. M. (1986), "Algorithms for maximum independent sets", *Journal of Algorithms* **7** (3): 425–440, doi:10.1016/0196-6774(86)90032-5.

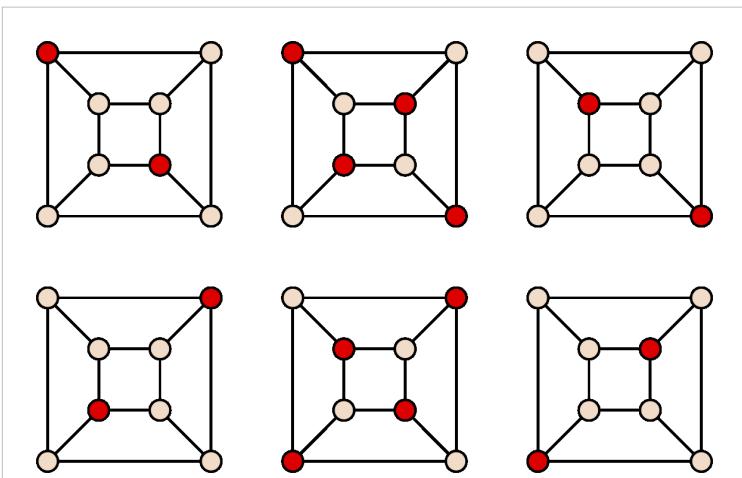
- Sbihi, Najiba (1980), "Algorithme de recherche d'un stable de cardinalité maximum dans un graphe sans étoile" (in French), *Discrete Mathematics* **29** (1): 53–76, doi:10.1016/0012-365X(90)90287-R, MR553650.

External links

- Weisstein, Eric W., "Maximal Independent Vertex Set (<http://mathworld.wolfram.com/MaximalIndependentVertexSet.html>)" from 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>)

Maximal independent set

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.^[2]

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.^[3] 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.^[4]

Certain families of graphs may, however, have much more restrictive bounds on the numbers of maximal independent sets or maximal cliques. For instance, if all graphs in a family of graphs have $O(n)$ edges, and the family is closed under subgraphs, then all maximal cliques have constant size and there can be at most linearly many maximal cliques.^[5]

Any maximal-clique irreducible graph, clearly, has at most as many maximal cliques as it has edges. A tighter bound is possible for interval graphs, and more generally chordal graphs: in these graphs there can be at most n maximal cliques.

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.^[6] 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.^[7] 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 $O(3^{n/3})$.^[8] 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.^[9] The time per maximal independent set is proportional to that for matrix multiplication in dense graphs, or faster in various classes of sparse graphs.^[10]

Notes

- [1] Erdős (1966) 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] Weigt & Hartmann (2001).
- [3] 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).
- [4] Byskov (2003). For related earlier results see Croitoru (1979) and Eppstein (2003).
- [5] Chiba & Nishizeki (1985). The sparseness condition is equivalent to assuming that the graph family has bounded arboricity.
- [6] Bisdorff & Marichal (2007); Euler (2005); Füredi (1987).
- [7] Eppstein (2003); Byskov (2003).
- [8] Eppstein (2003). For a matching bound for the widely used Bron–Kerbosch algorithm, see Tomita, Tanaka & Takahashi (2006).
- [9] Bomze et al. (1999); Eppstein (2005); Jennings & Motycková (1992); Johnson, Yannakakis & Papadimitriou (1988); Lawler, Lenstra & Rinnooy Kan (1980); Liang, Dhall & Lakshminarayanan (1991); Makino & Uno (2004); Mishra & Pitt (1997); Stix (2004); Tsukiyama et al. (1977); Yu & Chen (1993).
- [10] Makino & Uno (2004); Eppstein (2005).

References

- Bisdorff, R.; Marichal, J.-L. (2007), *Counting non-isomorphic maximal independent sets of the n-cycle graph*, arXiv: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.
- 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/cs.DS/0011009.
- 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.
- Erdős, P. (1966), "On cliques in graphs", *Israel J. Math.* **4** (4): 233–234, doi:10.1007/BF02771637, MR0205874.

- 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.
- 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.
- 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.
- 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.
- 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.
- Liang, Y. D.; Dhall, S. K.; Lakshminarayanan, S. (1991), *On the problem of finding all maximum weight independent sets in interval and circular arc graphs*, pp. 465–470
- Makino, K.; Uno, T. (2004), *New algorithms for enumerating all maximal cliques* (<http://www.springerlink.com/content/p9qbl6y1v5t3xc1w/>), 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, 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, MR0182577.
- 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
- 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.
- 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.
- 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, doi: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.

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 planar graph is just a vertex coloring of its planar 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.

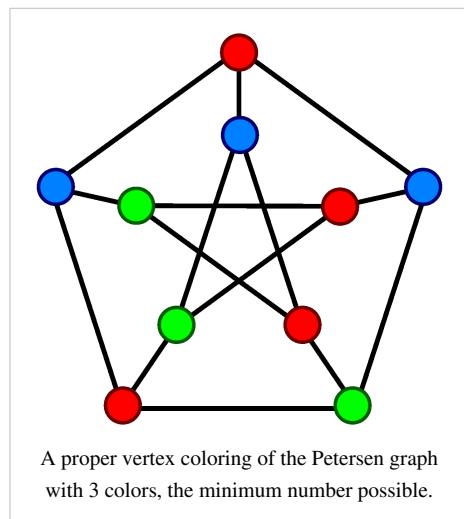
Graph coloring enjoys many practical applications as well as theoretical challenges. Beside the classical types of problems, different limitations can also be set on the graph, or on the way a color is assigned, or even on the color itself. It has even reached popularity with the general public in the 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

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. Perhaps surprisingly, the proof went back to the ideas of Heawood and Kempe and largely disregarded the intervening developments.^[2] 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,^[3] 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 in 2002 by Chudnovsky, Robertson, Seymour, Thomas 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.

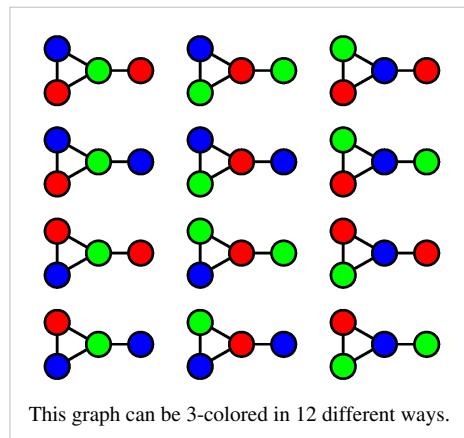
Definition and terminology

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 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,...}.

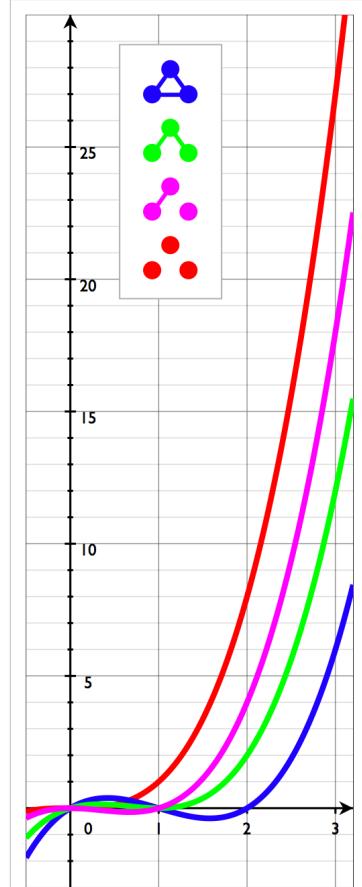
A coloring using at most k colors is called a (proper) **k -coloring**. The smallest number of colors needed to color a graph G is called its **chromatic number**, and is often denoted $\chi(G)$. Sometimes $\gamma(G)$ is used, since $\chi(G)$ is also used to denote the Euler characteristic of a graph. 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

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

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

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 .

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^[4] 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 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 is one 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 (de Bruijn & Erdős 1951).

Also, 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 polynomial $P(G, t)$ 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 unres-

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	$O(2^n n)^{[5]}$
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 forest, 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 every 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 $O(2.445^n)$.^[6] Using the principle of inclusion–exclusion and Yates's algorithm for the fast zeta transform, k -colorability can be decided in time $O(2^n n)$ ^[5] for any k . Faster algorithms are known for 3- and 4-colorability, which can be decided in time $O(1.3289^n)$ ^[7] and $O(1.7504^n)$,^[8] 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

$$P(G - uv, k) = P(G/uv, k) + P(G, k)$$

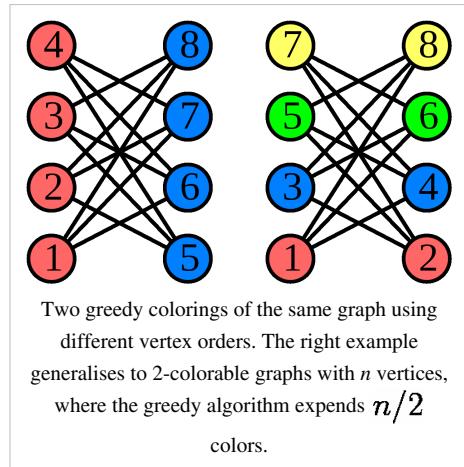
where u and v are adjacent vertices and $G - uv$ is the graph with the edge uv removed. $P(G - uv, k)$ represents the number of possible proper colorings of the graph, when the vertices may have same or different colors. The number of proper colorings therefore come from the sum of two graphs. If the 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 $((1 + \sqrt{5})/2)^{n+m} = O(1.6180^{n+m})$.^[9] The analysis can be improved to within a polynomial factor of the number $t(G)$ of spanning trees of the input graph.^[10] 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

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.

If the vertices are ordered according to their degrees, the resulting greedy coloring uses at most $\max_i \min\{d(v_i) + 1, i\}$ colors, at most one more than the graph's maximum degree. This heuristic is sometimes called the Welsh–Powell algorithm.^[11] 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.^[12] 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.^[13]

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.^[13]

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^[14] 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 $\text{poly}(\Delta) + O(\log^* n)$.^[15] The technique was extended to unit disk graphs by Schneider et al.^[16] The fastest deterministic algorithms for $(\Delta + 1)$ -coloring for small Δ are due to Leonid Barenboim, Michael Elkin and Fabian Kuhn.^[17] 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.^[18] 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). Somewhat surprisingly, 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 $k = 2$. Especially, it is NP-hard to compute the chromatic number. The 3-coloring problem remains NP-complete even on planar graphs of degree 4.^[19]

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.^[20] For all $\epsilon > 0$, approximating the chromatic number within $n^{1-\epsilon}$ is NP-hard.^[21]

It is also NP-hard to color a 3-colorable graph with 4 colors^[22] and a k -colorable graph with $k^{(\log k)/25}$ colors for sufficiently large constant k .^[23]

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 $k = 2$.^[24] There is no FPRAS for evaluating the chromatic polynomial at any rational point $k \geq 1.5$ except for $k = 2$ unless $\text{NP} = \text{RP}$.^[25]

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.^[26] In terms of approximation algorithms, Vizing's algorithm shows that the edge chromatic number can be approximated within $4/3$, and the hardness result shows that no $(4/3 - \epsilon)$ -algorithm exists for any $\epsilon > 0$ 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.^[27]

Applications

Scheduling

Vertex coloring models to a number of scheduling problems.^[28] 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

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.^[29] The compiler constructs an *interference graph*, where vertices are symbolic registers and an edge connects two nodes if they are needed at the same time. If the graph can be colored with k colors then the variables can be stored in 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

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 says 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	Centered coloring
Distance between two colors of adjacent vertices must not belong to fixed set T	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 Kubale (2004)
- [2] van Lint & Wilson (2001, Chap. 33)
- [3] Jensen & Toft (1995), p. 2
- [4] Brooks (1941)
- [5] Björklund, Husfeldt & Koivisto (2009)
- [6] Lawler (1976)
- [7] Beigel & Eppstein (2005)
- [8] Byskov (2004)
- [9] Wilf (1986)
- [10] Sekine, Imai & Tani (1995)
- [11] Welsh & Powell (1967)
- [12] Brélaz (1979)
- [13] Schneider (2010)

- [14] Cole & Vishkin (1986), see also Cormen, Leiserson & Rivest (1990, Section 30.5)
- [15] Goldberg, Plotkin & Shannon (1988)
- [16] Schneider (2008)
- [17] Barenboim & Elkin (2009); Kuhn (2009)
- [18] Panconesi (1995)
- [19] Dailey (1980)
- [20] Halldórsson (1993)
- [21] Zuckerman (2007)
- [22] Guruswami & Khanna (2000)
- [23] Khot (2001)
- [24] Jaeger, Vertigan & Welsh (1990)
- [25] Goldberg & Jerrum (2008)
- [26] Holyer (1981)
- [27] Crescenzi & Kann (1998)
- [28] Marx (2004)
- [29] Chaitin (1982)

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, 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
- Björklund, A.; Husfeldt, T.; Koivisto, M. (2009), "Set partitioning via inclusion–exclusion", *SIAM Journal on Computing* **39** (2): 546–563, doi:10.1137/070683933
- Brèlaz, D. (1979), "New methods to color the vertices of a graph", *Communications of the ACM* **22** (4): 251–256, doi: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
- 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
- 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, 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
- 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

- 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
- Fawcett, B. W. (1978), "On infinite full colourings of graphs", *Can. J. Math.* **XXX**: 455–457
- 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
- 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
- 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
- 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, 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
- Holyer, I. (1981), "The NP-completeness of edge-coloring", *SIAM Journal on Computing* **10** (4): 718–720, doi: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
- 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
- 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, 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, 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
- 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
- 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**, 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, ISBN 978-3-642-27874-7, MR2920058.
- 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, 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, 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
- 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
- Zykov, A. A. (1949), "О некоторых свойствах линейных комплексов (On some properties of linear complexes)" (<http://mi.mathnet.ru/eng/msb5974>) (in Russian), *Math. Sbornik*. **24(66)** (2): 163–188
- Jensen, Tommy R.; Toft, Bjarne (1995), *Graph Coloring Problems*, John Wiley & Sons, ISBN 0-471-02865-7, 9780471028659

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 such that every edge connects a vertex in U to one in V ; that is, U and V are each independent sets. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.^{[1][2]}

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.^{[3][4]} 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 . If a bipartite graph is not connected, it may have more than one bipartition;^[5] 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.^[3]

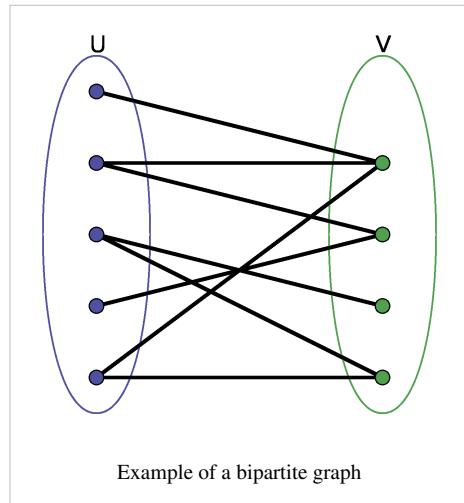
Examples

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.^[6]

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 as small a set of train stations 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.^[7]

More abstract examples include the following:

- Every tree is bipartite.^[4]
- Cycle graphs with an even number of vertices are bipartite.^[4]
- Every planar graph whose faces all have even length is bipartite.^[8] 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.^[9]
- The complete bipartite graph on n and m vertices, denoted by $K_{n,m}$ is the bipartite graph $G = (V, U, E)$, where V and U are disjoint sets of size n and m , respectively, and E connects every vertex in V with all vertices in U . It follows that $K_{n,m}$ has nm edges.^[10] Closely related to the complete bipartite graphs are the crown graphs, formed from complete bipartite graphs by removing the edges of a perfect matching.^[11]
- 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.^[12]



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.^[13]
- A graph is bipartite if and only if it is 2-colorable, (i.e. its chromatic number is less than or equal to 2).^[3]
- The spectrum of a graph is symmetric if and only if it's a bipartite graph.^[14]

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.^{[15][16]} 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.^[17] 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.^[18] 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.^[19]

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.^[20] 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.^[21]

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.^[22]

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.^[23]

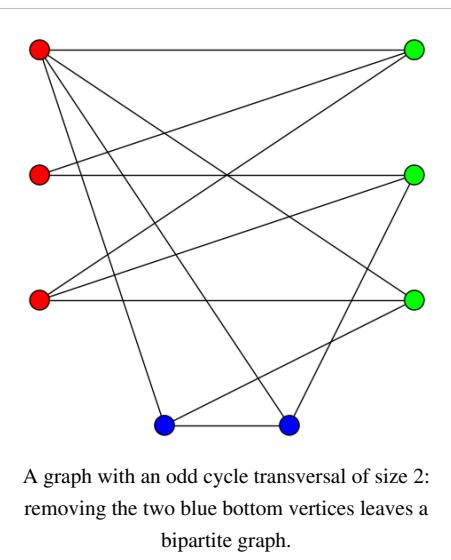
Alternatively, a similar procedure may be used with breadth-first search in place of depth-first search. Again, each node is given the opposite color to its parent in the search tree, in breadth-first order. If, when a vertex is colored, there exists an edge connecting it to a previously-colored vertex 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.^[24]

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.^[25]

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.^[26] 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 .^[27] More specifically, the time for this algorithm is $O(4k^k mn)$.

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.

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.^[28] In many cases, matching problems are simpler to solve on bipartite graphs than on non-bipartite graphs,^[29] and many matching algorithms such as the Hopcroft–Karp algorithm for maximum cardinality matching^[30] 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.^[31] 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.^[32]

The Dulmage–Mendelsohn decomposition is a structural decomposition of bipartite graphs that is useful in finding maximum matchings.^[33]

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.^[34] A factor graph is a closely related belief network used for probabilistic decoding of LDPC and turbo codes.^[35]

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.^[36]

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.^[37]

References

- [1] Diestel, Reinhard (2005). *Graph Theory, Grad. Texts in Math* (<http://diestel-graph-theory.com/>). Springer. ISBN 978-3-642-14278-9..
- [2] Arsatian, Armen S.; Denley, Tristan M. J.; Häggkvist, Roland (1998), *Bipartite Graphs and their Applications*, Cambridge Tracts in Mathematics, **131**, Cambridge University Press, ISBN 9780521593458.
- [3] Arsatian, Denley & Häggkvist (1998), p. 7.
- [4] Scheinerman, Edward A. (2012), *Mathematics: A Discrete Introduction* (<http://books.google.com/books?id=DZBHGD2sEYwC&pg=PA363>) (3rd ed.), Cengage Learning, p. 363, ISBN 9780840049421, .
- [5] Chartrand, Gary; Zhang, Ping (2008), *Chromatic Graph Theory* (http://books.google.com/books?id=_l4CJq46MXwC&pg=PA223), Discrete Mathematics And Its Applications, **53**, CRC Press, p. 223, ISBN 9781584888000, .
- [6] Wasserman, Stanley; Faust, Katherine (1994), *Social Network Analysis: Methods and Applications* (<http://books.google.com/books?id=CAm2DpIqRUIC&pg=PA299>), Structural Analysis in the Social Sciences, **8**, Cambridge University Press, pp. 299–302, ISBN 9780521387071, .
- [7] Niedermeier, Rolf (2006). *Invitation to Fixed Parameter Algorithms (Oxford Lecture Series in Mathematics and Its Applications)*. Oxford. pp. 20–21. ISBN 978-0-19-856607-6.

- [8] Soifer, Alexander (2008), *The Mathematical Coloring Book*, Springer-Verlag, pp. 136–137, ISBN 978-0-387-74640-1. 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.
- [9] Bandelt, H.-J.; Chepoi, V.; Eppstein, D. (2010), "Combinatorics and geometry of finite and infinite squaregraphs", *SIAM Journal on Discrete Mathematics* **24** (4): 1399–1440, arXiv:0905.4537, doi:10.1137/090760301.
- [10] Asratian, Denley & Häggkvist (1998), p. 11.
- [11] Archdeacon, D.; Debowsky, M.; Dinitz, J.; Gavlas, H. (2004), "Cycle systems in the complete bipartite graph minus a one-factor", *Discrete Mathematics* **284** (1–3): 37–43, doi:10.1016/j.disc.2003.11.021.
- [12] Ovchinnikov, Sergei (2011), *Graphs and Cubes*, Universitext, Springer. See especially Chapter 5, "Partial Cubes", pp. 127–181.
- [13] Asratian, Denley & Häggkvist (1998), Theorem 2.1.3, p. 8. Asratian et al. attribute this characterization to a 1916 paper by Dénes König.
- [14] Biggs, Norman (1994), *Algebraic Graph Theory* (<http://books.google.com/books?id=6TasRmIFOxQC&pg=PA53>), Cambridge Mathematical Library (2nd ed.), Cambridge University Press, p. 53, ISBN 9780521458979, .
- [15] König, Dénes (1931). "Gráfok és mátrixok". *Matematikai és Fizikai Lapok* **38**: 116–119..
- [16] Gross, Jonathan L.; Yellen, Jay (2005), *Graph Theory and Its Applications* (http://books.google.com/books?id=-7Q_POGh-2cC&pg=PA568), Discrete Mathematics And Its Applications (2nd ed.), CRC Press, p. 568, ISBN 9781584885054, .
- [17] Chartrand, Gary; Zhang, Ping (2012), *A First Course in Graph Theory* (<http://books.google.com/books?id=ocIr0RHyl8oC&pg=PA189>), Courier Dover Publications, pp. 189–190, ISBN 9780486483689, .
- [18] Béla Bollobás (1998), *Modern Graph Theory* (<http://books.google.com/books?id=SbZKSZ-1qrwC&pg=PA165>), Graduate Texts in Mathematics, **184**, Springer, p. 165, ISBN 9780387984889, .
- [19] Chudnovsky, Maria; Robertson, Neil; Seymour, Paul; Thomas, Robin (2006), "The strong perfect graph theorem" (<http://annals.princeton.edu/annals/2006/164-1/p02.xhtml>), *Annals of Mathematics* **164** (1): 51–229, doi:10.4007/annals.2006.164.51, .
- [20] Asratian, Denley & Häggkvist (1998), p. 17.
- [21] A. A. Sapozhenko (2001), "Hypergraph" (http://www.encyclopediaofmath.org/index.php?title=Main_Page), in Hazewinkel, Michiel, *Encyclopedia of Mathematics*, Springer, ISBN 978-1-55608-010-4,
- [22] Brualdi, Richard A.; Harary, Frank; Miller, Zevi (1980), "Bigraphs versus digraphs via matrices", *Journal of Graph Theory* **4** (1): 51–73, doi:10.1002/jgt.3190040107, MR558453. Brualdi et al. credit the idea for this equivalence to Dulmage, A. L.; Mendelsohn, N. S. (1958), "Coverings of bipartite graphs", *Canadian Journal of Mathematics* **10**: 517–534, doi:10.4153/CJM-1958-052-0, MR0097069.
- [23] Sedgewick, Robert (2004), *Algorithms in Java, Part 5: Graph Algorithms* (3rd ed.), Addison Wesley, pp. 109–111.
- [24] Kleinberg, Jon; Tardos, Éva (2006), *Algorithm Design*, Addison Wesley, pp. 94–97.
- [25] Eppstein, David (2009), "Testing bipartiteness of geometric intersection graphs", *ACM Transactions on Algorithms* **5** (2): Art. 15, arXiv:cs.CG/0307023, doi:10.1145/1497290.1497291, MR2561751.
- [26] Yannakakis, Mihalis (1978), "Node-and edge-deletion NP-complete problems", *Proceedings of the 10th ACM Symposium on Theory of Computing (STOC '78)*, pp. 253–264, doi:10.1145/800133.804355
- [27] Reed, Bruce; Smith, Kaleigh; Vetta, Adrian (2004), "Finding odd cycle transversals", *Operations Research Letters* **32** (4): 299–301, doi:10.1016/j.orl.2003.10.009, MR2057781.
- [28] Ahuja, Ravindra K.; Magnanti, Thomas L.; Orlin, James B. (1993), "12. Assignments and Matchings", *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, pp. 461–509.
- [29] Ahuja, Magnanti & Orlin (1993), p. 463: "Nonbipartite matching problems are more difficult to solve because they do not reduce to standard network flow problems."
- [30] 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.
- [31] Ahuja, Magnanti & Orlin (1993), Application 12.1 Bipartite Personnel Assignment, pp. 463–464.
- [32] Robinson, Sara (April 2003), "Are Medical Students Meeting Their (Best Possible) Match?" (<http://www.siam.org/pdf/news/305.pdf>), *SIAM News* (3): 36, .
- [33] Dulmage & Mendelsohn (1958).
- [34] Moon, Todd K. (2005), *Error Correction Coding: Mathematical Methods and Algorithms* (<http://books.google.com/books?id=adx8CRx5vQC&pg=PA638>), John Wiley & Sons, p. 638, ISBN 9780471648000, .
- [35] Moon (2005), p. 686.
- [36] Cassandras, Christos G.; Lafourcade, Stephane (2007), *Introduction to Discrete Event Systems* (<http://books.google.com/books?id=AxguNHDtO7MC&pg=PA224>) (2nd ed.), Springer, p. 224, ISBN 9780387333328, .
- [37] Grünbaum, Branko (2009), *Configurations of Points and Lines* (<http://books.google.com/books?id=mRw571GNa5UC&pg=PA28>), Graduate Studies in Mathematics, **103**, American Mathematical Society, p. 28, ISBN 9780821843086, .

External links

- Information System on Graph Class Inclusions (<http://wwwteo.informatik.uni-rostock.de/isgci/index.html>): bipartite graph (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_69.html)
- Weisstein, Eric W., " Bipartite Graph (<http://mathworld.wolfram.com/BipartiteGraph.html>)" from 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.

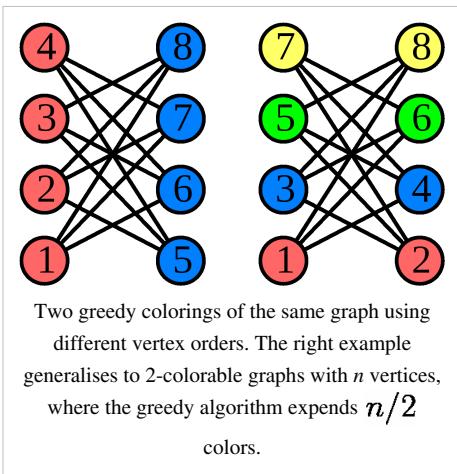
Greed is not always good

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.^[1] 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 some ordering of the vertices of G 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 .^[2]

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.^[3] 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.^[4] 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.^[5] 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.^[6]

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.^[7] However, for interval graphs, a constant competitive ratio is possible,^[8] while for bipartite graphs and sparse graphs a logarithmic ratio can be achieved.^[9] 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.^[9]

Notes

- [1] Kučera (1991).
- [2] Zaker (2006).
- [3] Chvátal (1984).
- [4] Middendorf & Pfeiffer (1990).
- [5] Welsh & Powell (1967); Johnson (1979); Syslo (1989); Maffray (2003).
- [6] Lovász (1975).
- [7] Lovász, Saks & Trotter (1989); Sz, Vishwanathan (1990).
- [8] Kierstead & Trotter (1981).
- [9] Irani (1994).

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.
- 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.
- 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.
- 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.
- 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, 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.
- 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.
- 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, 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.
- 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.

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 in-between functions as a calling convention (*interprocedural register allocation*).

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.

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 off of 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 v)  $\cup$  neighbors(second v)) < 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.

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", ESWEEK Tutorial <http://thedude.cc.gt.atl.ga.us/tutorials/1/>

Covering and domination

Vertex cover

In the mathematical discipline of graph theory, a **vertex 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.

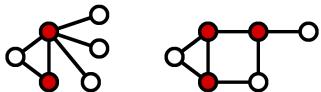
The minimum vertex cover problem can be formulated as a half-integral linear program whose dual linear program is the maximum matching problem.

Definition

Formally, a vertex cover of a graph G is a set C of vertices such that each edge of G is incident to at least one vertex in C . The set C is said to *cover* the edges of G . The following figure shows examples of vertex covers in two graphs (and the set C 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 two 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 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).^[1]

$$\begin{aligned} & \text{minimize} \quad \sum_{v \in V} c(v)x_v && \text{(minimize the total cost)} \\ & \text{subject to} \quad x_u + x_v \geq 1 \quad \text{for all } \{u, v\} \in E && \text{(cover every edge of the graph)} \\ & \quad x_v \in \{0, 1\} \quad \text{for all } v \in V. && \text{(every vertex is either in the vertex cover or not)} \end{aligned}$$

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^[2] and even in planar graphs of degree at most 3.^[3]

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.

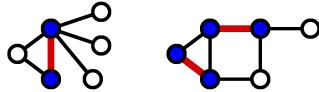
Fixed-parameter tractability

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 $1.2738^k \cdot n^{O(1)}$.^[4] 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.^[5] 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 $2^{o(\sqrt{k})} n^{O(1)}$.^[6]

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.^[7]

This simple algorithm was discovered independently by Fanica Gavril and Mihalis Yannakakis.^[7]

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.^[8]

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 $\mathbf{P} = \mathbf{NP}$.^[9] Moreover, if the unique games conjecture is true then minimum vertex cover cannot be approximated within any constant factor better than 2.^[10]

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-complete 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 $d - 1$.^[10]

Fixed-parameter tractability

For the hitting set problem, different parameterizations make sense.^[11] 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.^[12] 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] Vazirani 2001, pp. 122–123
- [2] Garey, Johnson & Stockmeyer 1974
- [3] Garey & Johnson 1977, pp. 190 and 195.
- [4] Chen, Kanj & Xia 2006
- [5] Demaine et al. 2005
- [6] Flum & Grohe (2006, p. 437)
- [7] Papadimitriou & Steiglitz 1998, p. 432, mentions both Gavril and Yannakakis. Garey & Johnson 1979, p. 134, cites Gavril.
- [8] Karakostas 2004
- [9] Dinur & Safra 2005
- [10] Khot & Regev 2008
- [11] Flum & Grohe (2006, p. 10ff)
- [12] O'Callahan & Choi 2003

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. 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://erikdемaine.org/papers/HMinorFree_JACM/). *Journal of the ACM* **52** (6): 866–893. doi:10.1145/1101821.1101823. Retrieved 2010-03-05.
- Dinur, Irit; Safra, Samuel (2005). "On the hardness of approximating minimum vertex cover" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.334&rep=rep1&type=pdf>). *Annals of Mathematics* **162** (1):

- 439–485. doi:10.4007/annals.2005.162.439. Retrieved 2011-11-03.
- 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.
 - 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
 - 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*.
 - 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.
 - 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.
 - 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

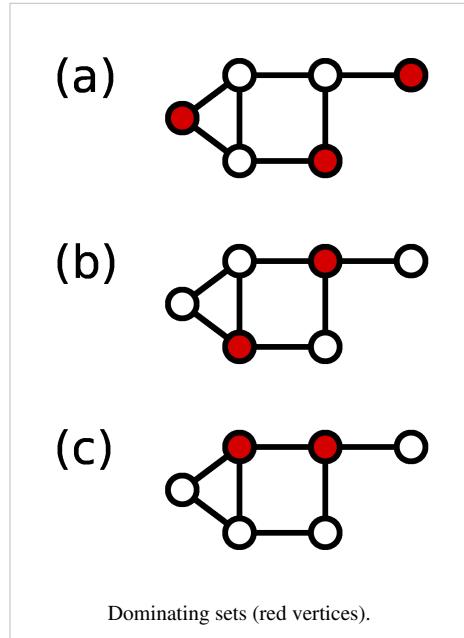
- Weisstein, Eric W., " Vertex Cover (<http://mathworld.wolfram.com/VertexCover.html>)" from MathWorld.

Dominating set

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 joined to at least one member of D by some edge. 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

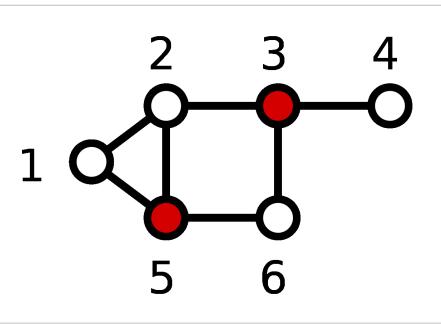
The following pair of reductions (Kann 1992, pp. 108–109) shows that the minimum dominating set problem and the set cover problem are equivalent under L-reductions: given an instance of one problem, we can construct an equivalent instance of the other problem.

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.

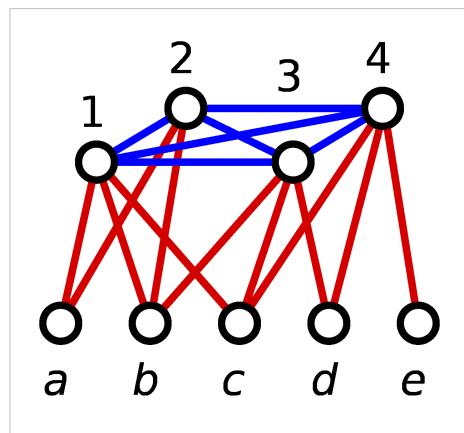
Now if $C = \{S_i : i \in D\}$ is a feasible solution of the set cover problem for some subset $D \subseteq I$, then D is a dominating set for G , with $|D| = |C|$: First, for each $u \in U$ there is an $i \in D$ such that $u \in S_i$, and by construction, u and i are adjacent in G ; hence u is dominated 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 domatic partition is a partition of the vertices into disjoint dominating sets. The domatic number is the maximum size of a domatic partition.

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.
- 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.
- Crescenzi, Pierluigi; Kann, Viggo; Halldórsson, Magnús; Karpinski, Marek; Woeginger, Gerhard (2000), "Minimum dominating set" [1], *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, MR1432221.
- 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.
- 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.
- 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.
- 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.

- Guha, S.; Khuller, S. (1998), "Approximation algorithms for connected dominating sets", *Algorithmica* **20** (4): 374–387, doi:10.1007/PL00009201.
- Haynes, Teresa W.; Hedetniemi, Stephen; Slater, Peter (1998a), *Fundamentals of Domination in Graphs*, Marcel Dekker, ISBN 0-8247-0033-3, OCLC 37903553.
- Haynes, Teresa W.; Hedetniemi, Stephen; Slater, Peter (1998b), *Domination in Graphs: Advanced Topics*, Marcel Dekker, ISBN 0-8247-0034-1, OCLC 38201061.
- 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.
- Kann, Viggo (1992), *On the Approximability of NP-complete Optimization Problems*^[2]. 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, 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.
- 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, ISBN 978-3-642-04127-3.

References

- [1] <http://www.nada.kth.se/~viggo/wwwcompendium/node11.html>
[2] <http://www.csc.kth.se/~viggo/papers/phdthesis.pdf>

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, genome assembly, 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 *edges* to make the graph cycle-free is equivalent to finding a minimum 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, see Karp (1972).

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.^[2] 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 $O(1.8638^n)$.^[3] 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.^[4] The parameterized versions of the directed and undirected problems are both fixed-parameter tractable.^[5]

Approximation

The problem is APX-complete, which directly follows from the APX-completeness of the vertex cover problem,^[6] and the existence of an approximation preserving L-reduction from the vertex cover problem to it.^[7] The best known approximation on undirected graphs is by a factor of two.^[8]

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.

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 (Silberschatz & Galvin 2008).

Furthermore, the feedback vertex set problem has applications in VLSI chip design (cf. Festa, Pardalos & Resende (2000)) and genome assembly.

Notes

- [1] unpublished results due to Garey and Johnson, cf. Garey & Johnson (1979): GT7
- [2] Fomin & Villanger (2010)
- [3] Fomin et al. (2008)
- [4] Razgon (2007)
- [5] Chen et al. (2008)
- [6] Dinur & Safra 2005
- [7] Karp (1972)
- [8] Becker & Geiger (1996)

References

Research articles

- Ann Becker, Reuven Bar-Yehuda, Dan Geiger: Randomized Algorithms for the Loop Cutset Problem. *J. Artif. Intell. Res. (JAIR)* 12: 219-234 (2000)
- Becker, Ann; Geiger, Dan (1996), "Optimization of Pearl's Method of Conditioning and Greedy-Like Approximation Algorithms for the Vertex Feedback Set Problem.", *Artif. Intell.* **83** (1): 167–188, doi:10.1016/0004-3702(95)00004-6
- Cao, Yixin; Chen, Jianer; Liu, Yang (2010), *On Feedback Vertex Set: New Measure and New Structures* (<http://www.springerlink.com/content/f3726432823626n7/>), in Kaplan, Haim, "SWAT 2010", *LNCS* **6139**: 93–104, doi:10.1007/978-3-642-13731-0
- Jianer Chen, Fedor V. Fomin, Yang Liu, Songjian Lu, Yngve Villanger: Improved algorithms for feedback vertex set problems. *J. Comput. Syst. Sci.* 74(7): 1188-1198 (2008)
- Jianer Chen, Yang Liu, Songjian Lu, Barry O'Sullivan, Igor Razgon: A fixed-parameter algorithm for the directed feedback vertex set problem. *J. ACM* 55(5): (2008)
- 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* **162** (1): 439–485, doi:10.4007/annals.2005.162.439, retrieved 2010-03-05.
- 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
- Fomin, Fedor V.; Villanger, Yngve (2010), "Finding Induced Subgraphs via Minimal Triangulations.", *Proc. of STACS 2010*, pp. 383–394, doi:10.4230/LIPIcs.STACS.2010.2470
- 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 A1.1: GT7, pg.191.
- Karp, Richard M. (1972), "Reducibility Among Combinatorial Problems" (<http://www.cs.berkeley.edu/~luca/cs172/karp.pdf>), *Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y.. New York: Plenum*, pp. 85–103
- I. Razgon : Computing Minimum Directed Feedback Vertex Set in $O^*(1.9977n)$. In: Giuseppe F. Italiano, Eugenio Moggi, Luigi Laura (Eds.), Proceedings of the 10th Italian Conference on Theoretical Computer Science 2007, World Scientific, pp. 70–81 (author's version (pdf) (<http://www.cs.ucc.ie/~ir2/papers/ictcsIgorCamera.pdf>), preliminary full version (pdf) (<http://www.cs.ucc.ie/~ir2/papers/mas1203.pdf>)).

Textbooks and survey articles

- P. Festa, P.M. Pardalos, and M.G.C. Resende, Feedback set problems, Handbook of Combinatorial Optimization, D.-Z. Du and P.M. Pardalos, Eds., Kluwer Academic Publishers, Supplement vol. A, pp. 209–259, 2000. author's version (pdf) (<http://www.research.att.com/~mgcr/doc/sfsp.pdf>)
- 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.^{[1][2]}

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 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.^[3]

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 find an edge set at most c times bigger than the optimal result. As of 2006, the highest value of c for which such an impossibility result is known is $c = 1.3606^{[4]}$. The best known approximation algorithm has ratio $O(\log n \log \log n)$.^[5] For the dual problem, of approximating the maximum number of edges in an acyclic subgraph, an approximation somewhat better than $1/2$ is possible.^{[6][7]}

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^[8].

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.

Notes

- [1] Di Battista, Giuseppe; Eades, Peter; Tamassia, Roberto; Tollis, Ioannis G. (1998), "Layered Drawings of Digraphs", *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, pp. 265–302, ISBN 9780133016154.
- [2] 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.
- [3] 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), doi:10.1145/1411509.1411511.
- [4] 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* **162** (1): 439–485, doi:10.4007/annals.2005.162.439, . (Preliminary version in STOC 2002, titled "The importance of being biased", doi:10.1145/509907.509915.)
- [5] G. Even, J. Noar, B. Schieber and M. Sudan. Approximating minimum feedback sets and multicut in directed graphs. *Algorithmica*, 20, (1998) 151-174.
- [6] Berger, B.; Shor, P. (1990), "Approximation algorithms for the maximum acyclic subgraph problem" (<http://dl.acm.org/citation.cfm?id=320176.320203>), *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pp. 236–243, .
- [7] Eades, P.; Lin, X.; Smyth, W. F. (1993), "A fast and effective heuristic for the feedback arc set problem", *Information Processing Letters* **47**: 319–323, doi:10.1016/0020-0190(93)90079-O.
- [8] Kenyon-Mathieu, C.; Schudy, W. (2007). *How to rank with few errors*. pp. 95. doi:10.1145/1250790.1250806. author's extended version (http://www.cs.brown.edu/people/ws/papers/fast_journal.pdf)

References

- Richard M. Karp. "Reducibility Among Combinatorial Problems." In *Complexity of Computer Computations*, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y.. New York: Plenum, p. 85-103. 1972.
- Pierluigi Crescenzi, Viggo Kann, Magnús Halldórsson, Marek Karpinski and Gerhard Woeginger. " Minimum Feedback Arc Set (<http://www.nada.kth.se/~viggo/wwwcompendium/node20.html>)". A *compendium of NP optimization problems* (<http://www.nada.kth.se/~viggo/wwwcompendium/>). Last modified March 20, 2000.
- Viggo Kann. pdf On the Approximability of NP-complete Optimization Problems (<http://www.nada.kth.se/~viggo/papers/phdthesis.pdf>). PhD thesis. Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm. 1992.
- 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. A1.1: GT8, pg.192.

Tours

Eulerian path

In graph theory, an **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.^[2]

For the existence of Eulerian trails it is necessary that no more than 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

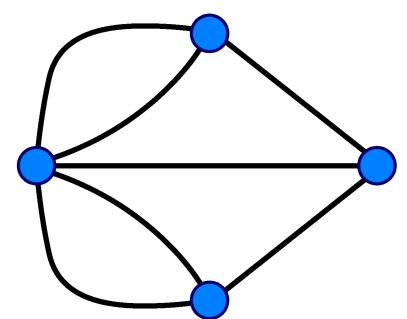
An **Eulerian trail**,^[3] or **Euler walk** in an undirected graph is a path that uses each edge exactly once. If such a path exists, the graph is called **traversable** or **semi-eulerian**.^[4]

An **Eulerian cycle**,^[3] **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**.^[5] The term "Eulerian graph" is also sometimes used in a weaker sense to denote a graph where every vertex has even degree. For 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.

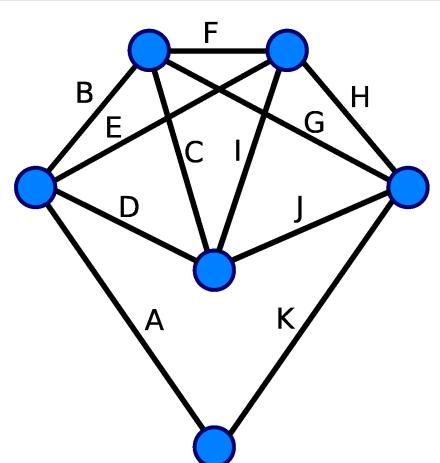
For directed graphs, "path" has to be replaced with *directed path* and "cycle" with *directed cycle*.

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.^[6] 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.^[7] Consider a graph known to have all edges in the same component and at most two vertices of odd degree. We start with a vertex of odd degree—if the graph has none, then start with any vertex. At each step we move across an edge whose deletion would not disconnect the graph, unless we have no choice, then we delete that edge. At the end of the algorithm there are no edges left, and the sequence of edges we moved across 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.^[8]

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.^[9] 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.^[10]

Special cases

The asymptotic formula for the number of Eulerian circuits in the complete graphs was determined by McKay and Robinson (1995):^[11]

$$ec(K_n) = 2^{(n+1)/2} \pi^{1/2} e^{-n^2/2 + 11/12} n^{(n-2)(n+1)/2} (1 + O(n^{-1/2+\epsilon})).$$

A similar formula was later obtained by M.I. Isaev (2009) for complete bipartite graphs:^[12]

$$ec(K_{n,n}) = (n/2 - 1)!^{2n} 2^{n^2-n+1/2} \pi^{-n+1/2} n^{n-1} (1 + O(n^{-1/2+\epsilon})).$$

Applications

Eulerian trails are used in bioinformatics to reconstruct the DNA sequence from its fragments.^[13]

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] C. L. Mallows, N. J. A. Sloane (1975). "Two-graphs, switching classes and Euler graphs are equal in number". *SIAM Journal on Applied Mathematics* **28** (4): 876–880. doi:10.1137/0128070. JSTOR 2100368.
- [3] 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.
- [4] Jun-ichi Yamaguchi, Introduction of Graph Theory (<http://jwilson.coe.uga.edu/EMAT6680/Yamaguchi/emat6690/essay1/GT.html>).
- [5] 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>).
- [6] Schrijver, A. (1983), "Bounds on the number of Eulerian orientations", *Combinatorica* **3** (3-4): 375–380, doi:10.1007/BF02579193, MR729790.
- [7] Fleury, M. (1883), "Deux problèmes de Géométrie de situation" (<http://books.google.com/books?id=l-03AAAAMAAJ&pg=PA257>) (in French), *Journal de mathématiques élémentaires*, 2nd ser. **2**: 257–261, .
- [8] Fleischner, Herbert (1991), "X.1 Algorithms for Eulerian Trails", *Eulerian Graphs and Related Topics: Part 1, Volume 2*, Annals of Discrete Mathematics, **50**, Elsevier, pp. X.1–13, ISBN 978-0-444-89110-5.

- [9] Brightwell and Winkler, " Note on Counting Eulerian Circuits (<http://www.cdam.lse.ac.uk/Reports/Files/cdam-2004-12.pdf>)", 2004.
- [10] Tetali, P.; Vempala, S. (2001). "Random Sampling of Euler Tours" (<http://www.springerlink.com/content/k5kbhmg4qkj7whcf/>).
Algorithmica **30**: 376–385. .
- [11] 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.
- [12] M.I. Isaev (2009). "Asymptotic number of Eulerian circuits in complete bipartite graphs" (in Russian). *Proc. 52-nd MFTI Conference* (Moscow): 111–114.
- [13] Pevzner, Pavel A.; Tang, Haixu; Waterman, Michael S. (2001). "An Eulerian trail approach to DNA fragment assembly" (<http://www.pnas.org/content/98/17/9748.long>). *Proceedings of the National Academy of Sciences of the United States of America* **98** (17): 9748–9753. Bibcode 2001PNAS...98.9748P. doi:10.1073/pnas.171285098. PMC 55524. PMID 11504945. .

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.
- Lucas, E., *Récréations Mathématiques IV*, Paris, 1921.
- Fleury, "Deux problemes de geometrie de situation", *Journal de mathematiques elementaires* (1883), 257-261.
- T. van Aardenne-Ehrenfest and N. G. de Bruijn, Circuits and trees in oriented linear graphs, *Simon Stevin*, 28 (1951), 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
- W. T. Tutte and C. A. B. Smith, On Unicursal Paths in a Network of Degree 4. *Amer. Math. Monthly*, 48 (1941), 233-237.

External links

- Discussion of early mentions of Fleury's algorithm (<http://mathforum.org/kb/message.jspa?messageID=3648262&tstart=135>)

Hamiltonian path

In the mathematical field of graph theory, a **Hamiltonian path** (or **traceable path**) is a path in an undirected 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 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.^[1]

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.

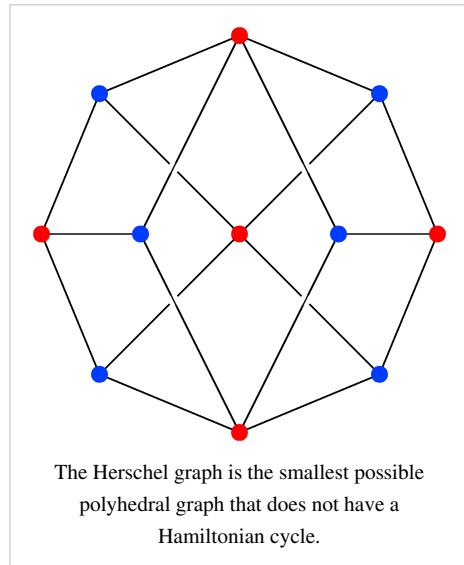
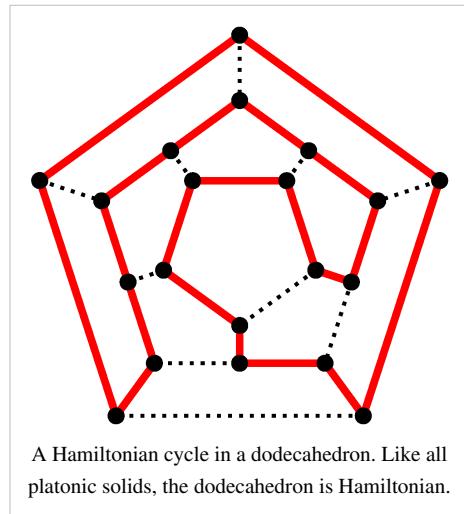
A *Hamiltonian cycle*, *Hamiltonian circuit*, *vertex tour* or *graph cycle* is a cycle that visits each vertex exactly once (except the vertex that is both the start and end, and so is visited twice). A graph that contains a Hamiltonian cycle is called a **Hamiltonian graph**.

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
- every platonic solid, considered as a graph, is Hamiltonian
- every prism is Hamiltonian
- The Deltoidal hexecontahedron is the only non-hamiltonian Archimedean dual
- Every antiprism is Hamiltonian
- A maximal planar graph with no separating triangles is Hamiltonian.



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.

The line graph of a Hamiltonian graph is Hamiltonian. The line graph of an Eulerian graph is Hamiltonian.

A tournament (with more than 2 vertices) is Hamiltonian if and only if it is strongly connected.

The problem of finding a Hamiltonian cycle may be used as the basis of a zero-knowledge proof.

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 each vertex has degree $n / 2$ or greater.^[2]

Ore (1960)

A graph with n vertices ($n \geq 3$) is Hamiltonian if, for each 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 some 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 some two 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.

Notes

- [1] Biggs, N. L. (1981), "T. P. Kirkman, mathematician", *The Bulletin of the London Mathematical Society* **13** (2): 97–120, doi:10.1112/blms/13.2.97, MR608093.
- [2] Graham, p. 20 (http://books.google.com/books?id=XicsNQIrC3sC&printsec=frontcover&source=gbs_summary_r&cad=0#PPA20,M1).

References

- Berge, Claude; Ghouila-Houiri, A. (1962), *Programming, games and transportation networks*, New York: John Wiley & Sons, Inc.
- DeLeon, Melissa, "A Study of Sufficient Conditions for Hamiltonian Cycles (<http://www.rose-hulman.edu/mathjournal/archives/2000/vol1-n1/paper4/v1n1-4pd.PDF>)". Department of Mathematics and Computer Science, Seton Hall University
- 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, MR0047308
- Graham, Ronald L., *Handbook of Combinatorics*, MIT Press, 1995. ISBN 978-0-262-57170-8.
- Hamilton, William Rowan, "Memorandum respecting a new system of roots of unity". *Philosophical Magazine*, 12 1856
- Hamilton, William Rowan, "Account of the Icosian Calculus". Proceedings of the Royal Irish Academy, 6 1858
- Meyniel, M. (1973), "Une condition suffisante d'existence d'un circuit hamiltonien dans un graphe orienté", *Journal of Combinatorial Theory, Ser. B* **14** (2): 137–147, doi:10.1016/0095-8956(73)90057-9
- Ore, O "A Note on Hamiltonian Circuits." *American Mathematical Monthly* 67, 55, 1960.
- Peterson, Ivars, "The Mathematical Tourist". 1988. W. H. Freeman and Company, NY
- Pósa, L. A theorem concerning hamilton lines. *Magyar Tud. Akad. Mat. Kutató Int. Kozl.* 7(1962), 225–226.
- Chuzo Iwamoto and Godfried Toussaint, "Finding Hamiltonian circuits in arrangements of Jordan curves is NP-complete," *Information Processing Letters*, Vol. 52, 1994, pp. 183–189.

External links

- Weisstein, Eric W., " Hamiltonian Cycle (<http://mathworld.wolfram.com/HamiltonianCycle.html>)" from MathWorld.
- Euler tour and Hamilton cycles (<http://www.graph-theory.net/euler-tour-and-hamilton-cycles/>)

Hamiltonian path problem

In the mathematical field of graph theory the **Hamiltonian path problem** and the **Hamiltonian cycle problem** are problems of determining whether a Hamiltonian path or a Hamiltonian cycle exists in a given graph (whether directed or undirected). Both problems are NP-complete.^[1]

Relation between problems

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 degree-1 vertices, 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.

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^[2] 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.^{[3][4]}

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 $O(1.414^n)$.^[5]

For graphs of maximum degree three, a careful backtracking search can find a Hamiltonian cycle (if one exists) in time $O(1.251^n)$.^[6]

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 distinct types of DNA molecule to participate in the reaction.^[7]

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,^[8] for directed planar graphs with indegree and outdegree at most two,^[9] for bridgeless undirected planar 3-regular bipartite graphs, and for 3-connected 3-regular bipartite graphs.^[10] 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.^[11] 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.^[12]

References

- [1] 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. A1.3: GT37–39, pp. 199–200.
- [2] Rubin, Frank (1974), "A Search Procedure for Hamilton Paths and Circuits", *Journal of the ACM* **21**: 576–80.
- [3] Bellman, R. (1962), "Dynamic programming treatment of the travelling salesman problem", *Journal of the ACM* **9**: 61–63, doi:10.1145/321105.321111.
- [4] Held, M.; Karp, R. M. (1962), "A dynamic programming approach to sequencing problems", *J. SIAM* **10** (1): 196–210, doi:10.1137/0110015.
- [5] Björklund, Andreas (2010), "Determinant sums for undirected Hamiltonicity", *Proc. 51st IEEE Symposium on Foundations of Computer Science (FOCS '10)*, pp. 173–182, arXiv:1008.0541, doi:10.1109/FOCS.2010.24.
- [6] Iwama, Kazuo; Nakashima, Takuya (2007), "An improved exact algorithm for cubic graph TSP", *Proc. 13th Annual International Conference on Computing and Combinatorics (COCOON 2007)*, Lecture Notes in Computer Science, **4598**, pp. 108–117, doi:10.1007/978-3-540-73545-8_13.
- [7] Adleman, Leonard (November), "Molecular computation of solutions to combinatorial problems", *Science* **266** (5187): 1021–1024, doi:10.1126/science.7973651, JSTOR 2885489, PMID 7973651.
- [8] Garey, M. R.; Johnson, D. S.; Stockmeyer, L. (1974), "Some simplified NP-complete problems", *Proc. 6th ACM Symposium on Theory of Computing (STOC '74)*, pp. 47–63, doi:10.1145/800119.803884.
- [9] Plesník, J. (1979), "The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree bound two" (<http://www.aya.or.jp/~babalabo/DownLoad/Plesnik 8.4.192-196.pdf>), *Information Processing Letters* **8** (4): 199–201, .
- [10] Akiyama, Takanori; Nishizeki, Takao; Saito, Nobuji (1980–1981), "NP-completeness of the Hamiltonian cycle problem for bipartite graphs" (http://www.nishizeki.ecei.tohoku.ac.jp/nszk/nishi/sub/j/DVD/PDF_J/J029.pdf), *Journal of Information Processing* **3** (2): 73–76, MR596313, .
- [11] Thomason, A. G. (1978), "Hamiltonian cycles and uniquely edge colourable graphs", *Advances in Graph Theory (Cambridge Combinatorial Conf., Trinity College, Cambridge, 1977)*, Annals of Discrete Mathematics, **3**, pp. 259–268, doi:10.1016/S0167-5060(08)70511-9, MR499124.
- [12] Papadimitriou, Christos H. (1994), "On the complexity of the parity argument and other inefficient proofs of existence", *Journal of Computer and System Sciences* **48** (3): 498–532, doi:10.1016/S0022-0000(05)80063-7, MR1279412.

Travelling salesman problem

The **travelling salesman problem (TSP)** is an NP-hard problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city. It is a special case of the travelling purchaser problem.

The problem was first formulated as a mathematical problem 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,^[1] a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

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 make the problem considerably harder.

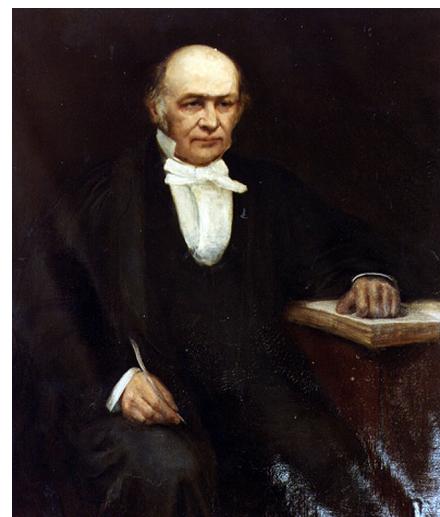
In the theory of computational complexity, the decision version of the TSP (where, given a length L , the task is to decide whether any tour is shorter than L) belongs to the class of NP-complete problems. Thus, it is likely that the worst-case running time for any algorithm for the TSP increases exponentially with the number of cities.

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 defined 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 2005, Cook and others computed an optimal tour through a 33,810-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 1% of an optimal tour.

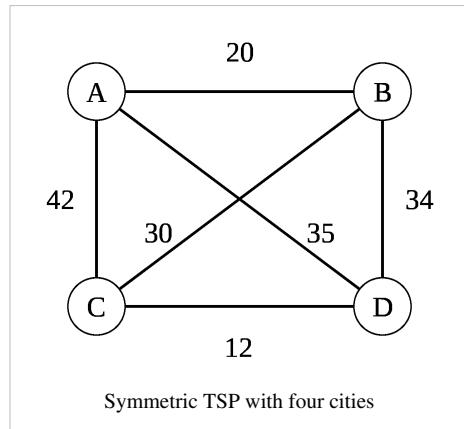
Description

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 travelling 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 travelling 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 "travelling 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 7054798^[6]. Surprisingly, Behzad and Modarres^[7] demonstrated that the generalised travelling salesman problem can be transformed into a standard travelling 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 travelling 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.

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.^[8] If the distance measure is a metric and symmetric, the problem becomes APX-complete^[9] and Christofides's algorithm approximates it within 1.5.^[10]

If the distances are restricted to 1 and 2 (but still are a metric) the approximation ratio becomes 7/6. In the asymmetric, metric case, only logarithmic performance guarantees are known, the best current algorithm achieves performance ratio $0.814 \log n$,^[11] 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.^[12] If the distance function is symmetric, the longest tour can be approximated within 4/3 by a deterministic algorithm^[13] and within $(33 + \epsilon)/25$ by a randomised algorithm.^[14]

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 $O(n^2 2^n)$.^[15]

The dynamic programming solution requires exponential space. Using inclusion–exclusion, the problem can be solved in time within a polynomial factor of 2^n and polynomial space.^[16]

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.^[17]

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; 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.^[18]

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 neighbour (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.^[19] 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.^[20] The NF operator can also be applied on an initial solution obtained by NN algorithm for further improvement in an elitist model, where better solutions are only 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 [21]), 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, or Lin–Kernighan heuristics

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 a more general method.

k-opt heuristic

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

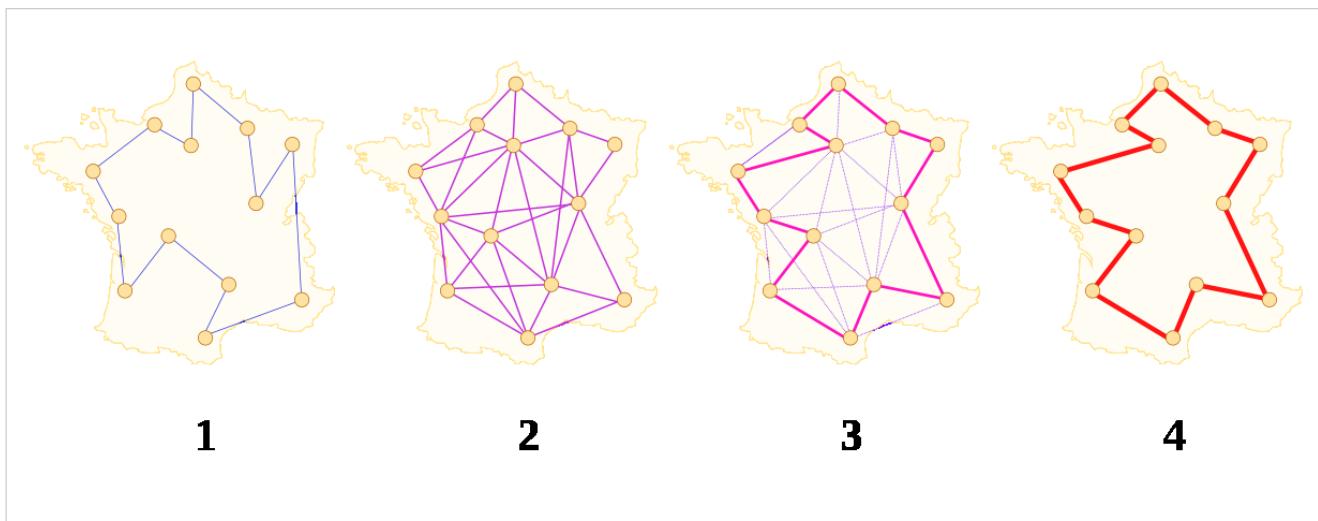
Optimized Markov chain algorithms which use local searching heuristic sub-algorithms can find a route extremely close to the optimal route for 700 to 800 cities.

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

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).^[22] 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, i.e., they satisfy the triangle inequality. This can be understood as the absence of "shortcuts", in the sense that the direct connection from A to B is never longer than the route via intermediate C :

$$d_{AB} \leq d_{AC} + d_{CB}$$

The edge lengths then form 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 length of the minimum spanning tree of the network G is a natural lower bound for the length 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 length 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 length is twice the length 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.

In the special case that distances between cities are all either one or two (and thus the triangle inequality is necessarily satisfied), there is a polynomial-time approximation algorithm that finds a tour of length at most $8/7$ times the optimal tour length.^[23] However, it is a long-standing (since 1975) open problem to improve the Christofides approximation factor of 1.5 for general metric TSP to a smaller constant. It is known that, unless $P = NP$, there is no polynomial-time algorithm that finds a tour of length at most $220/219=1.00456\dots$ times the optimal tour's length.^[24] In the case of bounded metrics it is known that there is no polynomial time algorithm that constructs a tour of length at most $321/320$ times the optimal tour's length, unless $P = NP$.^[25]

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 (and therefore the general metric TSP) is NP-complete.^[26] 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).^[27] 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 $2N$.

	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 $-\infty$.

	<i>A</i>	<i>B</i>	<i>C</i>	<i>A'</i>	<i>B'</i>	<i>C'</i>
<i>A</i>				$-\infty$	6	5
<i>B</i>				1	$-\infty$	4
<i>C</i>				2	3	$-\infty$
<i>A'</i>	$-\infty$	1	2			
<i>B'</i>	6	$-\infty$	3			
<i>C'</i>	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**^[28] 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 is observed that humans are able to produce good quality solutions quickly. The first issue of the Journal of Problem Solving^[29] is devoted to the topic of human performance on TSP.

TSP path length for random pointset in a square

Suppose N points are randomly distributed in a 1×1 square with $N \gg 1$. Consider many such squares. Suppose we want to know the average of the shortest path length (i.e. TSP solution) of each square.

Lower bound

$\frac{1}{2}\sqrt{N}$ is a lower bound obtained by assuming i be a point in the tour sequence and i has its nearest neighbor as its next in the path.

$\left(\frac{1}{4} + \frac{3}{8}\right)\sqrt{N}$ is a better lower bound obtained by assuming *is next is* is nearest, and *is previous is* is second nearest.

$\sqrt{\frac{N}{2}}$ is an even better lower bound obtained by dividing the path sequence into two parts as *before_i* and *after_i* with each part containing $N/2$ points, and then deleting the *before_i* part to form a diluted pointset (see discussion).

- David S. Johnson^[30] obtained a lower bound by computer experiment:
 $0.7080\sqrt{N} + 0.522$, where 0.522 comes from the points near square boundary which have fewer neighbors.
- Christine L. Valenzuela and Antonia J. Jones^[31] obtained another lower bound by computer experiment:
 $0.7078\sqrt{N} + 0.551$

Upper bound

By applying Simulated Annealing method on samples of $N=40000$, computer analysis shows an upper bound of

$$\left(\sqrt{\frac{N}{2}} + 0.72\right) \cdot 1.015, \text{ where } 0.72 \text{ comes from the boundary effect.}$$

Because the actual solution is only the shortest path, for the purposes of programmatic search another upper bound is the length of any previously discovered approximation.

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.

Popular Culture

Travelling Salesman, by director Timothy Lanzone, is the story of 4 mathematicians hired by the US Government to solve the most elusive problem in computer-science history: P vs. NP.^[32]

Notes

- [1] <http://www.mjc2.com/logistics-planning-complexity.htm> Why is vehicle routing hard - a simple explanation
- [2] "Der Handlungsreisende – wie er sein soll und was er zu thun [sic] 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 be sure to perform his tasks and have success in his business — by a high *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 Schrijver (2005). 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?vid=7054798>
- [7] Behzad, Arash; Modarres, Mohammad (2002), "New Efficient Transformation of the Generalized Traveling Salesman Problem into Traveling Salesman Problem", *Proceedings of the 15th International Conference of Systems Engineering (Las Vegas)*
- [8] Orponen (1987)
- [9] Papadimitriou (1983)
- [10] Christofides (1976)
- [11] Kaplan (2004)
- [12] Kosaraju (1994)
- [13] Serdyukov (1984)
- [14] Hassin (2000)
- [15] Bellman (1960), Bellman (1962), Held & Karp (1962)

- [16] Kohn (1977) Karp (1982)
- [17] Woeginger (2003)
- [18] Work by David Applegate, AT&T Labs – Research, Robert Bixby, ILOG and Rice University, Vašek Chvátal, Concordia University, William Cook, Georgia Tech, and Keld Helsgaun, Roskilde University is discussed on their project web page hosted by Georgia Tech and last updated in June 2004, here (<http://www.tsp.gatech.edu/sweden/>)
- [19] 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
- [20] 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.
- [21] 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>
- [22] 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>
- [23] P. Berman (2006). M. Karpinski, "8/7-Approximation Algorithm for (1,2)-TSP", Proc. 17th ACM-SIAM SODA (2006), pp. 641–648, ECCC TR05-069.
- [24] C.H. Papadimitriou and Santosh Vempala. On the approximability of the traveling salesman problem (<http://dx.doi.org/10.1007/s00493-006-0008-z>), *Combinatorica* 26(1):101–120, 2006.
- [25] L. Engebretsen, M. Karpinski, TSP with bounded metrics (<http://dx.doi.org/10.1016/j.jcss.2005.12.001>). *Journal of Computer and System Sciences*, 72(4):509–546, 2006.
- [26] Christos H. Papadimitriou. "The Euclidean travelling salesman problem is NP-complete". *Theoretical Computer Science* 4:237–244, 1977. doi:10.1016/0304-3975(77)90012-3
- [27] Sanjeev Arora. Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems. *Journal of the ACM*, Vol.45, Issue 5, pp.753–782. ISSN:0004-5411. September 1998. <http://citeseer.ist.psu.edu/arora96polynomial.html>.
- [28] <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>
- [29] <http://docs.lib.psu.edu/jps/>
- [30] David S. Johnson (<http://www.research.att.com/~dsj/papers/HKsoda.pdf>)
- [31] Christine L. Valenzuela and Antonia J. Jones (<http://users.cs.cf.ac.uk/Antonia.J.Jones/Papers/EJORHeldKarp/HeldKarp.pdf>)
- [32] Geere, Duncan. "'Travelling Salesman' movie considers the repercussions if P equals NP" (<http://www.wired.co.uk/news/archive/2012-04/26/travelling-salesman>). Wired. . Retrieved 26 April 2012.

References

- Applegate, D. L.; Bixby, R. M.; Chvátal, V.; Cook, W. J. (2006), *The Traveling Salesman Problem*, ISBN 0-691-12993-2.
- 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.
- 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.
- 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.
- 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.
- Karp, R.M. (1982), "Dynamic programming meets the principle of inclusion and exclusion", *Oper. Res. Lett.* **1** (2): 49–51, doi:10.1016/0167-6377(82)90044-X.
- Kohn, S.; Gottlieb, A.; Kohn, M. (1977), "A Generating Function Approach to the Traveling Salesman Problem", *ACM Annual Conference*, ACM Press, pp. 294–300.
- 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*.
- 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.
- 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>)
- Applegate, D. L.; Bixby, R. E.; Chvátal, V.; Cook, W. J. (2006), *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, ISBN 978-0-691-12993-8.
- 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.
- 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.
- 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, JSTOR 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*, New York: Addison-Wesley, 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.
- 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.
- 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.
- 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.
 - 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, PMID 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

- Traveling Salesman Problem (<http://www.tsp.gatech.edu/index.html>) at Georgia Tech
- 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
- *optimap* (<http://www.gebweb.net/optimap/>) an approximation using ACO on GoogleMaps with JavaScript
- *tsp* (<http://travellingsalesmanproblem.appspot.com/>) an exact solver using Constraint Programming on GoogleMaps
- Demo applet of a genetic algorithm solving TSPs and VRPTW problems (<http://www.dna-evolutions.com/dnaappletsample.html>)
- 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 Georgia Tech's Concorde and Chained Lin-Kernighan heuristics.
- Traveling Salesman (on IMDB) (<http://www.imdb.com/title/tt1801123/>)
- Traveling Salesman Movie (<http://www.travellingsalesmanmovie.com/>) Official webpage of Traveling Salesman film (2012)
- C++ implementation of simulated annealing applied to the travelling salesman problem (<http://www.technical-recipes.com/2012/c-implementation-of-hill-climbing-and-simulated-annealing-applied-to-travelling-salesman-problems/>)

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.^[1]

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] 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. A2.3: ND24, pg.212.
- [2] R. Garey Parker and Ronald L. Rardin (1984). *Guaranteed performance heuristics for the bottleneck traveling salesman problem*. Operations Research Letters. 2(6):269–272

Christofides' heuristic for the TSP

The goal of the **Christofides heuristic 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 the minimum spanning tree MST 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.

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 Mei-Ku Kuan in 1962.^[1]

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 whose odd-degree vertices are the vertices in T is called a **T -join**. (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 empty set, 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.^[2]

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.

Variants

A few variants of the Chinese Postman Problem have been studied and shown to be NP-complete.^[3]

- 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] ""Chinese Postman Problem"" (<http://www.nist.gov/dads/HTML/chinesePostman.html>). .
- [2] J. Edmonds and E.L. Johnson, Matching Euler tours and the Chinese postman problem, Math. Program. (1973).
- [3] Crescenzi, P.; Kann, V.; Halldórsson, M.; Karpinski, M.; Woeginger, G. "A compendium of NP optimization problems" (<http://www.nada.kth.se/~viggo/problemst/compendium.html>). KTH NADA, Stockholm. . Retrieved 2008-10-22.

External links

- Chinese Postman Problem (<http://mathworld.wolfram.com/ChinesePostmanProblem.html>)

Matching

Matching

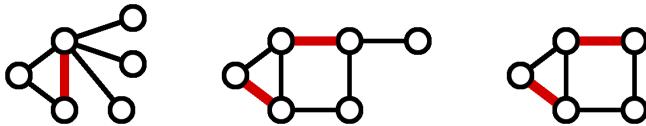
In the mathematical discipline of graph theory, a **matching** or **independent edge set** in a graph is a set of edges without common vertices. It may also be an entire graph consisting of edges without common vertices.

Definition

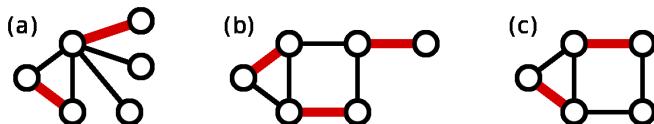
Given a graph $G = (V, E)$, a **matching** M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.

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** 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 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.^[1] 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

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

Maximum matchings in bipartite graphs

Matching problems are often concerned with bipartite graphs. Finding a **maximum bipartite matching**^[2] (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 $O(\sqrt{VE})$ time. Another approach is based on the fast matrix multiplication algorithm and gives $O(V^{2.376})$ complexity,^[3]

which is better in theory for sufficiently dense graphs, but in practice the algorithm is slower.

In a weighted bipartite graph, each edge has an associated value. A **maximum weighted bipartite matching**^[2] 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. It can be solved by using a modified shortest path search in the augmenting path algorithm. If the Bellman-Ford algorithm is used, the running time becomes $O(V^2 E)$, 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.^[4] The remarkable Hungarian algorithm solves the assignment problem and it was one of the beginnings of combinatorial

optimization algorithms. The original approach of this algorithm needs $O(V^2E)$ running time, but it could be improved to $O(\sqrt{VE})$ time with extensive use of priority queues.

Maximum matchings

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.^[5] Another algorithm by Mucha and Sankowski,^[3] 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.^[6] 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.^[7] Both problems can be approximated within factor 2 in polynomial time: simply find an arbitrary maximal matching M .^[8]

Counting problems

The problem of determining the number of perfect matchings in a given graph is #P Complete (see Permanent). However, 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. There exists a fully polynomial time randomized approximation scheme for counting the number of bipartite matchings.^[9]

For the problem of determining the total number of matchings in a given graph, see Hosoya index.

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 $O(VE)$ ^[10]. There exists a randomized algorithm that solves this problem in time $\tilde{O}(V^{2.376})$ ^[11]. 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^[12]; 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^[13], 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.^[14]

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] Gallai, Tibor (1959), "Über extreme Punkt- und Kantenmengen", *Ann. Univ. Sci. Budapest. Eötvös Sect. Math.* **2**: 133–138.
- [2] West, Douglas Brent (1999), *Introduction to Graph Theory* (2nd ed.), Prentice Hall, Chapter 3, ISBN 0-13-014400-2
- [3] Mucha, M.; Sankowski, P. (2004), "Maximum Matchings via Gaussian Elimination" (http://www.mimuw.edu.pl/~muchka/pub/mucha_sankowski_focs04.pdf), *Proc. 45st IEEE Symp. Foundations of Computer Science*, pp. 248–255,
- [4] Fredman, M.; Tarjan, R. (1987), "Fibonacci heaps and their uses in improved network optimization algorithms" (<http://dl.acm.org/citation.cfm?id=28874>), *Journal of the ACM (JACM)*, **34**, pp. 596–615,
- [5] 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.
- [6] Yannakakis, Mihalis; Gavril, Fanica (1980), "Edge dominating sets in graphs", *SIAM Journal on Applied Mathematics* **38** (3): 364–372, doi:10.1137/0138030.
- [7] 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. 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.
- [8] Ausiello, Giorgio; Crescenzi, Pierluigi; Gambosi, Giorgio; Kann, Viggo; Marchetti-Spaccamela, Alberto; Protasi, Marco (2003), *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*, Springer. 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/>).
- [9] Bezákova, Ivona; Štefankovič, Daniel; Vazirani, Vijay V.; Vigoda, Eric (2008). "Accelerating Simulated Annealing for the Permanent and Combinatorial Counting Problems". *SIAM Journal on Computing* **37** (5): 1429–1454. doi:10.1137/050644033.
- [10] de Carvalho, Marcelo H.; Cheriyan, Joseph (2005), "An $O(VE)$ algorithm for ear decompositions of matching-covered graphs", *SODA*, pp. 415–423.
- [11] Rabin, Michael O.; Vazirani, Vijay V. (1989), "Maximum matchings in general graphs through randomization", *J. of Algorithms* **10**: 557–567.
- [12] Tassa, Tamir (2012), "Finding all maximally-matchable edges in a bipartite graph", *Theoretical Computer Science* **423**: 50–58, doi:10.1016/j.tcs.2011.12.071.
- [13] Gionis, Aris; Mazza, Arnon; Tassa, Tamir (2008), "k-Anonymization revisited", *International Conference on Data Engineering (ICDE)*, pp. 744–753.
- [14] See, e.g., Trinajstić, Nenad; Klein, Douglas J.; Randić, Milan (1986), "On some solved and unsolved problems of chemical graph theory", *International Journal of Quantum Chemistry* **30** (S20): 699–742, doi:10.1002/qua.560300762.

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).
4. Michael L. Fredman and Robert E. Tarjan (1987), "Fibonacci heaps and their uses in improved network optimization algorithms", *Journal of the ACM* (ACM Press) **34** (3): 595–615, doi:10.1145/28869.28874.
5. S. J. Cyvin and Ivan Gutman (1988), *Kekulé Structures in Benzenoid Hydrocarbons*, Springer-Verlag

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 $O(m\sqrt{n})$ time in the worst case, where "O" refers to big O notation, m is the number of edges in the graph, and n is the number of vertices of the graph. In the case of dense graphs the time bound becomes $O(n^{5/2})$, 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 of size n , 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 $n + 1$. 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 unmatched and matched. 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 unmatched and matched 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{n} phases, in a graph with n vertices and m edges, take time $O(m\sqrt{n})$.

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{n} phases of the algorithm are complete, the shortest remaining augmenting path has at least \sqrt{n} 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{n} , there can be at most \sqrt{n} paths in the collection, and the size of the optimal matching can differ from the size of M by at most \sqrt{n} edges. Since each phase of the algorithm increases the size of the matching by at least one, there can be at most \sqrt{n} additional phases before the algorithm terminates.

Since the algorithm performs a total of at most $2\sqrt{n}$ phases, it takes a total time of $O(m\sqrt{n})$ 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 $O(\log n)$ phases and $O(m \log n)$ total 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, $O(n^{1.5}(m/\log n)^{0.5})$. 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 $O(\sqrt{n})$ 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; alternative methods for this problem were later described by other authors.^[3]

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)
        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] Ahuja, Magnanti & Orlin (1993), section 12.3, bipartite cardinality matching problem, pp. 469–470.
- [2] Chang & McCormick (1990); Darby-Dowman (1980); Setubal (1993); Setubal (1996).
- [3] Gabow & Tarjan (1989) and Blum (2001).

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 $\mathcal{O}(n^{1.5} \sqrt{\frac{m}{\log n}})$ ", *Information Processing Letters* **37** (4): 237–240, doi: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.
- 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, MR0177907.

- 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.
- 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.
- 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.
- 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.
- 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.

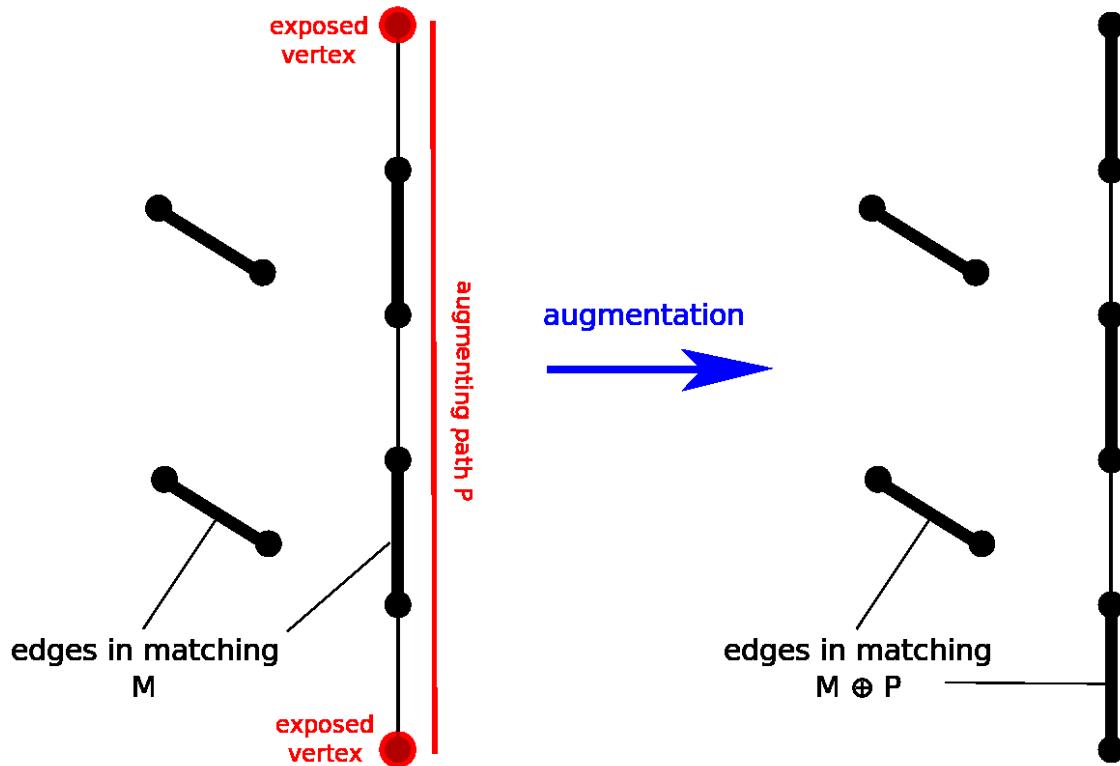
Edmonds's algorithm for maximum matching in non-bipartite graphs

The **blossom algorithm** is an algorithm in graph theory for constructing maximum matchings on graphs. The algorithm was discovered by Jack Edmonds in 1961,^[1] and published in 1965.^[2] 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.^[3] 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."^[4]

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 $M_1 = M \oplus P = (M \setminus P) \cup (P \setminus M)$.



We can prove^{[5][6]} 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

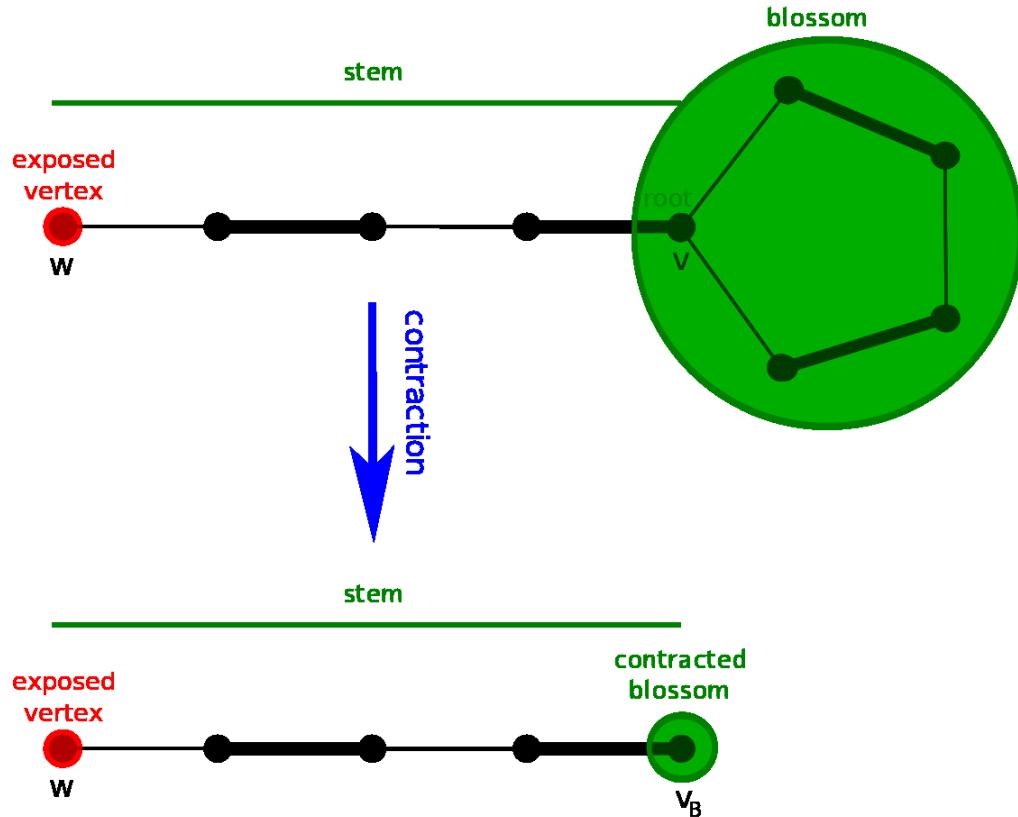
```

We still have to describe how augmenting paths can be found efficiently. The subroutine to find them uses blossoms and contractions.

Blossoms and contractions

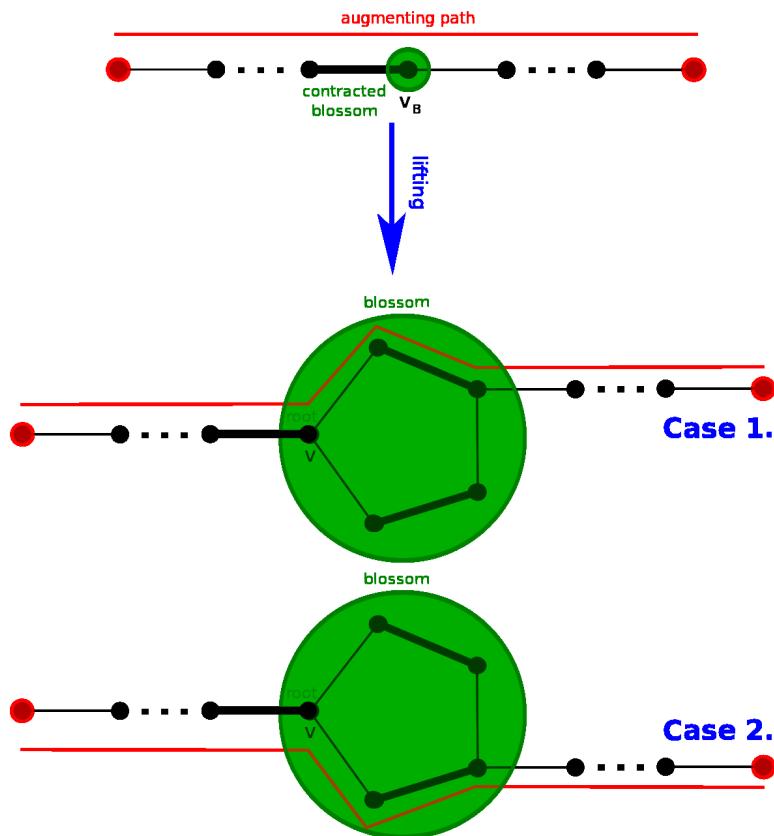
Given $G = (V, E)$ and a matching M of G , a *blossom* B is a cycle in G consisting of $2k + 1$ edges of which exactly k belong to M , and where one of the vertices v of the cycle (the *base*) is such that there exists an alternating path of even length (the *stem*) from v to an exposed vertex w .

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 .

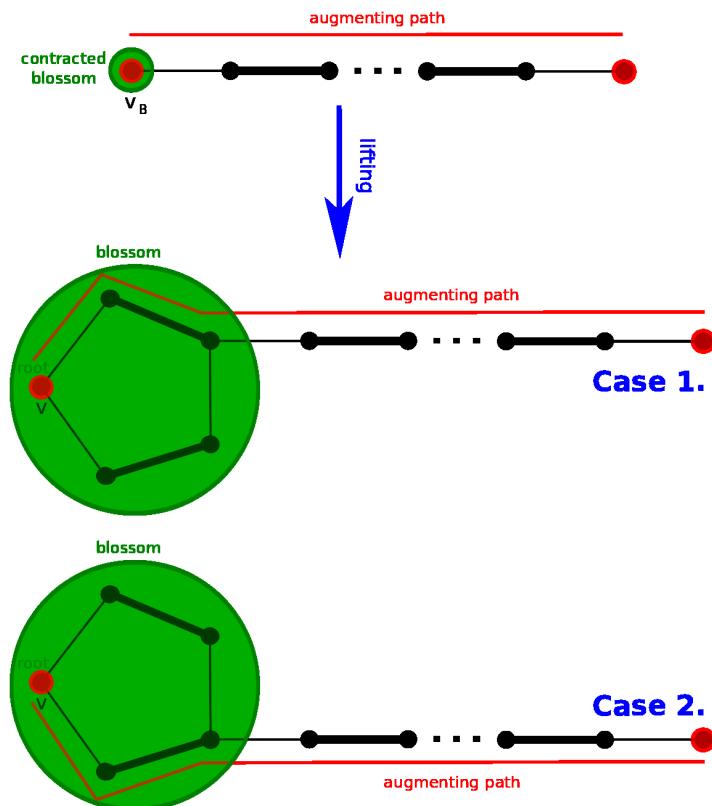


It can be shown^[7] that G' has an M' -augmenting path iff G has an M -augmenting path, and that any M' -augmenting path P' in G' can be **lifted** to a M -augmenting path in G by undoing the contraction by B so that the segment of P' (if any) traversing through v_B is replaced by an appropriate segment 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's algorithm.

Finding an augmenting path

The search for 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 iterations 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.^[7]

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 .

```

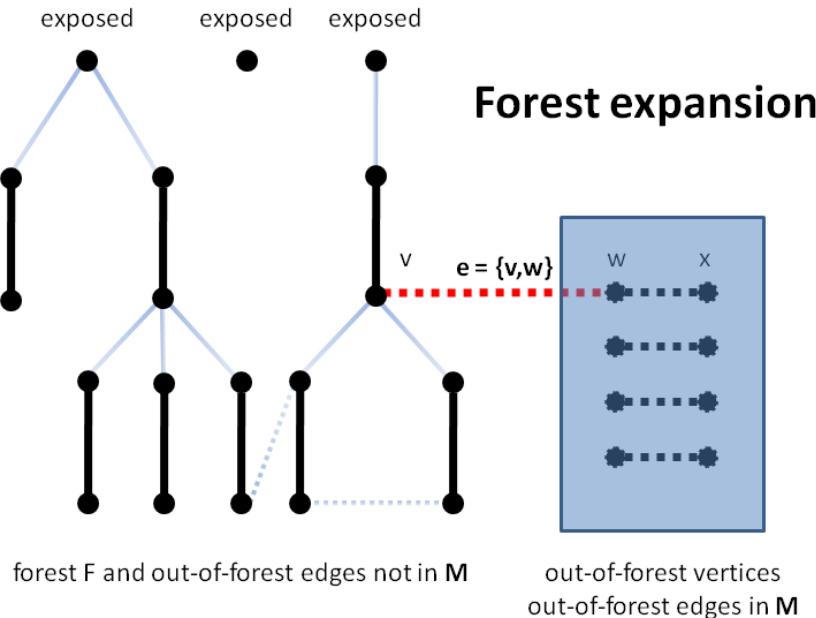
INPUT: Graph  $G$ , matching  $M$  on  $G$ 
OUTPUT: augmenting path  $P$  in  $G$  or empty path if none found

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         // Update  $F$ .
B12          $x \leftarrow$  vertex matched to  $w$  in  $M$ 
B13         add edges {  $v, w$  } and {  $w, x$  } to the tree of  $v$ 
B14       else
B15         if  $\text{distance}( w, \text{root}( w ) )$  is odd then
B16           do nothing
B17         else
B18           if  $\text{root}( v ) \neq \text{root}( w )$  then
B19             // Report an augmenting path in  $F \cup \{ e \}$ .
B20              $P \leftarrow$  path (  $\text{root}( v ) \rightarrow \dots \rightarrow v$  )  $\rightarrow$  (  $w \rightarrow \dots \rightarrow \text{root}( w )$  )
B21             return  $P$ 
B22       else
B23         // Contract a blossom in  $G$  and look for the path in the contracted graph.
B24          $B \leftarrow$  blossom formed by  $e$  and edges on the path  $v \rightarrow w$  in  $T$ 
B25          $G', M' \leftarrow$  contract  $G$  and  $M$  by  $B$ 
B26          $P' \leftarrow$  find_augmenting_path(  $G', M'$  )
B27          $P \leftarrow$  lift  $P'$  to  $G$ 
B28         return  $P$ 
B29       end if
B30     end while
B31   mark vertex  $v$ 
B32 end while
B33 return empty path
B34 end function

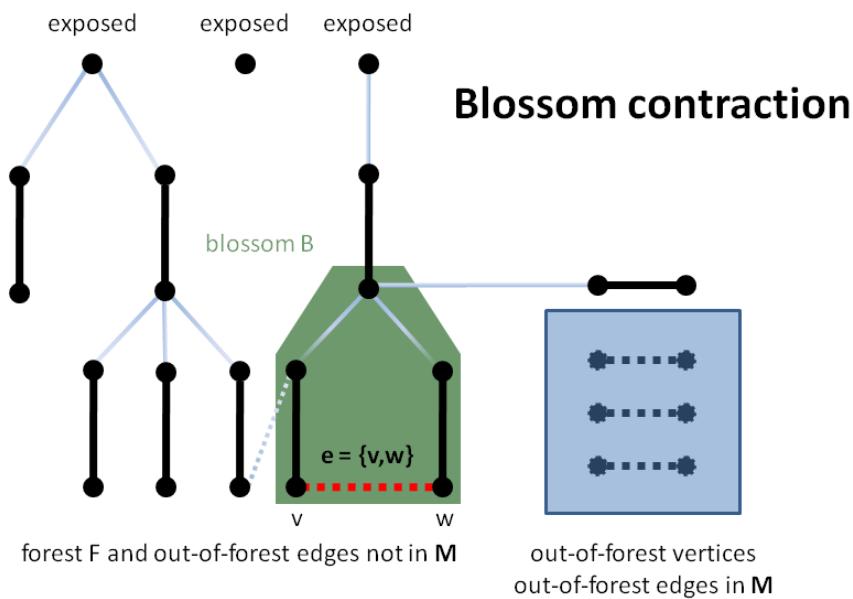
```

Examples

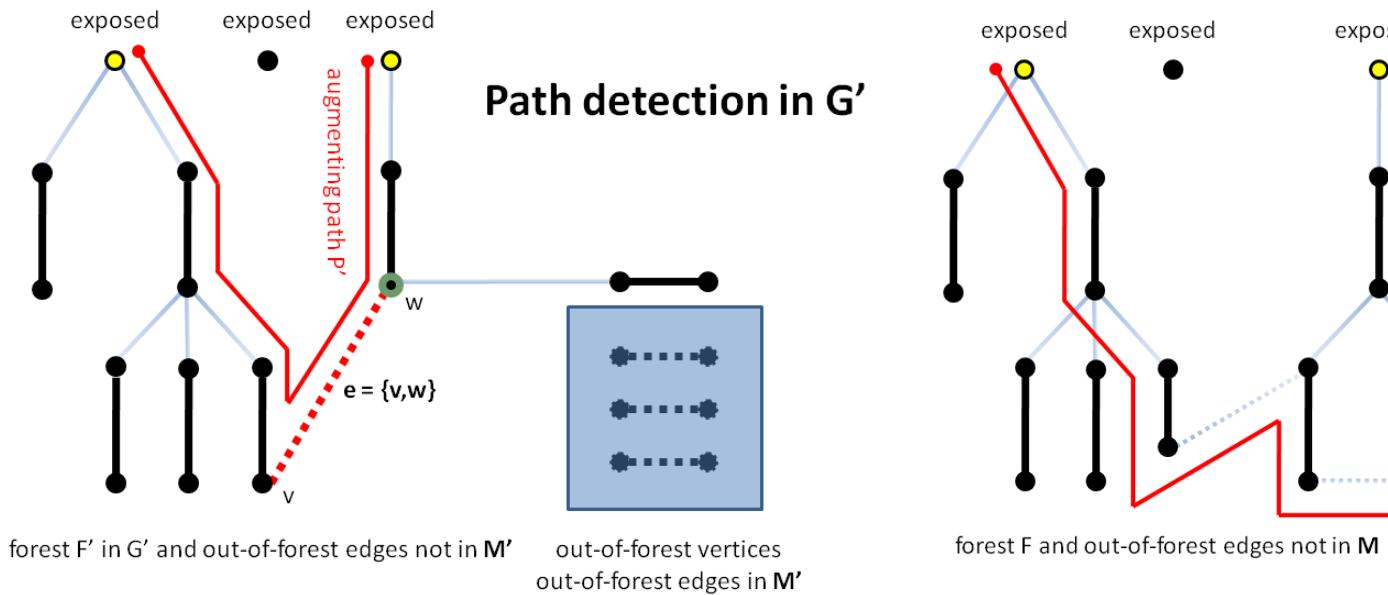
The following four figures illustrate the execution of the algorithm. We use dashed lines to indicate edges that are currently not present in the forest. First, the algorithm processes an out-of-forest edge that causes the expansion of the current forest (lines B10 – B12).



Next, it detects a blossom and contracts the graph (lines B20 – B22).



Finally, it locates an augmenting path P' in the contracted graph (line B23) and lifts it to the original graph (line B24). 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.^[8]

- 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 B21). It is easy to see that the running time is $O(|V|^4)$. Micali and Vazirani^[9] 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^[6] 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 B21 – B29 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.^[5] Kolmogorov provides an efficient C++ implementation of this.^[10]

References

- [1] Edmonds, Jack (1991), "A glimpse of heaven", in J.K. Lenstra, A.H.G. Rinnooy Kan, A. Schrijver, ed., *History of Mathematical Programming --- A Collection of Personal Reminiscences*, CWI, Amsterdam and North-Holland, Amsterdam, pp. 32-54
- [2] Edmonds, Jack (1965). "Paths, trees, and flowers". *Canad. J. Math.* **17**: 449–467. doi:10.4153/CJM-1965-045-4.
- [3] Edmonds, Jack (1965). "Maximum matching and a polyhedron with 0,1-vertices". *Journal of Research National Bureau of Standards Section B* **69**: 125–130.
- [4] Schrijver, Alexander. *Combinatorial Optimization: Polyhedra and Efficiency*. Algorithms and Combinatorics. **24**. Springer.
- [5] Lovász, László; Plummer, Michael (1986). *Matching Theory*. Akadémiai Kiadó. ISBN 963-05-4168-8.
- [6] Karp, Richard, "Edmonds's Non-Bipartite Matching Algorithm" (<http://www.cs.berkeley.edu/~karp/greatalgo/lecture05.pdf>), *Course Notes. U. C. Berkeley*,
- [7] Tarjan, Robert, "Sketchy Notes on Edmonds' Incredible Shrinking Blossom Algorithm for General Matching" (<http://www.cs.dartmouth.edu/~ac/Teach/CS105-Winter05/Handouts/tarjan-blossom.pdf>), *Course Notes, Department of Computer Science, Princeton University*,
- [8] Kenyon, Claire; Lovász, László, "Algorithmic Discrete Mathematics", *Technical Report CS-TR-251-90, Department of Computer Science, Princeton University*
- [9] Micali, Silvio; Vazirani, Vijay (1980), "An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs", *21st Annual Symposium on Foundations of Computer Science*, (IEEE Computer Society Press, New York): 17–27
- [10] Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm" (<http://www.cs.ucl.ac.uk/staff/V.Kolmogorov/papers/BLOSSOM5.html>), *Mathematical Programming Computation* **1** (1): 43–67,

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 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

- Burkard, Rainer; M. Dell'Amico, S. Martello (2012). *Assignment Problems (Revised reprint)*. SIAM. ISBN 978-1-61197-222-1.

Hungarian algorithm for the assignment problem

The **Hungarian method** is a combinatorial optimization algorithm which 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 Kőnig and Jenő Egerváry.

James Munkres reviewed the algorithm in 1957 and observed that it is (strongly) polynomial. 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.^[1]

Layman's explanation

Say you have three workers: **Jim**, **Steve** and **Alan**. You need to have one of them clean the bathroom, another sweep the floors & 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	\$1	\$2	\$3
Steve	\$3	\$3	\$3
Alan	\$3	\$3	\$2

Then the Hungarian algorithm, 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.^[2]

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'	d3'	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 till a closed loop is obtained.

x				
0	a2'	a3'	a4'	x
b1'	b2'	b3'	0'	
0'	c2'	c3'	c4'	x
d1'	0'	d3'	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'	d3'	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 1–4) 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
- Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterly*, 2:83–97, 1955. Kuhn's original publication.
- Harold W. Kuhn, "Variants of the Hungarian method for assignment problems", *Naval Research Logistics Quarterly*, 3: 253–258, 1956.
- J. Munkres, "Algorithms for the Assignment and Transportation Problems", *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957 March.
- M. Fischetti, "Lezioni di Ricerca Operativa", Edizioni Libreria Progetto Padova, Italia, 1995.
- R. Ahuja, T. Magnanti, J. Orlin, "Network Flows", Prentice Hall, 1993.

References

- [1] <http://www.lix.polytechnique.fr/~ollivier/JACOBI/jacobiEngl.htm>
- [2] Beryl Castello, The Hungarian Algorithm (<http://www.ams.jhu.edu/~castello/362/Handouts/hungarian.pdf>)

External links

- 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.

Implementations

(Note that not all of these satisfy the $O(n^3)$ time constraint.)

- Java implementation (<http://konstantinosnedas.com/dev/soft/munkres.htm>)
- Python implementation (<http://software.clapper.org/munkres/>) (see also here (<http://www.enseignement.polytechnique.fr/informatique/INF441/INF441b/code/kuhnMunkres.py>))
- Ruby implementation with unit tests (<http://github.com/evansenter/gene/blob/f515fd73cb9d6a22b4d4b146d70b6c2ec6a5125b/objects/extensions/hungarian.rb>)
- C# implementation (<http://noldorin.com/blog/2009/09/hungarian-algorithm-in-csharp/>)
- 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 (<http://search.cpan.org/perldoc?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>)

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?^[1] 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.^[2] In the early 1960s, the definition of *exactly solvable* was not rigorous.^[3] 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.^[4] 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^[5] and Temperley-Fisher^[6] 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.^{[7][8]}

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 $\text{pf}(B) = \text{PerfMatch}(G)$.^[9] 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 .

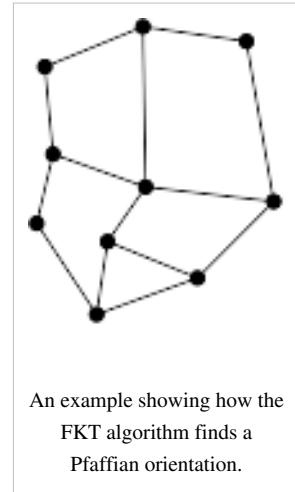
Finally, for any skew-symmetric matrix A ,

$$\text{pf}(A)^2 = \det(A),$$

where $\det(A)$ is the determinant of A . 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
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 which do not contain a subgraph homeomorphic to $K_{3,3}$ ^[10]. 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.^[3] 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.^[11]

References

- [1] Hayes, Brian (2008 January–February), "Accidental Algorithms" (<http://www.americanscientist.org/issues/pub/accidental-algorithms>), *American Scientist*,
- [2] Baxter, R. J. (2008) [1982]. *Exactly Solved Models in Statistical Mechanics* (<http://tpsrv.anu.edu.au/Members/baxter/book>) (Third ed.). Dover Publications. p. 11. ISBN 978-0-486-46271-4. .
- [3] Cai, Jin-Yi; Lu, Pinyan; Xia, Mingji (2010). "Holographic Algorithms with Matchgates Capture Precisely Tractable Planar #CSP". Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on (<http://www.egr.unlv.edu/~larmore/FOCS/focs2010/>). Las Vegas, NV, USA: IEEE. arXiv:1008.0683.
- [4] Kenyon, Richard; Okounkov, Andrei (2005). "What is a Dimer?" (<http://www.ams.org/notices/200503/what-is.pdf>). *AMS* **52** (3): 342–343. .
- [5] Kasteleyn, P. W. (1961), "The statistics of dimers on a lattice. I. The number of dimer arrangements on a quadratic lattice", *Physica* **27** (12): 1209–1225, doi:10.1016/0031-8914(61)90063-5
- [6] Temperley, H. N. V.; Fisher, Michael E. (1961). "Dimer problem in statistical mechanics-an exact result". *Philosophical Magazine* **6** (68): 1061–1063. doi:10.1080/14786436108243366.
- [7] Kasteleyn, P. W. (1963). "Dimer Statistics and Phase Transitions". *Journal of Mathematical Physics* **4** (2): 287–293. doi:10.1063/1.1703953.
- [8] Kasteleyn, P. W. (1967), "Graph theory and crystal physics", in Harary, F., *Graph Theory and Theoretical Physics*, New York: Academic Press, pp. 43–110
- [9] Thomas, Robin (2006). "A survey of Pfaffian orientations of graphs" (<http://people.math.gatech.edu/~thomas/PAP/pfafsuv.pdf>). **III**. International Congress of Mathematicians (<http://www.icm2006.org/>). Zurich: European Mathematical Society. pp. 963–984. .
- [10] 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.
- [11] Valiant, Leslie G. (2004). "Holographic Algorithms (Extended Abstract)". *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*. FOCS'04 (<http://www.cs.brown.edu/~aris/focs04/>). Rome, Italy: IEEE Computer Society. pp. 306–315. doi:10.1109/FOCS.2004.34. ISBN 0-7695-2228-9.

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 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:

- 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 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 a 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]

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]}

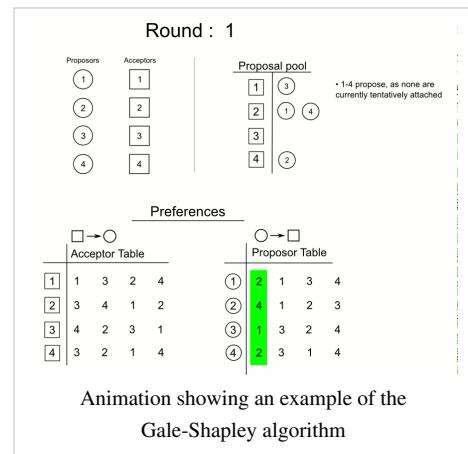
The **Gale-Shapley algorithm** involves a number of "rounds" (or "iterations") where each unengaged man "proposes" to the most-preferred woman to whom he has not yet proposed. Each woman then considers all her suitors and tells the one she most prefers "Maybe" and all the rest of them "No". She is then provisionally "engaged" to the suitor she most prefers so far, and that suitor is likewise provisionally engaged to her. 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. 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 such 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 **weighted matching 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]

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.

Textbooks and other important references not cited in the text

- Dubins, L., and Freedman, D. (1981). Machiavelli and the Gale-Shapley algorithm. *The 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, especially in the fields of game theory and combinatorics, 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 does not require that a set is broken up into male and female subsets. Any person can prefer anyone in the same set.

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 (unordered) 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, the stable roommates may not, in general, have a solution. For a minimal counterexample, consider 4 people A, B, C and D such that the 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,C is the most favorite of someone. In any solution, one of A,B,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 (see below).

The algorithm consists of two phases. In the first phase, 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 – this situation can be represented as a set S of ordered pairs of the form (p,q) , where q holds a proposal from p – we say that q is p 's current *favorite*. In the case that this set represents a matching, i.e., (q,p) is in S whenever (p,q) is, the algorithm terminates with this matching, which is bound to be stable.

Otherwise the algorithm enters Phase 2, in which the set S is repeatedly changed by the application of so-called rotations. Suppose that (p,q) is in the set S but (q,p) is not. For each such p we identify his current *second favorite* to be the first successor of q in p 's preference list who would reject the proposal that he holds in favor of p . A *rotation* relative to S is a sequence $(p_0, q_0), (p_1, q_1), \dots, (p_{k-1}, q_{k-1})$ such that (p_i, q_i) is in S for each i , and q_{i+1} is p_i 's current second favorite (where $i + 1$ is taken modulo k). If, such a rotation $(p_0, q_0), \dots, (p_{2k}, q_{2k})$, of odd length, is found such that $p_i = q_{i+k+1}$ for all i (where $i + k + 1$ is taken modulo $2k + 1$), this is what is referred to as an *odd party*, which is also an indicator that there is no stable matching. Otherwise, application of the rotation involves replacing the pairs (p_i, q_i) , in S by the pairs (p_i, q_{i+1}) , (where $i + 1$ is again taken modulo k).

Phase 2 of the algorithm can now be summarized as follows:

```

S = output of Phase 1;
while (true) {
    identify a rotation r in S;
    if (r is an odd party)
        return null; (there is no stable matching)
    else
        apply r to S;
        if (S is a matching)
            return S; (guaranteed to be stable)
}

```

Example

The following are the preference lists for a Stable Roommates instance involving 6 participants $p_1, p_2, p_3, p_4, p_5, p_6$:

$p_1: p_3 \ p_4 \ p_2 \ p_6 \ p_5$
 $p_2: p_6 \ p_5 \ p_4 \ p_1 \ p_3$
 $p_3: p_2 \ p_4 \ p_5 \ p_1 \ p_6$
 $p_4: p_5 \ p_2 \ p_3 \ p_6 \ p_1$
 $p_5: p_3 \ p_1 \ p_2 \ p_4 \ p_6$
 $p_6: p_5 \ p_1 \ p_3 \ p_4 \ p_2$

A possible execution of Phase 1 consists of the following sequence of proposals and rejections, where \rightarrow represents *proposes to* and \times represents *rejects*.

$p_1 \rightarrow p_3$
 $p_2 \rightarrow p_6$
 $p_3 \rightarrow p_2$
 $p_4 \rightarrow p_5$
 $p_5 \rightarrow p_3; \ p_3 \times p_1$
 $p_1 \rightarrow p_4$
 $p_6 \rightarrow p_5; \ p_5 \times p_6$
 $p_6 \rightarrow p_1$

So Phase 1 ends with the set $S = \{(p_1, p_4), (p_2, p_6), (p_3, p_2), (p_4, p_5), (p_5, p_3), (p_6, p_1)\}$.

In Phase 2, the rotation $r_1 = (p_1, p_4), (p_3, p_2)$ is first identified. This is because p_2 is p_1 's second favorite, and p_4 is the second favorite of p_3 . Applying r_1 gives the new set $S = \{(p_1, p_2), (p_2, p_6), (p_3, p_4), (p_4, p_5), (p_5, p_3), (p_6, p_1)\}$. Next, the rotation $r_2 = (p_1, p_2), (p_2, p_6), (p_4, p_5)$ is identified, and application of r_2 gives $S = \{(p_1, p_6), (p_2, p_5), (p_3, p_4), (p_4, p_2), (p_5, p_3), (p_6, p_1)\}$. Finally, the rotation $r_3 = (p_2, p_5), (p_3, p_4)$ is identified, application of which gives $S = \{(p_1, p_6), (p_2, p_4), (p_3, p_5), (p_4, p_2), (p_5, p_3), (p_6, p_1)\}$. This is a matching, and is guaranteed to be 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
- Irving, Robert W.; Manlove, David F. (2002), "The Stable Roommates Problem with Ties" ^[1], *Journal of Algorithms* **43** (1): 85–105, doi:10.1006/jagm.2002.1219

References

[1] <http://eprints.gla.ac.uk/11/01/SRT.pdf>

Permanent

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 (2×2 matrix),

$$\text{perm} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad + bc.$$

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. 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). A formula similar to Laplace's for the development of a determinant along a row or column is also valid for the permanent; all signs have to be ignored for the permanent.

The permanent of a matrix A is denoted $\text{per } A$, $\text{perm } A$, 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 (1812) as "fonctions symétriques permanentes" for a related type of functions, and was used by Muir (1882) in the modern, more specific, sense.

Properties and applications

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 biadjacency 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.

0-1 permanents: counting in unweighted graphs

In an unweighted, directed, simple graph, 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 01-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]

Minimal permanent

Of all the doubly stochastic matrices, the matrix $a_{ij} = 1/n$ (that is, the uniform matrix) has strictly the smallest permanent. This was conjectured by van der Waerden, and proved in the late 1970s independently by Falikman and Egorychev.^[2] The proof of Egorychev is an application of the Alexandrov–Fenchel inequality.

Computation

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 (matrix whose entries are 0 or 1) is #P-complete. Thus, if the permanent can be computed in polynomial time by any method, then $\mathbf{FP} = \#P$, which is an even stronger statement than $P = NP$. 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.^[3]

References

- [1] Dexter Kozen. *The Design and Analysis of Algorithms*. (http://books.google.com/books?id=L_AMnf9UF9QC&pg=PA141&dq=%22permanent+of+a+matrix%22+valiant&as_brr=3&ei=h8BKScClJYOUNtTP6LEO#PPA142,M1) Springer-Verlag, New York, 1991. ISBN 978-0-387-97687-7; pp. 141–142
- [2] Van der Waerden's permanent conjecture (<http://planetmath.org/?op=getobj&from=objects&id=6935>), PlanetMath.org.
- [3] Jerrum, M.; Sinclair, A.; Vigoda, E. (2004), "A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries", *Journal of the ACM* **51**: 671–697, doi:10.1145/1008731.1008738

Further reading

- Cauchy, A. L. (1815), "Mémoire sur les fonctions qui ne peuvent obtenir que deux valeurs égales et de signes contraires par suite des transpositions opérées entre les variables qu'elles renferment." (<http://gallica.bnf.fr/ark:/12148/bpt6k90193x/f97>), *Journal de l'École Polytechnique* **10**: 91–169
- Minc, H. (1978). "Permanents". *Encyclopedia of Mathematics and its Applications* (Addison–Wesley) **6**. ISSN 0953-4806. OCLC 3980645.
- Muir, Thomas; William H. Metzler. (1960) [1882]. *A Treatise on the Theory of Determinants*. New York: Dover. OCLC 535903.

External links

- Permanent at PlanetMath (<http://planetmath.org/encyclopedia/Permanent.html>)

Computing the permanent

In mathematics, the **computation of the permanent of a matrix** is a problem that is believed to be more complex 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 assigns a ± 1 sign to each of these products, the permanent does not.

While the determinant can be computed in polynomial time by Gaussian elimination, Gaussian elimination cannot be used to compute the permanent. In computational complexity theory, a theorem of Valiant states that computing permanents, even of matrices in which all entries are 0 or 1, is #P-complete Valiant (1979) putting 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 ACC⁰ circuits (Allender & Gore 1994).

Despite, or perhaps because of, its computational difficulty, there has been much research on exponential-time exact algorithms and polynomial time approximation algorithms for the permanent, both for the case of the 0-1 matrices arising in the graph matching problems and more generally.

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 fastest known ^[1] general exact algorithm is due to Herbert John Ryser (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^[3]

$$\text{perm}(A) = (-1)^n \sum_{S \subseteq \{1, \dots, n\}} (-1)^{|S|} \prod_{i=1}^n \sum_{j \in S} a_{ij}.$$

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.

Glynn formula

Another formula that appears to be as fast as Ryser's is closely related to the polarization identity for a symmetric tensor (Glynn 2010).

It has the formula (when the characteristic of the field is not two)

$$\text{perm}(A) = \left[\sum_{\delta} \left(\prod_{k=1}^m \delta_k \right) \prod_{j=1}^m \sum_{i=1}^m \delta_i a_{ij} \right] / 2^{m-1},$$

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 $K_{3,3}$.^[4]

George Pólya had asked the question^[5] 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 using the determinants of the principal submatrices of the matrix:

$$\text{perm}(A) = (-1)^{m-1} \sum_{U \subset \{2, \dots, m\}} \det(A_U) \cdot \det(A_{\bar{U}}),$$

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 $\{1, \dots, m\}$.

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 $\epsilon > 0$ 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 Rempala & Wesolowski (2008)
- [2] van Lint & Wilson (2001) p. 99 (<http://books.google.com/books?id=5l5ps2JkyT0C&pg=PA108&dq=permanent+ryser&lr=#PPA99,M1>)
- [3] CRC Concise Encyclopedia of Mathematics ()
- [4] Little (1974), Vazirani (1988)
- [5] Pólya (1913), Reich (1971)

References

- Allender, Eric; Gore, Vivec (1994), "A uniform circuit lower bound for the permanent", *SIAM J. Comput.* **23** (5): 1026–1049
- David G. Glynn (2010), "The permanent of a square matrix", *European Journal of Combinatorics* **31** (7): 1887–1891, doi:10.1016/j.ejc.2010.01.010
- 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, ECCC TR00-079
- 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
- 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>), *J. Combin. Theory Ser. B* **18** (3): 187–208, doi:10.1016/0095-8956(75)90048-9
- Marcus, M.; Minc, H. (1961), "On the relation between the determinant and the permanent", *Illinois J. Math.* **5**: 376–381
- Pólya, G. (1913), "Aufgabe 424", *Arch. Math. Phys.* **20** (3): 27
- Reich, Simeon (1971), "Another solution of an old problem of polya", *American Mathematical Monthly* **78** (6): 649–650, doi:10.2307/2316574, JSTOR 2316574

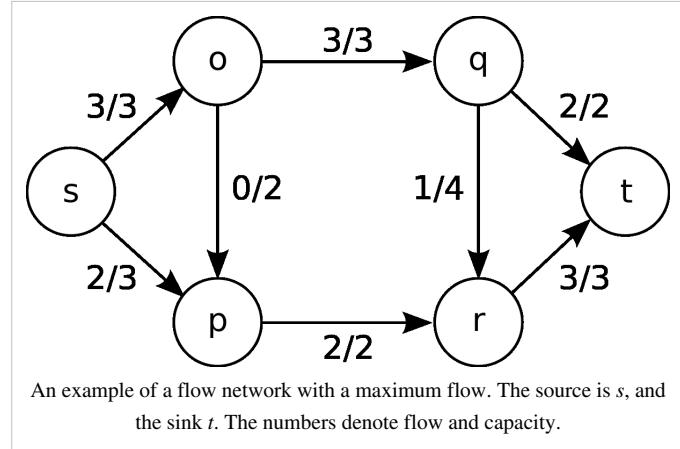
- Rempała, Grzegorz A.; Wesolowski, Jacek (2008), *Symmetric Functionals on Random Matrices and Random Matchings Problems*, pp. 4, ISBN 0-387-75145-9
- Ryser, H. J. (1963), *Combinatorial Mathematics*, The Carus mathematical monographs, The Mathematical Association of America
- Vazirani, Vijay V. (1988), "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
- Leslie G. Valiant (1979), "The Complexity of Computing the Permanent", *Theoretical Computer Science* (Elsevier) **8** (2): 189–201, doi: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, the **maximum flow problem** is to find 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 is equal to the minimum capacity of an s - t cut 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 as a simplified model of Soviet railway traffic flow.^[1] In 1955, Lester R. Ford and Delbert R. Fulkerson created the first known algorithm, the Ford–Fulkerson algorithm.^{[2][3]}

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.

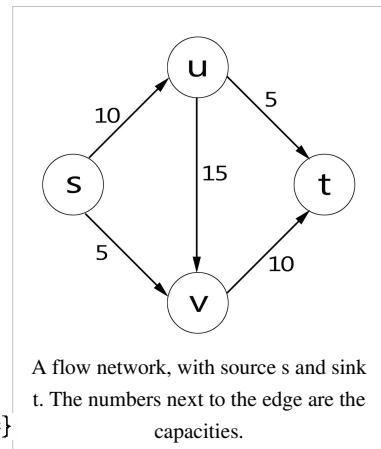
The **capacity** of an edge is a mapping $c:E \rightarrow \mathbb{R}^+$, denoted by c_{uv} or $c(u,v)$. It represents the maximum amount of flow that can pass through an edge.

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 $|f| = \sum_{v \in V} f_{sv}$, 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 .

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 <i>FIFO</i> 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 GR97	$O(E \min(V^{2/3}, \sqrt{E}) \log(V^2/E) \log U)$	The value U corresponds to the maximum capacity of the network.
MPM (Malhotra, Pramodh-Kumar and Maheshwari) algorithm	$O(V^3)$	

For a more extensive list, see GT88.

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. $E' = \{(u, v) \in (V_{out} \times V_{in}) : (u, v) \in E\}$.

Then it can be shown 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 .

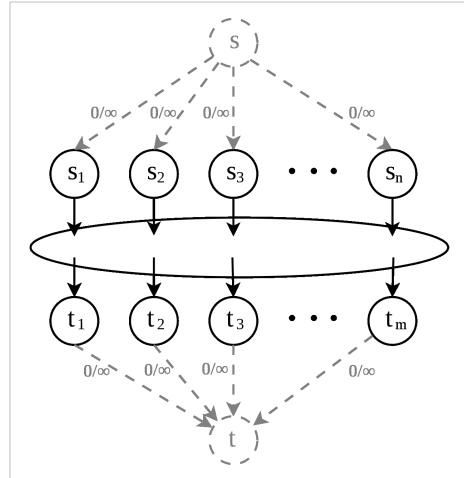


Fig. 4.1.1. Transformation of a multi-source multi-sink maximum flow problem into a single-source single-sink maximum flow problem

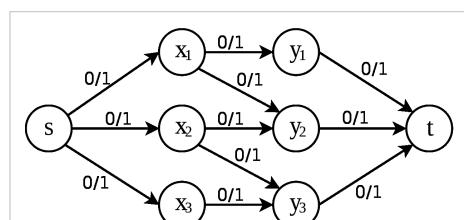


Fig. 4.3.1. Transformation of a maximum bipartite matching problem into a maximum flow problem

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 $c: V \rightarrow R^+$, 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.

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). In this expanded network, the vertex capacity constraint is removed and therefore the problem can be treated as the original maximum flow problem.

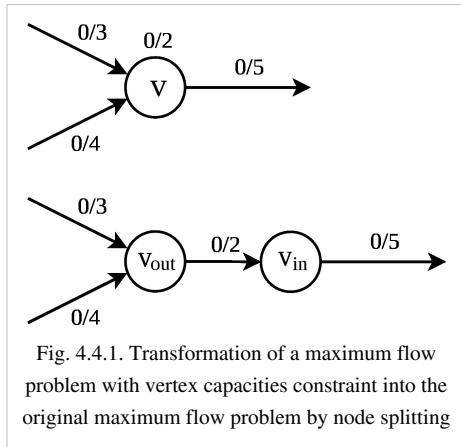


Fig. 4.4.1. Transformation of a maximum flow problem with vertex capacities constraint into the original maximum flow problem by node splitting

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.

References

- [1] Schrijver, Alexander, "On the history of the transportation and maximum flow problems" (<http://homepages.cwi.nl/~lex/files/histtpclean.pdf>), *Mathematical Programming* 91 (2002) 437-445
- [2] Ford, L.R., Jr.; Fulkerson, D.R., "Maximal Flow through a Network", *Canadian Journal of Mathematics* (1956), pp.399-404.
- [3] Ford, L.R., Jr.; Fulkerson, D.R., *Flows in Networks*, Princeton University Press (1962).

Notes

1. Andrew V. Goldberg and S. Rao (1998). "Beyond the flow decomposition barrier". *J. Assoc. Comput. Mach.* **45** (5): 753–782. doi:10.1145/290179.290181.
2. Andrew V. Goldberg and Robert E. Tarjan (1988). "A new approach to the maximum-flow problem". *Journal of the ACM* (ACM Press) **35** (4): 921–940. doi:10.1145/48014.61051. ISSN 0004-5411.
3. 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.
4. 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. ISSN 0022-0000.
5. Daniel D. Sleator and Robert E. Tarjan (1985). "Self-adjusting binary search trees" (<http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf>). *Journal of the ACM* (ACM Press) **32** (3): 652–686. doi:10.1145/3828.3835. ISSN 0004-5411.
6. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2001). "26. Maximum Flow". *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill. pp. 643–668. ISBN 0-262-03293-7.
7. 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 which 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 $c: E \rightarrow R^+$, 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 $f: E \rightarrow R^+$, 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 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

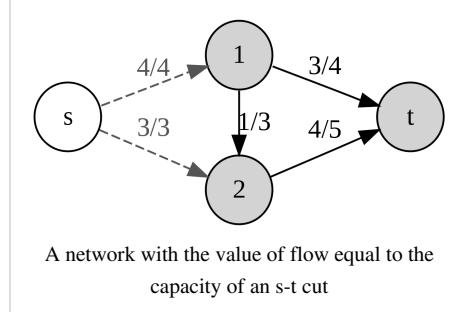
The max-flow problem and min-cut problem can be formulated as two primal-dual linear programs.

Max-flow (Primal)	Min-cut (Dual)
maximize $ f = \nabla_s$	minimize $\sum_{(i,j) \in E} c_{ij} d_{ij}$
subject to	subject to
$\begin{array}{lcl} f_{ij} & \leq & c_{ij} & (i, j) \in E \\ \sum_{j:(j,i) \in E} f_{ji} - \sum_{j:(i,j) \in E} f_{ij} & \leq & 0 & i \in V, i \neq s, t \\ \nabla_s + \sum_{j:(j,s) \in E} f_{js} - \sum_{j:(s,j) \in E} f_{sj} & \leq & 0 \\ \nabla_t + \sum_{j:(t,j) \in E} f_{jt} - \sum_{j:(j,t) \in E} f_{tj} & \leq & 0 \\ f_{ij} & \geq & 0 & (i, j) \in E \end{array}$	$\begin{array}{lcl} d_{ij} - p_i + p_j & \geq & 0 & (i, j) \in E \\ p_s & = & 1 \\ p_t & = & 0 \\ p_i & \geq & 0 & i \in V \\ d_{ij} & \geq & 0 & (i, j) \in E \end{array}$

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

$$\sum_{v \in V} f_{iv} \leq c(v) \text{ for each } v \in V \setminus \{s, t\}.$$

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

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

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 $r(p_i)$ and $c(q_j)$ are positive, this maximization problem can be formulated as a minimization problem instead, that is,

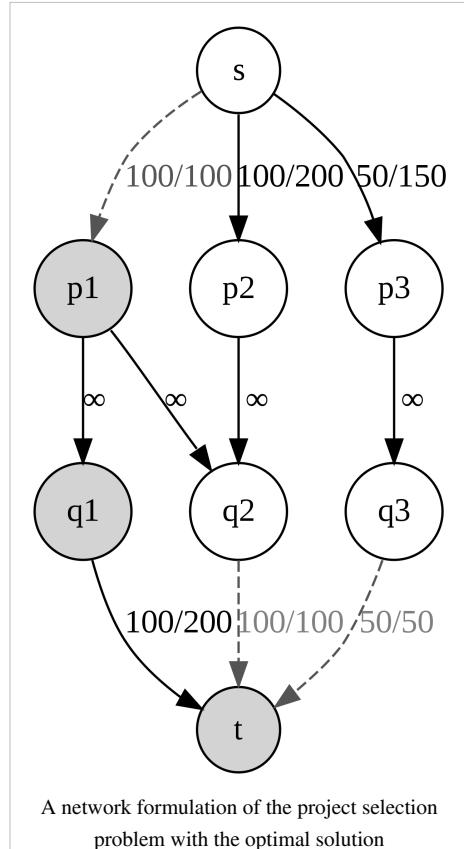
$$\min\{g'\} = \sum_{p_i \in P} r(p_i) + \sum_{q_j \in Q} c(q_j).$$

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 .



History

The **max-flow min-cut theorem** was proved by P. Elias, A. Feinstein, and C.E. Shannon in 1956, and independently also by L.R. Ford, Jr. and D.R. Fulkerson in the same year.

References

- 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.
- 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.
- 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** (named for L. R. Ford, Jr. and D. R. Fulkerson) is an algorithm which computes the maximum flow in a flow network. It was published in 1954. 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 very simple. 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: $\forall u \in V : u \neq s \wedge u \neq t \Rightarrow \sum_{w \in V} f(u, w) = 0$ 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.

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 Graph G with flow capacity c , a source node s , and a sink node t

Output A flow f from s to t which is a maximum

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.

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

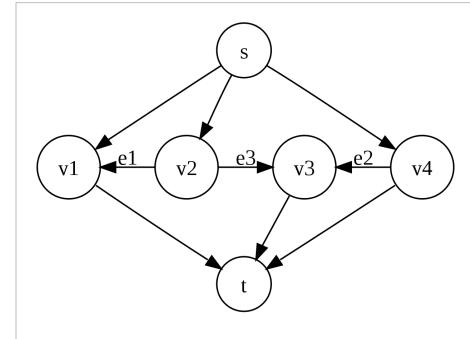
The following example shows the first steps of Ford–Fulkerson in a flow network with 4 nodes, source A and sink D . This example shows the worst-case behaviour of the algorithm. In each step, only a flow of 1 is sent across the network. If breadth-first-search were used instead, only two steps would be needed.

Path	Capacity	Resulting flow network
	Initial flow network	
A, B, C, D	$\min(c_f(A, B), c_f(B, C), c_f(C, D)) =$ $\min(c(A, B) - f(A, B), c(B, C) - f(B, C), c(C, D) - f(C, D)) =$ $\min(1000 - 0, 1 - 0, 1000 - 0) = 1$	
A, C, B, D	$\min(c_f(A, C), c_f(C, B), c_f(B, D)) =$ $\min(c(A, C) - f(A, C), c(C, B) - f(C, B), c(B, D) - f(B, D)) =$ $\min(1000 - 0, 0 - (-1), 1000 - 0) = 1$	
	After 1998 more steps ...	
	Final flow network	

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.^[1]

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):
        if vertex not in self.adj:
            self.adj[vertex] = []
        if vertex not in self.flow:
            self.flow[vertex] = 0

```

```

    self.adj[vertex] = []

    def get_edges(self, v):
        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 not (edge,residual) in path:
                result = self.find_path( edge.sink, sink, path +
[ (edge,residual) ] )
                if result != None:
                    return result

    def max_flow(self, source, sink):
        path = self.find_path(source, sink, [])
        while path != None:
            flow = min(res for edge,res in path)
            for edge,res 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()
map(g.add_vertex, ['s','o','p','q','r','t'])
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')
```

Output: 5

Notes

- [1] Zwick, Uri (21 August 1995). "The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate". *Theoretical Computer Science* **148** (1): 165–170. doi:10.1016/0304-3975(95)00022-O.

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.
- Ford, L. R.; Fulkerson, D. R. (1954). "Maximal flow through a network". *Canadian Journal of Mathematics* **8**: 399–404.

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>)

Edmonds–Karp algorithm for maximum flows

In computer science and graph theory, 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. It is asymptotically slower than the relabel-to-front algorithm, which runs in $O(V^3)$ time, but it is often faster in practice for sparse graphs. The algorithm was first published by Yefim (Chaim) Dinic in 1970^[1] and independently published by Jack Edmonds and Richard Karp in 1972^[2]. 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, 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^[3].

Pseudocode

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

```

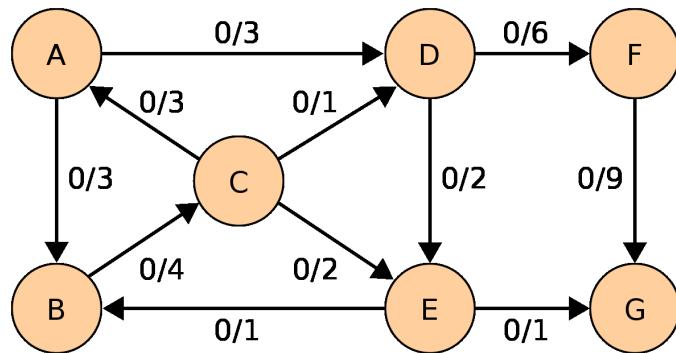
```

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

```

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

- [1] Dinic, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". *Soviet Math. Doklady* **11**: 1277–1280.
- [2] Edmonds, Jack; Karp, Richard M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". *Journal of the ACM* (Association for Computing Machinery) **19** (2): 248–264. doi:10.1145/321694.321699.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2001). "26.2". *Introduction to Algorithms* (second ed.). MIT Press and McGraw–Hill. pp. 660–663. ISBN 0-262-53196-8.

References

1. Algorithms and Complexity (see pages 63–69). <http://www.cis.upenn.edu/~wilf/AlgComp3.html>

Dinic's algorithm for maximum flows

Dinic'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.^[1] 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,

1. if $(u, v) \in E$,
- $$c_f(u, v) = c(u, v) - f(u, v)$$
- $$c_f(v, u) = f(u, v)$$
2. $c_f(u, v) = 0$ 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.

1. Set $f(e) = 0$ for each $e \in E$.
2. Construct G_L from G_f of G . If $\text{dist}(t) = \infty$, stop and output f .
3. Find a blocking flow f' in G_L .
4. 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.^[2]

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] Yefim Dinitz (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation" (<http://www.cs.bgu.ac.il/~dinitz/D70.pdf>). *Doklady Akademii nauk SSSR* **11**: 1277–1280..
- [2] 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

The **push-relabel algorithm** is one of the most efficient algorithms to compute a maximum flow. The general algorithm has $O(V^2E)$ time complexity, while the implementation with FIFO vertex selection rule has $O(V^3)$ running time, the highest active vertex selection rule provides $O(V^2\sqrt{E})$ complexity, and the implementation with Sleator's and Tarjan's dynamic tree data structure runs in $O(VE \log(V^2/E))$ time. Asymptotically, it is more efficient than the Edmonds-Karp algorithm, which runs in $O(VE^2)$ time.

Algorithm

Given a flow network $G(V, E)$ with capacity from node u to node v given as $c(u, v)$, source s and sink t , we want to find the maximum amount of flow you can send from s to t through the network. Two types of operations are performed on nodes, *push* and *relabel*. Throughout we maintain:

- $f(u, v)$. Flow from u to v . Available capacity is $c(u, v) - f(u, v)$.
- $height(u)$. We only *push* from u to v if $height(u) > height(v)$. For all u , $height(u)$ is a non-negative integer.
- $excess(u)$. Sum of flow to and from u .

After each step of the algorithm, the flow is a **preflow**, satisfying:

- $f(u, v) \leq c(u, v)$. The flow between u and v , does not exceed the capacity.
- $f(u, v) = -f(v, u)$. We maintain the net flow.
- $\sum_v f(u, v) = excess(u) \geq 0$ for all nodes $u \neq s$. Only the source may produce flow.

Notice that the last condition for a preflow is relaxed from the corresponding condition for a legal flow in a regular flow network.

We observe that the longest possible path from s to t is $|V|$ nodes long. Therefore it must be possible to assign $height$ to the nodes such that for any legal flow, $height(s) = |V|$ and $height(t) = 0$, and if there is a positive flow from u to v , $height(u) > height(v)$. As we adjust the height of the nodes, the flow goes through the network as water through a landscape. Differing from algorithms such as Ford-Fulkerson, the flow through the network is not necessarily a legal flow throughout the execution of the algorithm. In short words, the heights of nodes (except s and t) is adjusted, and flow is sent between nodes, until all possible flow has reached t . Then we continue increasing the height of internal nodes until all the flow that went into the network, but did not reach t , has flowed back into s . A node can reach the height $2|V| - 1$ before this is complete, as the longest possible path back to s excluding t is $|V| - 1$ long, and $height(s) = |V|$. The height of t is kept at 0.

Push

A *push* from u to v means sending a part of the excess flow into u on to v . Three conditions must be met for a *push* to take place:

- $excess(u) > 0$. More flow into the node than out of it so far.
- $c(u, v) - f(u, v) > 0$. Available capacity from u to v .
- $height(u) > height(v)$. Can only send to lower node.

We send an amount of flow equal to $\min(excess(u), c(u, v) - f(u, v))$.

Relabel

Doing a *relabel* on a node u is increasing its height until it is higher than at least one of the nodes it has available capacity to. Conditions for a *relabel*:

- $excess(u) > 0$. There must be a reason to relabel.
- $height(u) \leq height(v)$ for all v such that $c(u, v) - f(u, v) > 0$. The only nodes we have available capacity to are higher.

When relabelling u , we set $height(u)$ to be the lowest value such that $height(u) > height(v)$ for some v where $c(u, v) - f(u, v) > 0$.

Push-relabel algorithm

Push-relabel algorithms in general have the following layout:

1. As long as there is legal *push* or *relabel* operation
 1. Perform a legal push, or
 2. a legal relabel.

The running time for these algorithms are in general $O(V^2E)$ (argument omitted).

Discharge

In *relabel-to-front*, a *discharge* on a node u is the following:

1. As long as $excess(u) > 0$:
 1. If not all neighbours have been tried since last *relabel*:
 1. Try to *push* flow to an untried neighbour.
 2. Else:
 1. *Relabel* u

This requires that for each node, it is known which nodes have been tried since last *relabel*.

Relabel-to-front algorithm, ie. using FIFO heuristic

In the *relabel-to-front algorithm*, the order of the *push* and *relabel* operations is given:

1. Send as much flow from s as possible.
2. Build a list of all vertices except s and t .
3. As long as we have not traversed the entire list:
 1. *Discharge* the current vertex.
 2. If the height of the current vertex changed:
 1. Move the current vertex to the front of the list
 2. Restart the traversal from the front of the list.

The running time for *relabel-to-front* is $O(V^3)$ (proof omitted).

Sample implementation

C implementation:

```
#include <stdlib.h>
#include <stdio.h>

#define NODES 6
#define MIN(X, Y) X < Y ? X : Y
#define INFINITE 10000000

void push(int **C, int **F, int *excess, int u, int v) {
    int send = MIN(excess[u], C[u][v] - F[u][v]);
    F[u][v] += send;
    F[v][u] -= send;
    excess[u] -= send;
    excess[v] += send;
}

void relabel(int **C, int **F, int *height, int u) {
    int v;
    int min_height = INFINITE;
    for (v = 0; v < NODES; v++) {
        if (C[u][v] - F[u][v] > 0) {
            min_height = MIN(min_height, height[v]);
            height[u] = min_height + 1;
        }
    }
}
```

```

        }
    }

    void discharge(int **C, int **F, int *excess, int *height, int
*seen, int u) {
        while (excess[u] > 0) {
            if (seen[u] < NODES) {
                int v = seen[u];
                if ((C[u][v] - F[u][v] > 0) && (height[u] >
height[v])) {
                    push(C, F, excess, u, v);
                }
            } else {
                relabel(C, F, height, u);
                seen[u] = 0;
            }
        }
    }

    void moveToFront(int i, int *A) {
        int temp = A[i];
        int n;
        for (n = i; n > 0; n--) {
            A[n] = A[n-1];
        }
        A[0] = temp;
    }

    int pushRelabel(int **C, int **F, int source, int sink) {
        int *excess, *height, *list, *seen, i, p;

        excess = (int *) calloc(NODES, sizeof(int));
        height = (int *) calloc(NODES, sizeof(int));
        seen = (int *) calloc(NODES, sizeof(int));

        list = (int *) calloc((NODES-2), sizeof(int));

        for (i = 0, p = 0; i < NODES; i++) {
            if ((i != source) && (i != sink)) {
                list[p] = i;
                p++;
            }
        }
    }
}

```

```

height[source] = NODES;
excess[source] = INFINITE;
for (i = 0; i < NODES; i++)
    push(C, F, excess, source, i);

p = 0;
while (p < NODES - 2) {
    int u = list[p];
    int old_height = height[u];
    discharge(C, F, excess, height, seen, u);
    if (height[u] > old_height) {
        moveToFront(p, list);
        p=0;
    }
    else
        p += 1;
}
int maxflow = 0;
for (i = 0; i < NODES; i++)
    maxflow += F[source][i];

return maxflow;
}

void printMatrix(int **M) {
    int i,j;
    for (i = 0; i < NODES; i++) {
        for (j = 0; j < NODES; j++)
            printf("%d\t",M[i][j]);
        printf("\n");
    }
}

int main(void) {
    int **flow, **capacities, i;
    flow = (int **) calloc(NODES, sizeof(int*));
    capacities = (int **) calloc(NODES, sizeof(int*));
    for (i = 0; i < NODES; i++) {
        flow[i] = (int *) calloc(NODES, sizeof(int));
        capacities[i] = (int *) calloc(NODES, sizeof(int));
    }

    //Sample graph
    capacities[0][1] = 2;
    capacities[0][2] = 9;
    capacities[1][2] = 1;
    capacities[1][3] = 0;
}

```

```

    capacities[1][4] = 0;
    capacities[2][4] = 7;
    capacities[3][5] = 7;
    capacities[4][5] = 4;

    printf("Capacity:\n");
    printMatrix(capacities);

    printf("Max Flow:\n%d\n", pushRelabel(capacities, flow, 0,
5));

    printf("Flows:\n");
    printMatrix(flow);

    return 0;
}

```

Python implementation:

```

def relabel_to_front(C, source, sink):
    n = len(C) # C is the capacity matrix
    F = [[0] * n for _ in xrange(n)]
    # residual capacity from u to v is C[u][v] - F[u][v]

    height = [0] * n # height of node
    excess = [0] * n # flow into node minus flow from node
    seen = [0] * n # neighbours seen since last relabel
    # node "queue"
    list = [i for i in xrange(n) if i != source and i != sink]

    def push(u, v):
        send = min(excess[u], C[u][v] - F[u][v])
        F[u][v] += send
        F[v][u] -= send
        excess[u] -= send
        excess[v] += send

    def relabel(u):
        # find smallest new height making a push possible,
        # if such a push is possible at all
        min_height = infinity
        for v in xrange(n):
            if C[u][v] - F[u][v] > 0:
                min_height = min(min_height, height[v])
                height[u] = min_height + 1

    def discharge(u):
        while excess[u] > 0:

```

```

    if seen[u] < n: # check next neighbour
        v = seen[u]
        if C[u][v] - F[u][v] > 0 and height[u] > height[v]:
            push(u, v)
        else:
            seen[u] += 1
    else: # we have checked all neighbours. must relabel
        relabel(u)
        seen[u] = 0

height[source] = n # longest path from source to sink is less
than n long
excess[source] = Inf # send as much flow as possible to neighbours
of source
for v in xrange(n):
    push(source, v)

p = 0
while p < len(list):
    u = list[p]
    old_height = height[u]
    discharge(u)
    if height[u] > old_height:
        list.insert(0, list.pop(p)) # move to front of list
        p = 0 # start from front of list
    else:
        p += 1

return sum(F[source])

```

Note that the above implementation is not very efficient. It is slower than Edmonds-Karp algorithm even for very dense graphs. To speed it up, you can do at least two things:

1. Make neighbour lists for each node, and let the index `seen[u]` be an iterator over this, instead of the range $0..n-1$.
2. Use a **gap heuristic**. If there is a k such that for no node, $\text{height}(u) = k$, you can set $\text{height}(u) = \max(\text{height}(u), \text{height}(s) + 1)$ for all nodes except s for which $\text{height}(u) > k$. This is because any such k represents a minimal cut in the graph, and no more flow will go from the nodes $S = \{u | \text{height}(u) > k\}$ to the nodes $T = \{v | \text{height}(v) < k\}$. If (S, T) was not a minimal cut, there would be an edge (u, v) such that $u \in S, v \in T$ and $c(u, v) - f(u, v) > 0$. But then $\text{height}(u)$ would never be set higher than $\text{height}(v) + 1$, contradicting that $\text{height}(u) > k$ and $\text{height}(v) < k$.

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 26.4: Push-relabel algorithms, and section 26.5: The relabel-to-front-algorithm.
- Andrew V. Goldberg, Robert E. Tarjan. A new approach to the maximum flow problem^[1]. Annual ACM Symposium on Theory of Computing, Proceedings of the eighteenth annual ACM symposium on Theory of computing, 136–146. ISBN 0-89791-193-8, 1986

References

[1] <http://doi.acm.org/10.1145/12130.12144>

Closure problem

A **Closure problem** is a problem in graph theory for finding a set of vertices in a directed graph such that there are no edges from the set to the rest of the graph. More specifically, the minimum closure problem asks for a set of this type with the minimum possible weight in a weighted graph.

Closure problem

Definition

In a directed graph $G = (V, A)$, a set S of vertices is said to be *closed* if every successor of every vertex in S is also in S . Equivalently, S is closed if it has no outgoing edge.

It may be assumed without loss of generality that G is a directed acyclic graph. For, if it is not acyclic, each of its strongly connected components must either be entirely contained in or entirely disjoint from any closed set. Therefore, the closures of G are in one-to-one correspondence with the closures of the condensation of G , a directed acyclic graph that has one vertex for each strongly connected component of G . In weighted closure problems, one may set the weight of any vertex of the condensation to the sum of the weights of the vertices in the corresponding strongly connected component of G .

Minimum closure problem

For a directed graph $G = (V, A)$ with vertex weights w_i , the minimum closure problem is to find a closed set of total minimum weight. If all the weights are positive or negative, the minimum closure problem is trivial. Indeed, if all weights are positive the empty subgraph is a solution, and if all weights are negative, the whole graph is solution. We may therefore assume that G has both positive and negative weights.

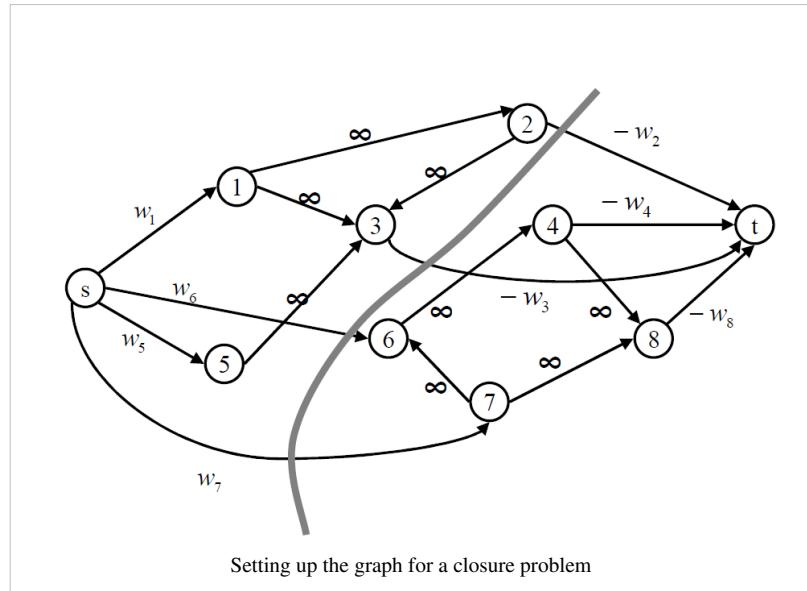
Open-pit mining

Picard studied the closure problem on the open-pit mining problem, which he modeled as a maximum-closure problem.

Maximum and minimum closure problem

In the maximum closure problem a directed graph $G = (V, A)$ with vertex weights w_i is given and we want to find a closed subset of vertices V' such that $\sum_{i \in V'} w_i$ is maximum. Picard showed

that the maximum closure problem can be solved using a minimum cut procedure. Additionally it is clear that the selection problem is closure problem on a bipartite graph therefore the selection problem is a special case of the closure problem. Maximum and minimum closure problems can be converted to each other by negating the vertex weights. A maximum closure problem can be formulated as follows



$$\begin{aligned} & \max \sum_{j \in V} w_j x_j \\ & \text{subject to } x_j \leq x_i \quad \forall (i, j) \in A \\ & x_i \in \{0, 1\} \quad \forall j \in V \end{aligned}$$

Where x_i is 1 if the vertex is in the closure and it is 0 otherwise, and the first constraint ensures that if a vertex is in the closure its successor is also in it. Since each row has at most one 1 and one -1 we know that the constraint matrix is totally unimodular and an integer solution is obtained by solving the LP relaxation of the problem.

In order to solve the maximum closure problem we can use the max-flow min-cut theorem. Let us construct the s-t graph for maximum closure problem. The graph has a vertex j for each variable x_j . We also add a source s and a sink vertex t . If the weight of the variable w_j is positive we include an arc from the source to the vertex j with capacity w_j . If the weight is negative we add the arc from j to the sink vertex (t) with capacity $-w_j$. Each inequality $x_j \leq x_i$ is associated with an arc (i, j) with capacity ∞ . Let V^+ be the set of vertices with positive weights and V^- the set of vertices with negative weights. Figure [\ref{fig:cl4}](#) shows the graph constructed for the closure problem. We can find the minimum cut in this graph by solving a max-flow problem from source to the sink . The source set of a minimum cut separating s from t is a maximum closure in the graph. This statement holds because the minimum cut is finite and cannot include any arc from A that has the weight equal to ∞ [5]. Minimizing the value of the finite cut is equivalent to maximizing the sum of weights of the vertices in the source set of the finite cut. Denote (A, B) the collection of arc with tails at A and heads at B . Also $C(A, B) = \sum_{i \in A, j \in B} c_{ij}$ where c_{ij} is the capacity of arc (i, j) .

Let $w(A) = \sum_{j \in A} w_j$. For a finite cut $(\{s\} \cup S, \bar{S} \cup \{t\})$ we have:

$$\begin{aligned} \min_{\bar{S} \subseteq V} [C(\{s\} \cup S, \bar{S} \cup \{t\})] &= \min_{\bar{S} \subseteq V} \sum_{j \in \bar{S} \cap V^+} w_j + \sum_{j \in S \cap V^-} (-w_j) \\ \min_{\bar{S} \subseteq V} \sum_{j \in \bar{S} \cap V^+} w_j - \left(\sum_{i \in V^-} w_j - \sum_{i \in \bar{S} \cap V^-} w_i \right) \\ \min_{\bar{S} \subseteq V} \sum_{j \in \bar{S}} w_j - w(V^-). \end{aligned}$$

In the last expression $w(V^-)$ is a constant. Therefore the closed set \bar{S} of minimum weight is also the sink set of the minimum cut and vice versa—the sink set of a minimum cut (without t), which has to be finite, also minimized the weight of the closure.

History

During the 1970s J.C. Picard was working on the *open-pit mining* problem and during that time worked on generalizing the selection problem to the *closure problem*. During that time the mining industry was developing optimization methods on their own. Picard's contribution introduced the closure problem to the mining industry.

References

- Hochbaum, Dorit S. (2005), "Complexity and algorithms for convex network optimization and other nonlinear problems", *4OR* **3** (3): 171–216, doi:10.1007/s10288-005-0078-6, MR2177960.
- Hochbaum, Dorit S. (2001), "An efficient algorithm for image segmentation, Markov random fields and related problems", *Journal of the ACM* **48** (4): 686–701 (electronic), doi:10.1145/502090.502093, MR2144926.
- Hochbaum, Dorit S. (2004), "Selection, provisioning, shared fixed costs, maximum closure, and implications on algorithmic methods today" [1], *Management Science* **50** (6): 709–723
- Hochbaum, Dorit S. (2008), "The pseudoflow algorithm: a new algorithm for the maximum-flow problem", *Operations Research* **56** (4): 992–1009, doi:10.1287/opre.1080.0524, MR2455709.
- Hochbaum, Dorit S.; Queyranne, Maurice (2003), "Minimizing a convex cost closure set", *SIAM Journal on Discrete Mathematics* **16** (2): 192–207 (electronic), doi:10.1137/S0895480100369584, MR1982135.
- Hochbaum, Dorit S.; Shanthikumar, J. George (1990), "Convex separable optimization is not much harder than linear optimization", *Journal of the ACM* **37** (4): 843–862, doi:10.1145/96559.96597, MR1083654.

References

[1] http://hkilter.com/files/articles/hochbaum_6_04.pdf

Minimum cost flow problem

The **minimum-cost flow problem** is finding the cheapest possible way of sending a certain amount of flow through a flow network.

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 $a(u, v) \geq 0$. 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 ^{AMO93}. 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.^{K67}
- *Minimum mean cycle canceling*: a simple strongly polynomial algorithm.^{GT89}
- *Successive shortest path and capacity scaling*: dual methods, which can be viewed as the generalizations of the Ford–Fulkerson algorithm.^{EK72}
- *Cost scaling*: a primal-dual approach, which can be viewed as the generalization of the push-relabel algorithm.^{GT90}
- *Network simplex*: a specialized version of the linear programming simplex method.

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.
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.
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.
5. Andrew V. Goldberg and Robert E. Tarjan (1990). "Finding minimum-cost circulations by successive approximation". *Math. Oper. Res.* **15** (3): 430–466.

External links

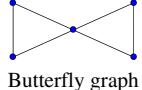
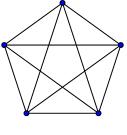
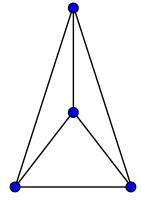
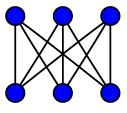
- LEMON C++ library with several maximum flow and minimum cost circulation algorithms ^[1]

References

[1] <http://lemon.cs.elte.hu/>

Graph drawing and planar graphs

Planar graph

Example graphs	
Planar	Nonplanar
 Butterfly graph	 K_5
 The complete graph K_4 is planar	 $K_{3,3}$

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.^[1] 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).

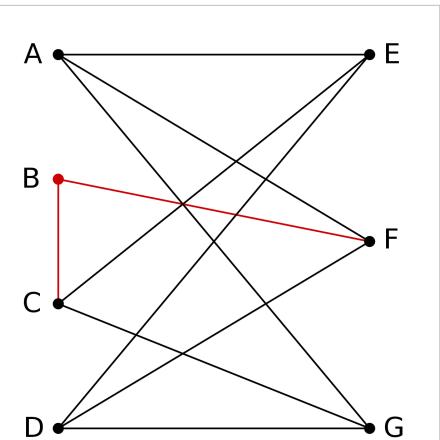
A subdivision of a graph results from inserting vertices into edges (for example, changing an edge $\bullet-\bullet-\bullet$ to $\bullet-\bullet-\bullet$) zero or more times. Equivalent formulations of this theorem, also known as "Theorem P" include

A finite graph is planar if and only if it does not contain a subgraph that is homeomorphic to K_5 or $K_{3,3}$.

In the Soviet Union, Kuratowski's theorem was known as the **Pontryagin–Kuratowski theorem**, as its proof was allegedly first given in Pontryagin's unpublished notes. By a long-standing academic tradition, such references are not taken into account in determining priority, so the Russian name of the theorem is not acknowledged internationally.

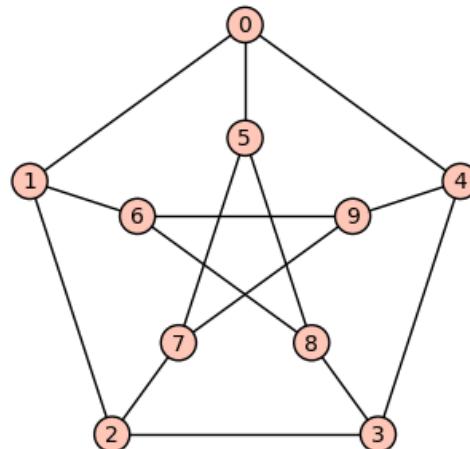
Instead of considering subdivisions, **Wagner's theorem** deals with minors:

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.



An animation showing that the Petersen graph contains a minor isomorphic to the $K_{3,3}$ graph

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 planarity criteria 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 $e \leq 2v - 4$.

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;
- Fraysseix–Rosenstiehl's 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

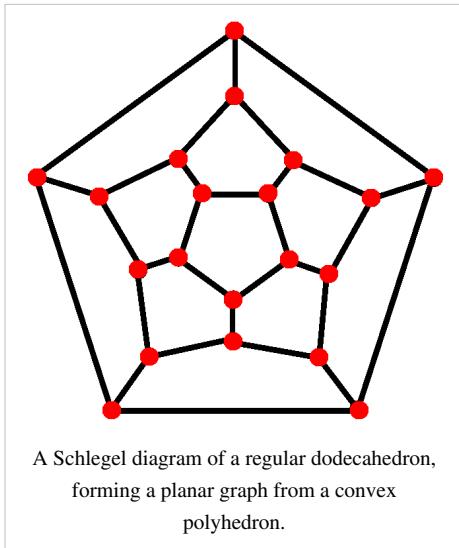
Euler's formula states that if a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely large region), then

$$v - e + f = 2.$$

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

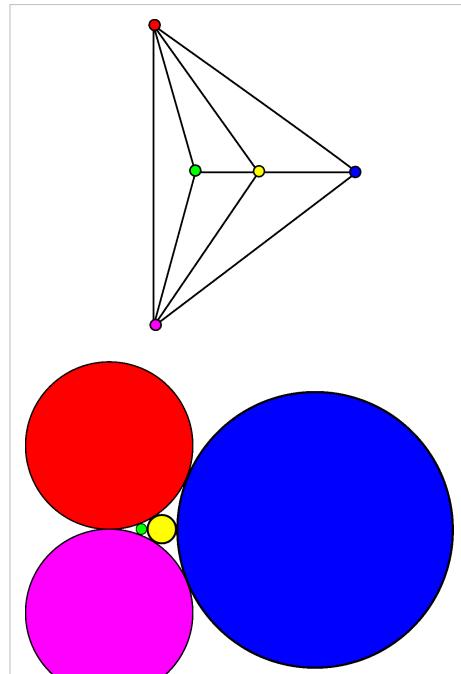
From $v - e + f = 2$ and $3f \leq 2e$ (one face has minimum 3 edges and each edge has maximum two faces) it follows via algebraic transformations that the average degree is strictly less than 6. Otherwise the given graph can't be planar.

Coin graphs

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

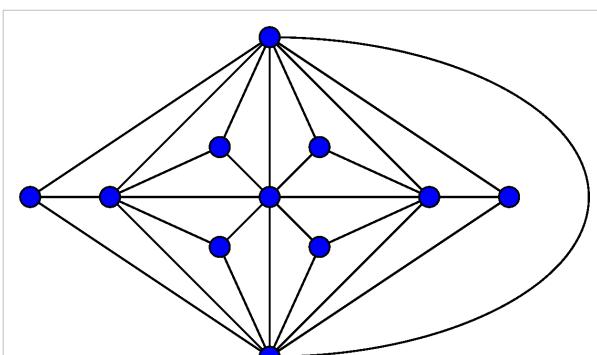


Example of the circle packing theorem on K_5 , the complete graph on five vertices, minus one edge.

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 (even the outer one) are then bounded by three edges, explaining the alternative term **plane triangulation**. The alternative names "triangular graph"^[2] or "triangulated graph"^[3] 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.

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.

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 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 graph onto which the graph may be embedded; planar graphs have genus zero and nonplanar toroidal graphs have genus one.

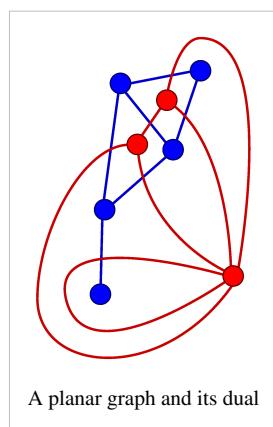
Any graph may be embedded into three-dimensional space without crossings. However, a three-dimensional analogue of the planar graphs is provided by the linklessly embeddable graphs, graphs that can be embedded into three-dimensional space in such a way that no two cycles are topologically linked with 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.

Other facts and definitions

Every planar graph without loops 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. Similarly, 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.

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.



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] Trudeau, Richard J. (1993). *Introduction to Graph Theory* (<http://store.doverpublications.com/0486678709.html>) (Corrected, enlarged republication. ed.). New York: Dover Pub.. pp. 64. ISBN 978-0-486-67870-2. . Retrieved 8 August 2012. "Thus a planar graph, when drawn on a flat surface, either has no edge-crossings or can be redrawn without them."
- [2] Schnyder, W. (1989), "Planar graphs and poset dimension", *Order* **5**: 323–343, doi:10.1007/BF00353652, MR1010382.
- [3] Bhasker, Jayaram; Sahni, Sartaj (1988), "A linear algorithm to find a rectangular dual of a planar triangulated graph", *Algorithmica* **3** (1–4): 247–278, doi:10.1007/BF01762117.
- [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

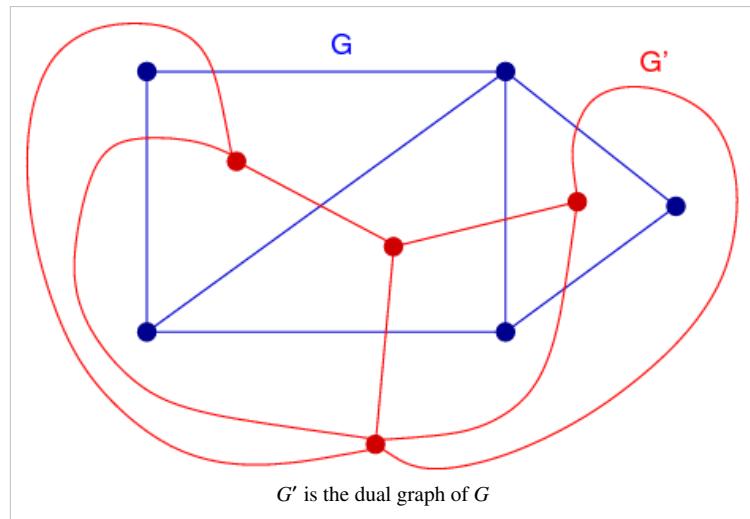
- Kuratowski, Kazimierz (1930), "Sur le problème des courbes gauches en topologie" (<http://matwbn.icm.edu.pl/ksiazki/fm/fm15/fm15126.pdf>) (in French), *Fund. Math.* **15**: 271–283.
- Wagner, K. (1937), "Über eine Eigenschaft der ebenen Komplexe", *Math. Ann.* **114**: 570–590, doi:10.1007/BF01594196.
- 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.
- 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. 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.

External links

- 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 mathematics, the **dual graph** of a given planar graph G is a graph which has a vertex corresponding to each plane region of G , and an edge joining two neighboring regions for each edge in G , for a certain embedding of 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 on manifolds.



Properties

- The dual of a planar graph is a planar multigraph - multiple edges.^[1]
- If G is a connected graph and if G' is a dual of G then G is a dual of G' .

- Dual graphs are not unique, in the sense that the same graph can have non-isomorphic dual graphs because the dual graph depends on a particular plane embedding. In the picture, red graphs are not isomorphic because the upper one has a vertex with degree 6 (the outer region).

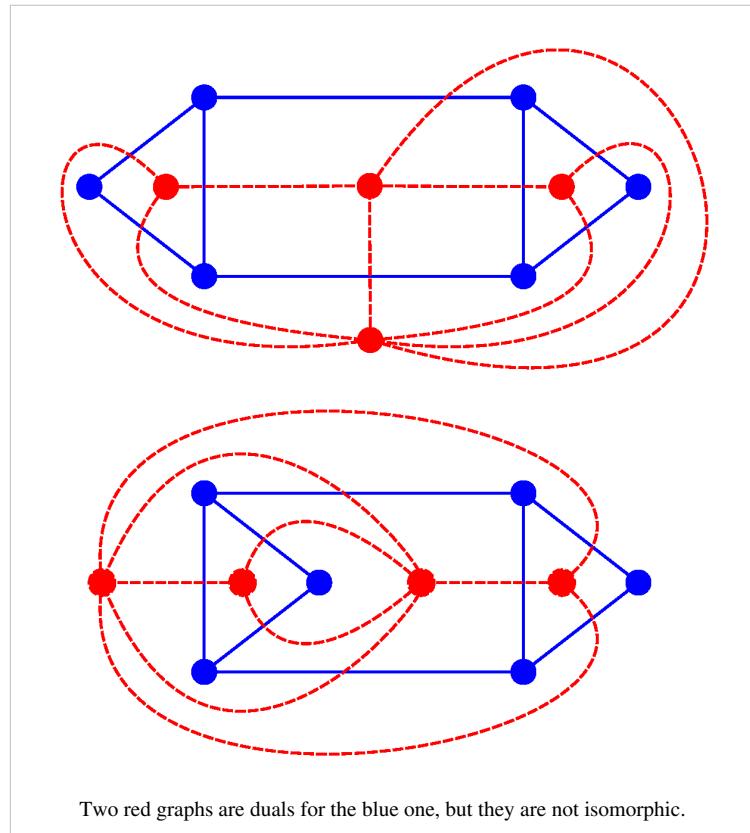
Because of the duality, any result involving counting regions and vertices can be dualized by exchanging them.

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 Whitney:^[2]

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 .



Two red graphs are duals for the blue one, but they are not isomorphic.

Weak dual

The **weak dual** of an embedded planar graph is the subgraph of the dual graph whose vertices correspond to the bounded faces of the primal graph. A planar graph is outerplanar if and only if its weak dual is a forest, and a planar graph is a Halin graph if and only if its weak dual is biconnected and outerplanar. For any embedded planar graph G , let G^+ be the 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 planar dual of G^+ .^{[3][4]}

Complex networks

In the context of complex network theory, edge dual of a random network preserves many of its properties such as small-world property and the shape of its degree distribution function^[5]

Notes

- [1] Here we consider that graphs may have loops and multiple edges to avoid uncommon considerations
- [2] Whitney, Hassler (1932), "Non-separable and planar graphs", *Transactions of the American Mathematical Society* **34** (2): 339–362, doi:10.1090/S0002-9947-1932-1501641-2.
- [3] Fleischner, Herbert J.; Geller, D. P.; Harary, Frank (1974), "Outerplanar graphs and weak duals", *Journal of the Indian Mathematical Society* **38**: 215–219, MR0389672.
- [4] Sysło, Maciej M.; Proskurowski, Andrzej (1983), "On Halin graphs", *Graph Theory: Proceedings of a Conference held in Lagów, Poland, February 10–13, 1981*, Lecture Notes in Mathematics, **1018**, Springer-Verlag, pp. 248–256, doi:10.1007/BFb0071635.
- [5] Ramezanpour A, Karimipour V. and Mashaghi A., Generating correlated networks from uncorrelated ones, Phys. Rev. E 67, 046107 (2003) <http://pre.aps.org/abstract/PRE/v67/i4/e046107>

External links

- Weisstein, Eric W., " Dual graph (<http://mathworld.wolfram.com/DualGraph.html>)" from MathWorld.
- Weisstein, Eric W., " Self-dual graph (<http://mathworld.wolfram.com/Self-DualGraph.html>)" from MathWorld.

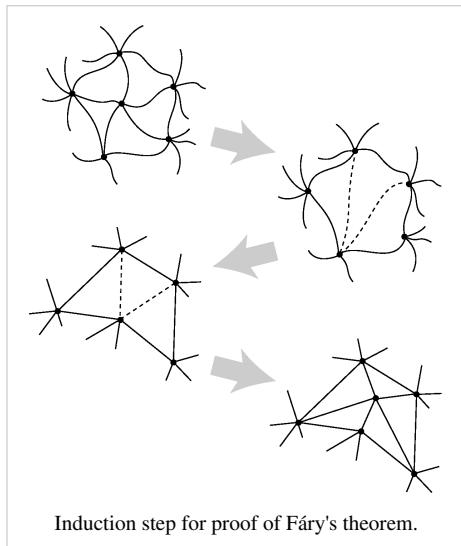
Fáry's theorem

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

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 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.

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. 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.

Heiko Harborth raised the question of whether every planar graph has a straight line representation in which all edge lengths are integers.^[1] The answer remains unknown as of 2009. However, integer-distance straight line embeddings are known to exist for cubic graphs.^[2]

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] Harborth et al. (1987); Kemnitz & Harborth (2001); Mohar (2001, 2003).

[2] Geelen, Guo & McKinnon (2008).

References

- Fáry, István (1948), "On straight-line representation of planar graphs", *Acta Sci. Math. (Szeged)* **11**: 229–233, MR0026311.
- 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, 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, MR1075065.
- 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.
- 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.

- 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. problem 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, 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, JSTOR 2031777, MR0041425.
- 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, MR0158387.
- Wagner, Klaus (1936), "Bemerkungen zum Vierfarbenproblem" (<http://www.digizeitschriften.de/index.php?id=resolveppn&PPN=GDZPPN002131633>), *Jahresbericht. German. Math.-Verein.* **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.^[3] Steinitz's theorem is named after Ernst Steinitz, who proved it in 1922.^[4] Branko Grünbaum has called this theorem "the most important and deepest known result on 3-polytopes."^[2]

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,^[5]
- 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,^[6] and
- Steinitz's vectorial generalization of the Riemann series theorem on the rearrangements of conditionally convergent series.^{[7][8][9][10]}

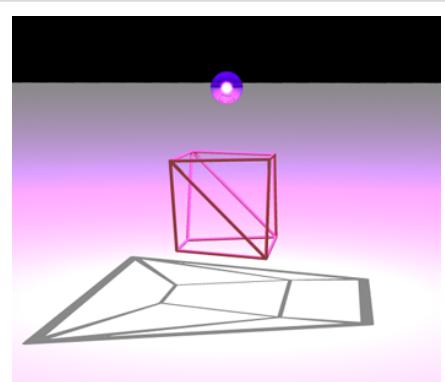
Definitions and statement of the theorem

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 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.^[11]

The other, more difficult, direction of Steinitz's theorem states that every planar 3-connected graph is the graph of a convex polyhedron.



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.

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.^[12] However, subsequent researchers have found graph drawing algorithms that use only $O(n)$ bits per vertex.^{[13][14]} 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 $O(n)$.^[15]

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 .^[16] 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.^[17]

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.^[18] 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 .^[19]

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.^[20]

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 Gunter M. Ziegler, 2003, ISBN 0-387-40409-0, ISBN 978-0-387-40409-7, 466pp.
- [3] Weisstein, Eric W., "Polyhedral graph (<http://mathworld.wolfram.com/PolyhedralGraph.html>)" from MathWorld.
- [4] Steinitz, E. (1922), "Polyeder und Raumeinteilungen", *Encyclopädie der mathematischen Wissenschaften, Band 3 (Geometries)*, pp. 1–139.
- [5] Zynel, Mariusz (1996), "The Steinitz theorem and the dimension of a vector space" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.1707&rep=rep1&type=pdf>), *Formalized Mathematics* **5** (8): 423–428, .
- [6] Kirkpatrick, David; Mishra, Bhubaneswar; Yap, Chee-Keng (1992), "Quantitative Steinitz's theorems with applications to multifingered grasping", *Discrete & Computational Geometry* **7** (1): 295–318, doi:10.1007/BF02187843.
- [7] Rosenthal, Peter (1987), "The remarkable theorem of Lévy and Steinitz", *American Mathematical Monthly* **94** (4): 342–351, JSTOR 2323094.
- [8] Banaszczyk, Wojciech (1991). "Chapter 3.10 The Lévy-Steinitz theorem". *Additive subgroups of topological vector spaces*. Lecture Notes in Mathematics. **1466**. Berlin: Springer-Verlag. pp. viii+178. ISBN 3-540-53917-4. MR1119302.
- [9] Kadets, V. M.; Kadets, M. I. (1991). "Chapter 6 The Steinitz theorem and B -convexity". *Rearrangements of series in Banach spaces*. Translations of Mathematical Monographs. **86** (Translated by Harold H. McFaden from the Russian-language (Tartu) 1988 ed.). Providence, RI: American Mathematical Society. pp. iv+123. ISBN 0-8218-4546-2. MR1108619.
- [10] Kadets, Mikhail I.; Kadets, Vladimir M. (1997). "Chapter 2.1 Steinitz's theorem on the sum range of a series, Chapter 7 The Steinitz theorem and B -convexity". *Series in Banach spaces: Conditional and unconditional convergence*. Operator Theory: Advances and Applications. **94** (Translated by Andrei Iacob from the Russian-language ed.). Basel: Birkhäuser Verlag. pp. viii+156. ISBN 3-7643-5401-1. MR1442255.
- [11] Balinski, M. L. (1961), "On the graph structure of convex polyhedra in n -space" (<http://projecteuclid.org/euclid.pjm/1103037323>), *Pacific Journal of Mathematics* **11** (2): 431–434, MR0126765, .
- [12] Venkatasubramanian, Suresh (2006), *Planar graphs and Steinitz's theorem* (<http://geomblog.blogspot.com/2006/08/planar-graphs-and-steinitzs-theorem.html>), .
- [13] Ribó Mor, Ares; Rote, Günter; Schulz, André, "Small Grid Embeddings of 3-Polytopes", *Discrete & Computational Geometry* **45** (1): 65–87, doi:10.1007/s00454-010-9301-0, .
- [14] Buchin, Kevin; Schulz, André (2010), "On the number of spanning trees a planar graph can have", *Algorithms - 18th Annual European Symposium (ESA 2010)*, Lecture Notes in Computer Science, **6346**, Springer-Verlag, pp. 110–121, doi:10.1007/978-3-642-15775-2.
- [15] Schulz, André (2011), "Drawing 3-polytopes with good vertex resolution" (<http://jgaa.info/accepted/2011/Schulz2011.15.1.pdf>), *Journal of Graph Algorithms and Applications* **15** (1): 33–52, .
- [16] Barnette, David; Grünbaum, Branko (1970), "Preassigning the shape of a face" (<http://projecteuclid.org/euclid.pjm/1102977361>), *Pacific Journal of Mathematics* **32** (2): 299–306, MR0259744, .
- [17] Barnette, David W. (1970), "Projections of 3-polytopes", *Israel Journal of Mathematics* **8** (3): 304–308, doi:10.1007/BF02771563.
- [18] Ziegler, Günter M. (2007), "Convex polytopes: extremal constructions and f -vector shapes. Section 1.3: Steinitz's theorem via circle packings", in Miller, Ezra; Reiner, Victor; Sturmfels, Bernd, *Geometric Combinatorics*, IAS/Park City Mathematics Series, **13**, American Mathematical Society, pp. 628–642, ISBN 978-0-8218-3736-8.
- [19] Schramm, Oded (1992), "How to cage an egg", *Inventiones Mathematicae* **107** (3): 543–560, doi:10.1007/BF01231901, MR1150601.
- [20] Richter-Gebert, Jürgen (1996). *Realization Spaces of Polytopes*. Lecture Notes in Mathematics. **1643**. Springer-Verlag. ISBN 978-3-540-62084-6.

Planarity testing

In graph theory, the **planarity testing** problem asks whether, given a graph, that graph is a planar graph (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.

Simple algorithms and planarity characterizations

By Fáry's theorem we can assume the edges in the graph drawing, if any, are straight line segments. Given such a drawing for the graph, we can verify that there are no crossings using well-known line segment intersection algorithms that operate in $O(n \log n)$ time. However, this is not a particularly good solution, for several reasons:

- There's no obvious way to find a drawing, a problem which is considerably more difficult than planarity testing;
- Line segment intersection algorithms are more expensive than good planarity testing algorithms;
- It does not extend to verifying nonplanarity, since there is no obvious way of enumerating all possible drawings.

For these reasons, planarity testing algorithms take advantage of theorems in graph theory that characterize the set of planar graphs in terms that are independent of graph drawings. One of these is Kuratowski's theorem, which states that:

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).

A graph can be demonstrated to be nonplanar by exhibiting a subgraph matching the above description, and this can be easily verified, which places the problem in co-NP. However, this also doesn't by itself produce a good algorithm, since there are a large number of subgraphs to consider (K_5 and $K_{3,3}$ are fixed in size, but a graph can contain $2^{\Omega(m)}$ subdivisions of them).

A simple theorem allows graphs with too many edges to be quickly determined to be nonplanar, but cannot be used to establish planarity. If v is the number of vertices (at least 3) and e is the number of edges, then the following imply nonplanarity:

$$e > 3v - 6 \text{ or};$$

There are no cycles of length 3 and $e > 2v - 4$.

For this reason n can be taken to be either the number of vertices or edges when using big O notation with planar graphs, since they differ by at most a constant multiple.

Path addition method

The classic *path addition* method of Hopcroft and Tarjan^[1] was the first published linear-time planarity testing algorithm in 1974.

PQ tree vertex addition method

The *vertex addition* method began with an inefficient $O(n^2)$ method conceived by Lempel, Even and Cederbaum in 1967.^[2] It was improved by Even and Tarjan, who found a linear-time solution for the s,t -numbering step,^[3] 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.^[4] This method was also extended to allow a planar embedding (drawing) to be efficiently computed for a planar graph.^[5]

PC tree vertex addition method

In 1999, Shih and Hsu developed a planarity testing algorithm that was significantly simpler than classical methods based on a new type of data structure called the PC tree and a postorder traversal of the depth-first search tree of the vertices.^[6]

Edge addition method

In 2004, Boyer and Myrvold^[7] 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^{[8][9]}). See^[10] 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.^[11] The source code for the planarity test^{[12][13]} and the extraction of multiple Kuratowski subdivisions^[12] is publicly available. Algorithms that locate a Kuratowski subgraph in linear time in vertices were developed by Williamson in the 1980s.^[14]

References

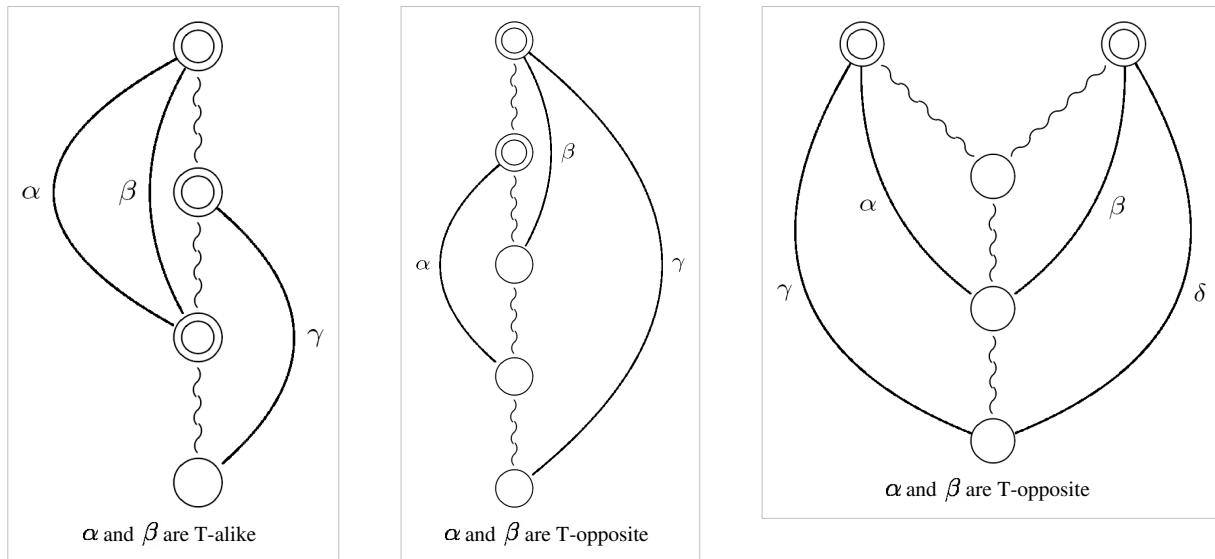
- [1] Hopcroft, John; Tarjan, Robert E. (1974), "Efficient planarity testing", *Journal of the Association for Computing Machinery* **21** (4): 549–568, doi:10.1145/321850.321852.
- [2] Lempel, A.; Even, S.; Cederbaum, I. (1967), "An algorithm for planarity testing of graphs", in Rosenstiehl, P., *Theory of Graphs*, New York: Gordon and Breach, pp. 215–232.
- [3] Even, Shimon; Tarjan, Robert E. (1976), "Computing an *st*-numbering", *Theoretical Computer Science* **2** (3): 339–344, doi:10.1016/0304-3975(76)90086-4.
- [4] Boyer & Myrvold (2004), p. 243: "Its implementation in LEDA is slower than LEDA implementations of many other O(n)-time planarity algorithms."
- [5] Chiba, N.; Nishizeki, T.; Abe, A.; Ozawa, T. (1985), "A linear algorithm for embedding planar graphs using PQ-trees", *Journal of Computer and Systems Sciences* **30** (1): 54–76, doi:10.1016/0022-0000(85)90004-2.
- [6] Shih, W. K.; Hsu, W. L. (1999), "A new planarity test", *Theoretical Computer Science* **223** (1–2): 179–191, doi:10.1016/S0304-3975(98)00120-0.
- [7] Boyer, John M.; Myrvold, Wendy J. (2004), "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, .
- [8] de Fraysseix, H.; Ossona de Mendez, P.; Rosenstiehl, P. (2006), "Trémaux Trees and Planarity", *International Journal of Foundations of Computer Science* **17** (5): 1017–1030, doi:10.1142/S0129054106004248.
- [9] Brandes, Ulrik (2009), *The left-right planarity test* (<http://www.inf.uni-konstanz.de/algo/publications/b-lrpt-sub.pdf>), .
- [10] Boyer, John M.; Cortese, P. F.; Patrignani, M.; Battista, G. D. (2003), "Stop minding your P's and Q's: implementing a fast and simple DFS-based planarity testing and embedding algorithm", *Proc. 11th Int. Symp. Graph Drawing (GD '03)*, Lecture Notes in Computer Science, **2912**, Springer-Verlag, pp. 25–36
- [11] Chimani, M.; Mutzel, P.; Schmidt, J. M. (2008), "Efficient extraction of multiple Kuratowski subdivisions", *Proc. 15th Int. Symp. Graph Drawing (GD'07)*, Lecture Notes in Computer Science, **4875**, Sydney, Australia: Springer-Verlag, pp. 159–170.
- [12] <http://www.ogdf.net>
- [13] http://www.boost.org/doc/libs/1_40_0/libs/graph/doc/boyer_myrvold.html
- [14] Williamson, S. G. (1984), "Depth First Search and Kuratowski Subgraphs", *Journal Association of Computing Machinery* **31**: 681–693

Fraysseix–Rosenstiehl's planarity criterion

In graph theory, a branch of mathematics, **Fraysseix–Rosenstiehl's planarity criterion** is a characterization of planarity based on the properties of the trémaux tree defined by a depth-first search. It is named after Hubert de Fraysseix and Pierre Rosenstiehl.

Considering any depth-first search of a graph G , the edges encountered when discovering a vertex for the first time define a **DFS-tree** T of G . The remaining edges form the **cotree**. Three types of patterns define two relations on the set of the cotree edges, namely 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 (with label of the edge put near the curved arc). In the first figure, α and β are T -alike (it means that their low extremities will be on the same side of the tree in every planar drawing); in the next two figures, they are T -opposite (it means that their low extremities will be on different sides of the tree in every planar drawing).



Let G be a graph and let T be a DFS-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.

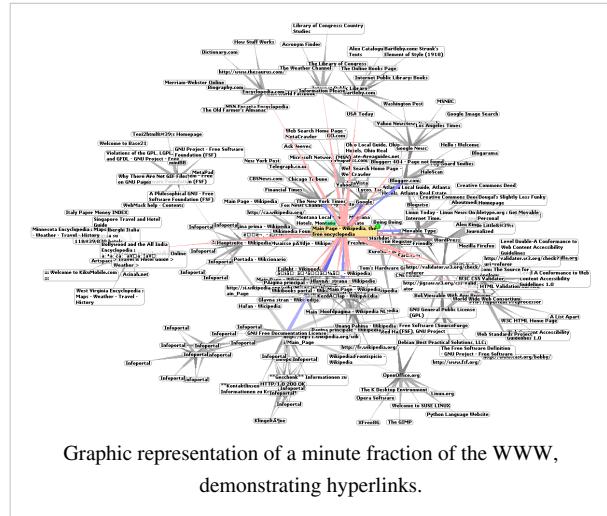
References

- H. de Fraysseix and P. Rosenstiehl, *A depth-first search characterization of planarity*, Annals of Discrete Mathematics **13** (1982), 75–80.

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.^[3] 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^[4].

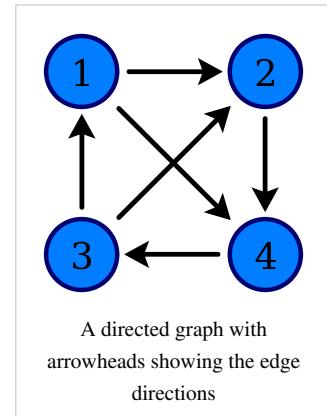


Graphical conventions

Graphs are frequently drawn as **node-link diagrams** in which the vertices are represented as disks or boxes and the edges are represented as line segments, polylines, or curves in the Euclidean plane.^[3]

In the case of directed graphs, arrowheads form a commonly used graphical convention to show their orientation;^[2] however, user studies have shown that other conventions such as tapering provide this information more effectively.^[5]

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.^[6] 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.^[7]

- 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.^[8]
- 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.^[9]
- 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.^[9]

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.^[10]
- Spectral layout methods use as coordinates the eigenvectors of a matrix such as the Laplacian derived from the adjacency matrix of the graph.^[11]
- 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.^[12]
- Tree layout algorithms 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.^[13]
- 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.^[14]

- 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.^[15]

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^[16]
- Hasse diagrams, a type of graph drawing specialized to partial orders^[17]
- Dessin d'enfants, a type of graph drawing used in algebraic geometry^[18]
- State diagrams, graphical representations of finite state machines^[19]
- Computer network diagrams, depictions of the nodes and connections in a computer network.
- 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.

In addition, the placement and routing steps of electronic design automation are similar in many ways to graph drawing, 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:

- Graphviz, an open-source graph drawing system from AT&T^[20]
- yEd, a widely used graph editor with graph layout functionality^[21]
- Microsoft Automatic Graph Layout, a .NET library (formerly called GLEE) for laying out graphs^[22]
- Tom Sawyer Software^[23] 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)^[24]
- Gephi, an open-source network analysis and visualization software
- Cytoscape

Notes

[1] Di Battista et al. (1994), pp. vii–viii; Herman, Melançon & Marshall (2000), Section 1.1, "Typical Application Areas".

[2] Di Battista et al. (1994), p. 6.

[3] Di Battista et al. (1994), p. viii.

[4] Misue et al. (1995)

[5] Holten & van Wijk (2009); Holten et al. (2011).

[6] Di Battista et al. (1994), Section 2.1.2, Aesthetics, pp. 14–16; Purchase, Cohen & James (1997).

[7] Di Battista et al. (1994), p 14.

[8] Di Battista et al. (1994), p. 16.

[9] Pach & Sharir (2009).

[10] Di Battista et al. (1994), Section 2.7, "The Force-Directed Approach", pp. 29–30, and Chapter 10, "Force-Directed Methods", pp. 303–326.

[11] Beckman (1994); Koren (2005).

[12] Di Battista et al. (1994), Chapter 5, "Flow and Orthogonal Drawings", pp. 137–170; (Eiglsperger, Fekete & Klau 2001).

[13] Herman, Melançon & Marshall (2000), Section 2.2, "Traditional Layout – An Overview".

- [14] Sugiyama, Tagawa & Toda (1981); Bastert & Matuszewski (2001); Di Battista et al. (1994), Chapter 9, "Layered Drawings of Digraphs", pp. 265–302.
- [15] Doğrusöz, Madden & Madden (1997).
- [16] Scott (2000).
- [17] Di Battista et al. (1994), pp. 15–16, and Chapter 6, "Flow and Upward Planarity", pp. 171–214; Freese (2004).
- [18] Zapponi (2003).
- [19] Anderson & Head (2006).
- [20] "Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools", by John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull, in Jünger & Mutzel (2004).
- [21] "yFiles – Visualization and Automatic Layout of Graphs", by Roland Wiese, Markus Eiglsperger, and Michael Kaufmann, in Jünger & Mutzel (2004).
- [22] Nachmanson, Robertson & Lee (2008).
- [23] Madden et al. (1996).
- [24] "Tulip – A Huge Graph Visualization Framework", by David Auber, in Jünger & Mutzel (2004).

References

- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.

- Jünger, Michael; Mutzel, Petra (2004), *Graph Drawing Software*, Springer-Verlag, ISBN 978-3-540-00881-1.
- 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, MR2154691.
- 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.
- 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.
- 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.
- 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.
- 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, Mitsuhiro (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, MR0611436.
- 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, ISSN 0002-9920.

External links

- 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 the Open Directory Project for many additional links related to graph drawing.

Force-based graph drawing algorithms

Force-based or **force-directed** 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. The idea originated in the 1980s with a spring-embedder model of Eades and Kamada-Kawai; the term **force-directed** comes from Fruchterman & Reingold's 1990 University of Illinois technical report (UIUCDCS-R-90-1609).

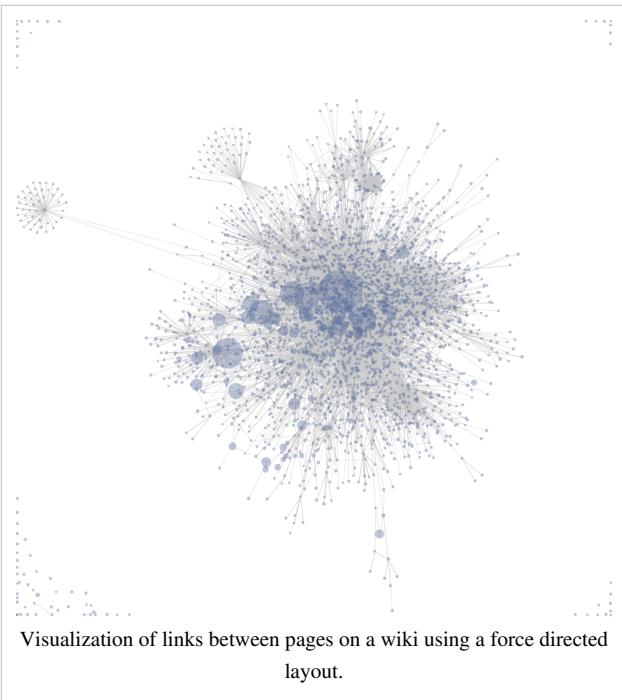
The force-directed algorithms achieve this by assigning forces among the set of edges and the set of nodes; the most straightforward method is to assign forces as if the edges were springs (see Hooke's law) and the nodes were electrically charged particles (see Coulomb's law). The entire graph is then simulated as if it were a physical system. 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 an equilibrium state; i.e., their relative positions do not change anymore from one iteration to the next. At that moment, the graph is drawn. The physical interpretation of this equilibrium state is that all the forces are in mechanical equilibrium.

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 . In this model, there is no need for a separate repulsive force. Note that minimizing the difference (usually the squared difference) between euclidean and ideal distances between nodes is then equivalent to a metric multidimensional scaling problem. 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.

A force-directed graph can involve forces other than mechanical springs and electrical repulsion; examples include logarithmic springs (as opposed to linear springs), gravitational forces (which aggregate connected components in graphs that are not singly connected), magnetic fields (so as to obtain good layouts for directed graphs), and electrically charged springs (in order to avoid overlap or near-overlap in the final drawing). In the case of spring-and-charged-particle graphs, 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).

While graph drawing is a difficult problem, force-directed algorithms, being physical simulations, usually require no special knowledge about graph theory such as planarity.

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,^[1] 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 (such as the one given in pseudo-code in this article) 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.^[2] 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 such as the one used in the pseudo-code below, 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,^[3] multi-layer drawing and other methods related to N-body simulation. For example, the Barnes–Hut simulation-based method FADE^[4] 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.^[4] Force-directed algorithm, when combined with a multilevel approach, can draw graphs of millions of nodes.^[5]

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^[6] to quickly generate a reasonable initial layout and then the Fruchterman-Reingold algorithm^[7] to improve the placement of neighbouring nodes. Another technique to achieve global minimum is to use a multilevel approach.

Pseudocode

Each node has x,y position and dx,dy velocity and mass m. There is usually a spring constant, s, and damping: $0 < \text{damping} < 1$. The force toward and away from nodes is calculated according to Hooke's Law and Coulomb's law or similar as discussed above. The example can be trivially expanded to include a z position for 3D representation.

```

set up initial node velocities to (0,0)
set up initial node positions randomly // make sure no 2 nodes are in exactly the same position
loop
    total_kinetic_energy := 0 // running sum of total kinetic energy over all particles
    for each node
        net-force := (0, 0) // running sum of total force on this particular node

        for each other node
            net-force := net-force + Coulomb_repulsion( this_node, other_node )

        next node

        for each spring connected to this node
            net-force := net-force + Hooke_attraction( this_node, spring )
        next spring

        // without damping, it moves forever
        this_node.velocity := (this_node.velocity + timestep * net-force) * damping
        this_node.position := this_node.position + timestep * this_node.velocity
        total_kinetic_energy := total_kinetic_energy + this_node.mass * (this_node.velocity)^2
    next node
until total_kinetic_energy is less than some small number // the simulation has stopped moving

```

References

- [1] Vose, Aaron. "3D Phylogenetic Tree Viewer" (<http://www.aaronvose.com/phytree3d/>). . Retrieved 3 June 2012.
- [2] de Leeuw, J. (1988)
- [3] Harel, D., Koren Y. (2002)
- [4] Quigley A., Eades P. (2001)
- [5] "A Gallery of Large Graphs" (<http://www2.research.att.com/~yifanhu/GALLERY/GRAPHS/>). . Retrieved 1 July 2012.
- [6] Kamada, T., Kawai, S. (1989)
- [7] Fruchterman, T. M. J., & Reingold, E. M. (1991)

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.
- Eades, Peter (1984). "A Heuristic for Graph Drawing". *Congressus Numerantium* **42** (11): 149–160.
- Fruchterman, Thomas M. J.; Reingold, Edward M. (1991). "Graph Drawing by Force-Directed Placement" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.8444>). *Software – Practice & Experience* (Wiley) **21** (11): 1129–1164. doi:10.1002/spe.4380211102.
- Harel, David; Koren, Yehuda (2002). "Graph Drawing by High-Dimensional Embedding" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.5390>). *Proceedings of the 9th International Symposium on Graph Drawing*. pp. 207–219. ISBN 3-540-00158-1.
- Kamada, Tomihisa; Kawai, Satoru (1989). "An algorithm for drawing general undirected graphs". *Information Processing Letters* (Elsevier) **31** (1): 7–15. doi:10.1016/0020-0190(89)90102-6.
- 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. ISBN 978-3-540-42062-0.
- de Leeuw, Jan (1988). "Convergence of the majorization method for multidimensional scaling". *Journal of Classification* (Springer) **5** (2): 163–180. doi:10.1007/BF01897162.
- Quigley, Aaron; Eades, Peter (2001). "FADE: Graph Drawing, Clustering, and Visual Abstraction" (<http://www.cs.ucd.ie/staff/aquigley/home/downloads/aq-gd2000.pdf>) (PDF). *Proceedings of the 8th International Symposium on Graph Drawing*. pp. 197–210. ISBN 3-540-41554-8.

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>)
- Short explanation of the Kamada-Kawai spring-based graph layout algorithm featuring a picture (<https://nwb.slis.indiana.edu/community/?n=VisualizeData.Kamada-Kawaii>)
- Short explanation of Fruchterman-Reingold algorithm. The algorithm implements a variable step width ("temperature") to guarantee that the system reaches equilibrium state (<https://nwb.slis.indiana.edu/community/?n=VisualizeData.Fruchterman-Rheingold>)
- 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)
- Implementation of a Force Directed Graph with C# including video demonstration (<http://chris.widdowson.id.au/?p=406>)
- Interactive and real-time force directed graphing algorithms used in an online database modeling tool (<http://www.anchormodeling.com/modeler>)

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.

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".^[1]

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**).^[2] A **2-cell embedding** or **map** is an embedding in which every face is homeomorphic to an open disk.^[3] 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 .^[2]

Combinatorial embedding

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".^{[4][5][6]}

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).^[7]

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.^[8]

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.^[9]

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.

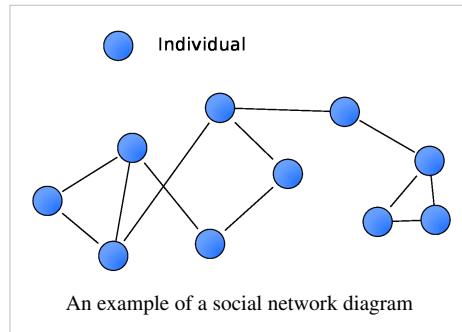
An embedding of a graph into three-dimensional space in which no two of the cycles are topologically linked is called a linkless embedding. A graph has a linkless embedding if and only if it does not have one of the seven graphs of the Petersen family as a minor.

References

- [1] Katoh, Naoki; Tanigawa, Shin-ichi (2007), "Enumerating Constrained Non-crossing Geometric Spanning Trees", *Computing and Combinatorics, 13th Annual International Conference, COCOON 2007, Banff, Canada, July 16-19, 2007, Proceedings*, Lecture Notes in Computer Science, **4598**, Springer-Verlag, pp. 243–253, doi:10.1007/978-3-540-73545-8_25, ISBN 978-3-540-73544-1.
- [2] Gross, Jonathan; Tucker, Tom (2001), *Topological Graph Theory*, Dover Publications, ISBN 0-486-41741-7.
- [3] Lando, Sergei K.; Zvonkin, Alexander K. (2004), *Graphs on Surfaces and their Applications*, Springer-Verlag, ISBN 3-540-00203-0.
- [4] Mutzel, Petra; Weiskircher, René (2000), "Computing Optimal Embeddings for Planar Graphs", *Computing and Combinatorics, 6th Annual International Conference, COCOON 2000, Sydney, Australia, July 26–28, 2000, Proceedings*, Lecture Notes in Computer Science, **1858**, Springer-Verlag, pp. 95–104, doi:10.1007/3-540-44968-X_10, ISBN 978-3-540-67787-1.
- [5] Didjев, Hristo N. (1995), "On drawing a graph convexly in the plane", *Graph Drawing, DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994, Proceedings*, Lecture Notes in Computer Science, **894**, Springer-Verlag, pp. 76–83, doi:10.1007/3-540-58950-3_358.
- [6] Duncan, Christian; Goodrich, Michael T.; Kobourov, Stephen (2010), "Planar Drawings of Higher-Genus Graphs", *Graph Drawing, 17th International Symposium, GD 2009, Chicago, IL, USA, September 22–25, 2009, Revised Papers*, Lecture Notes in Computer Science, **5849**, Springer-Verlag, pp. 45–56, doi:10.1007/978-3-642-11805-0_7, ISBN 978-3-642-11804-3.
- [7] Thomassen, Carsten (1989), "The graph genus problem is NP-complete", *Journal of Algorithms* **10** (4): 568–576, doi:10.1016/0196-6774(89)90006-0
- [8] Filotti, I. S.; Miller, Gary L.; Reif, John (1979), "On determining the genus of a graph in $O(v O(g))$ steps(Preliminary Report)", *Proc. 11th Annu. ACM Symposium on Theory of Computing*, pp. 27–37, doi:10.1145/800135.804395.
- [9] Mohar, Bojan (1999), "A linear time algorithm for embedding graphs in an arbitrary surface", *SIAM Journal on Discrete Mathematics* **12** (1): 6–26, doi:10.1137/S089548019529248X

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.

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

- Free software tool (Win & Mac) to make Sociograms. (<http://www.phenotyping.com/sociogram>)

Application: Concept maps

For concept maps in generic programming, see *Concept (generic programming)*. A **concept map** is a diagram showing the relationships among concepts. It is a graphical tool for organizing and representing knowledge.

Concepts, usually represented as boxes or circles, are connected with labeled arrows in a downward-branching hierarchical structure. The relationship between concepts can be articulated in linking phrases such as "gives rise to", "results in", "is required by," or "contributes to".^[1]

The technique for visualizing these relationships among different concepts is called "concept mapping".

Concept maps are used to 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 is connected to another and linked 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.^[2]

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. There is research evidence that knowledge is stored in the brain in the form of productions (situation-response conditionals) that act on declarative memory content which is also referred to as chunks or propositions.^{[3][4]} 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 philosophical basis, which "makes concepts, and propositions composed of concepts, the central elements in the structure of knowledge and construction of meaning."^[5] 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 conceptual center.

History

The technique of concept mapping was developed by Joseph D. Novak^[6] 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.

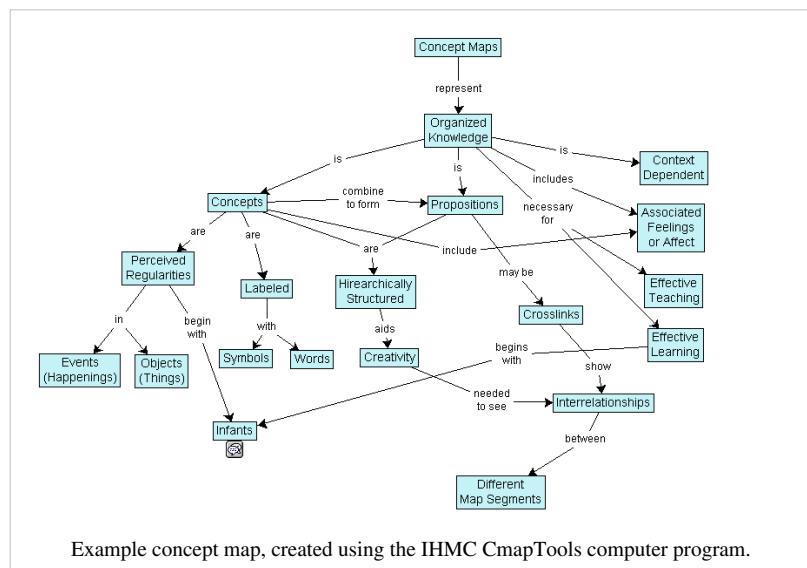
Novak's work is based on the cognitive theories of David Ausubel (assimilation theory), who stressed the importance of prior knowledge in being able to learn new concepts: "The most important single factor influencing learning is what the learner already knows. Ascertain this and teach accordingly."^[7] 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. For example, concept mapping is 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 gleaning 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" which 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.

- Increasing meaningful learning for example through writing activities where concept maps automatically generated from an essay are shown to the writer.^[13]
- 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

References

- [1] 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.
- [2] CONCEPT MAPPING FUELS (http://www.energyeducation.tx.gov/pdf/223_inv.pdf). Accessed 24 Nov 2008.
- [3] Anderson, J. R., & Lebiere, C. (1998). The atomic components of thought. Mahwah, NJ: Erlbaum.
- [4] 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.
- [5] Novak, J.D. & Gowin, D.B. (1996). Learning How To Learn, Cambridge University Press: New York, p. 7.
- [6] "Joseph D. Novak" (<http://www.ihmc.us/users/user.php?UserID=jnovak>). Institute for Human and Machine Cognition (IHMC).. Retrieved 2008-04-06.
- [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] Villalon, Jorge; Rafael Calvo (2011). "Concept maps as cognitive visualizations of writing assignments" (http://www.ifets.info/download_pdf.php?j_id=52&a_id=1149) (PDF). *Journal of Educational Technology and Society* 14 (3): 16–27.. Retrieved 2011-11-16.

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 multiset of intervals on the real line. It has one vertex for each interval in the set, and an edge between every pair of vertices corresponding to intervals that intersect.

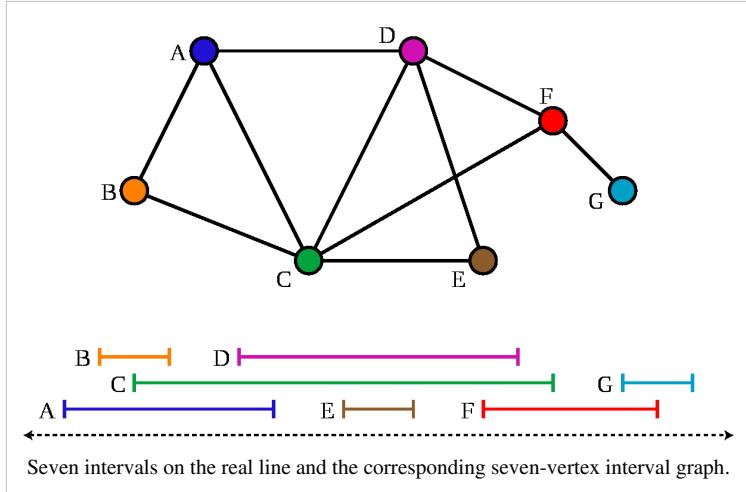
Definition

Let $\{I_1, I_2, \dots, I_n\} \subset P(R)$ be a set of intervals.

The corresponding interval graph is $G = (V, E)$, where

- $V = \{I_1, I_2, \dots, I_n\}$, and
- $\{I_\alpha, I_\beta\} \in E$ if and only if $I_\alpha \cap I_\beta \neq \emptyset$.

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 , $i \leq j \leq k$.^[1]



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.^{[1][2]}

Related families of graphs

Interval graphs are chordal graphs and hence perfect graphs.^{[1][2]} Their complements belong to the class of comparability graphs,^[3] and the comparability relations are precisely the interval orders.^[1]

The interval graphs that have an interval representation in which every two intervals are either disjoint or nested are the trivially perfect graphs.

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. Every proper interval graph is a claw-free graph. However, the converse is not true. Every claw-free graph is not necessarily a proper interval graph.^[4] If the collection of segments in question is a set, i.e., no repetitions of segments is allowed, then the graph is unit interval graph if and only if it is proper interval graph.

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.^[5]

The connected triangle-free interval graphs are exactly the caterpillar trees.^[6]

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.^[7] Cohen applied interval graphs to mathematical models of population biology, specifically food webs.^[8]

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.^[9] Interval graphs are used to represent resource allocation problems in operations research and scheduling theory. Each interval represents a request for a resource 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.^[10] Interval graphs also play an important role in temporal reasoning.^[11]

Notes

[1] (Fishburn 1985)

[2] Golumbic (1980).

[3] Gilmore & Hoffman (1964)

[4] Faudree, Flandrin & Ryjáček (1997), p. 89.

[5] Bodlaender (1998).

[6] Eckhoff (1993).

[7] Cohen (1978, pp. ix-10 (http://books.google.se/books/princeton?hl=en&q=interval+graph&vid=ISBN9780691082028&redir_esc=y#v=snippet&q=%22interval%20graph%22&f=false))

[8] Cohen (1978, pp. 12–33 (http://books.google.se/books/princeton?hl=en&q=interval+graph&vid=ISBN9780691082028&redir_esc=y#v=snippet&q=%22interval%20graph%22&f=false))

[9] Zhang et al. (1994).

[10] Bar-Noy et al. (2001).

[11] Golumbic & Shamir (1993).

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.
- 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.
- 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.
- 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.
- 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, MR1432221.
- 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.
- 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.
- 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.
- 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.
- 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.

External links

- "interval graph" (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_234.html). *Information System on Graph Class Inclusions* (<http://wwwteo.informatik.uni-rostock.de/isgci/index.html>).
- Weisstein, Eric W., " Interval graph (<http://mathworld.wolfram.com/IntervalGraph.html>)" from MathWorld.

Chordal graph

In the mathematical area of graph theory, a graph is **chordal** if each of its cycles of four or more nodes has a *chord*, which is an edge joining two nodes that are not adjacent in the cycle. An equivalent definition is that any chordless cycle has at most three nodes. Some also state this as that a chordal graph is a graph with no induced cycles of length more than three.

Chordal graphs are a subset of the perfect graphs. They are sometimes also called **rigid circuit graphs**^[1] or **triangulated graphs** (the latter term is sometimes erroneously used for plane triangulations (maximal planar graphs)).^[2]

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 (Fulkerson & Gross 1965).

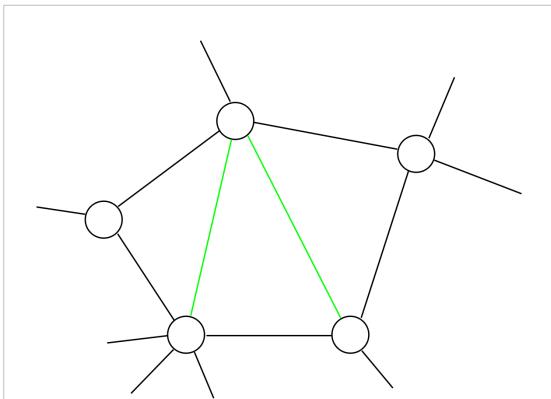
Rose, Lueker & Tarjan (1976) (see also Habib et al. 2000) show that a perfect elimination ordering of a chordal graph may be found efficiently using an algorithm known as lexicographic breadth-first search. This algorithm maintains a partition of the vertices of the graph into a sequence of sets; initially 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 (Bodlaender, Fellows & Warnow 1992), whereas the probe graph problem on chordal graphs has polynomial-time complexity (Berry, Golumbic & Lipshteyn 2007).

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 (Maffray



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.

2003).

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), the chordal graphs are exactly the 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**.^[3]

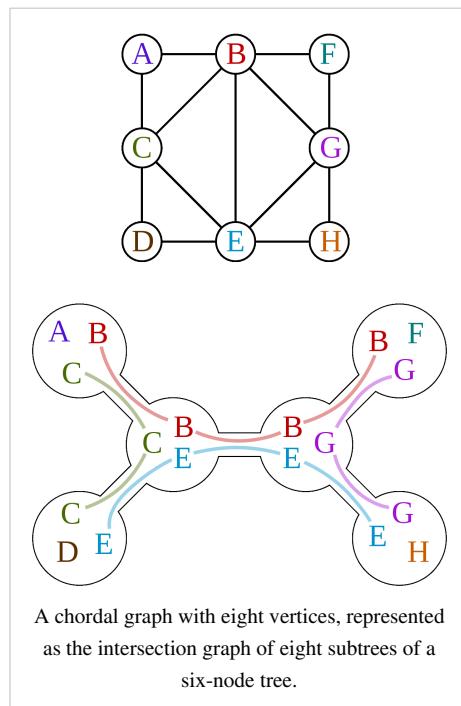
Intersection graphs of subtrees

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. As Gavril showed, 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



Subclasses

Interval graphs are the intersection graphs of subtrees of path graphs, a special case of trees; therefore, they are a subfamily of the chordal graphs.

Split graphs are exactly the 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 exactly the graphs that are both chordal and distance hereditary. Quasi-threshold graphs are a subclass of ptolemaic graphs that are exactly the 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 the 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.

The k -trees are the chordal graphs in which all maximal cliques and all maximal clique separators have the same size.^[4] The Apollonian networks are the chordal maximal planar graphs, or equivalently the planar 3-trees.^[4] The 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).

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, MR0770128.
- 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.
- 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*.
- Chandran, L. S.; Ibarra, L.; Ruskey, F.; Sawada, J. (2003), "Enumerating and characterizing the perfect elimination orderings of a chordal graph"^[5], *Theoretical Computer Science* **307** (2): 303–317, doi: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, MR0130190.
- Fulkerson, D. R.; Gross, O. A. (1965), "Incidence matrices and interval graphs"^[6], *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.
- 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"^[7], *Theoretical Computer Science* **234**: 59–84, doi: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, ISBN 0-387-95434-1.
- Patil, H. P. (1986), "On the structure of k -trees", *Journal of Combinatorics, Information and System Sciences* **11** (2–4): 57–64, MR966069.
- 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.

Notes

- [1] Dirac (1961).
- [2] Weisstein, Eric W., " Triangulated Graph (<http://mathworld.wolfram.com/TriangulatedGraph.html>)" from MathWorld.
- [3] Peter Bartlett. "Undirected Graphical Models: Chordal Graphs, Decomposable Graphs, Junction Trees, and Factorizations:" (<http://www.stat.berkeley.edu/~bartlett/courses/241A-spring2007/graphnotes.pdf>). .
- [4] Patil (1986).
- [5] <http://www.cis.uoguelph.ca/~sawada/papers/chordal.pdf>
- [6] <http://projecteuclid.org/Dienst/UI/1.0/Summarize/euclid.pjm/1102995572>
- [7] <http://www.cs.colostate.edu/~rmm/lexbfs.ps>

External links

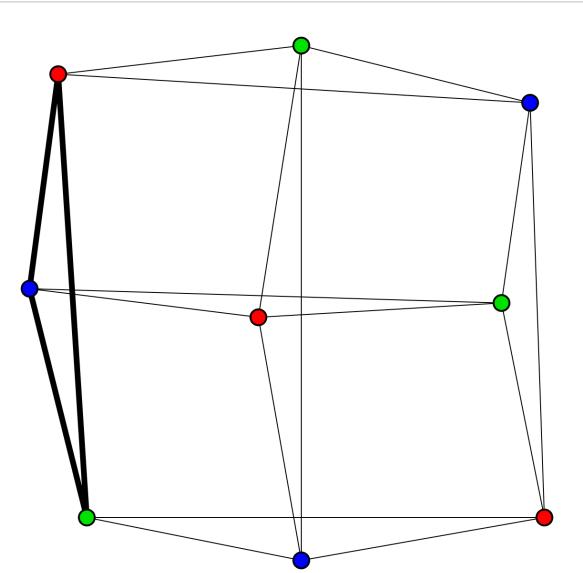
- Information System on Graph Class Inclusions (<http://wwwteo.informatik.uni-rostock.de/isgci/index.html>): chordal graph (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_32.html)
- Weisstein, Eric W., " Chordal Graph (<http://mathworld.wolfram.com/ChordalGraph.html>)" from MathWorld.

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. Perfect graphs are the same as the **Berge graphs**, graphs that have no odd-length induced cycle or induced complement of an odd cycle.

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 later proven as the strong perfect graph theorem.



The Paley graph of order 9, colored with three colors and showing a clique of three vertices. In this graph and each of its induced subgraphs the chromatic number equals the clique number, so it is a perfect graph.

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

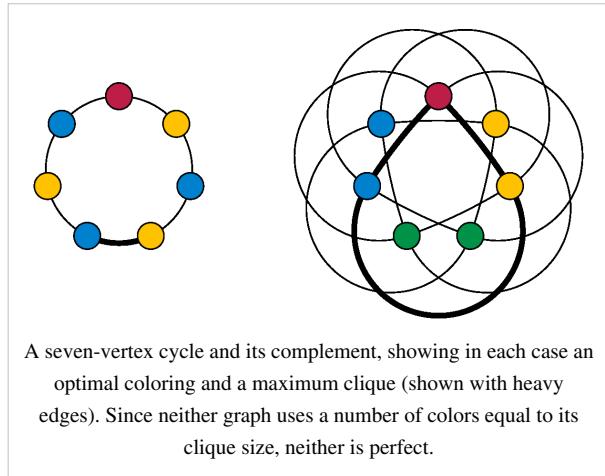
perfection of the line graphs of bipartite graphs.

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 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.
- Chudnovsky, Maria; Robertson, Neil; Seymour, Paul; Thomas, Robin (2006). "The strong perfect graph theorem"^[1]. *Annals of Mathematics* **164** (1): 51–229. doi:10.4007/annals.2006.164.51.
- Gallai, Tibor (1958). "Maximum-minimum Sätze über Graphen". *Acta Math. Acad. Sci. Hungar.* **9** (3-4): 395–434. doi:10.1007/BF02020271.
- Golumbic, Martin Charles (1980). *Algorithmic Graph Theory and Perfect Graphs*^[2]. 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.
- 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.
- 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*^[3] by Václav Chvátal.
- Open problems on perfect graphs^[4], maintained by the American Institute of Mathematics.
- *Perfect Problems*^[5], maintained by Václav Chvátal.
- Information System on Graph Class Inclusions^[6]: perfect graph^[7]

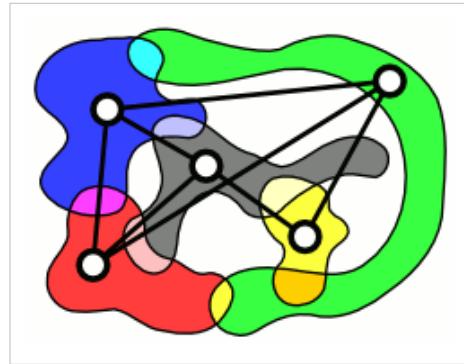
References

- [1] <http://annals.princeton.edu/annals/2006/164-1/p02.xhtml>
- [2] http://www.elsevier.com/wps/find/bookdescription.cws_home/699916/description#description
- [3] <http://www.cs.concordia.ca/~chvatal/perfect/spt.html>
- [4] <http://www.aimath.org/WWN/perfectgraph/>
- [5] <http://www.cs.concordia.ca/~chvatal/perfect/problems.html>
- [6] <http://wwwteo.informatik.uni-rostock.de/isgci/index.html>
- [7] http://wwwteo.informatik.uni-rostock.de/isgci/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 may be represented as an intersection graph, but some important special classes of graphs may be defined by the types of sets that are used to form an intersection representation of them.

For an overview of the theory of intersection graphs, and of important special classes of intersection graphs, see McKee & McMorris (1999).



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.
- 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.
- 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.

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, MR0176940.
- Erdős, Paul; Goodman, A. W.; Pósa, Louis (1966), "The representation of a graph by set intersections" ^[1], *Canadian Journal of Mathematics* **18** (1): 106–112, doi:10.4153/CJM-1966-014-3, MR0186575.
- 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, MR1672910
- Szpilrajn-Marczewski, E. (1945), "Sur deux propriétés des classes d'ensembles", *Fund. Math.* **33**: 303–307, MR0015448.
- Schaefer, Marcus (2010), "Complexity of Some Geometric and Topological Problems" ^[2], *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, ISBN 978-3-642-11804-3.

External links

- Jan Kratochvíl, A video lecture on intersection graphs (June 2007) ^[3]
- E. Prisner, A Journey through Intersection Graph County ^[4]

References

- [1] http://www.renyi.hu/~p_erdos/1966-21.pdf
[2] <http://ovid.cs.depaul.edu/documents/convex.pdf>
[3] http://videolectures.net/sicgt07_kratochvil_gig/
[4] <http://www.eprisner.de/Journey/Rahmen.html>

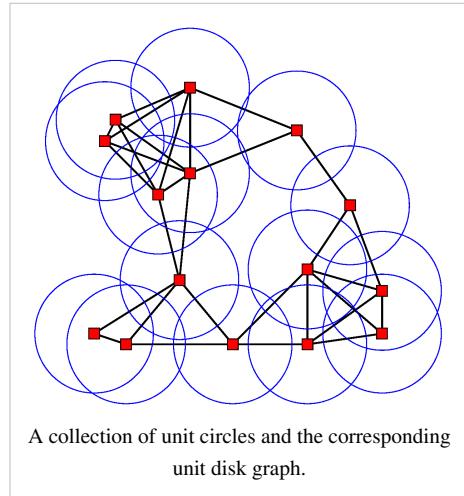
Unit disk graph

In geometric graph theory, a **unit disk graph** is the intersection graph of a family of unit circles in the Euclidean plane. That is, we form a vertex for each circle, and connect two vertices by an edge whenever the corresponding circles cross each other.

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. 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

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.^[3]

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,^[4] and the maximum clique problem can be solved exactly for these graphs in polynomial time, given a disk representation.^[5] 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.^[6]

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.^[7] 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., Dall & Christensen (2002).
- [2] Breu & Kirkpatrick (1998); Kang & Müller (2011).
- [3] McDiarmid & Mueller (2011).
- [4] Marathe et al. (1994); Matsui (2000).
- [5] Clark, Colbourn & Johnson (1990).
- [6] Raghavan & Spinrad (2003).
- [7] Miyamoto & Matsui (2005).

References

- Breu, Heinz; Kirkpatrick, David G. (1998), "Unit disk graph recognition is NP-hard", *Computational Geometry Theory and Applications* **9** (1–2): 3–24.
- 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.
- Dall, Jesper; Christensen, Michael (2002), "Random geometric graphs", *Phys. Rev. E* **66**: 016121, arXiv:cond-mat/0203026, doi: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, 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.
- 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, ISBN 978-3-540-67181-7.
- McDiarmid, Colin; Mueller, Tobias (2011), *Integer realizations of disk and segment graphs*, arXiv: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, 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, MR2006100.

Line graph

In graph theory, the **line graph** $L(G)$ of 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 (Hemminger & Beineke 1978, p. 273). 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** (Hemminger & Beineke 1978, p. 273), as well as the **edge graph**, the **interchange graph**, the **adjoint graph**, and the **derived graph** (Balakrishnan 1997, p. 44).

One of the earliest and most important theorems about line graphs is due to Hassler Whitney (1932), who proved that with one exceptional case the structure of G can be recovered completely from its line graph. In other words, with that one exception, the entire graph can be deduced from knowing the adjacencies of edges ("lines").

Formal definition

Given a graph G , its line graph $L(G)$ is a graph such that

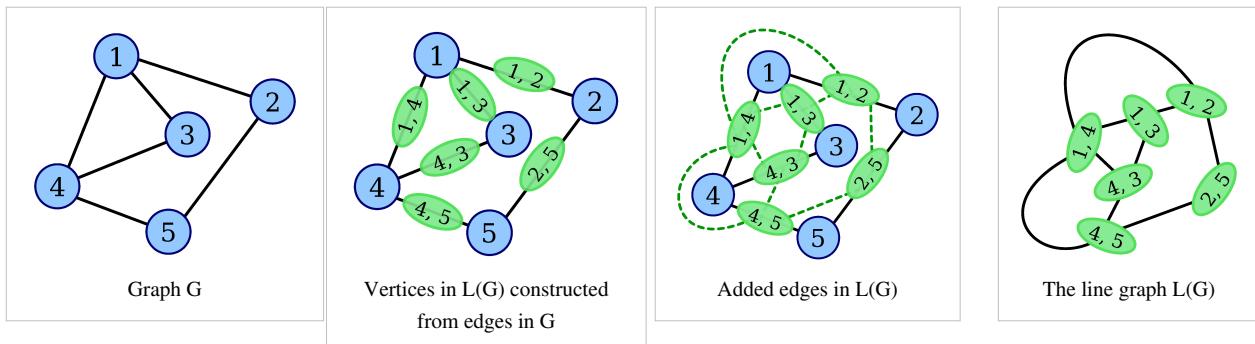
- each vertex of $L(G)$ represents an edge of G ; and
- two vertices of $L(G)$ are adjacent if and only if their corresponding edges share a common endpoint ("are adjacent") in G .

That is, it is the intersection graph of the edges of G , representing each edge by the set of its two endpoints.

Examples

Example construction

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).



Triangular graphs

The line graph of the complete graph K_n was previously known as the **triangular graph**. An important theorem is that the triangular graphs are characterized by their spectra, except for $n = 8$, where there are three other graphs with the same spectrum as $L(K_8)$. The exceptions are explained by graph switching.

Line graphs of convex polyhedra

A source of examples from geometry are the line graphs of the graphs of simple polyhedra. Taking the line graph of the graph of the tetrahedron one gets the graph of the octahedron; from the graph of the cube one gets the graph of a cuboctahedron; from the graph of the dodecahedron one gets the graph of the icosidodecahedron, etc. Geometrically, the operation consists in cutting each vertex of the polyhedron with a plane cutting all edges adjacent to the vertex at their midpoints; it is sometimes named rectification.

If a polyhedron is not simple (it has more than three edges at a vertex) the line graph will be nonplanar, with a clique replacing each high-degree vertex.

Medial graph

The medial graph is a variant of the line graph of a planar graph, in which two vertices of the medial graph are adjacent if and only if the corresponding two edges are consecutive on some face of the planar graph. For simple polyhedra, the medial graph and the line graph coincide, but for non-simple graphs the medial graph remains planar. Thus, the medial graphs of the cube and octahedron are both isomorphic to the graph of the cuboctahedron, and the medial graphs of the dodecahedron and icosahedron are both isomorphic to the graph of the icosidodecahedron.

Properties

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.

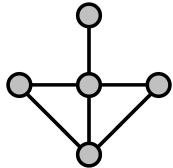
Thus,

- 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.
- 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 $L(G)$.
- The line graph of an edge-transitive graph is vertex-transitive.
- 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.)
- Line graphs are claw-free graphs, graphs without an induced subgraph in the form of a three-leaf tree.
- The line graphs of trees are exactly the claw-free block graph.

Characterization and recognition

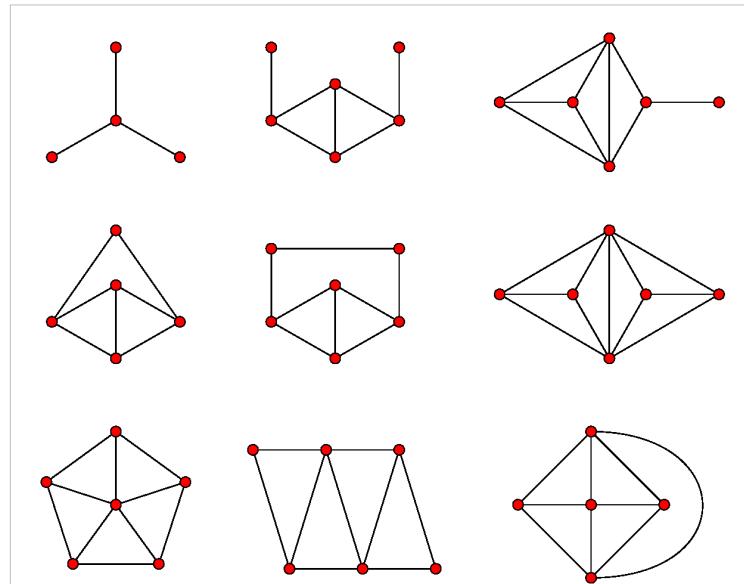
A graph G is the line graph of some other graph if and only if it is possible to find a collection of cliques in G , partitioning the edges of G , such that each vertex of G belongs to exactly two of the cliques. In order to do this, it may be necessary for some of the cliques to be single vertices. By Whitney's theorem (Whitney 1932),^[1] if G is not a triangle, there can be only one partition of this type. If such a partition exists, we can recover the original graph for which G is a line graph, by creating a vertex for each clique, and connecting two cliques by an edge whenever G contains a vertex belonging to both cliques. Therefore, except for the case of K_3 and $K_{1,3}$, if the line graphs of two connected graphs are isomorphic then the graphs are isomorphic. Roussopoulos (1973) used this observation as the basis for a linear time algorithm for recognizing line graphs and reconstructing their original graphs.

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.

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 (Metelsky & Tyshkevich 1997). This result is similar to the results of Line graphs of hypergraphs.^[2]



The nine minimal non-line graphs, from Beineke's forbidden-subgraph characterization of line graphs. A graph is a line graph if and only if it does not contain one of these nine graphs as an induced subgraph.

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 .
- 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 .

Relations to other families of graphs

Every line graph is a claw-free graph. Some of the properties of claw-free graphs are generalizations of those of line graphs.

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 perfect graph theorem. A special case is the rook's graphs, line graphs of complete bipartite graphs.

Generalizations

Multigraphs

The concept of the line graph of G may naturally be extended to the case where G is a multigraph, although in that case Whitney's uniqueness theorem no longer holds; 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.

Line digraphs

It is also possible to generalize line graphs to directed graphs.^[3] 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 (Zhang & Lin 1987).

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, Whitney's theorem (Whitney 1932) 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 (Evans & Lambiotte 2009). 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 .

Line graphs of hypergraphs

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] See also Krausz (1943).
- [2] Weisstein, Eric W., "Line Graph (<http://mathworld.wolfram.com/LineGraph.html>)" from MathWorld.
- [3] Harary, Frank, and Norman, Robert Z., "Some properties of line digraphs", *Rend. Circ. Mat. Palermo, II. Ser.* 9 (1960), 161-168.

References

- Balakrishnan, V. K. (1997), *Schaum's Outline of Graph Theory* (1st ed.), McGraw-Hill, ISBN 0-07-005489-4.
- 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, MR0262097.
- Brandstädt, Andreas; Le, Van Bang; Spinrad, Jeremy P. (1999), *Graph Classes: A Survey*, SIAM Monographs on Discrete Mathematics and Applications, ISBN 0-89871-432-X.
- Evans, T.S.; Lambiotte, R. (2009), "Line Graphs, Link Partitions and Overlapping Communities", *Phys. Rev. E* 80: 016105, doi:10.1103/PhysRevE.80.016105.
- 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.
- 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.
- Krausz, J. (1943), "Démonstration nouvelle d'un théorème de Whitney sur les réseaux", *Mat. Fiz. Lapok* 50: 75–85, MR0018403.
- 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.
- 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.
- 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, MR0424435.
- Whitney, H. (1932), "Congruent graphs and the connectivity of graphs", *American Journal of Mathematics* 54 (1): 150–168, doi:10.2307/2371086, JSTOR 2371086.
- Zhang, Fu Ji; Lin, Guo Ning (1987), "On the de Bruijn–Good graphs", *Acta Math. Sinica* 30 (2): 195–205, MR0891925.

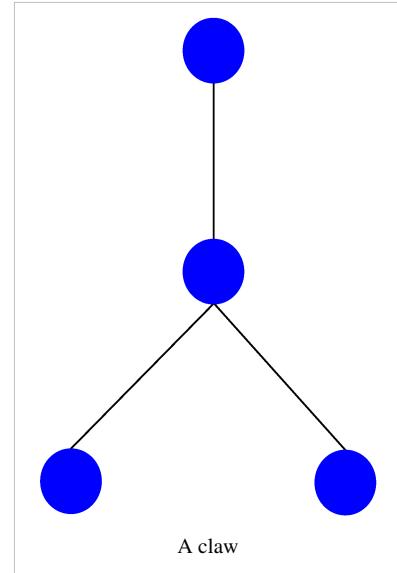
External links

- *line graphs* (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_249.html), Information System on Graph Class Inclusions (<http://wwwteo.informatik.uni-rostock.de/isgci/index.html>)

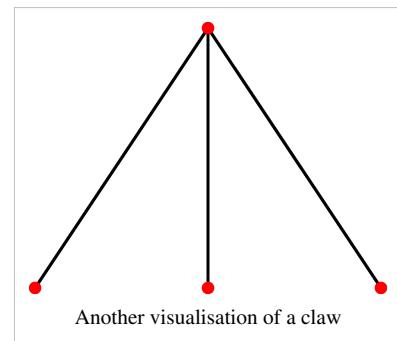
Claw-free graph

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.

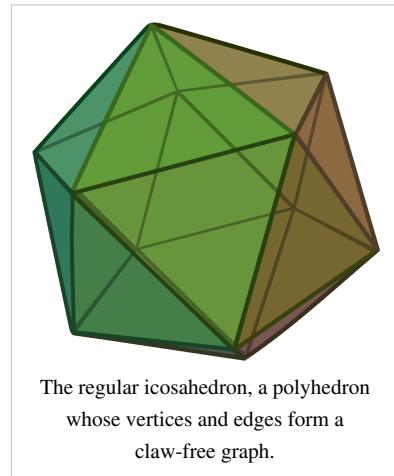


Claw-free graphs were initially studied as a generalization of line graphs, and gained additional motivation through three key discoveries about them: the fact that all claw-free connected graphs of even order have perfect matchings, the discovery of polynomial time algorithms for finding maximum independent 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.^[1]



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,^[2] the claw is the simplest of these nine graphs. This characterization provided the initial motivation for studying claw-free graphs.^[1]
- 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.^[3] 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.^[3]
- 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,^[4] and the graph of the 16-cell.
- The Schläfli graph, a strongly regular graph with 27 vertices, is claw-free.^[4]



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.^[5] 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.^[6] 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.

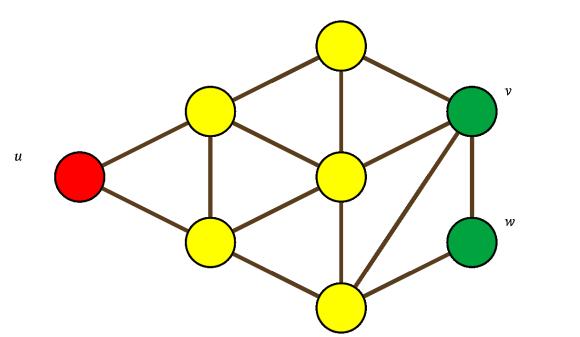
Matchings

Sumner (1974) and, independently, Las Vergnas (1975) proved that every claw-free connected graph with an even number of vertices has a perfect matching.^[7] 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.^[7]

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 depth-first search, as well as efficient parallel algorithms for the same problem.

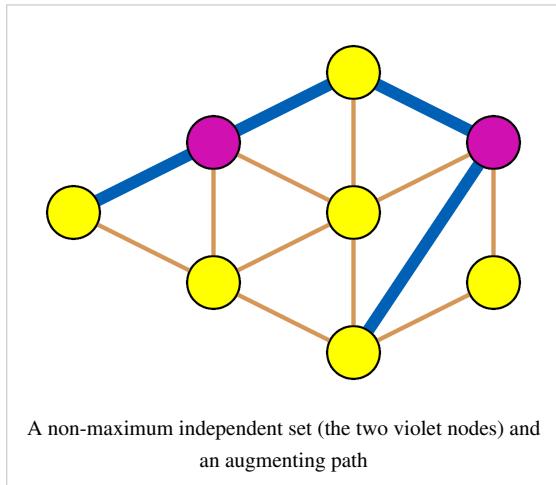
Faudree, Flandrin & Ryjáček (1997) list several related results, including the following: $(r - 1)$ -connected $K_{1,r}$ -free graphs of even order have perfect matchings for any $r \geq 2$; claw-free graphs of odd order with at most one degree-one vertex may be partitioned into an odd cycle and a matching; 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.



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.

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.^[8] 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.^[8]



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*).^[9] 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.^[10]

If a claw-free graph is not perfect, it is NP-hard to find to find its largest clique.^[5] 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.^[5] 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.^[11]

Despite this domination perfectness property, it is NP-hard to determine the size of the minimum dominating set in a claw-free graph.^[5] 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.^[12]

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.^[9] 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,^[4] as well as to new fixed-parameter-tractable algorithms for dominating sets in claw-free graphs.^[13]

Notes

- [1] Faudree, Flandrin & Ryjáček (1997), p. 88.
- [2] Beineke (1968).
- [3] Faudree, Flandrin & Ryjáček (1997), p. 89.
- [4] Chudnovsky & Seymour (2005).
- [5] Faudree, Flandrin & Ryjáček (1997), p. 132.
- [6] Itai & Rodeh (1978).
- [7] Faudree, Flandrin & Ryjáček (1997), pp. 120–124.
- [8] Faudree, Flandrin & Ryjáček (1997), pp. 133–134.
- [9] Chudnovsky et al. (2006).
- [10] Faudree, Flandrin & Ryjáček (1997), pp. 135–136.
- [11] Faudree, Flandrin & Ryjáček (1997), pp. 124–125.
- [12] Cygan et al. (2010); Hermelin et al. (2010).
- [13] Hermelin et al. (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.
- 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.
- 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.
- 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, MR2187738.
- 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..
- Edmonds, Jack (1965), "Paths, Trees and Flowers", *Canadian J. Math* **17** (0): 449–467, doi:10.4153/CJM-1965-045-4, MR0177907.
- 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, MR1432221.
- Hermelin, Danny; Mnich, Matthias; van Leeuwen, Erik Jan; Woeginger, Gerhard (2010). "Domination when the stars are out". arXiv: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, MR0508603.

- 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, MR1761552.
- Las Vergnas, M. (1975), "A note on matchings in graphs", *Cahiers du Centre d'Études de Recherche Opérationnelle* **17** (2-3-4): 257–260, MR0412042.
- 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, MR579076.
- 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, MR1972075.
- 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, MR553650.
- 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, JSTOR 2039666, MR0323648.

External links

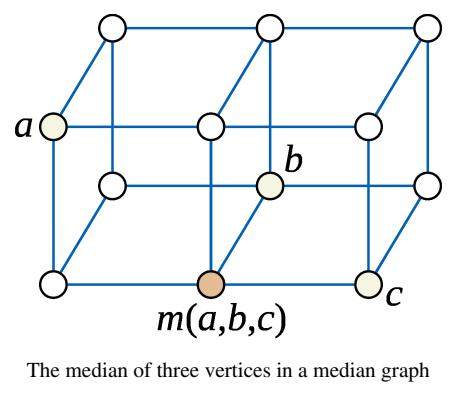
- Claw-free graphs (http://wwwteo.informatik.uni-rostock.de/isgci/classes/gc_62.html), Information System on Graph Class Inclusions
- Mugan, Jonathan William; Weisstein, Eric W., "Claw-Free Graph" (<http://mathworld.wolfram.com/Claw-FreeGraph.html>) from MathWorld.

Median graph

In mathematics, and more specifically graph theory, a **median graph** is an undirected graph in which any three vertices a , b , and c have a unique *median*: a vertex $m(a,b,c)$ that belongs to shortest paths between any two 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".^[1] In phylogenetics, the Buneman graph representing all maximum parsimony evolutionary trees is a median graph.^[2] 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.^[3]

Additional surveys of median graphs are given by Klavžar & Mulder (1999), Bandelt & Chepoi (2008), and Knuth (2008).

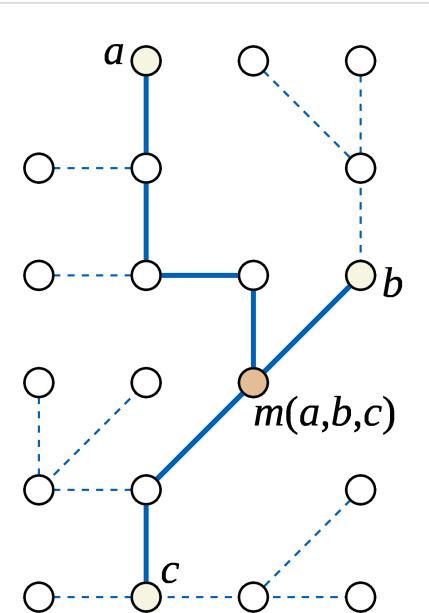


The median of three vertices in a median graph

Examples

Any tree is a median graph.^[4] To see this, observe that in a tree, the union of the three shortest paths between any 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 any 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.^[5]

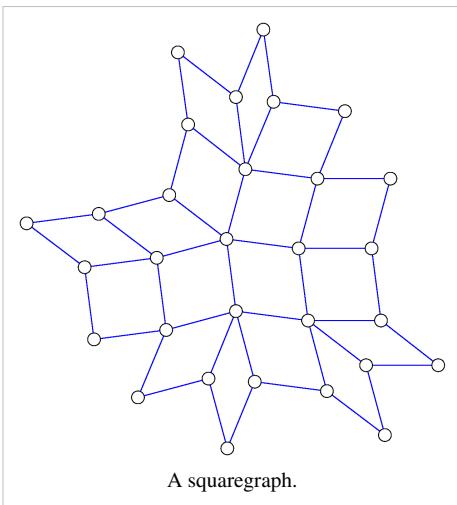


The median of three vertices in a tree, showing the subtree formed by the union of shortest paths between the vertices.

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.^[6] A polyomino is a special case of a squaregraph and therefore also forms a median graph.

The simplex graph $\kappa(G)$ of any 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 any three 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 any three vertices.

No cycle graph of length other than four can be a median graph, because any 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.



A squaregraph.

Equivalent definitions

In any graph, for any two vertices a and b , define the *interval* of vertices that lie on shortest paths

$$I(a,b) = \{v \mid d(a,b) = d(a,v) + d(v,b)\}.$$

A median graph is defined by the property that, for any 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^[7]

$$m(a,b,c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \wedge (b \vee c),$$

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 any 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 any 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 .

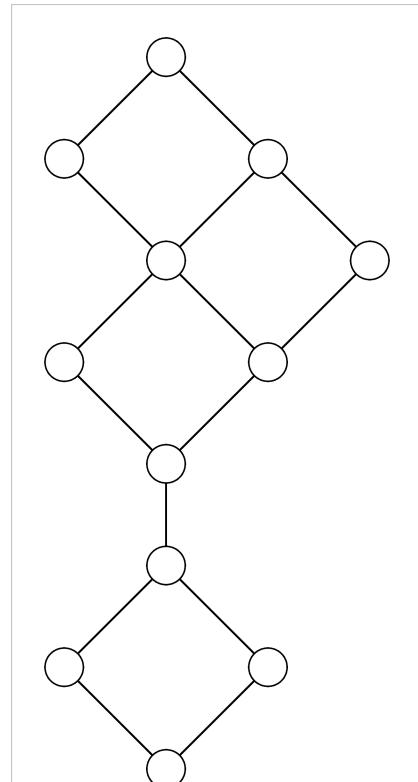
The distributive law may be replaced by an associative law:^[8]

- 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:

$$I(a,b) = \{x \mid m(a,x,b) = x\} = \{x \mid a \wedge b \leq x \leq a \vee b\}.^{[9]}$$

The graph of a finite distributive lattice has an edge between any two vertices a and b whenever $I(a,b) = \{a,b\}$. For any 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 any a , b , and c , $m(a,b,c)$ is the unique intersection of the three intervals $I(a,b)$, $I(a,c)$, and $I(b,c)$.^[10] Therefore, the graph of any finite distributive lattice is a median graph. Conversely, if a



The graph of a distributive lattice, drawn as a Hasse diagram.

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.^[11]

Duffus & Rival (1983) characterize graphs of distributive lattices directly as diameter-preserving retracts of hypercubes. More generally, any median graph gives rise to a ternary operation m satisfying idempotence, commutativity, and distributivity, but possibly without the identity elements of a distributive lattice. Any 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.^[12]

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 any three points at least two of which are from S . Observe that the intersection of any two convex sets is itself convex.^[13]

The convex sets in a median graph have the Helly property: if F is any family of pairwise-intersecting convex sets, then all sets in F have a common intersection.^[14] 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 any shortest path from a to b must lie within T by convexity, and similarly any 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 any two of the 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 any 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 any 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.^[15]

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 any solution is formed by negating a set of variables that are all constrained to be equal or unequal to each other.

Conversely, any 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 . Any 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 .

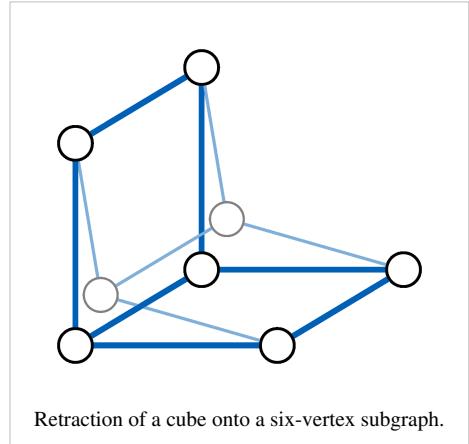
Retracts of hypercubes

A *retraction* of a graph G is an adjacency-preserving map from G to one of its subgraphs.^[16] More precisely, it is graph homomorphism φ from G to itself such that $\varphi(v) = v$ for any 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 any 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 any 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 any 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.^[17]

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 any 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.^[18] 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



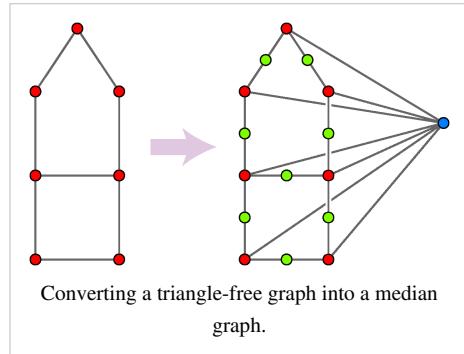
Retraction of a cube onto a six-vertex subgraph.

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.^[19]

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.^[20] Therefore, the best known time bound for testing whether a graph is triangle-free, $O(m^{1.41})$,^[21] 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.

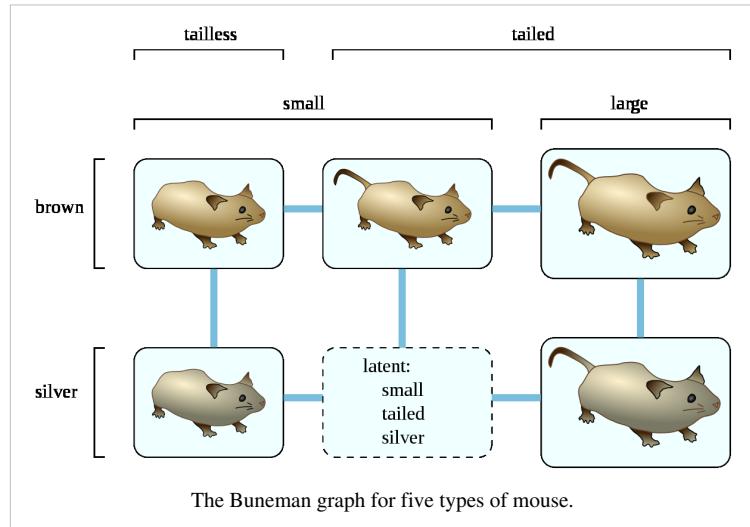


In one direction, suppose one is given as input a graph G , and must test whether G is triangle-free. From G , construct a new graph H having as vertices each set of zero, one, or two adjacent vertices of G . Two such sets are adjacent in H when they differ by exactly one vertex. An equivalent description 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*^[22] and is a type of phylogenetic networks. Any 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.

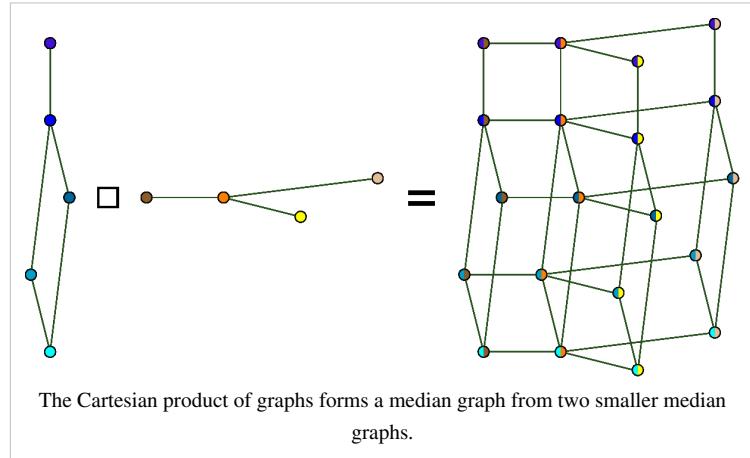
To form the Buneman graph for a set of species and characteristics, first, eliminate redundant species that are indistinguishable from some other species and redundant characteristics that are always the same as some other characteristic. Then, form a latent vertex for every combination of characteristic 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 any 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: any pairwise intersecting family of sets 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.^[23]

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 any 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 $t_i = m(t_{i-1}, s_i, s_{i+1})$.^[1]
- The property of having a unique median is also called the *unique Steiner point property*.^[1] An optimal Steiner tree for any 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, any 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 any 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.^[24]
- The only regular median graphs are the hypercubes.^[25]



Notes

- [1] Chung, Graham & Saks (1987).
- [2] Buneman (1971); Dress et al. (1997); Dress, Huber & Moulton (1997).
- [3] Bandelt & Barthélémy (1984); Day & McMorris (2003).
- [4] Imrich & Klavžar (2000), Proposition 1.26, p. 24.
- [5] This follows immediately from the characterization of median graphs as retracts of hypercubes, described below.
- [6] Soltan, Zambitskii & Pris'acaru (1973); Chepoi, Dragan & Vaxès (2002); Chepoi, Fanciullini & Vaxès (2004).
- [7] Birkhoff & Kiss (1947) credit the definition of this operation to Birkhoff, G. (1940), *Lattice Theory*, American Mathematical Society, p. 74.
- [8] Knuth (2008), p. 65, and exercises 75 and 76 on pp. 89–90. Knuth states that a simple proof that associativity implies distributivity remains unknown.
- [9] 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 Birkhoff & Kiss (1947).
- [10] Birkhoff & Kiss (1947), Theorem 2.
- [11] Birkhoff & Kiss (1947), p. 751.
- [12] Avann (1961).
- [13] Knuth (2008) 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.
- [14] Imrich & Klavžar (2000), Theorem 2.40, p. 77.

- [15] Bandelt & Chepoi (2008), Proposition 2.5, p.8; Chung, Graham & Saks (1989); Feder (1995); Knuth (2008), Theorem S, p. 72.
- [16] Hell (1976).
- [17] Imrich & Klavžar (2000), Proposition 1.33, p. 27.
- [18] Bandelt (1984); Imrich & Klavžar (2000), Theorem 2.39, p.76; Knuth (2008), p. 74.
- [19] The technique, which culminates in Lemma 7.10 on p.218 of Imrich and Klavžar, consists of applying an algorithm of Chiba & Nishizeki (1985) 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 Knuth (2008), Algorithm H, p. 69.
- [20] For previous median graph recognition algorithms, see Jha & Slutzki (1992), Imrich & Klavžar (1998), and Hagauer, Imrich & Klavžar (1999). For triangle detection algorithms, see Itai & Rodeh (1978), Chiba & Nishizeki (1985), and Alon, Yuster & Zwick (1995).
- [21] Alon, Yuster & Zwick (1995), 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 $m = O(n \log n)$.
- [22] Mulder & Schrijver (1979) described a version of this method for systems of characteristics not requiring any latent vertices, and Barthélémy (1989) gives the full construction. The Buneman graph name is given in Dress et al. (1997) and Dress, Huber & Moulton (1997).
- [23] Mulder & Schrijver (1979).
- [24] Škrekovski (2001).
- [25] Mulder (1980).

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, MR14111787.
- 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, JSTOR 2034206, MR0125807.
- Bandelt, Hans-Jürgen (1984), "Retracts of hypercubes", *Journal of Graph Theory* **8** (4): 501–510, doi:10.1002/jgt.3190080407, MR0766499.
- 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, MR0743019.
- 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, PMID 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, PMID 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, PMID 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, MR1002234.
- 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, MR0021540.
- 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.
- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM Journal on Computing* **14**: 210–223, doi:10.1137/0214017, MR0774940.
- 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, MR0910939.
- 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.
- Day, William H. E.; McMorris, F. R. (2003), *Axiomatic Concensus [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, MR1630739.
- 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, MR1630743.
- 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, JSTOR 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, MR1678773.
- 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, MR0543779.
- 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, MR1642702.
- Imrich, Wilfried; Klavžar, Sandi (2000), *Product Graphs: Structure and Recognition*, Wiley, ISBN 0-471-37039-8, MR788124.
- 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, MR1666073.
- Itai, A.; Rodeh, M. (1978), "Finding a minimum circuit in a graph", *SIAM Journal on Computing* **7** (4): 413–423, doi:10.1137/0207033, MR0508603.
- Jha, Pranava K.; Slutzki, Giora (1992), "Convex-expansion algorithms for recognizing and isometric embedding of median graphs", *Ars Combinatoria* **34**: 75–92, MR1206551.
- Klavžar, Sandi; Mulder, Henry Martyn (1999), "Median graphs: characterizations, location theory and related structures", *Journal of Combinatorial Mathematics and Combinatorial Computing* **30**: 103–127, MR1705337.
- 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, MR1630736.
- 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, MR0558458.
- 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, MR0522746.
- Nebesk'y, Ladislav (1971), "Median graphs", *Commentationes Mathematicae Universitatis Carolinae* **12**: 317–325, MR0286705.

- Škrekovski, Riste (2001), "Two relations for median graphs", *Discrete Mathematics* **226** (1): 351–353, doi:10.1016/S0012-365X(00)00120-5, MR1802603.
- Soltan, P.; Zambitskii, D.; Prisăcaru, C. (1973) (in Russian), *Extremal problems on graphs and algorithms of their solution*, 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.

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.

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**.

Example

The two graphs shown below are isomorphic, despite their different looking drawings.

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

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, see the example above. 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

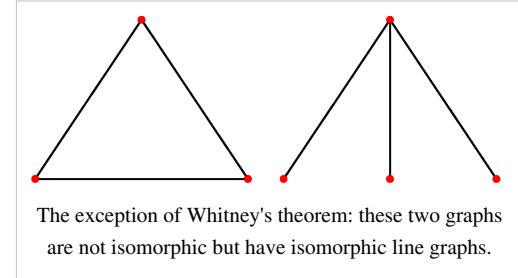
$$\sum_{v \in V(G)} v \cdot \deg v$$

may be different for two isomorphic graphs.

Recognition of graph isomorphism

Whitney theorem

The **Whitney graph isomorphism theorem**,^[1] 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.^[2]



Algorithmic approach

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.^[3]

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.

References

- [1] Whitney, Hassler (January 1932). "Congruent Graphs and the Connectivity of Graphs" (<http://www.jstor.org/stable/2371086>). *American Journal of Mathematics (Am. J. Math.)* (The Johns Hopkins University Press) **54** (1): 150–168. . Retrieved 17 August 2012.
- [2] Dirk L. Vertigan, Geoffrey P. Whittle: A 2-Isomorphism Theorem for Hypergraphs. *J. Comb. Theory, Ser. B* 71(2): 215–230. 1997.
- [3] Schöning, Uwe (1988). "Graph isomorphism is in the low hierarchy". *Journal of Computer and System Sciences* **37**: 312–323.
- 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

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 the best algorithm (Luks, 1983) has run time $2^{O(\sqrt{n} \log n)}$ for graphs with n vertices.^{[2][3]}

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.^[4]

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.^[5]

This problem is a special case of the subgraph isomorphism problem,^[6] 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.^[7]

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.^[8]

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.^[9]

Solved special cases

A number of important special cases of the graph isomorphism problem have efficient, polynomial-time solutions:

- Trees^[10]
- Planar graphs^[11] (In fact, planar graph isomorphism is in log space,^[12] a class contained in P.)
- Interval graphs^[13]
- Permutation graphs^[14]
- Partial k -trees^[15]
- Bounded-parameter graphs
 - Graphs of bounded genus^[16] (Note: planar graphs are graphs of genus 0)
 - Graphs of bounded degree^[17]
 - Graphs with bounded eigenvalue multiplicity^[18]
 - k -Contractible graphs (a generalization of bounded degree and bounded genus)^[19]
 - 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.^[20]

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.^[21] 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^[22]^[23]. 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}.^[24] 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:^[25]

- digraphs^[25]
- labelled graphs, with the proviso that an isomorphism is not required to preserve the labels,^[25] 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)^[25]
- 2-colored graphs^[25]
- explicitly given finite structures^[25]
- multigraphs^[25]

- hypergraphs^[25]
- finite automata^[25]
- Markov Decision Processes^[26]
- commutative class 3 nilpotent (i.e., $xyz = 0$ for every elements x, y, z) semigroups^[25]
- finite rank associative algebras over a fixed algebraically closed field with zero squared radical and commutative factor over the radical^{[25][27]}
- context-free grammars^[25]
- balanced incomplete block designs^[25]
- Recognizing combinatorial isomorphism of convex polytopes represented by vertex-facet incidences.^{[2][28]}

This list is incomplete.

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:^[25]

- connected graphs^[25]
- graphs of diameter 2 and radius 1^[25]
- directed acyclic graphs^[25]
- regular graphs.^[25]
- bipartite graphs without non-trivial strongly regular subgraphs^[25]
- bipartite Eulerian graphs^[25]
- bipartite regular graphs^[25]
- line graphs^[25]
- chordal graphs^[25]
- regular self-complementary graphs^[25]
- polytopal graphs of general, simple, and simplicial convex polytopes in arbitrary dimensions^[28]

This list is incomplete.

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.^[29]
- 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 $(n - \epsilon)$ -clique in a M -graph of size n^2 is NP-complete for arbitrarily small positive ϵ .^[30]
- The problem of homeomorphism of 2-complexes.^[31]

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.^[32]
- 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.^[28]

Program checking

Blum and Kannan^[33] 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.^[34] 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.^[35] 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.^[36]

Notes

- [1] The latest one resolved was minimum-weight triangulation, proved to be NP-complete in 2006. Mulzer, Wolfgang; Rote, Günter (2008), "Minimum-weight triangulation is NP-hard", *Journal of the ACM* **55** (2): 1, arXiv:cs.CG/0601002, doi:10.1145/1346330.1346336.
- [2] Johnson 2005
- [3] Babai & Codenotti (2008).
- [4] 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
- [5] McKay 1981
- [6] Ullman (1976).
- [7] Cristopher Moore; Alexander Russell; Schulman, Leonard J. (2005). "The Symmetric Group Defies Strong Fourier Sampling: Part I". arXiv:quant-ph/0501056v3 [quant-ph].
- [8] 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.

- [9] 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.
- [10] P.J. Kelly, "A congruence theorem for trees" Pacific J. Math., 7 (1957) pp. 961–968; Aho, Hopcroft & Ullman 1974.
- [11] Hopcroft & Wong 1974
- [12] Datta, S.; Limaye, N.; Nimbhorkar, P.; Thierauf, T.; Wagner, F. (2009). "Planar Graph Isomorphism is in Log-Space". *2009 24th Annual IEEE Conference on Computational Complexity*. pp. 203. doi:10.1109/CCC.2009.16. ISBN 978-0-7695-3717-7.
- [13] Booth & Lueker 1979
- [14] Colbourn 1981
- [15] Bodlaender 1990
- [16] Miller 1980; Filotti & Mayer 1980.
- [17] Luks 1982
- [18] Babai, Grigoryev & Mount 1982
- [19] Gary L. Miller: Isomorphism Testing and Canonical Forms for k -Contractable 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.)
- [20] Eugene Luks, "Parallel algorithms for permutation groups and graph isomorphism", Proc. IEEE Symp. Foundations of Computer Science, 1986, 292-302
- [21] Booth & Colbourn 1977; Köbler, Schöning & Torán 1993
- [22] http://qwiki.stanford.edu/wiki/Complexity_Zoo:S#spp
- [23] Köbler, Schöning & Torán 1992; Arvind & Kurur 2006
- [24] Arvind & Köbler 2000
- [25] Zemlyachenko, Korneenko & Tyshkevich 1985
- [26] "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.
- [27] 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, doi:10.1007/BF01084390 (translation of a 1981 Russian language article)
- [28] 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.
- [29] Colbourn M.J., Colbourn Ch.J. "Graph isomorphism and self-complementary graphs", *SIGACT News*, 1978, vol. 10, no. 1, 25–29.
- [30] Kozen 1978.
- [31] J. Shawe-Taylor, T.Pisanski, "Homeomorphism of 2-Complexes is Graph Isomorphism Complete", *SIAM Journal on Computing*, 23 (1994) 120 – 132 .
- [32] R. Mathon, "A note on the graph isomorphism counting problem", *Information Processing Letters*, 8 (1979) pp. 131–132; Johnson 2005.
- [33] Designing Programs that Check their Work (<ftp://ftp.cis.upenn.edu/pub/kannan/jacm.ps.gz>)
- [34] Christophe-André Mario Irniger (2005) "Graph Matching: Filtering Databases of Graphs Using Machine Learning", ISBN 1-58603-557-6
- [35] "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)
- [36] Baird, HS and Cho, YE (1975). "An artwork design verification system" (<http://dl.acm.org/citation.cfm?id=809095>). Proceedings of the 12th Design Automation Conference. IEEE Press. pp. 414–420. .

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.
- 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.
- 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, 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.
 - 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.
 - 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.
 - 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, 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.
 - 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.
 - Köbler, Johannes; Schöning, Uwe; Torán, Jacobo (1992), "Graph isomorphism is low for PP", *Computational Complexity* **2** (4): 301–330, doi: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.
 - Kozen, Dexter (1978), "A clique problem equivalent to graph isomorphism", *ACM SIGACT News* **10** (2): 50–52, doi: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.
 - 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, 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, 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.

Surveys and monographs

- Read, Ronald C.; Corneil, Derek G. (1977), "The graph isomorphism disease", *Journal of Graph Theory* **1** (4): 339–363, MR0485586..
- 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. (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. (From the book cover: The book 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. (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 finding a **canonical form of a graph** G , which is a graph $\text{Canon}(G)$ isomorphic to G such that $\text{Canon}(H) = \text{Canon}(G)$ if and only if H is isomorphic to G . The canonical form of a graph is an example of a complete graph invariant.^{[1][2]} Since the vertex sets of (finite) graphs are commonly identified with the intervals of integers $1, \dots, n$, where n is the number of the vertices of a graph, a canonical form of a graph is commonly called **canonical labeling** of a graph.^[3] Graph canonization is also sometimes known as **graph canonicalization**.

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.

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.^[2]

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 a simple vertex classification algorithm after only two refinement steps produces a canonical labeling of an n -vertex random graph with probability $1 - \exp(-O(n))$. Small modifications and added depth-first search step produce canonical labeling of all graphs in linear average time. This result sheds some light on the fact why many reported graph isomorphism algorithms behave well in practice.^[4] 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.

The computation of the lexicographically smallest graph is NP-Hard.^[1]

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.^[5]

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

- [1] "A Logspace Algorithm for Partial 2-Tree Canonization" (<http://books.google.com/books?id=tU8KG7D03DAC&pg=PA40&dq=%22graph+canonization%22#PPA40,M1>)
- [2] "the Space Complexity of k -Tree Isomorphism" (<http://books.google.com/books?id=vCoVUKRhfl4C&pg=PA823&dq=%22graph+canonization%22#PPA823,M1>)
- [3] Laszlo Babai, Eugene Luks, "Canonical labeling of graphs", Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 171–183.
- [4] L. Babai, "On the Isomorphism Problem", unpublished manuscript, 1977
 - L. Babai and L. Kucera. Canonical labeling of graphs in average linear time. Proc. 20th Annual IEEE Symposium on Foundations of Computer Science (1979), 39–46.
- [5] "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)
- Yuri Gurevich, *From Invariants to Canonization*, The Bull. of Euro. Assoc. for Theor. Computer Sci., no. 63, 1997. (<http://research.microsoft.com/en-us/um/people/gurevich/opera/131.pdf>)

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.^[2]

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 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.

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 number of vertices 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.^[3]

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.^[2]

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.^[4] 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,^[5] the bioinformatics of protein-protein interaction networks,^[6] and in exponential random graph methods for mathematically modeling social networks.^[7]

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.

Notes

- [1] The original Cook (1971) paper that proves the Cook–Levin theorem already showed subgraph isomorphism to be NP-complete, using a reduction from 3-SAT involving cliques.
- [2] Eppstein (1999)
- [3] Here Ω invokes Big Omega notation.
- [4] Ullmann (1976)
- [5] Kuramochi & Karypis (2001).
- [6] Pržulj, Corneil & Jurisica (2006).
- [7] Snijders et al. (2006).

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.
- 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.
- 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, 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, 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, PMID 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.
- Ullmann, Julian R. (1976), "An algorithm for subgraph isomorphism", *Journal of the ACM* **23** (1): 31–42, doi: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

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.^[2] 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^[6] 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 $\{1, 2, \dots, |V|\}$ to $\{1, 2, \dots, k\}$ can be obtained.

In the case of derandomizing well-coloring, where each vertex on the subgraph is colored consecutively, 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 it is done by using $nk \log k$ random bits that are almost $k \log k$ independent, and the size of the resulting k -perfect family will be $n^{O(k)}$. 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.

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.

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.^{[1][2]}

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.^[3] 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.

References

- [1] Syslo, Maciej M. (1982), "The subgraph isomorphism problem for outerplanar graphs", *Theoretical Computer Science* **17** (1): 91–97, doi:10.1016/0304-3975(82)90133-5, MR644795.
- [2] Johnson, David S. (1985), "The NP-completeness column: an ongoing guide", *Journal of Algorithms* **6** (3): 434–451, doi:10.1016/0196-6774(85)90012-4, MR800733.
- [3] Ramanujacharyulu, C.; Menon, V. V. (1964), "A note on the snake-in-the-box problem", *Publ. Inst. Statist. Univ. Paris* **13**: 131–135, MR0172736.

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 induced subgraph of G_1 isomorphic to an induced subgraph of G_2 ?

The associated decision problem, i.e., given G_1 , G_2 and an integer k , deciding whether G_1 contains an induced subgraph of at least k edges isomorphic to an induced 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.^[1] Recently, the uniform graph partition problem has gained importance due to its application for clustering and detection of cliques in social, pathological and biological networks.^[2]

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^{[3][2]}. However, uniform graph partitioning or a balanced graph partition problem can be shown to be NP-complete to approximate within any finite factor^[1]. Even for special graph classes such as trees and grids, no reasonable approximation algorithms exist^[4], 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^[4].

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.^[1] 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,

$$\max_i |V_i| \leq (1 + \varepsilon) \frac{|V|}{k}.$$

We already know that $(2, 1)$ cut is the minimum bisection problem and it is NP complete.^[5] Next we assess a 3-partition problem wherein $n = 3k$, which is also bounded in polynomial time.^[1] 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 $P = NP$.^[1]

Graph partition methods

Since graph partitioning is a hard problem, the solutions are based on heuristics. There are two broad categories of these partitioning approaches. While the first one works locally, the second one considers global connectivity. Well known classical methods for partitioning are the Kernighan–Lin algorithm, and Fiduccia–Mattheyses algorithm which were the first effective 2-way cuts by local search strategies. The major drawback was the arbitrary initial partitioning of the vertex set. 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. Global approaches such as the spectral partitioning, is based on the spectra of the adjacency matrices.

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.^[6] 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,^[7] a graph partitioner, and hMETIS, the corresponding partitioner for hypergraphs.^[8]

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) [9] 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 has recently demonstrated that Q suffers a resolution limit, what produces unreliable results when dealing with small communities. In this context, Surprise^[10] has been proposed as a novel promising approach for evaluating the quality of a partition.

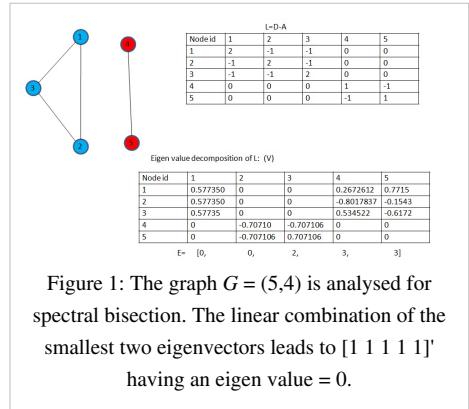


Figure 1: The graph $G = (5,4)$ is analysed for spectral bisection. The linear combination of the smallest two eigenvectors leads to $[1 1 1 1 1]'$ having an eigen value = 0.

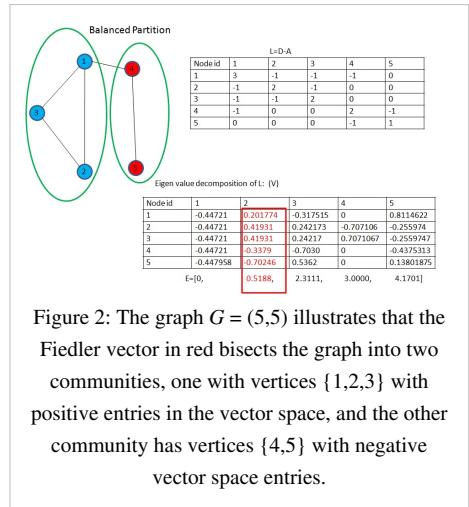


Figure 2: The graph $G = (5,5)$ illustrates that the Fiedler vector in red bisects the graph into two communities, one with vertices $\{1,2,3\}$ with positive entries in the vector space, and the other community has vertices $\{4,5\}$ with negative vector space entries.

Other graph partition methods

Spin models have used for clustering of multivariate data wherein similarities are translated into coupling strengths.^[11] 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^[12]

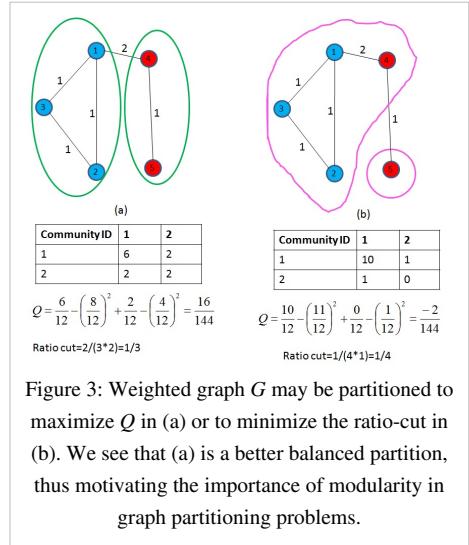


Figure 3: Weighted graph G may be partitioned to maximize Q in (a) or to minimize the ratio-cut in (b). We see that (a) is a better balanced partition, thus motivating the importance of modularity in graph partitioning problems.

References

- [1] Andreev, Konstantin and Räcke, Harald, (2004). "Balanced Graph Partitioning". *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures* (Barcelona, Spain): 120–124. doi:10.1145/1007912.1007931. ISBN 1-58113-840-7.
- [2] Sachin B. Patkar and H. Narayanan (2003). "An Efficient Practical Heuristic For Good Ratio-Cut Partitioning". *VLSI Design, International Conference on*: 64. doi:10.1109/ICVD.2003.1183116.
- [3] Feldmann, Andreas Emil and Foschini, Luca (2012). "Balanced Partitions of Trees and Applications". *Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science* (Paris, France): 100–111.
- [4] Feldmann, Andreas Emil (2012). "Fast Balanced Partitioning is Hard, Even on Grids and Trees". *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science* (Bratislava, Slovakia).
- [5] Garey, Michael R. and Johnson, David S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman & Co.. ISBN 0-7167-1044-7.
- [6] Hendrickson, B. and Leland, R. (1995). "A multilevel algorithm for partitioning graphs". Proceedings of the 1995 ACM/IEEE conference on Supercomputing. ACM. pp. 28.
- [7] Karypis, G. and Kumar, V. (1999). "A fast and high quality multilevel scheme for partitioning irregular graphs". *SIAM Journal on Scientific Computing* **20** (1): 359. doi:10.1137/S1064827595287997.
- [8] Karypis, G. and Aggarwal, R. and Kumar, V. and Shekhar, S. (1997). "Multilevel hypergraph partitioning: application in VLSI domain". *Proceedings of the 34th annual Design Automation Conference*. pp. 526–529.
- [9] Newman, M. E. J. (2006). "Modularity and community structure in networks". *PNAS* **103** (23): 8577–8696. doi:10.1073/pnas.0601602103. PMC 1482622. PMID 16723398.
- [10] Rodrigo Aldecoa and Ignacio Marín (2011). "Deciphering network community structure by Surprise" (<http://www.plosone.org/article/info:doi/10.1371/journal.pone.0024195>). *PLoS ONE* **6** (9): e24195. doi:10.1371/journal.pone.0024195. PMID 21909420. .
- [11] Reichardt, Jörg and Bornholdt, Stefan (Jul 2006). "Statistical mechanics of community detection". *Phys. Rev. E* **74** (1): 016110. doi:10.1103/PhysRevE.74.016110.
- [12] Carlos Alzate and Johan A.K. Suykens (2010). "Multiway Spectral Clustering with Out-of-Sample Extensions through Weighted Kernel PCA". *IEEE Transactions on Pattern Analysis and Machine Intelligence* (IEEE Computer Society) **32** (2): 335–347. doi:10.1109/TPAMI.2008.292. ISSN 0162-8828. PMID 20075462.

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

- 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. pp. 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. 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. PMID 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.. 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.^{[1][2]}

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 .^[1]

Pseudocode

See ^[2]

```

1  function Kernighan-Lin( $G(V, E)$ ):
2      determine a balanced initial partition of the nodes into sets  $A$  and  $B$ 
3      do
4           $A_1 := A$ ;  $B_1 := B$ 
5          compute  $D$  values for all  $a$  in  $A_1$  and  $b$  in  $B_1$ 
6          for ( $i := 1$  to  $/2$ )
7              find  $a[i]$  from  $A_1$  and  $b[i]$  from  $B_1$ , such that  $g[i] = D[a[i + D[b[i]] - 2*c[a[i]][b[i]]]$  is maximal
8              move  $a[i]$  to  $B_1$  and  $b[i]$  to  $A_1$ 
9              remove  $a[i]$  and  $b[i]$  from further consideration in this pass
10             update  $D$  values for the elements of  $A_1 = A_1 / a[i]$  and  $B_1 = B_1 / b[i]$ 
11         end for
12         find  $k$  which maximizes  $g_{max}$ , the sum of  $g[1], \dots, g[k]$ 

```

```

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

- [1] Kernighan, B. W.; Lin, Shen (1970). "An efficient heuristic procedure for partitioning graphs". *Bell Systems Technical Journal* **49**: 291–307.
 [2] Ravikumār, Si. Pi; Ravikumar, C.P (1995). *Parallel methods for VLSI layout design* (<http://books.google.com/?id=VPXAxkTKxXIC>). Greenwood Publishing Group. pp. 73. ISBN 978-0-89391-828-6. OCLC 2009-06-12. .

Tree decomposition

In graph theory, a **tree decomposition** is a mapping of a graph into a tree that can be used to speed up solving certain problems on the original graph. The **treewidth** measures the number of graph vertices mapped onto any tree node in an optimal tree decomposition. While it is NP-hard to determine the treewidth of a graph, many NP-hard combinatorial problems on graphs are solvable in polynomial time when restricted to graphs of bounded treewidth.

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.

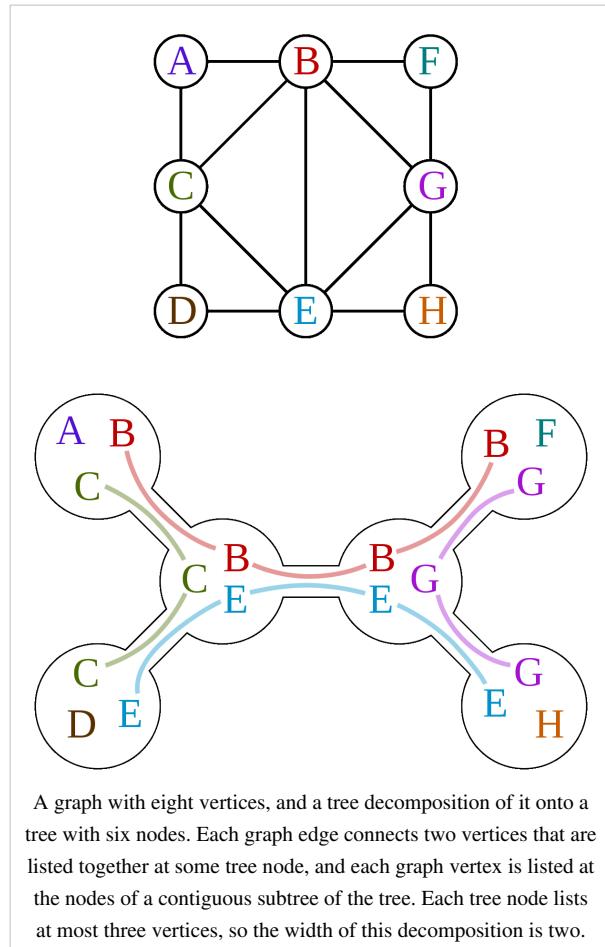
The concept of tree decompositions and treewidth was originally introduced by Halin (1976). Later it was rediscovered by Robertson & Seymour (1984) and has since been studied by many other authors^[1].

Definition

Intuitively, a tree decomposition represents the vertices of the given graph as subtrees of a tree, in such a way that vertices are adjacent only when the corresponding subtrees intersect. Thus, the graph 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.



A graph with eight vertices, and a tree decomposition of it onto a tree with six nodes. Each graph edge connects two vertices that are listed together at some tree node, and each graph vertex is listed at the nodes of a contiguous subtree of the tree. Each tree node lists at most three vertices, so the width of this decomposition is two.

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 $X_i \cap X_j \subseteq X_k$.

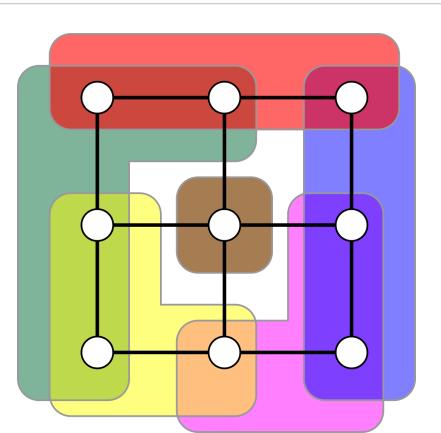
The tree decomposition of a graph is far from unique; for example, a trivial tree decomposition contains all vertices of the graph in its single root node.

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. Equivalently, the treewidth of G is one less than the size of the largest clique in the chordal graph containing G with the smallest clique number. The graphs with treewidth at most k are also called partial k -trees.

It is NP-complete to determine whether a given graph G has treewidth at most a given variable k .^[3] 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.^[4] The time dependence of this algorithm on k is exponential.

Treewidth may also be characterized in terms of havens, functions describing an evasion strategy for a certain pursuit-evasion game defined on a graph. A graph G has treewidth k if and only if it has a haven of order $k + 1$ but of no higher order, where a haven of order $k + 1$ is a function β that maps each set X of at most k vertices in G into one of the connected components of $G \setminus X$ and that obeys the monotonicity property that $\beta(Y) \subseteq \beta(X)$ whenever $X \subseteq Y$. A similar characterization can also be made using brambles, sets of connected subgraphs that all touch each other.^[5]

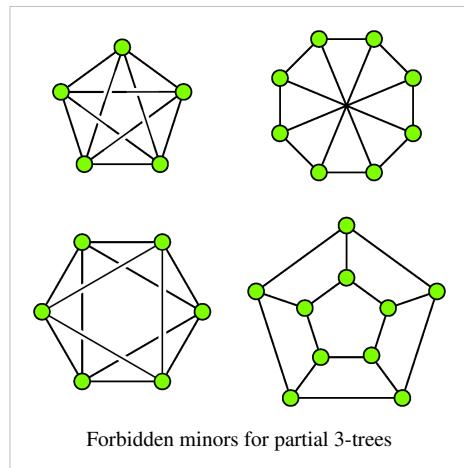


A bramble of order four in a 3×3 grid graph, the existence of which shows that the graph has treewidth at least 3

Graph minors

For any fixed constant k , the partial k -trees are closed under the operation of graph minors, and therefore, by the Robertson–Seymour theorem, this family can be characterized in terms of a finite set of forbidden minors. For partial 1-trees (that is, forests), the single forbidden minor is a triangle, and for the partial 2-trees the single forbidden minor is the complete graph on four vertices. However, the number of forbidden minors increases for larger values of k : for partial 3-trees there are four forbidden minors, the complete graph on five vertices, the octahedral graph with six vertices, the eight-vertex Wagner graph, and the pentagonal prism with ten vertices.^[6]

The planar graphs do not have bounded treewidth, because the $n \times n$ grid graph is a planar graph with treewidth n . Therefore, if F is a minor-closed graph family with bounded treewidth, it cannot include all planar graphs. Conversely, if some planar graph cannot occur as a minor for graphs in family F , then there is a constant k such that all graphs in F have treewidth at most k . That is, the following three conditions are equivalent to each other:^[7]



Forbidden minors for partial 3-trees

1. F is a minor-closed family of bounded-treewidth graphs;
2. One of the finitely many forbidden minors characterizing F is planar;
3. F is a minor-closed graph family that does not include all planar graphs.

Families of graphs with bounded treewidth include the cactus graphs, pseudoforests, series-parallel graphs, outerplanar graphs, Halin graphs, and Apollonian networks.^[6] The control flow graphs arising in the compilation of structured programs also have bounded treewidth, which allows certain tasks such as register allocation to be performed efficiently on them.^[8]

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*,^[9] a parameter related to treewidth. Later, several authors independently observed at the end of the 1980s^[10] 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.^[4]

Treewidth of cliques

In any tree decomposition (T, X) of a graph containing a clique there is a node i in T such that X_i contains all the nodes of the clique. This is shown by induction on the size of the clique. The base cases are cliques of size 1 and 2, which follow from the conditions 1 and 2 on a tree decomposition. The inductive case is a graph containing a clique of size $k+1$, where k is 2 or greater. Let C be the set of nodes in the clique. Since $k+1 \geq 3$, there are at least three nodes in the clique, call them x, y and z . We know from the induction hypothesis that there are nodes a, b and c in the tree decomposition with

$$X_a \supseteq C - \{x\}, X_b \supseteq C - \{y\}, X_c \supseteq C - \{z\}.$$

In a tree there is exactly one path between any two nodes. A second property of trees is that the three paths between a, b and c have a non-empty intersection. Let v be a node in this intersection. From condition 3 on a tree decomposition we get that

$$X_v \supseteq X_a \cap X_b \supseteq C - \{x, y\}$$

$$X_v \supseteq X_b \cap X_c \supseteq C - \{y, z\}$$

$$X_v \supseteq X_a \cap X_c \supseteq C - \{x, z\}$$

This implies that $X_v \supseteq C$.

It follows from this that the treewidth of a k -clique is $k-1$.

Treewidth of trees

A connected graph with at least two vertices has treewidth 1 if and only if it is a tree. This can be shown in two steps, first that a tree has treewidth 1, second, that a connected graph with treewidth 1 is a tree.

To show the former, use induction on the number of vertices in the tree to show that it has a tree decomposition with no bag with size larger than two. The base case is a tree with two vertices, in which case the tree decomposition with one single node is sufficient. The inductive case is a tree G with $k+1$ vertices, where k is any integer greater than 1. If we remove a leaf v from the graph, we get a tree of size k . From the induction hypothesis we can create a tree decomposition (T, X) of width 1 of this graph. Let u be the unique neighbour of v in G and i some node in T such that u is in X_i . Let T_1 be T added a node j with i as its only neighbour and let X' be X with the addition that $X'_j = \{i, j\}$. Now (T_1, X') is a tree decomposition of G with width 1.

Now it remains to show that a connected graph with treewidth 1 is a tree. The contrapositive statement is that a graph with a cycle does not have treewidth 1. A graph with a cycle has the 3-clique as a minor, which from the statement in the previous section has treewidth 2. Since the partial 2-trees are closed under minors, the graph therefore has treewidth 2 or greater.

Notes

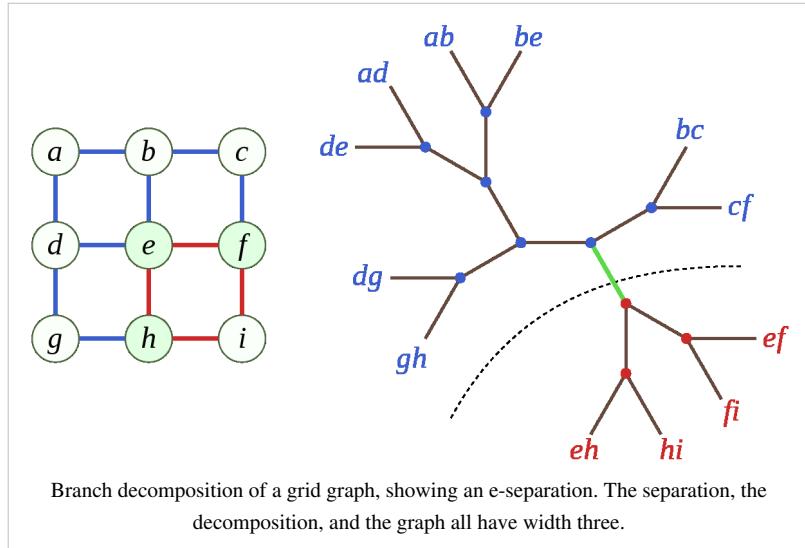
- [1] Diestel (2005) pp.354–355
- [2] Diestel (2005) section 12.3
- [3] Arnborg, Corneil & Proskurowski (1987).
- [4] Bodlaender (1996).
- [5] Seymour & Thomas (1993).
- [6] Bodlaender (1998).
- [7] Robertson & Seymour (1986).
- [8] Thorup (1998).
- [9] Bertelé (1972)
- [10] Arnborg & Proskurowski (1989); Bern, Lawler & Wong (1987); Bodlaender (1988).

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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Seymour, Paul D.; Thomas, Robin (1993), "Graph Searching and a Min-Max Theorem for Tree-Width.", *Journal of Combinatorial Theory, Series B* **58** (1): 22–33, doi:10.1006/jctb.1993.1027.
- Thorup, Mikkel (1998), "All structured programs have small tree width and good register allocation", *Information and Computation* **142** (2): 159–181, doi:10.1006/inco.1997.2697.

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.^[2]

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.^[2] However, the graphs with branchwidth at most k form a minor-closed family of graphs,^[3] 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.^[4]

For planar graphs, the branchwidth can be computed exactly in polynomial time.^[2], this in contrast to treewidth for which the complexity on planer graphs is a well known open problem.

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

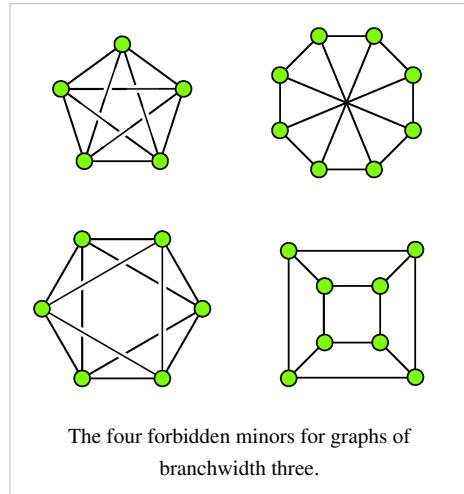
It is also possible to define a notion of branch-decomposition for matroids that generalizes branch-decompositions of graphs.^[6] A branch-decomposition of a matroid is a hierarchical clustering of the matroid elements, represented as an unrooted binary tree with the elements of the matroid at its leaves. 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.^[7] However, for graphs that are not trees, the branchwidth of the graph is equal to the branchwidth of its associated graphic matroid.^[8] 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.^[7]

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,^[9] and plays a bigger role than branchwidth in the theory of graph minors, branchwidth has more convenient properties in the matroid setting.^[10] 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.^[11] 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.^[12]

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).^[13] 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.^[13] 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.^[13] 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.^[14]



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.^[15]

Notes

- [1] Robertson & Seymour 1991, Theorem 5.1, p. 168.
- [2] Seymour & Thomas (1994).
- [3] Robertson & Seymour (1991), Theorem 4.1, p. 164.
- [4] Bodlaender & Thilikos (1997). Fomin, Mazoit & Todinca (2009) 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.
- [5] Hicks (2000); Hliněný (2003).
- [6] Robertson & Seymour 1991. Section 12, "Tangles and Matroids", pp. 188–190.
- [7] Mazoit & Thomassé (2007).
- [8] Mazoit & Thomassé (2007); Hicks & McMurray (2007).
- [9] Hliněný & Whittle (2006).
- [10] Geelen, Gerards & Whittle (2006).
- [11] Geelen, Gerards & Whittle (2002); Geelen, Gerards & Whittle (2006).
- [12] Geelen, Gerards & Whittle (2006); Geelen, Gerards & Whittle (2007).
- [13] Robertson & Seymour (1991), Theorem 4.2, p. 165.
- [14] Bodlaender & Thilikos (1999). 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.
- [15] Hall et al. (2002); Geelen et al. (2003).

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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Hall, Rhianon; 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.
- 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.
- 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
- 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.
 - 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.
- 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.
- 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.
- Seymour, Paul D.; Thomas, Robin (1994), "Call routing and the ratcatcher", *Combinatorica* **14** (2): 217–241, doi: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,^[1] 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,^[2] 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**.^[3]

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,^[2] and the "vortices" appearing in the general structure theory for minor-closed graph families have bounded pathwidth.^[4] 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.^{[5][6]} 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 .^[7] Additionally, for several special classes of graphs, such as trees, the pathwidth may be computed in polynomial time without dependence on k .^{[8][9]} 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.^[10] Path decomposition may also be used to measure the space complexity of dynamic programming algorithms on graphs of bounded treewidth.^[11]

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 .^[2]

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.^[12] 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.

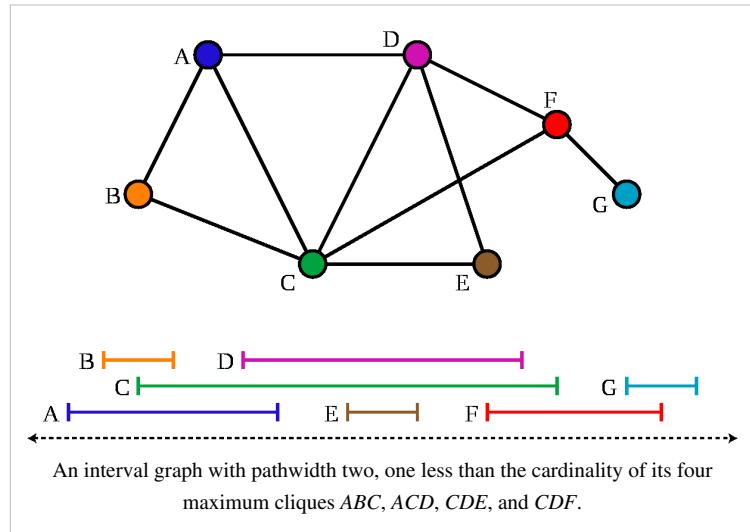
In one direction, suppose a path decomposition of G is given. Then one may represent the nodes of the decomposition as points on a line (in path order) and represent each vertex v as a closed interval having these points as endpoints. In this way, the path decomposition nodes containing v correspond 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 .^[13] 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.

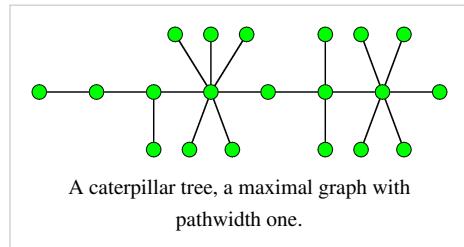


Node searching number

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 in which the non-leaf vertices induce a k -path. In particular the maximal graphs of pathwidth one are exactly the caterpillar trees.^[14]



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.^[15] For similar reasons, the cutwidth is at most the pathwidth times the maximum degree of the vertices in a given graph.^[16]

Any n -vertex forest has pathwidth $O(\log n)$.^[17] 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 $O(t \log n)$.^[18] 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.^[19] 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.^[20]

In any planar graph, the pathwidth is at most proportional to the square root of the number of vertices.^[21] 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.^[22] Since, like planar graphs, the graphs in any fixed minor-closed graph family have separators of size $O(\sqrt{n})$,^[23] 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^[24] and for polyhedral graphs.^[25] For 2-connected planar graphs, the pathwidth of the dual graph less than the pathwidth of the line graph.^[26] 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,^[27] permutation graphs,^[28] the complements of comparability graphs,^[29] and the comparability graphs of interval orders.^[30]

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.^[31]

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.^[5] 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 .^[32] 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.^[7] 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 .^[33] 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,^[34] planar graphs,^[34] planar graphs of bounded degree,^[34] chordal graphs,^[35] chordal dominoes,^[36] the complements of comparability graphs,^[29] and bipartite distance-hereditary graphs.^[37] 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.^[37]

However, the pathwidth may be computed in linear time for trees and forests,^[9] and for graphs of bounded treewidth including series-parallel graphs, outerplanar graphs, and Halin graphs.^[7] It may also be computed in polynomial time for split graphs,^[38] for the complements of chordal graphs,^[39] for permutation graphs,^[28] for cographs,^[27] for circular-arc graphs,^[40] for the comparability graphs of interval orders,^[30] 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.^[6] The best known approximation ratio of a polynomial time approximation algorithm for pathwidth is $O((\log n)^{3/2})$.^[41] 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.

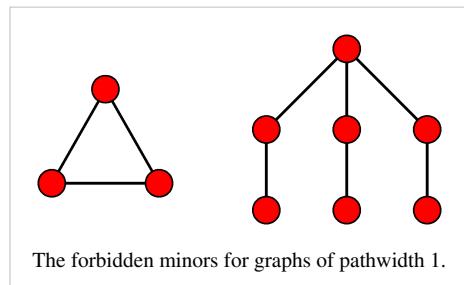
Excluding a forest

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.^[42] 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)^[2], 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 k levels has pathwidth $k - 1$, 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 $n - 2$.^[43]

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.^{[42][44]}



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 $(p!)^2$.^[44] The complete set X_2 of forbidden minors for pathwidth-2 graphs has been computed; it contains 110 different graphs.^[45]

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.^[4] This theory, in which pathwidth is intimately connected to arbitrary minor-closed graph families, has important algorithmic applications.^[46]

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.^[34]

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^[47] 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^[48] 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.^[49] The same algorithmic approach can also be used to model folding problems in programmable logic arrays.^[50]

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.^[51]
- 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.^[52]
- 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.^[53]

- 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.^[54]

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.^[55]

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.^[55]

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,^[56] 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.^[10] 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 $O(2^w n)$.^[31] On graphs of bounded pathwidth, this approach leads to fixed-parameter tractable algorithms, parametrized by the pathwidth.^[49] 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.^[11]

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.^[31] A similar approach leads to improved exponential-time algorithms for the maximum cut and minimum dominating set problems in cubic graphs,^[31] and for several other NP-hard optimization problems.^[57]

Notes

- [1] Diestel & Kühn (2005).
- [2] Robertson & Seymour (1983).
- [3] Bodlaender (1998).
- [4] Robertson & Seymour (2003).
- [5] Kashiwabara & Fujisawa (1979); Ohtsuki et al. (1979); Lengauer (1981); Arnborg, Corneil & Proskurowski (1987).
- [6] Bodlaender et al. (1992).
- [7] Bodlaender (1996); Bodlaender & Kloks (1996).
- [8] Bodlaender (1994).
- [9] Möhring (1990); Scheffler (1990); Ellis, Sudborough & Turner (1994); Coudert, Huc & Mazauric (1998); Peng et al. (1998); Skodinis (2000); Skodinis (2003).
- [10] Arnborg (1985).
- [11] Aspvall, Proskurowski & Telle (2000).
- [12] Bodlaender (1998), Theorem 29, p. 13.
- [13] Kinnersley (1992); Bodlaender (1998), Theorem 51.
- [14] Proskurowski & Telle (1999).
- [15] Korach & Solel (1993), Lemma 3 p.99; Bodlaender (1998), Theorem 47, p. 24.
- [16] Korach & Solel (1993), Lemma 1, p. 99; Bodlaender (1998), Theorem 49, p. 24.
- [17] Korach & Solel (1993), Theorem 5, p. 99; Bodlaender (1998), Theorem 66, p. 30. Scheffler (1992) gives a tighter upper bound of $\log_{3(2n+1)}$ on the pathwidth of an n -vertex forest.
- [18] Korach & Solel (1993), Theorem 6, p. 100; Bodlaender (1998), Corollary 24, p.10.
- [19] Gurski & Wanke (2007).
- [20] Golovach (1993).
- [21] Bodlaender (1998), Corollary 23, p. 10.
- [22] Bodlaender (1998), Theorem 20, p. 9.
- [23] Alon, Seymour & Thomas (1990).
- [24] Bodlaender & Fomin (2002); Coudert, Huc & Sereni (2007).
- [25] Fomin & Thilikos (2007); Amini, Huc & Pérennes (2009).
- [26] Fomin (2003).
- [27] Bodlaender & Möhring (1990).
- [28] Bodlaender, Kloks & Kratsch (1993).
- [29] Habib & Möhring (1994).
- [30] Garbe (1995).
- [31] Fomin & Høie (2006).
- [32] Fomin et al. (2008).
- [33] Downey & Fellows (1999), p.12.
- [34] Monien & Sudborough (1988).
- [35] Gusted (1993).
- [36] Kloks, Kratsch & Müller (1995). A chordal domino is a chordal graph in which every vertex belongs to at most two maximal cliques.
- [37] Kloks et al. (1993).
- [38] Kloks & Bodlaender (1992); Gusted (1993).
- [39] Garbe (1995) 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.
- [40] Suchan & Todinca (2007).
- [41] Feige, Hajiaghayi & Lee (2005).
- [42] Robertson & Seymour (2004).
- [43] Bienstock et al. (1991); Diestel (1995); Cattell, Dinneen & Fellows (1996).
- [44] Kinnersley (1992); Takahashi, Ueno & Kajitani (1994); Bodlaender (1998), p. 8.
- [45] Kinnersley & Langston (1994).
- [46] Demaine, Hajiaghayi & Kawarabayashi (2005).
- [47] Berge (1967).
- [48] Lopez & Law (1980).
- [49] Fellows & Langston (1989).
- [50] Möhring (1990); Ferreira & Song (1992).
- [51] Hliněny (2003).
- [52] Suderman (2004).
- [53] Dujmović et al. (2008).
- [54] Dujmović, Morin & Wood (2003).

- [55] Bodlaender, Gustedt & Telle (1998).
- [56] Miller (1956).
- [57] Kneis et al. (2005); Björklund & Husfeldt (2008).

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.
- 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.
- Arnborg, Stefan (1985), "Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey", *BIT* **25** (1): 2–23, doi: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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Bodlaender, Hans L.; Gustedt, 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.
- 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.
- 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.
- 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.
- 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, doi:10.1007/978-3-540-87779-0_36.
 - 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.
 - Diestel, Reinhard (1995), "Graph Minors I: A short proof of the path-width theorem", *Combinatorics, Probability and Computing* **4** (01): 27–30, doi: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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.

- 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.
- 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.
- 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.
- Gusted, Jens (1993), "On the pathwidth of chordal graphs", *Discrete Applied Mathematics* **45** (3): 233–248, doi: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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Lengauer, Thomas (1981), "Black-white pebbles and graph separation", *Acta Informatica* **16** (4): 465–475, doi: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, Also in the joint issue, *IEEE Journal of Solid-State Circuits* **15** (4): 736–740, 1980, doi: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, PMID 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.
 - 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.

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

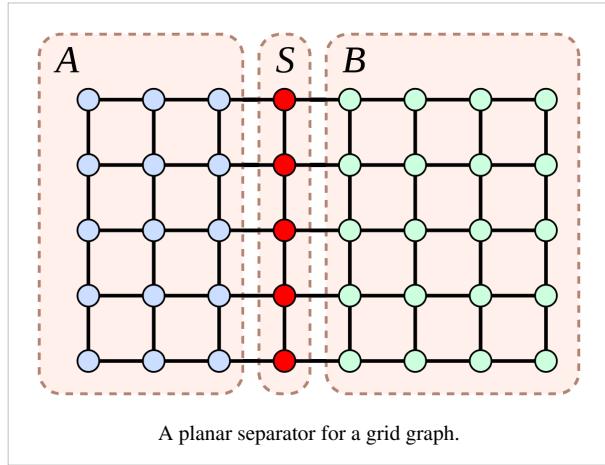
As it is usually stated, the separator theorem states that, in any n -vertex planar graph $G = (V, E)$, there exists a partition of the vertices of G into three sets A , S , and B , such that each of A and B has at most $2n/3$ vertices, S has $O(\sqrt{n})$ vertices, and there are no edges with one endpoint 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**.^[1] 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.^[2]

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 $n/2$.^[3]



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 .^[4] 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.^[5]

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.^{[6][7]}

This method leads to a randomized algorithm that finds such a separator in linear time,^[6] and a less-practical deterministic algorithm with the same linear time bound.^[8] 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

$$\sqrt{\frac{2\pi}{\sqrt{3}}} \left(\frac{1 + \sqrt{3}}{2\sqrt{2}} + o(1) \right) \sqrt{n} \approx 1.84\sqrt{n}.^{[9]}$$

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.^[10]

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.^[11]

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.^[12] 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 $O(\sqrt{(\Delta n)})$.^[13]

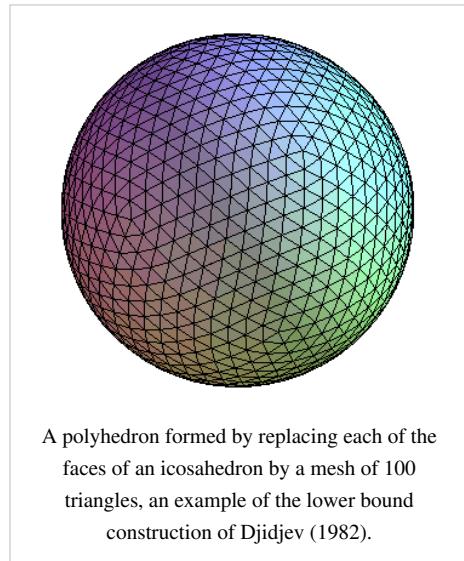
A simple cycle separator in the dual graph of a planar graph forms an edge separator in the original graph.^[14] Applying the simple cycle separator theorem of Gazit & Miller (1990) to the dual graph of a given planar graph strengthens the $O(\sqrt{(\Delta n)})$ bound for the size of an edge separator 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 $1.56\sqrt{n}$.^[2] 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 have 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.^[15]

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.^[16]

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.^[17]

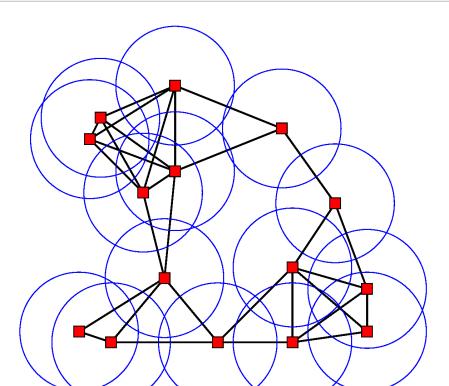
Possibly the earliest known separator theorem is a result of Jordan (1869) that any tree can be partitioned into subtrees of at most $2n/3$ vertices each by the removal of a single vertex.^[6] In particular, the vertex that minimizes the maximum component size has this property, for if it did not then 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 $O((g\Delta n)^{1/2})$.^[18]

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 $\varepsilon > 0$.^[19]

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,^[6] and to the graphs arising from finite element meshes.^[20] 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 $O(k^{1/d} n^{1-1/d})$.^[6]

Applications



An intersection graph of disks, with at most $k = 5$ disks covering any point of the plane.

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 $O(n \log^2 n)$ time,^[21] and the shortest cycle in an unweighted undirected planar graph (its girth) may be found in time $O(n)$,^[22] but although these algorithms both use a similar divide and conquer strategy the divide parts of the algorithms are not based on the separator theorem.

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.^[3] The fill-in of this method (the number of nonzero coefficients of the resulting Cholesky decomposition of the matrix) is $O(n \log n)$,^[23] allowing this method to be competitive with iterative methods for the same problems.^[3]

The separator based divide and conquer paradigm has also been used to design data structures for dynamic graph algorithms^[24] and point location,^[25] algorithms for polygon triangulation,^[15] shortest paths,^[26] and the construction of nearest neighbor graphs,^[27] and approximation algorithms for the maximum independent set of a planar graph.^[25]

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.^[28] 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.^[29]

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 .^[30]

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^[15] 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.^[31] 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.^[32] 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.^{[33][34]}

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.^[35]

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.^[35]

Notes

- [1] Alon, Seymour & Thomas (1990).
- [2] Djidjev (1982).
- [3] George (1973). 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.
- [4] Lipton & Tarjan (1979).
- [5] Gazit & Miller (1990).
- [6] Miller et al. (1997).
- [7] Pach & Agarwal (1995).
- [8] Eppstein, Miller & Teng (1995).
- [9] Spielman & Teng (1996).
- [10] Gremban, Miller & Teng (1997).
- [11] Har-Peled (2011).
- [12] Donath & Hoffman (1972); Fiedler (1973).
- [13] Miller (1986) proved this result for 2-connected planar graphs, and Diks et al. (1993) extended it to all planar graphs.
- [14] Miller (1986); Gazit & Miller (1990).
- [15] Goodrich (1995).
- [16] Seymour & Thomas (1994).
- [17] Lipton & Tarjan (1979); Erdős, Graham & Szemerédi (1976).
- [18] Sýkora & Vrt'o (1993).
- [19] Kawarabayashi & Reed (2010). For earlier work on separators in minor-closed families see Alon, Seymour & Thomas (1990), Plotkin, Rao & Smith (1994), and Reed & Wood (2009).
- [20] Miller et al. (1998).
- [21] Chalermsook, Fakcharoenphol & Nanongkai (2004).
- [22] Chang & Lu (2011).
- [23] Lipton, Rose & Tarjan (1979); Gilbert & Tarjan (1986).
- [24] Eppstein et al. (1996); Eppstein et al. (1998).
- [25] Lipton & Tarjan (1980).
- [26] Klein et al. (1994); Tazari & Müller-Hannemann (2009).
- [27] Frieze, Miller & Teng (1992).
- [28] Bern (1990); Deineko, Klinz & Woeginger (2006); Dorn et al. (2005); Lipton & Tarjan (1980).
- [29] Smith & Wormald (1998).
- [30] Alber, Fernau & Niedermeier (2003); Fomin & Thilikos (2006b).
- [31] Bar-Yehuda & Even (1982); Chiba (1981).
- [32] He, Kao & Lu (2000).
- [33] Blandford, Blellock & Kash (2003).
- [34] Blellock & Farzan (2010).
- [35] Babai et al. (1982); Bhatt et al. (1989); Chung (1990).

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.
- 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.
- Alon, Noga; Seymour, Paul; Thomas, Robin (1994), "Planar separators", *SIAM Journal on Discrete Mathematics* **7** (2): 184–193, doi: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.
- 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, 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.
- 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.
- 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, 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.
- 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.
- 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.
- 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.
- Djidjev, Hristo N.; Venkatesan, Shankar M. (1997), "Reduced constants for simple cycle graph separation", *Acta Informatica* **34** (3): 231–243, doi: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, 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.
- 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.

- 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.
- 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.
- 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, 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.
- 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, JSTOR 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.
- Gilbert, John R.; Tarjan, Robert E. (1986), "The analysis of a nested dissection algorithm", *Numerische Mathematik* **50** (4): 377–404, doi: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.
- 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.
- Har-Peled, Sariel (2011), *A Simple Proof of the Existence of a Planar Separator*, arXiv: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/S009753999359117.
- 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.
- 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, ISBN 0-89791-663-8.
- 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, JSTOR 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Seymour, Paul D.; Thomas, Robin (1994), "Call routing and the ratcatcher", *Combinatorica* **14** (2): 217–241, doi: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, 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.
- 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.
- 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.
- 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.
- 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.
- Chang, Hsien-Chih; Lu, Hsueh-I, *Computing the girth of a planar graph in linear time*, arXiv:1104.4892.
- Wulff-Nilsen, Christian (2009), *Girth of a planar digraph with real edge weights in $O(n \log^3 n)$ time*, arXiv:0908.0697.

Graph minors

In graph theory, an undirected graph H is called a **minor** of the graph G if H is isomorphic to a graph that can be obtained by zero or more edge contractions on a subgraph of G .

The theory of graph minors began with Wagner's theorem that a graph is planar if and only if it does not contain the complete graph K_5 nor the complete bipartite graph $K_{3,3}$ as a minor.^[1] The Robertson–Seymour theorem states that the relation "being a minor of" is a well-quasi-ordering on the isomorphism classes of graphs, and implies that many other families of graphs have forbidden minor characterizations similar to that for the planar graphs.^[2]

Definitions

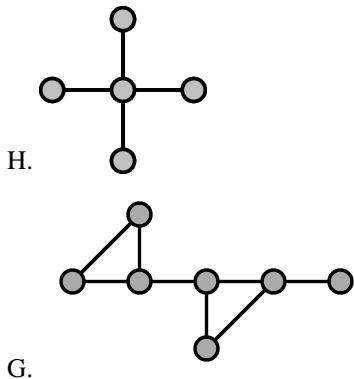
An edge contraction is an operation which removes an edge from a graph while simultaneously merging the two vertices it used to connect. An undirected graph H is a minor of another undirected graph G if a graph isomorphic to H can be obtained from G by contracting some edges, deleting some edges, and 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.

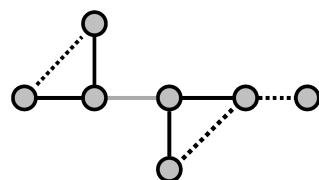
In other contexts (such as with the study of pseudoforests) it makes more sense to allow the deletion of a cut-edge, and to allow disconnected graphs, but to forbid multigraphs. In this variation of graph minor theory, a graph is always simplified after any edge contraction to eliminate its self-loops 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 satisfies the transitive property (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. 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.^[7] 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.^[8] 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 the removal of which splits G into two (possibly disconnected) subgraphs with at most $2n/3$ vertices per subgraph.^[9]

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.^[10] The case $k = 5$ is a restatement of the four color theorem. The Hadwiger conjecture has been proven only for $k \leq 6$,^[11] but remains unproven 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.^{[12][13]}

Minor-closed graph families

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.^[2] 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.^[1]

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,^[14] 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,^[15] 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).^[16] 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.^[17] 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 .^[18]

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 .^[19] It is easy to see that every topological minor is also a minor. The converse however is not true in general, but holds for graph with maximum degree not greater than three.^[20]

The topological minor relation is not a well-quasi-ordering on the set of finite graphs 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 (u,w) . In the case where (u,w) already was present, u 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 .

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.

Immersions and edge-disjoint paths

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.

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 .^[21] 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.^[22]

Notes

- [1] Lovász (2006), p. 77; Wagner (1937a).
- [2] Lovász (2006), theorem 4, p. 78; Robertson & Seymour (2004).
- [3] Lovász (2006) 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] Diestel (2005), Chapter 12: Minors, Trees, and WQO; Robertson & Seymour (2004).
- [5] Lovász (2006), p. 76.
- [6] Lovász (2006), pp. 80–82; Robertson & Seymour (2003).
- [7] Mader (1967).
- [8] Kostochka (1982); Kostochka (1984); Thomason (1984); Thomason (2001).
- [9] Alon, Seymour & Thomas (1990); Plotkin, Rao & Smith (1994); Reed & Wood (2009).
- [10] Hadwiger (1943).
- [11] Robertson, Seymour & Thomas (1993).
- [12] 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,
- [13] Thomas, Robin, *Recent Excluded Minor Theorems for Graphs* (<http://people.math.gatech.edu/~thomas/PAP/bcc.pdf>), p. 14,
- [14] Robertson & Seymour (1983).
- [15] Lovász (2006), Theorem 9, p. 81; Robertson & Seymour (1986).
- [16] Eppstein (2000); Demaine & Hajiaghayi (2004).
- [17] Robertson & Seymour (1993); Demaine, Hajiaghayi & Thilikos (2002).
- [18] Wagner (1937a); Wagner (1937b); Hall (1943).
- [19] Diestel 2005, p. 20
- [20] Diestel 2005, p. 22
- [21] Robertson & Seymour (1995).
- [22] Fellows & Langston (1988).

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, JSTOR 1990903, MR1065053.
- 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.
- 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.
- 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
- 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.
- Eppstein, David (2000), "Diameter and treewidth in minor-closed graph families", *Algorithmica* **27**: 275–291, arXiv:math.CO/9907126, doi:10.1007/s004530010020, MR2001c: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.
- 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.

- Kostochka, Alexandr V. (1982), "The minimum Hadwiger number for graphs with a given mean degree of vertices" (in Russian), *Metody Diskret. Analiz.* **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.
- Lovász, László (2006), "Graph minor theory", *Bulletin of the American Mathematical Society* **43** (1): 75–86, doi: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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Wagner, Klaus (1937a), "Über eine Eigenschaft der ebenen Komplexe", *Math. Ann.* **114**: 570–590, doi: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>)" from 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.^{[1][2][3]}

In this context, the graph is described by a set of vertices V and a binary adjacency relation, 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. 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

$$\exists R, G, B \left(\forall v \in V(v \in R \vee v \in G \vee v \in B) \right) \wedge \\ (\forall u, v ((u \in R \wedge v \in R) \vee (u \in G \wedge v \in G) \vee (u \in B \wedge v \in B)) \rightarrow \neg \text{adj}(u, v)).$$

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. Thus, by Courcelle's theorem, 3-colorability of graphs of bounded treewidth may be tested in linear time.

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.^[2]

References

- [1] Courcelle, Bruno (1990), "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs", *Information and Computation* **85** (1): 12–75, doi:10.1016/0890-5401(90)90043-H, MR1042649
- [2] Kneis, Joachim; Langer, Alexander (2009), "A practical approach to Courcelle's theorem", *Electronic Notes in Theoretical Computer Science* **251**: 65–81, doi:10.1016/j.entcs.2009.08.028.
- [3] Lampis, Michael (2010), "Algorithmic meta-theorems for restrictions of treewidth", in de Berg, Mark; Meyer, Ulrich, *Proc. 18th Annual European Symposium on Algorithms*, Lecture Notes in Computer Science, **6346**, Springer, pp. 549–560, doi:10.1007/978-3-642-15775-2_47.

Robertson–Seymour theorem

In graph theory, the **Robertson–Seymour theorem** (also called the **graph minor theorem**^[1]) states that the undirected graphs, partially ordered by the graph minor relationship, form a well-quasi-ordering.^[2] 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.^[3] 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.^[4]

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.^[5]

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.^[6]

A preorder is said to form a well-quasi-ordering if it contains neither an infinite descending chain nor an infinite antichain.^[7] 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).^[8] 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.^[8] 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.^[9] 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 the finite set H of minimal forbidden minors in F must be exactly the minimal elements in S , so S must have only a finite number of minimal elements.

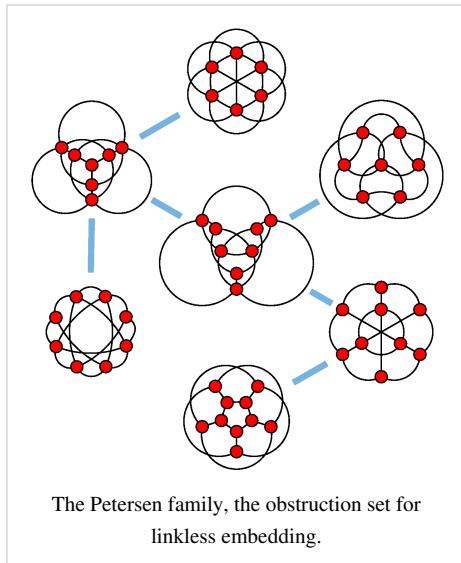
Examples of minor-closed families

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;^[10]
- graphs that are linklessly embeddable in Euclidean 3-space, and graphs that are knotlessly embeddable in Euclidean 3-space;^[10]
- 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.^[11]

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). 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.^[12]

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.^[13] 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] Bienstock & Langston (1995).
- [2] Robertson & Seymour (2004).
- [3] Robertson and Seymour (1983, 2004); Diestel (2005, p. 333).
- [4] Diestel (2005, p. 355).
- [5] Diestel (2005, pp. 335–336); Lovász (2005), Section 3.3, pp. 78–79.
- [6] E.g., see Bienstock & Langston (1995), Section 2, "well-quasi-orders".
- [7] Diestel (2005, p. 334).
- [8] Lovász (2005, p. 78).
- [9] Bienstock & Langston (1995), Corollary 2.1.1; Lovász (2005), Theorem 4, p. 78.
- [10] Lovász (2005, pp. 76–77).
- [11] Chambers (2002).
- [12] Robertson & Seymour (1995); Bienstock & Langston (1995), Theorem 2.1.4 and Corollary 2.1.5; Lovász (2005), Theorem 11, p. 83.
- [13] Fellows & Langston (1988); Bienstock & Langston (1995), 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.
- 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.
- 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.
- 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.
- 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.

- 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.
- 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.

External links

- Weisstein, Eric W., " Robertson-Seymour Theorem (<http://mathworld.wolfram.com/Robertson-SeymourTheorem.html>)" from 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.^[1]

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

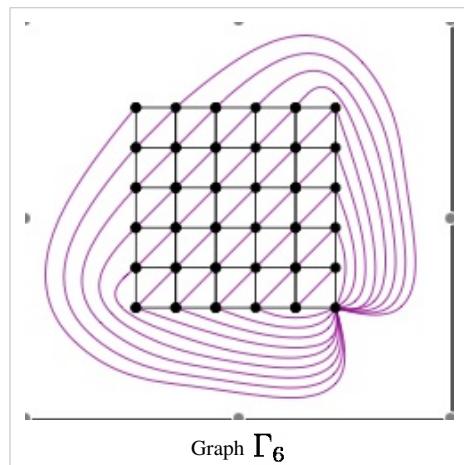
1. For any pair of graphs H, G , such that H is a minor of G and integer k , $(G, k) \in \Pi$ yields that $(H, k) \in \Pi$. In other words, contracting or deleting an edge in a graph G cannot increase the parameter; and
2. there is $\delta > 0$ such that for every $(r \times r)$ -grid R , $(R, k) \notin \Pi$ 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.



Excluding 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]
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 .^[3]

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 (PTAS)

Bidimensionality theory has been used to obtain Polynomial Time Approximation Schemes (PTAS) for many bidimensional problems. If a minor (contraction) bidimensional problem has several additional properties^{[4][5]} then the problem poses Efficient Polynomial Time Approximation Scheme (EPTAS) 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, Max Induced at most d-Degree Subgraph, Max Internal Spanning Tree, Induced Matching, Triangle Packing, Partial r-Dominating Set and Partial Vertex Cover admit an EPTAS on apex-minor-free graphs.

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.^[6]

Notes

- [1] Demaine et al. (2005)
- [2] Demaine & Hajiaghayi (2008)
- [3] Fomin, Golovach & Thilikos (2009)
- [4] Fomin et al. (2010)
- [5] Demaine & Hajiaghayi (2005)
- [6] Fomin et al. (2010)

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.
- 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.
- 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.
- 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.
- 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.
- Fomin, Fedor V.; Lokshtanov, Daniel; Raman, Venkatesh; Saurabh, Saket (2010), "Bidimensionality and EPTAS", arXiv: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=507985297> *Contributors:* APH, Aaditya7, Aaronzat, Abeg92, Agro1986, Ajweinstein, Aknxy, Akutagawa10, AlanAu, Alansohn, Alcidesfonseca, AlexBakharev, Allens, Almi, Altenmann, Ams80, AndreEngels, Andreal05, AndreasKaufmann, Andris, Ankitbhatt, AnnaLincoln, AnonymousDissident, Anthonynow12, Aquae, Arbor, Armoren10, Arvindn, Astrophil, AxelBoldt, AydaDBact, Bananastalktome, Bazuz, Beda42, Bender235, Bereziny, Berteun, Bkell, BlairSutton, BlckKngh, BoleslavBobcik, Booyabazooka, Brad7777, BrentGulanowski, BrickThrower, Brona, Bumbuski, C_S, CRGreathouse, Camembert, CanadianLinuxUser, Caramdir, Cerber, Cesarb, Chalst, CharlesMatthews, Chaszzzbrown, Chmod007, Conversionscript, CorpX, Cs177, D75304, DFRussia, DamienKarras, DanielQuinlan, DavidEppstein, DavidFstr, DawnBard, Dbenbenn, Delaszk, DerHexer, Dgrant, Dicklyon, DiegoUFCG, Dina, Disavian, Discopinsters, Dittaeva, Doctorbozzball, Doradus, Dr.S.Ramachandran, Duncharris, Dycedarg, Dysprosia, Dze27, ElBenevolente, Eleuther, Elf, Eric.weigle, Eugeneiiim, Favonian, FayssalF, Fehrsto, Feeeshboy, Fivelittlemonkeys, FredBradstadt, Fredrik, FvdP, GGordonWorleyIII, GTBacchus, GaiusCornelius, Gandalf61, Garyzx, Geometryguy, GeorgeBurgess, Gianfranco, Gifflite, GiveAFishABone, Glinos, Gmeli, GoShow, Goochelaar, Gragragra, Graham87, GraphTheoryPwns, GregorB, Grog, Gutza, Gyan, HannesEder, HansAdler, Hazmat2, Hbruhs, Headbomb, HenningMakholm, Hirzel, Idiosyncratic-bumblebee, Ignatzmice, Igodard, InoShiro, JakobVoss, Jalpar75, Jaxl, JeLiuf, JeanJulius, Jeronimo, Jheuristic, Jinna, JoelB.Lewis, Joespiff, JohnBlackburne, Johnsopc, Jojifb, JonAwbrey, JonHar, JorgeStolfi, Jwissick, Kalogeropoulos, KarlE.V.Palmen, KellyCoinGuy, Kilom691, KingBee, Knutux, Kope, Kpjnas, Kruusamagi, LC, LOL, Lanem, Ldecola, Liao, Linas, Looxix, Low-frequencyinternal, Lpgeffen, Lupin, Madewokherd, MagisterMathematicae, Magmi, Maherlite, Mailerdiablo, Mani1, Marek69, Marianocewski, MarkFoskey, MarkRenier, MathMartin, Matusz, Maurobio, MaximeDeboschere, McKay, Meekohi, MelbourneStar, Melchoir, MichaelHardy, MichaelSlone, Miguel, MildBillHiccup, Miym, Mlprk, Morphh, Msh210, Mn, Myanw, Myasuda, MynameisJayden, Naerbnic, Nanshu, Nasnema, NethaHussain, Nictich, Ntisimp, Obankston, ObradovicGoran, Ohnoitsjamie, OlegAlexandro, OliFilth, Omigotanaccount, Onorem, Optim, OrangeDog, Orosa, OskarFlordal, Outragedduck, Pashute, Patrick, PaulAugust, PaulMurray, PaulHoadley, PaulTanenbaum, Pcb21, PeterKwok, Photonic, PianaNonTroppi, PierreAbbat, PoorYorick, Populus, Powerthirst123, Protont, Pstanton, Pucky, Pugget, Quaeler, Qutezue, R'n'B, RUARD, Radagast3, RandomAct, Ratiocinante, Renice, Request, Restname, Rev3nt, RexNL, Rjwilmsi, Rmiesen, Roachmeister, RobertMerkel, Robinklein, RobinK, Ronz, RuudKoot, SCEhardt, Sacredmind, Sangak, Shanes, Shd, Shepazu, Shikhar1986, Shizhao, Sibian, SixWingedSeraph, Slawekb, SlumdogAramis, Smoke73, SofiaKarapataki, Solitude, Sonia, Spacefarer, Stochata, Sundar, Sverdrup, TakuyaMurata, Tangi-tamma, Tarotcards, Taw, Taxipom, Tckma, Template namespaceinitialisationscript, TheCaveTroll, ThIsiah, Thesilverbail, ThV, Titanic4000, Tomo, Tompw, Trinitrix, Tyir, TylerMcHenry, UncleDick, Usien6, Vacio, Vonkje, Watersmeetfreak, Whiteknex, Whyfish, Wommer, Wshun, Wsu-dm-jb, Wsu-f, XJamRastafire, Xdenizen, Xiong, XxjwxuX, Yecril, Ylloh, Youngster68, Zaslav, Zero0000, Zoicon5, Zundark, Александър, Канеюк, 448anonymous edits

Glossary of graph theory *Source:* <http://en.wikipedia.org/w/index.php?oldid=509450005> *Contributors:* 3mta3, A1kmm, Aarond144, Achab, Algebraist, Altenmann, Altoid, Andreycric, AnonymousDissident, Archelon, ArnoldReinhold, Bender235, Bethnim, Bkell, Booyabazooka, BrickThrower, Brona, Buenasdiaz, CharlesMatthews, Citrus538, Closedmouth, Cokaban, Csaracho, D6, DVAnDyk, DamianYerrick, DavidEppstein, Dbenbenn, Dcljr, Dcoetze, Denisaroma, DocHoncho, Doradus, Dysprosia, Edward, Eggwadi, ElC, Elwikipedista, EmanueleMinotto, Eric119, Ferris37, GGordonWorleyIII, GTBacchus, GaiusCornelius, Gifflite, Grubber, Happynomad, Headbomb, HorsePunchKid, Hyacinth, IvanŠtambuk, JLLeander, Jalal0, Jitnett, JokesFree4Me, Joriki, JustinWSmith, Jwanders, Jérôme, Kjonnele, Kope, Lansey, Lasuncty, Linas, Markhurd, Mastergreg82, MathMartin, MattGiua, MaximusRex, MclD, Meand, MentorMentor, Mggreenbe, MichaelHardy, MichaelHardy, Michaelnosivad.com, MikhailDvorkin, Miym, Mzamora2, N8wilson, Nonenmac, OlegAlexandro, Ott2, Patrick, PaulAugust, PaulTanenbaum, PeterKwok, Pmq20, Pojo, Populus, Prunesqualer, Quaeler, Quintopia, Quixplusions, R'n'B, Raftox, Rdvdijk, Reinariemann, RekishiEJ, RichFarmbrough, RickNorwood, Ricky81682, Rssetsch, RuudKoot, Salgueiro, Salixalba, SanitySolipsism, Scarpy, Shadowjams, SixWingedSeraph, Skaraoke, Spanningtree, Starcrab, Stux, Sunayana, Sundar, Szbenisz, TakuyaMurata, TheTransliterator, Thechao, Thehotelambush, Tizio, Tomo, Twri, Vonkje, Whorush, Wshun, XJamRastafire, Xiong, Yitzhak, Zaslav, 142anonymous edits

Undirected graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=508868651> *Contributors:* 3ICE, ABF, Abdull, Ahorstermeier, Aitiias, Ajraddatz, Aknorals, Akutagawa10, Algont, Altenmann, Amintora, Anabus, AnandJeyahar, Andrewsky00, Andros1337, Andyman100, Athenray, Barras, BenRG, Bethnim, Bhadani, BiT, Bkell, Bobo192, Booyabazooka, Borgx, Brentdax, Burnin1134, CRGreathouse, Can'tsleep, clownwilleatme, Catgut, Cbordset, Cbuckley, CentroBabbage, Ch'marr, Chewings72, Chinasaur, Chmod007, ChrisTheSpeller, Chronist, Citrus538, Cjfsyntropy, Cornflakepirate, Corti, Crisofilax, Cybercobra, DARTH SIDIOUS2, DRAGON BOOSTER, Danrah, DavidEppstein, DavidFstr, Dbenbenn, Dcoetze, Ddxc, Debolaurus, DenJättraderank, Deor, Dicklyon, Djk3, Dockfish, Dreamster, Drebien, Dureo, Dysprosia, E_mraedarab, Editor70, Ekren, Erhudhy, EricLengyl, Falcon8765, GaiusCornelius, Gandalf61, Gauge, Gene_arbit, George100, Gifflite, Grolmusz, Gutin, Gwaihir, HairyDude, HannesEder, HansAdler, HansDunkelberg, Harp, Headbomb, HenryDelforn(Old), Huynl, Ijdejter, IliaKr., Ilya, InsanityIncarnate, J.delanoy, JNW, JasonQuinn, JeffErickson, Jiang, Joeblakesley, JokesFree4Me, JonAwbrey, JonHar, Joriki, Jpeeling, Jpwi, JuPiTeEr, Jwanders, Karada, KarlScherer3, Kine, Kingmash, Knutux, Kruusamägi, Kuria, LKensington, Liao, Libcup, Lipedia, LuckyWizard, MER-C, Mrg90, Maghnus, MagisterMathematicae, Mahanga, ManningBartlett, MarcvanLeeuwen, Materialscientist, MathMartin, MattCrypto, Mayooranathan, McKay, Mdd, MichaelHardy, MichaelSlone, MikeBorkowski, Minder2k, Mindmatrix, Mitmaro, Miym, Mountain, Myasuda, Mymyhoward16, Nat2, Neilc, Neutrapt, Nihonjoe, Nowhither, Nsk92, Ohnoitsjamie, Oliphant, Oxymoron83, Pafcu, Paleorthid, Patrick, PaulAugust, PaulTanenbaum, PeterKwok, PeterL, Phegy181, PhotoBox, Pinethicket, Possum, Powerthirst123, Prunesqualer, Quaeler, R'n'B, Radagast3, RandomAct, RaseaC, Repied, Request, Regettast, Rglegg, RichFarmbrough, Rjwilmsi, RobertBorgersen, Robertstdeadman, RobinK, Royerloic, Salixalba, Sdrucker, Shadowjams, Shanel, Siddhart, Silversmith, SixWingedSeraph, Someguy1221, SophomoricPendant, Sswn, Stevertigo, StruthiousBandersnatch, Stux, Suchap, Super-Magician, Svick, THENWHO WASPHONE?, Tangi-tamma, Tbc2, Tempodivalse, Tgv8925, TheAnome, Theone256, TimBentley, Timflutre, Timrollpickering, Tman159, TobiasBergemann, Tomharrison, Tomhubbard, Tomo, Tompw, Tomruen, Tosh, Tslocum, Twri, Tyw7, Ulrich1313, Urdutext, Vacio, VaughanPratt, VictorPorton, Void-995, W, Wandrer2, WesleyMoy, WestAndrew, g, Wgunther, WhiteTrillium, WikiDao, WikidsP, Willking1979, Wknight94, Wshun, XJamRastafire, Xavgoem, Xiong, Yath, Yecril, Yitzhak, Ylloh, Zachlipton, Zaslav, Zero0000, Zocky, Zven, Kaneюк, ПикаПика, 400anonymous edits

Directed graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=507858811> *Contributors:* Alesiom, Altenmann, AndreasKaufmann, AnneBauval, BiT, Bkell, Booyabazooka, Bryan.burgers, Calle, Catgut, Corruptcopper, DavidEppstein, DziedBulbash, Einkl, Gifflite, Grue, Hamaryns, Headbomb, HenryDelforn(Old), JustinWSmith, Linas, M-le-mot-dit, Marcuse7, MarkRenier, Miym, Ms.wiki.us, Nbarth, NuclearWarfare, PaulTanenbaum, Pcap, R'n'B, RicardoFerreiraDeOliveira, Rinix, Sinuhe20, SixWingedSeraph, Stepa, Twri, Vadmium, WookieInHeat, Zaslav, 36anonymous edits

Directed acyclic graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=507568439> *Contributors:* AndreasToth, Ap, BrickThrower, BryanDerkens, C-sonic, CRGreathouse, Calculuslover, ChrisHoward, ColoniesChris, Comps, DamianYerrick, Davehi1, DavidEppstein, Dcoetze, Ddxc, Deflective, Delirium, DoostdarWKP, Doradus, Edward, EmreD, Epbr123, Esap, Farisori, Gambette, Gifflite, Greenrd, HannesEder, Henrygb, Homerjay, Jaredwf, Jesser77, Jmr, Jonon, Kaaphi, Kierano, Kwamikagami, MarcGirod, MathMartin, Mauritsmaartendejong, Mindmatrix, MisterSheik, Mitchan, Mpoluton, Nbarth, NoblerSavager, Ort43v, OskarSigvardsson, Patrick, PaulTanenbaum, Pleasantville, Radagast83, RexNL, SamohylJan, Sanya, Sartak, Stephenbez, StevePowell, TakuyaMurata, Template namespaceinitialisationscript, Tiggaen, TrevorMacInnis, Trovatore, Twri, Vonkje, Watcher, WileE.Heresiarch, Wren337, Zaslav, 79anonymous edits

Computer representations of graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=463712910> *Contributors:* 31stCenturyMatt, Aleg, Aaronzat, Alink, AndreasKaufmann, AnyKey, AvicAWB, Avoided, Bluebusy, Booyabazooka, Bruyninc, C4Cypher, Chochopk, Chrisholland, Cooldudefx, Cybercobra, DRAGON BOOSTER, DavidEppstein, Dcoetze, Dysprosia, Epimetheus, FedericoMenaQuintero, Gallando, Gifflite, Gmelli, Graphicalx, Gvanrossum, Hariva, Hobsonlane, Jojifb, JonAwbrey, JonHarder, JorgeStolfi, Juliancolton, Kate4341, Kazubon, Kbrose, KellyCoinGuy, KendrickHang, Klortho, KristjanJonasson, Labraun90, Liao, MaxTerry, NerdyNSK, Nmz787, ObradovicGoran, Ovi1, P0nc, Pbirnie, Pieleric, QuiteUnusual, R.S.Shaw, RG2, Rabarberski, Rborrego, Rd232, Rhanekom, RuudKoot, Sae1962, Saimhe, Salixalba, ScNewcastle, ScaledLizard, SimonFuhrmann, SimonFairfax, SiobhanHansa, Skippydo, Stphung, TFloto, Timwi, Tyir, UKoch, Zoicon5, ZorrollII, ZweiOhren, 140anonymous edits

Adjacency list *Source:* <http://en.wikipedia.org/w/index.php?oldid=507861707> *Contributors:* AndreasKaufmann, Ash211, Beetstra, Bobbertface, Booyabazooka, Chmod007, ChrisTheSpeller, Cob, CraigPemberton, DavidEppstein, Dcoetze, Dysprosia, Fredrik, Garyzx, Gifflite, Hariva, Hobsonlane, Iridescent, Jamelan, JustinWSmith, Jwpurple, Kku, Krazelke, MathMartin, MichaelHardy, NSR, OlegAlexandrov, Only2sea, Patmorin, Rduke, RicardoFerreiraDeOliveira, SPTWriter, Sabrinamagers, Schneelocke, Serketan, ThEcRaCkEr, Twri, 26anonymous edits

Adjacency matrix *Source:* <http://en.wikipedia.org/w/index.php?oldid=507736145> *Contributors:* Abdull, AbhaJain, Aleph4, Arthouse, AxelBoldt, Beetstra, BenFrantzDale, Bender235, Bender2k14, Bitsianshash, Booyabazooka, Burn, Calle, Churnett, Chenopodiaceous, DavidEppstein, Dcoetze, Debresser, Dredstar, Dysprosia, ElBenevolente, FelixHoffmann, Fredrik, Garyzx, Gauge, Gifflite, Headbomb, Hu12, JackSchmidt, JamesBWWatson, JeanHonorio, Jokers, JohnofReading, Jpbowen, Juffi, Kneufeld, Kompot3, Lepida, MarkSweep, MathMartin, Mbogelund, Mdrine, MichaelHardy, Miym, Morre, Natalewis, OlegAlexandrov, Olenielsen, Only2sea, Patmorin, Paulish, Periergeia, Phils, Reinariemann, Rgdboer, Rheth, RichFarmbrough, RobinK, SPTWriter, Salgueiro, Schneelocke, Senfo, Shilpi4560, Snowcream, SlawomirBialy, TakuyaMurata, Tamfang, Tbackstr, TimQ.Wells, Timendum, Tomo, Ttzz, Twri, X7q, YoavHaCohen, YuryKirienko, Zaslav, 72anonymous edits

Implicit graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=491885918> *Contributors:* AndreasKaufmann, Cybercobra, DavidEppstein, MichaelHardy, Olsonist, PhnomPencil, Twri

Depth-first search *Source:* <http://en.wikipedia.org/w/index.php?oldid=506237369> *Contributors:* A5b, ABCD, AndreEngels, Antiuser, Apalsola, Apanag, Batkins, Bbi5291, Beetstra, BenRG, Braddjwinter, BryanDerkens, Bubba73, CesarB, Citrus538, Clacker, CraigBarkhouse, Curtmack, DHN, DavidEppstein, Davideebot, Dcoetze, Dreske, DuncanHope, Dysprosia, EdiTOr, EdLee, ErikvanB, Fkodama, Fragle81, Frecklefoot, G8moon, Gifflite, Gcschroyer, Gurch, Gustavb, Hansamurai, Hemanshu, Hermel, Itai, Ifxd64, Jaredwf, Jef41341, Jeltz, Jerf, Jnl, Jonon, JustinMauger, JustinWSmith, Kate, Kdau, Kesla, Koereta, Kromped, LittleDan, MarkSweep, Marked, MartynasPatasius, Mcpty, Mentifisto, MichaelHardy, Miknight, MildBillHiccup, Miles, Mipadi, Miym, Moosebumps, Nizonstolz, NunoTavares, Nvrmd, Pmussler, PoorYorick, Pukeye, PurpyPupple, Qrantik, Qwertys, Qx2020, Regnaron, RichFarmbrough, RickNorwood, Rrnr, RxS, SPTWriter, Shuroo, Smallman12q, Snowlolf, Srrgei, StaszekLem, Stefan, Steverapaport, Stimp, Subh83, Svick, Tauwasser, Taxipom, Thegeneralguy, Thesilverbail, Thumperward, TimBentley, Vroo, Vsh3r, Waldir, Wavelength, Yonestar, Yuide, Z10x, 205anonymous edits

Breadth-first search Source: <http://en.wikipedia.org/w/index.php?oldid=507504165> Contributors: 28bytes, A5b, Adam Zivner, Akmenon, AllenDowney, Amelio Vázquez, Andr Engels, Asp-GentooLinux, Bazonka, BenFrantzDale, Bender05, Bkkbrad, BlueNovember, Bmathereny, Bonadea, Braintrain0000, CBM, Cananian, CesarB, Cheethdj, Clacker, Corti, CyberShadow, Cybercobra, DHR, Davapepe, David Eppstein, DavidCary, Davideyoth, Dbeatty, Dcoetzee, Dedicatedecoy, Dianna, Dileep98490, Dysprosia, Earobinson, Ecb29, Edaelon, Elayazalmahfouz, EricTalevich, ErikvanB, Ffaarr, Fkodama, Folletto, Frankrod44, Galaxiaad, Gdr, GeeksHaveFeelings, Gelwood, GeorgeBills, Giftlite, Gilliam, Headbomb, Hmwirth, IRP, J Di, Jacobvitkoski, Jan Winnicki, Jbellessa87, Jerf, Jludwig, Jogloran, Justin W Smith, Kdau, Kdnewton, Kenyon, Kesla, Kiand, Kostmo, LOL, Lemontea, LittleDan, Liuofficial, Lixinso, Loading, Lvsmart, M.aznaveh, Madmardigan53, Mahue, Marksweep, Martynas Patasius, Matt.smart, Mcld, Mentifisto, Michael Hardy, Michael Slone, Miia, Miym, Mmtux, Mrwojo, Mxn, Night Gyr, Nisargatanna, NotAnonymous0, Nuno Tavares, Nvrmmrd, Oblatenhaller, Ott2, P b1999, Pcap, Phoe6, Polveroj, Pomte, Poor Yorick, Qranic, Qwertus, Ragzouken, Regnarom, RobinK, Robost, Saurabhbenjari, Seefan, Smallman124, Smig, Sonett72, Spoonz, Staszek Lem, Steverapaport, Szabolcs Nagy, TWiStErRob, TakuyaMurata, Thesilverbail, Timwi, Tooto, Tracytheta, Valery.vv, Velella, VictorAnyakin, Viebel, W3bb0, Wtmitchell, Xodarap00, Zad68, Тиверополник, 266 anonymous edits

Lexicographic breadth-first search Source: <http://en.wikipedia.org/w/index.php?oldid=490753402> Contributors: A3 nm, Arpi Ter-Aragelyan, Bob5972, David Eppstein, Headbomb, 5 anonymous edits

Iterative deepening depth-first search Source: <http://en.wikipedia.org/w/index.php?oldid=509749695> Contributors: APShinobi, Arabani, Arvindn, BenFrantzDale, BryanDerksen, Buptcharlie, CesarB, CharlesGillingham, Codecrux, DerekRoss, DougBell, Dysprosia, EmrysK, ErikvanB, Firenu, Frikle, GPHemsley, Haoyao, HarshalJahagirdar, IlmariKaronen, JMCorye, Jafet, Jamelan, Kesla, Malcohol, MariamAsatryan, Marked, MichaelSlone, Pgan002, Pipasharto, PoisonedQuill, Quadrescence, Qwertus, Regnaron, Solberg, Styfle, Thesilverbail, Wolfkeeper, 30 anonymous edits

Topological sorting Source: <http://en.wikipedia.org/w/index.php?oldid=508804267> Contributors: A3 nm, Abaumgar, Aeons, Apanag, Armine badalyan, Arvindn, Atlantia, BACbKA, Baby112342, Bjarneh, Bluegrass, Bosmon, Branonm, CJLL Wright, Captainfranz, Churnett, Cfete77, Charles Matthews, Chenopodiaceous, Chub, Csl77, Cybercobra, David Eppstein, Dcoetze, Delaszk, DmitriTrix, Dmitry Dzhus, DomQ, Esmond.pitt, Freshenesz, Gmaxwell, Greennrd, GregorB, Grimboy, Gurt Posh, Hannes Eder, Headbomb, Immotminkus, Jim Huggins, Johnwon, Jokes Free4Me, Jonsafari, Joy, Karada, Kenyon, Kingpin13, Leland McInnes, Matthew Woodcraft, Mgreenbe, Michael Hardy, Miym, Mordomo, Obradovic Goran, Oskar Sigvardsson, Plass666, Pasixxx, Pekinenius, Planetscape, Quarl, Qwertys, Rattatoss, RazorICE, RobinK, Slaniel, Stimp, Sundar, Surturz, Torsornalingam, Taxipom, Tobias Bergemann, TomasRiker, Unbitwise, Vonbrand, Vromascantu, Workaphobia, Zundark, پانی، ۸۸ anonymous edits

Application: Dependency graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=497693562> *Contributors:* Abdull, Antonielly, Bryan.burgers, ChrisDailey, Colin Greene, Cthulhu88, David Eppstein, Dmccreary, Ego White Tray, GrAfFiT, Greenrd, Headbomb, Ibasota, Linas, SAE1962, Sam Pointon, SummerWithMorons, Tetha, Twri, Vadmium, 18 anonymous edits

Connected components Source: <http://en.wikipedia.org/w/index.php?oldid=495354773> Contributors: Alansohn, Arlekean, Arthena, AxelBoldt, DPoon, DavePeixotto, David Eppstein, Dcoetzee, Dicktlyon, Edemaine, Eggwadi, Fae, Giftlite, Graue, Headbomb, Hermel, Igor Yalovecky, Jhbdel, JoergenB, Jpbowen, Kostmo, LokiClock, Mashiah Davidson, Meekohi, Mikaeys, Msh210, PerryTachett, Pklosek, R'n'B, RichardVeryard, Ross m mcconnell, Rswarbrick, Rxxtreme, Shadowjams, Staszek Lem, Stdazi, Subversive.sound, Twri, Wavelength, Zundark, 24 anonymous edits

Edge connectivity *Source:* <http://en.wikipedia.org/w/index.php?oldid=508897146> *Contributors:* AmirOnWiki, Booyabazooka, CBM, David Eppstein, Flamholz, Headbomb, Inozem, Justin W Smith, KD5TVI, Mike Fikes, Radagast3, Sjcioosten, Sopoforic, WikHead, 3 anonymous edits

Vertex connectivity *Source:* <http://en.wikipedia.org/w/index.php?oldid=444333449> *Contributors:* David Eppstein, Dcoetzee, Douglas R. White, Gifflite, Headbomb, Justin W Smith, McKay, Mike Fikes, Miym, Nonenmac, Radagast3, Ratfox, Sopoforic, 5 anonymous edits

Menger's theorems on edge and vertex connectivity Source: <http://en.wikipedia.org/w/index.php?oldid=509688222> Contributors: Arthena, Asiantuntija, BeteNoir, Bigbluefish, Blankfaze, Charles Matthews, Cykerway, David Eppstein, Delaszk, Douglas R. White, Dricherby, Giftlike, Headbomb, Heinzi.at, Jason Quinn, Kope, MathMartin, Michael Hardy, Nonenmac, Oleg Alexandrov, Prsenhone1674, RDBurk, Schmitt Seb35, Silverfish & anonymous edits

Far decomposition. Source: <http://en.wikipedia.org/w/index.php?oldid=508060628>. Contributors: David Eppstein, Michael Hardy, Tmjglen, Jitse Niesen, Rjv, Sverre, Wouter, Zvika.

Algorithms for 2-edge-connected components. Source: <http://en.wikipedia.org/w/index.php?oldid=507169980> Contributors: Algebraist, At-par, BearMachine, Bedwyr, Booyabazooka, Creidicie, DaedalusInfinity, DarkMarsSasha, David Eppstein, Dcoetelet, El Roith, Flyrev, Giftlike, GregorB, Headbomb, Herr Satz, Igor Yalovecky, Joshuancooper, Klundarr, Leen Droogendijk, MacKay, Macrolord, Mizarred, Phyx, Pseudoguru, Sun Castro, Vassilis2, XianRastafaria, 20 anonymous edits

Algorithms for 2-vertex-connected components Source: <http://en.wikipedia.org/w/index.php?oldid=502024133> Contributors: David Eppstein, Dcoetzee, Edemaine, Eggwadi, FactorialG, FrantisekAnt, Gilfita, Goceony, Jachyzx, Justin W Smith, Koenan Popper, Lunes, Meliberty, MethaPoetry, Michael Hardy, Ott2, Pmacchabdi, SteinbachTirize, Yashbat, Zvach, 17 anonymous editors

Algorithms for 3-vertex-connected components Source: <http://en.wikipedia.org/w/index.php?oldid=502004709> Cite as: 1941: Hwang, H. and K. Kihara. 1992. Kite and L-shaped graphs. In: F. Harary and B. Randerup (eds.), Topics in Graph Theory, Vol. 2, pp. 103-116. Academic Press, London.

Karger's algorithm for general vertex connectivity Source: <http://en.wikipedia.org/w/index.php?oldid=502123629> Contributors: Bruyninc, ColdthroatEskimo, Daiyuda, David Eppstein, Garg, P.Hall, J.L.J.L., V.L.Q., v080, Sasho, Satoru, W.Steven, 4.11 users, 1 anonymous user

Cyberj0ac, DPoon, David Eppstein, Dcoetze, Giftite, Gioti, Hairy Dude, Headbom, Hvn0413, Integr8e, Jamie King, Jpbown, Justin W Smith, Martin Bravenboer, Mashiah Davidson, M. Crypt0, Nutcracker2007, Oleg Alexandrov, Plbogen, Pypmannetjes, Oleafriver, Quackor, R'n'B, RichardVaryard, Sho Uemura, Silverfish, Sleske, Thiijswijs, Twri, 34 anonymous edits

Tarjan's strongly connected components algorithm Source: <http://en.wikipedia.org/w/index.php?oldid=507100157> Contributors: Andreas Kaufmann, Arthas702, BlueNovember, CBM, Chachutin10, David Epstein, Dcoetzee, Ghazer, Gitlfite, Grubber, Headbomb, Jedaily, Johan.de.Ruiter, LiliHelpa, MattGiua, Michael Veksler, Neilc, Ott2, Rich Farmbrough, Stephen70Edwards, VirtualDemon, 61 anonymous edits

Path-based strong component algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=489068458> *Contributors:* Andreas Kaufmann, Darguz Parsilvan, David Eppstein, DavidCBryant, IOL, Nutcracker2007, Oleg Alexandrov, Ruud Koot, Vadimium, WPPatrol, 4 anonymous edits

Kosaraju's strongly connected components algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=503306045> *Contributors:* Andre.bittar, Boggle3343, David Eppstein, Dcoetzee, Emilkeyder, Favourian, Forbsey, Fridaydal13, Funandrvl, Giftlite, Headbomb, Justin W Smith, Matt Crypto, NickLewicky, Octahedron80, Omnipaediast, Phatsphere, Playswithfire, Prashmohan, RobinK, Ruud Koot, Stifte Svick, 17 anonymous edits

Application: 2-satisfiability Source: <http://en.wikipedia.org/w/index.php?oldid=493412883> Contributors: Aleph4, Andris, Booyabazooka, Bsilverthorn, C. lorenz, CBM, Chalst, Charles Matthews, CharlotteWebb, Chayant, Creidieki, David Eppstein, Dcoetzee, EagleFan, EdH, Fratrep, GoingBatty, Gregbard, Headbomb, Hermel, Igorpak, Jacobolus, Leibniz, Mboverload, Mets501, Mivm, Nitewhneils, Pushpendra, RobinK, Schneelocke, Sun Creator, Tiff098, Twin Bird, Wavelength, Yaronf, Ylloh, Zundark, 19 anonymous edits

Shortest path problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509463388> *Contributors:* Aaron Rotenberg, Alcidesfonseca, Alexander Anoprienko, Alfredo J. Herrera Lago, Altenmann, Andreas Kaufmann, Andris, AxelBoldt, B4hand, BenFrantzDale, Booyabazooka, Brian0918, C_lorenz, Caesura, Camw, Chris Capoccia, Cipher1024, Cthe, Daveagp, David Eppstein, Dcoetzee, Deanphd, Delaszk, Denisarona, Devis, Dmforcier, Download, Dysprosia, ESkog, Edepot, Esess, Fæ, Gaius Cornelius, General Wesc, Giftlite, Glrx, Graph Theory page blanker, Happyuk, Hari6389, Hariva, Headbomb, Jason.surratt, JeLuF, Justin W Smith, Khrose, Kope, LC, Lourakis, Lpgeffen, Lqs, Lzur, MadScientistVX, Marc van Leeuwen, Marcelkes, Mathiasck, Metaprimer, Michael Hardy, Mindmatrix, MiroBrada, MisterSheik, Mkroeger, Nethgirb, Nuno Tavares, Ohnoitsjamie, Oliphant, OverlordQ, Phelanpt, Pigrorsch, Pmlineditor, Qwertys, Rasmusdf, Rich Farmbrough, Rjwilmst, Robert Geisberger, RobinK, RomanSpa, Ruud Koot, Shd, Sho Uemura, Squire55, Taemyr, Templatetypedef, TerraFrost, The Anome, Tommyjb, ToneDaBass, Treessmill, Wolfkeeper, 109 anonymous edits

Dijkstra's algorithm for single-source shortest paths with positive edge lengths Source: <http://en.wikipedia.org/w/index.php?oldid=509768770> Contributors: 2001:628:408:104:222:4DFF:FE50:969, 4v4l0n42, 90 Auto, AJim, Abu adam, Adamarnesen, Adamianash, AgadaUrbanit, Agthorr, Ahy1, AlanUS, Alanb, AlcoholVat, Alex.mccarthy, Allan speck, Alquantor, Altenmann, Andreasneumann, Angus Lepper, Anog, Apalamarchuk, Aragorn2, Arrenlex, Arsstyleh, AxelBoldt, Aydee, B3vigr3b, B6s, BACbKA, B^4, Benof, Beatle Fab Four, Behco, BenFrantzDale, Bgwhite, Bkell, Blueshifting, Boemanneke, Borgx, Brona, CGamesPlay, CambridgeBayWeather, Charles Matthews, Christopher Parham, Cicconetti, Cincourtaprabu, Clementi, Coralinzum, Crazy george, Crefrog, Css, Csurgine, Cyde, Damn7az, Daveag, David Eppstein, Davub, Dcoetzee, Decrypt3f, Deflective, Diego UFCG, Digwuren, Dionyziz, Dmforcier, Dmitri666, DorisSmith, Dosman, Dreske, Drostie, Dudzcom, Dysprosia, Edemaime, ElonNarai, Erel Segal, Eric Burnett, Esrogs, Ewedistrict, Ezrakilti, Ezubaric, Faure.thomas, Foobaz, Foxj, FrankTobia, Frankrod44, Fresheneesz, GRuban, Gaiacarraca, Galoubet, Gerel, Geron82, Gerrit, Giftlite, Gordonnovak, Graemel, Graham87, Grantstevens, GregorB, Guanaco, Gutza, Hadal, Haham hanuka, Hao2lian, Happyuk, Hell112342, HereToHelp, Harry12, Huazheng, Ibmuua, IgushevEdward, Illnab204, Iridescent, Itai, JFocco, JForget, Jacobolus, Jarble, Jaredwf, Jason Rafe Miller, Jellworry, Jeltz, Jevillico, Jheiv, Jim1138, Joelinlimil, JohnBlackbrane, Joneman koo, Jonisocpol98, Jorvis, Julesed, Justin W Smith, K3rb, Khorkb, Keilana, Kesla, Kh naba.

Kku, Kndiaye, Koo, Kostmo, LC, LOL, Laurinkus, Lavaka, Leonard G., LordArtemis, LunaticFringe, Mahanga, Mameisam, MarkSweep, Martynas Patasius, MathMartin, Mathiestck, MattGiucci, Matusz, Mcraig, McCuley, Megharajv, MementoVivere, Merlin444, Mgreeben, Michael Sloane, Mikeo, Mikrosami Akademija 2, Milcke, MindAfterMath, Mkwh813, MusicScience, Mwarren us, Nanobear, NetRoller3D, Nethgirb, Nixdorf, Noogz, Obradovic Goran, Obscurans, Oleg Alexandrov, Oliphant, Optikos, Owen, PesoSw, Piojo, Pol098, PoliticalJunkie, Possum, ProBoj1, Pseudomanas, Pshanks, Psjkje, Pxtreme75, Quidquam, RISHARTHA, Radim Baca, Ramr, RamiWissa, Recognition, Reliableforever, Rhanekom, Rjwilmis, Robert Southworth, RodrigoCamargo, RoyBoy, Ruud Koot, Ryan Roos, Ryanli, SQGibbon, Sambayless, Sarkar112, Shd, Sheepetgrass, Shizny, Shiram, Shuroo, Sidonath, SiobhanHansa, Sk2613, Slakr, Sniedo, Sokari, Some else, Soptynum, Sprphodes, Stdazi, SteveJothen, Subbh83, Sundar, Svick, T0ljan, Tehwikipwnerer, Tesse, Thayts, The Arbitr, TheRingess, Thijswijs, Thom2729, ThomasCHenry, Timwi, Tobei, Tomisti, Torla42, Turketwh, VTBassMatt, Vecrumba, Vevek, Watcher, Wierdy1024, WikiSlasher, Wikipelli, Wildcat dunny, Wphamilton, X7q, Xerox 5B, Ycl6, Yworo, ZeroOne, Zhaladshar, Zr2d2, 571 anonymous edits

Bellman–Ford algorithm for single-source shortest paths allowing negative edge lengths Source: <http://en.wikipedia.org/w/index.php?oldid=506745452> Contributors: Aaron Rotenberg, Abednigo, Aedrichols, Aege, Agthor, AlexCovarrubias, Altenmann, Anabus, Andris, Arleken, B3vigr3b, BackslashForwardslash, BenFranzDale, Bjozen, Bkell, BlankVerse, Brona, CBM, Charles Matthews, CiapAn, Ciphergoth, David Epstein, Dcoetzee, Docu, Drvilelli44, E2ch9, Enochlau, Ephr123, Ferengi, FrankTobia, Fredrik, Fvv, Gadfium, Giftlike, GregorB, Guahnalha, Happyuk, Headbomb, Heineman, Helix84, Iceblock, Istanton, Itai, J.delanoy, Jamelan, Jaredwf, Jellyworld, Jftuga, Josteinaj, Justin W Smith, Konstable, LOL, Lavv17, Mario777Zelda, Mathimke, Mazin07, Meld, Michael Hardy, Miym, N Shar, Narozna, Nihilrites, Nilo Grimsimo, Orbst, P_b1999, PanLevan, Pion, Pjrm, Poor Yoricke, Posix4e, Psksj, Quuxpluseone, Rjwilmis, RobinK, Rspere, SQL, Salix alba, Sam Hocevar, Shuroo, Sigkill, Solon.KR, SpyMagician, Stdazi, Stderr.dk, Stern, Str82no1, Tomo, ToneDaBass, Tsunanet, Tvidas, Waldir, Wmahan, Writer130, Zholadas, 162 anonymous edits

Johnson's algorithm for all-pairs shortest paths in sparse graphs Source: <http://en.wikipedia.org/w/index.php?oldid=494972965> Contributors: Abtinb, AlexTG, Altenmann, Bob5972, Brahle, Charles Matthews, Cyrus Grisham, Danielx, David Epstein, Drilnoth, Gaius Cornelius, Josteinaj, Karl-Henner, Kl4m, MarkSweep, MicGrigni, MorganGreen, Nanobear, Netvor, Octahedron80, Ruud Koot, Twexcom, Welsh, 22 anonymous edits

Floyd–Warshall algorithm for all-pairs shortest paths in dense graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=506712678> *Contributors:* 16@r, 2001:41B8:83F:1004:0:0:27A6, Aenima23, AlanUS, AlexandreZ, Altenmann, Anharrington, Barfooz, Beardybloke, Buagg, C. Siebert, Cachedio, Closedmouth, Cquimper, Daveagp, David Eppstein, Dcoetzee, Dfrankow, Dittymathew, Donhalcon, DrAndrewColes, Fatal1955, Fintler, Frankrod44, Gaius Cornelius, Giftlite, Greenleaf, Greenmattter, GregorB, Gutworth, Harrigan, Hjferrybe, Ingr, J. Finkelstein, JLaTondre, Jaredwf, Jellyphool, Jerryobjetc, Joy, Julian Mendez, Justin W Smith, Kanitani, Kenyon, Kesa, KitMarlow, Kletos, Leycecy, LiDaobing, Luv2run, Mac1421, Magioladitis, MarkSweep, Md. aftabuddin, Michael Hardy, Minghong, Minority Report, Mqchen, Mwk soul, Nanobear, Netvor, Nicapicella, Nishantjr, Nneonneo, Obradovic Goran, Oliver, Opium, Pandemias, Phil Boswell, Pilotguy, Pjrm, Polymerbringer, Poor Yorick, Pxtrremet75, Quuxplusone, Rabarberski, Raknarf44, RobinK, Roman Munich, Ropez, Roseperrone, Ruud Koot, Sakanarm, SchreiberBike, Shadowjaws, Shyamal, Simoneau, Smurfix, Soumya92, Specs112, Sr3d, SteveJothen, Strainu, Svick, Taejo, Teles, Treyshonuff, Volkan YAZICI, W3bb0, Wickethewok, Yuzzzy n, 214 anonymous edits

Suurballe's algorithm for two shortest disjoint paths *Source:* <http://en.wikipedia.org/w/index.php?oldid=491907235> *Contributors:* David Eppstein, Dekart, Robertofcf1, Tedder, Vegaswikan, Vladhed, 7 anonymous edits

Bidirectional search *Source:* <http://en.wikipedia.org/w/index.php?oldid=495769299> *Contributors:* Cobi, David Eppstein, Ddccc, Delirium, Jamelan, JohnBlackburne, Mcld, Michael Devore, Mohammadali69, NathanHurst, Peatar, Quuxplusone, Qwertyus, Regnaron, Xavier Combelle, Xbao, 22 anonymous edits

A* **search algorithm** Source: <http://en.wikipedia.org/w/index.php?oldid=509381675> Contributors: IForTheMoney, 316 student, Abraham, B.S., Acdx, Aedieder, Ahoerstemeier, Albertzeyer, Ale_jrb, Alejandro.isaza, Alex_Krainov, Alex_mccarthy, AlexAlex, Alexander_Shekhtovsov, AllanBz, Altenmann, Andrewrosenberg, Aninhumer, Antonbarkhamsan, Arronax50, AxelBoldt, BACbKA, BIS_Suma_Bdesham, Beau, BenKovitz, Bjorn_Reese, Blogjack, Boom1234567, Braphael, Brona, C. A. Russell, C7protal, Catskul, Chaos5023, Charles_Matthews, Chire, CountingPine, Cycling-professor, DVdm, Damian_Yerrick, Daveage, David_Eppstein, DavidHarkness, Dcoetzee, Ddccc, Difyonzyk, Disavian, Dmytersturnbull, Docu, Dr0b3rt, Dreske, Dysprosia, Electro, Efestefect, Eyal0, Falsedef, Faure_thomas, FelipeVargasRigo, Fiarr, Frasmog, Frecklefoot, Fredrik, Fresheneesz, Furykef, GHPMesley, Gaganbanks1123, Geofadams, George126, Ghodsnia, GiM, Gifflite, Glenstamp, Gmentat, Grantstevens, GregorB, Gulliveig, Gökhān, Hadal, Hallows AG, HandsomeFella, Hart, Headbomb, HebrewHammerTime, Henke37, Hiihammuk, Hodja_Nasreddin, Hornbydd, IMSO_P, IanOsgood, JCarriker, JLaTondre, JackSchmidt, Jacob_Finn, Jamesfisherer, Jiri_Pavelka, Jill_Joelpt, JohnBlackburne, Johnnianfang, JonH, JoshReeves2, Joshua_Issac, Jrouquie, Julesd, Justin_W_Smith, JustinJh, KamuiShirou, Katieh5584, Keenan_Pepper, Keithphw, Kee-son, Kesla, Kevin, Khanser, Kingpin13, Kku, Kndiaye, Korrawit, Kotniski, Kru, Laurens, Lee_J_Haywood, Leirbag.arc, Lotje, MIT_Trekkie, Maco1421, Malcolm, Markulf, Martyr2566, Mat-C, MattiGiua, Mac, McCulley, Mcld, Mellum, Michael_Slone, Millerdl, Mintleaf, MithrandirAgain, Mortense, Mrwojo, Muhenuds, Nanobear, Naugurt, Neil.steiner, Neilc, Nick, Nick_Levine, Nohat, Orea456, Phil_Boswell, Piano non troppo, PierreAbbat, Piojo, Pjrm, Quuxplusone, Qwertys, Rankiri, Rdsmith4, Regnarorn, Remko, Rich_Farmbrough, Ripe, Ritchy, Rjwilmsi, RoySmith, Rspeer, Rufous, Runtime, Ruud_Koot, Salzahran, Samkass, Sanderd17, Sigmundur, Silnarm, Simeon, SiobhanHansa, Siroxo, SkyWalker, Stephenb, SteveJothen, Subbh83, Svick, Sylveron, Sythiss, Szabolcs_Nagy, Tac-Tics, Taejo, Talldean, Tassedethe, Tekhnofriend, TheGoblin, Thsgrn, Timwi, Tobias_Bergemann, TrentonLipscomb, Trisweb, Trudslev, Vanished user 04, VernoWhitney, Whosyourjudas, WillIUther, Yknott, Yuval_Baror, ZeroOne, 368 anonymous edits

Longest path problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509361211> *Contributors:* Andreas Kaufmann, David Eppstein, Dcoetzee, Delaszk, Dfarrell07, Djbwiki, Enricorpq, Eraserhead1, Gifttite, GregorB, Headbomb, Jits Niesen, Michael Hardy, Reyk, RobinK, S.zahiri, Seaphoto, Sidgalt, Skysmith, Some jerk on the Internet, Tetha, Thor Husfeldt, 21 anonymous edits

Widest path problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=498867163> *Contributors:* Daveagp, David Eppstein, Giftlite, KConWiki, Michael Hardy, Ost316, Rjwilmsi, Woohookitty

Canadian traveller problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=500713922> *Contributors:* Algebraist, Andreas Kaufmann, C. lorenz, Cybercobra, Dominus, Draaglom, Erik Warmelink, Giftlite, Mathew5000, Michael Hardy, Reyk, Rjwilmsi, RobinK, Twri, Uxekpat, Vanish2, Welsh, Woohooikitv, Zahy, 6 anonymous edits

Application: Centrality analysis of social networks *Source:* <http://en.wikipedia.org/w/index.php?oldid=507513256> *Contributors:* BAxelrod, Bkkbrad, Borgatts, Chmod644, ChristopherE, Compssim, DarwinPeacock, David Eppstein, Douglas R. White, Eassis, EbrushERB, Edchi, Ergotius, Fabercap, Flyingspuds, Frozen4322, Giftlite, Harthur, Hathawayc, Headbomb, Icarins, J.delaney, Kazastankas, Kku, Loodog, Marlon'n'marion, Mdvs, Mgwalker, Michael Hardy, Michael Rogers, Minimac, Mrocklin, Nburden, Netzwerkerin, Orubt, Piotrus, RafG, RedHouse18, Reedy, Rjwilmsi, Robd75, Sandal bandit, Seanandjason, Sepreece, Sevenp, Stochata, Sundirac, Tgr, Thegzak, Tore.opsisahl, Trumpsterator, Utopiah, Vyznev Xnebara, WaddSpoiley, 64 anonymous edits

Application: Schulz voting system Source: <http://en.wikipedia.org/w/index.php?oldid=509253821> Contributors: Angus, Argyriou, ArielGlenn, ArnoNymus, AugustinMa, Avij, AxelBoldt, Bayberrylane, Baylink, Bdesham, BenjaminMakoHill, BorgHunter, Borgx, BrainMarble, CRGreathouse, Cacycle, Carn, CataTony, Chealer, Daveagp, DavidJWilson, Deathphoenix, Dismas, Dissident, DixonD, Diza, Drbug, DuncanHill, Eequor, Electionworld, EnSamulili, Esrogs, Fahrenheit451, Falkvinge, FrenchTourist, Gamsbart, Geni, Giftlite, Gioto, Grasyop, GregorB, H2g2bob, HairyDude, Hermitage, HippopotamusLogic, IlmariKaronen, Insertrealmname, Iota, IronMan3000, Izno, JamesMcStub, Jguk, Joy, Liangent, Mailerdiablo, MarkusSchulze, McCart42, Meno25, MichaelHardy, Nethigra, Nishkid64, Obersachse, Physicistjdi, Pot, Purodha, Ricardopoppi, RobLa, RobertP.O'Shea, RobertPhilipAshcroft, RodCrosby, SEWilco, SEppley, Sanbec, ScottRitchie, SeL, SimonD, Sjwheeler, Skyfaller, Smartswe, TWCarlson, Tchof, The ed17, ThurnerRupert, Tmh, Tomruen, TpBradbury, TreasuryTag, Twilsonb, Ummit, VBGFScJn3, VyznevXnebars, Wat20, Wingelesubmariner, Wli, Xeroc, Xmath, Yaway, Yaway, 180 anonymous edits

Minimum spanning tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=506583833> *Contributors:* 25or6to4, Addihockey10, AdjustShift, Aednichols, Akashssp, Akerbos, Alangowitz, Alex.mccarthy, Altenmann, Americanhero, Amit_kumar22, Andreas_Kaufmann, Andrewbadr, Andris, Aranee, Asymmetric, B6s, Bender2k14, Blueharmony, Boleslav Bobcik, Booyabazooka, CRGreathouse, Chris the speller, Christian Thiemann, Cojoco, Coverback, DARTH SIDIOUS 2, DHN, Daf, David Eppstein, David Martland, Dcoetzee, DevastatorIIC, Drsadeq, Duckax, Dysprosia, Eda eng, Ejarendt, Eric Kvaalen, Evoluzion, George94, GeorgeBills, Giftlite, Haham hanuka, Hariva, Headbomb, Hiiiiiiiiiiiiiiiiiiii, Hike395, HsuanHan, Itai, J.delaney, Jan Wassenberg, Jeff abrahamson, Jgarrett, Jogloran, Julesd, Juniuswikia, K3rb, KKoolstar, Kallerdis, Keenan Pepper, Kenyon, King of Hearts, Kyle.cackett, LC, LOL, LSG1, LX, Ladida, Leeming, Levin, Lourakis, Luisbona, Lyrixoid84, Magnus Manske, MathMartin, Matpiw, Mbclimber, Mebden, Michael Angelkovich, Michael Hardy, Michael Slone, Mindmatrix, Mislerou, Miym, Mohitsinghr, Musically ut, Niccecupotea, Nixdorf, Obelix83, Oleg Alexandrov, Oxymoron83, Pakaran, Pashute, PeterDz, Poor Yorick, Pqrstuv, Q309185, Qwertyus, R'n'B, Rami R, Rjwilmsi, Romanman, Shmomuffin, Shaymal, Sun Creator, Superninja, Swfng8, Thehelpfulone, Timo Honkasalo, Tomekpe, Tricorne, Turinghasaposse, Vevek, Vitaly, Voomoo, Wladston, [Wzha553_X7a_Zzen_150_anonymous_edits](http://www.cgi.org.Wzha553_X7a_Zzen_150_anonymous_edits)

Borůvka's algorithm. Source: <http://en.wikipedia.org/w/index.php?oldid=508816742> Contributors: Adam Zivner, Agamir, Alieseraj, Altenmann, Angr, Auringero, Captain Segfault, Charles Matthews, Cobi, David Eppstein, Davif, Dcoetelet, Dysprosia, Electricfish2, Giftlike, Grendelhan, Hike395, Jaredwf, Jk2q3jrkls, Jogloran, Kaeso, Kendrick7M, Mgreenbe, Michael Sloane, Nikothebrain, Pierrick Abbat, Rich Ermabroulh, Rov, Sanjith, Sfurasz, Tencsej, 27 anonymous edits

Mgreenbe, Michael Angelkovich, Michael Slone, Michael.jaeger, Mikaea, MindAfterMath, MiqayelMinasyan, Miserlou, Mitchellduffy, Mmtux, MrOllie, Msh210, Mysid, NawlinWiki, Oli Filth, Omargamil, Oskar Sigvardsson, Panarchy, Pinethicket, Poor Yorick, Pranay Varma, Qwertys, R3m0t, Sarcelles, Shelfreef, Shen, Shreyasjoshi, Simeon, SiobhanHansa, Spangineer, THEN WHO WAS PHONE?, The Anome, Tokatara, Treyshonoff, Wakimakirrolls, Wapcaplet, Wavelength, YahkoKa, Ycl6, Zero2000, 162 anonymous edits

Prim's algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=509659409> *Contributors:* 4v4l0n42, A5b, Abu adam, Adam Zivner, Ahy1, Alexandru.Olteanu, Altenmann, Arrandale, BAxelrod, Berteun, Bigmonachus, Carmitsp, Cesarsorm, Charles Matthews, Connelly, David Eppstein, Dcoetzee, Dessertsheep, Dmn, Dysprosia, EBusiness, Ecb29, Epachamo, Erpingham, Ertzeid, Fresheneesz, Gatoatigrado, Giftlite, Hairy Dude, Hariva, Harry12, Hike395, Igoldste, IgushevEdward, Inquam, Isnow, Jamelan, Jaredwf, Jirkka6, Joe Schmedley, Justin W Smith, K3rb, Karlhendriks, Kingsindian, Kiril Simeonovski, Krivokon.dmitry, Krp, LC, LOL, Lambiam, MER-C, Magioladitis, Matusz, Maurobio, McIntosh Natura, Michael Hardy, Mihiirpmehtha, Mindbleach, Mitchellduffy, Miym, Moink, N4nojahn, Obradovic Grob, Opium, Pit, Platypus222, Poor Yorick, Prasanth.moothedath, Purpy Purple, R6144, Rami R, Rbrewer42, Rhmhughes, Romanmann, Ruud Koot, SchuminWeb, Seano1, Sethleb, Shen, SiobhanHansa, Squizzz, SuperSack56, Swfung8, Tameralkuly, Teimu.tm, Treyshonuff, Turketwh, UrsusArctos, Whosyourjudas, Wikimol, Wikizoli, Wtmitchell, Ycl6, Ysangkok, ZeroOne, 184 anonymous edits

Edmonds's algorithm for directed minimum spanning trees *Source:* <http://en.wikipedia.org/w/index.php?oldid=411605033> *Contributors:* Almit39, Bender2k14, Cronholm144, Edaeda, Erik Sjölund, Finity, Gifflite, GregorB, Headbomb, John of Reading, Kallerdis, Kinglag, Lida Hayrapetyan, Mcld, Michael Hardy, Neuralwiki, Omnipaedia, Senu, Silly rabbit, Yuide, 15 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=500727197> *Contributors:* 2001:DA8:8000:E0C2:226:B9FF:FE19:F8B8, Darklawhf, David Eppstein, MildBill Hiccup, Miym, Quxplusone, SpunkeyLepton, 4 anonymous edits

K-minimum spanning tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=476260203> *Contributors:* A.A.Graff, Charles Matthews, Cronholm144, David Eppstein, Kilom691, R'n'E, Sdudah, Silverfish, Timo Honkasalo, Tmigler, 4 anonymous edits

Capacitated minimum spanning tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=405141503> *Contributors:* Beeblebrox, Headbomb, IceCreamAntisocial, Lynxoid84, Rjwilmsi, 3 anonymous edits

Application: Single-linkage clustering. **Source:** <http://en.wikipedia.org/w/index.php?oldid=499751613> **Contributors:** 3mta3, Ankit Rakha, Chire, Cindamuse, David Eppstein, Fabrictramp, Gary King, Golddan Gin, Headbomb, Lynxoid84, Manuel freire, Melcombe, Michael Hardy, Pot, Wosam, XLRater, Zeno Gantner, 5 anonymous edits

Application: **Maze generation** *Source:* <http://en.wikipedia.org/w/index.php?oldid=506097861> *Contributors:* 041744, Ap, Arichnad, Brandnewotter, Bryan Derksen, Charles Matthews, Chris Roy, Cruiser1, Cyp, Dabigkid, DerGraph, Dysprosia, Eequor, Ellenshuman, EvanBarr, Frependulum, Furykef, GregorB, Grendelkan, Haham hanuka, Hu12, JDougherty, Javawizard, Karada, Keerthan, Kevin Saff, Melcombe, Mernen, Mfoltin, Michael Hardy, Mr2001, Nandhp, Nicolas.Rougier, Nskillen, Plugwash, Purpy Puppie, Qutezue, RHJC, RJJFJR, Radiant chains, Riking8, SchiftyThree, ShardK, SimonP, Simplex, Staszek Lem, SveinMarvin, Technicat, The Earwig, Timwi, 61 anonymous edits

Clique problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=508843284> *Contributors:* Alanyst, Andreas Kaufmann, Andrewpmk, Ascnder, Bender2k14, Bkell, Charles Matthews, Chmod007, Chris the speller, David Eppstein, Dcoetzee, Elias, Gambette, Gifflite, Headbomb, Huntster, JidGom, Joannatrykowska, JoeKearney, Justin W Smith, Kimbly, Kwertii, Leuko, Lilac Soul, LokiClock, M1ss1ontomars2k4, Mailer diablo, Mastergreg82, Materialscientist, MathMartin, Maximus Rex, Mellum, Michael Hardy, Miym, Neilc, Nilesj, Obradovic Goran, Oddbd07, Ott2, PV=nRT, Peap, Phil Boswell, Poor Yorick, R'n'B, Rampion, Redrose64, Rentier, RobinK, Shanes, Silversmith, Singleheart, Taw, Thore Husfeldt, Timwi, Tonysonneyp, Toobaz, Twocts, Twri, Unyoyega, W, Wavelength, Whbm1058, Xnn, Ylloh, Yurik, Zachlipiton, 42 anonymous edits

Bron–Kerbosch algorithm for listing all maximal cliques *Source:* <http://en.wikipedia.org/w/index.php?oldid=498423900> *Contributors:* Akuchaev, Andreas Kaufmann, Benhiller, Damiens.rf, David Eppstein, Headbomb, Michael Hardy, Mooncake2012, R'n'B, RobinK, Switchercat, 10 anonymous edits

Independent set problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=503324822> *Contributors:* Adi.akbartauhidin, Alexmf, Aliabbas aa, Amelio Vázquez, Andris, Arjayay, Bkell, Bramschoenmakers, C_lorenz, Calle, Citrus538, Cjoa, CommonsDelinker, Davepape, David Eppstein, Dcoetzee, Deuler, Dysprosia, Freeman0119, Giffile, Hv, Jamelan, Jeffreywarrenor, Jittat, Justin W Smith, LandruBek, Life of Riley, Mahue, Mailer diablo, Manuel Anastacio, Marozols, MathMartin, Maxal, Mellum, Michael Hardy, Miym, Petter Strandmark, RobinK, Rocchini, Spoon!, The Anome, Thore Husfeldt, TwanVl, Twri, YasarM, Ylloh, Zavash, 29 anonymous edits

Maximal independent set *Source:* <http://en.wikipedia.org/w/index.php?oldid=503185509> *Contributors:* Austin512, Booyabazooka, CBM, Charles Matthews, David Eppstein, Hermel, Jvimal, Kswenson, Leithp, Life of Riley, Miym, Phil Boswell, RobertBorgersen, RobinK, Silverfish, Twri, Yllohh, Zaslav, 9 anonymous edits

Bipartite graph Source: <http://en.wikipedia.org/w/index.php?oldid=509527747> Contributors: 149AFK, Altenmann, BiT, Bkkbrad, Bobo192, Booyabazooka, Brighterorange, Burn, CGRGreathouse, Catgofire, Chariecurri, Corti, David Eppstein, Delirium, DoostdarWP, Eric119, Freiberg, Gassa, Giftlite, H@rld, Hoarayfortunes, Howard McCay, Jason Klaus, Jdpipe, Jdudar, JoergenB, Jon Awbrey, Jpbrown, Kevimmon, Manojmp, MathMartin, Melchior, Michael Dinolfo, Michael Hardy, Mild Bill Hiccup, MistYi, Miym, Nethgirb, Netpilot43556, Nonenonam, Paul August, PaulTanenbaum, Pgdx, Pineticket, Robert Illes, Shahab, Shmomuffin, TakuyaMurata, Tom harrison, Twry, 92 anonymous edits

Greedy coloring *Source:* <http://en.wikipedia.org/w/index.php?oldid=506481529> *Contributors:* David Eppstein, DrpickeM, Hermel, Igorpak, Martynas Patasius, 2 anonymous edits

Application: Register allocation *Source:* <http://en.wikipedia.org/w/index.php?oldid=502503222> *Contributors:* Abdull, AndrewHowse, CanisRufus, Corti, Danfuzz, David Eppstein, DavidCary, Dcoeteur, Doradus, Ecb29, FireIce, Fred Gandy, FretPySpang, Guy Harris, Headtbody, Jamelan, Jonathan Watt, Jpo, Jter, Kdcoperc, Lecanard, Liao, Mdf, Mike Van Emmerik, Msiddalingaiah, Naasking, Ndanielm, Prari, Pronestox, Radagast83, Robost, Sparshong, Speight, SteveJothan, Taw, Tedickey, Walk&check, Wlievens, 55 anonymous edits

Vertex cover Source: <http://en.wikipedia.org/w/index.php?oldid=507478419> Contributors: Bardia90, Bender2k14, C. lorenz, David Eppstein, Dcoetzee, Dedevelop, Exol, Gdr, Giftlite, Goshafigosha, IvanLanin, Jamelan, JeremyBooms, JoergerB, Justin W Smith, MathMartin, Mellum, Michael Hardy, Michael Sloane, Miym, Mycroft80, Njahnke, Ricardo Ferreira de Oliveira, Riwiimski, RobinK, Ross Fraser, Thore Husfeldt, YlloB 25 anonymous edits

Dominating set *Source:* <http://en.wikipedia.org/w/index.php?oldid=491721693> *Contributors:* Benandorsqueaks, Bodlaender, Charles Matthews, Danielx, David Eppstein, Dmcq, Evilbu, Goshafigosha, Headbomb, Isopnark, Justin W Smith, Michael Hardy, Miym, Ntisimp, PieFlu, Oink, SuneI, Thore Husfeldt, 14 anonymous edits

Feedback vertex set *Source:* <http://en.wikipedia.org/w/index.php?oldid=501854453> *Contributors:* A3 nm, AED, Booyabazooka, David Eppstein, Dcoetzee, George Laparis Funk Studio, Gosha figosha, Hadal, Headbomb, Hermel, JoshRosen, Jrsteenhuisen, Mellum, Michael Hardy, Mr. Lefty, Ott2, Paul August, RobinK, Serggasp, Stevey7788, That Guy, From That Show!, Yixin.cao, Nihal, 8 anonymous edits

Feedback are set *Source:* <http://en.wikipedia.org/w/index.php?oldid=499876854> *Contributors:* CRGreathouse, Charles Matthews, David Eppstein, Dcoetzee, Eyal0, Graphile, Greenrd, Handbamb, Hormel, Kelly Martin, LoliClock, Mallym, Paul August, RobinK, That Guy From That Show!, Turi Vadrum, Yixin eno, 18 anonymous edits

Eulerian path Source: <http://en.wikipedia.org/w/index.php?oldid=509654974> Contributors: Altenmann, Anakata, Armend, Artem M. Pelenitsyn, Arthena, Arthur Rubin, At-par, Audiovideo, Bender235, Boemanneke, Booyabazooka, Bueller 007, Bugthink, CRGreathouse, Cbrarick1, Charles Matthews, CommonsDelinker, Dan100, David Eppstein, DoostdarWKP, Doradus, Dwyer, Dysprosia, Esnyder, Ekarulf, El Roih, Eythian, Ffangs, Fr33kmr, Frazz, Fredrik, Friginator, GTubio, Gelingvistoj, Giftlite, Gogo Dodo, Hanru, Haterade111, Henning Makholm, Herbee, Hippophaë, Igorpak, Ilmari Karonen, Iosif, Jacobolus, Jason Quinn, John Sheu, 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, Omnipaedista, Peter Kwok, Pixeltoot, PonyToast, Powerpanda, Red Bulls Fan, Ricardo Ferreira de Oliveira, Rjwilmsi, Rompelstompe1777, Rschwieb, Taxman, Tide rolls, Tommy2010, Trevor Andersen, Twri, Uni4dxf, Vonkie, Yatin mirsikow, Zero0000, 114 anonymous edits

Hamiltonian path *Source:* <http://en.wikipedia.org/w/index.php?oldid=509263160> *Contributors:* Alansohn, Albywind, Algebraist, Algont, Altenmann, Americanhero, Andrej.westermann, Anikingos, Armanbuet09, ArnoldReinhold, Aronzak, Arthena, Arvindn, Austinmohr, Bigmonachus, Bkell, Bryan.burgers, Catgut, Cdunn2001, CharonX, Cliff, Da Joe, David Eppstein, Daviddaved, Deljr, Dcoetzee, Dimadima, Dwyerj, Dysprosia, ElommolE, Erwinrcat, Etaoin, Ettrig, Ewlyahoocom, Giftlite, GodfriedToussaint, Haham hanuka, Hairy Dude, Headbomb, Henning Makholm, Henrygb, Henrylaxen, Igorpak, InverseHypercube, Ifsfisk, Jic, JoergenB, John Vandenberg, Justin W Smith, Jwpat7, Kenb215, KirbyRider, Kope, Kri, Kurkku, Lambiam, Lemmio, Lesnail, M4gnom0n, Martynas Patasius, MathMartin, Matt Cook, McKay, Meld, Mhym, Michael Hardy, Michael Slone, Miym, Myasuda, Nafsdadh, Nguyen Thanh Quang, Ninjagecko, Nr9, Nsantha2, Obradovic Goran, Ojigiri, Oleg Alexandrov, OneWeirdDude, Paul August, Paul bryner, Pcb21, Perryar, Peruvianllama, Peter Kwok, Populus, RDBury, Radagast3, Rafaelmonteiro, Reddi, Robomaster, Rocchini, Rupert Clayton, SamAMac, Shell Kinney, Shreevatsa, Sirommit, SkyLined, SlumdogAramis, Smijg, Status quo not acceptable, Stdazi, TakuyaMurata, Tanvir 09, Thore Husfeldt, Thue, Twri, Vinayak Pathak, Yecril, Zanetu, ۱۲۸ anonymous edits

Hamiltonian path problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=501133866> *Contributors:* Altenmann, Andreas Kaufmann, Antidrugue, Araujoraphael, Aronzak, Arvindn, Contesteen, CryptoDerk, Davepape, David Eppstein, Dcoetzee, Dominius, Dwca89, Ecb29, Ewlyahoocom, Giftlite, Graph Theory page blanker, Henrylaxen, Hqb, Ixfd64, Jamelan, Laurusnobilis, Luuva, MKoltnow, Mariehuynh, MathMartin, Matt Cook, Michael Slone, Miym, Nr9, Obradovic Goran, Pahari Sahib, Rjwilmsi, RobinK, Rupert Clayton, Sachmo2, Sbluen, Shreevatsa, Smaug123, Taxman, Thore Husfeldt, Vichter, Wicher Minnaard, 41 anonymous edits

Travelling salesman problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509706308> *Contributors:* 130.233.251.xxx, 1exec1, 28421u2232nfencenc, 4ndyD, 62.202.117.xxx, ANONYMOUS COWARD0xCODE, Aaronbrick, Adammathias, Aftermath1983, Ahoerstemeier, Akokskis, Alan.ca, AlanUS, Aldie, Altenmann, Andreas Kaufmann, Andreasr2d2, Andris, Angus Lepper, Apanag, ArglebargleIV, Aronisstat, Astral, AstroNomer, Azotlichid, B4hand, Bathysphere, Bender2k14, BenjaminTsai, Bensin, Bernard Teo, Bjornson81, Bo Jacoby, Bongwarrior, Boothinator, Brian Gunderson, Brucevdk, Brw12, Bubba73, C. Iorenz, CRGreathouse, Can't sleep, clown will eat me, Capricorn42, ChangChienFu, Chris-gore, ChrisCork, Classicalacon, Cngoulimis, Coconut7594, Conversion script, CountingPine, DVdm, Daniel Karapetyan, David Eppstein, David.Mestel, David.Monniaux, David.hillshafer, DavidBiesack, Davidhorman, Dbfirs, Dcoetze, Devis, Dino, Disavian, Donarreiskoffer, Doradas, Downtown dan seattle, DragonflySixtyseven, DreamGuy, Dwdhwh, Dysprosia, Edward, El C, Ellywa, ErnestSDavis, Fanis84, Ferris37, Fioravante Patrone, Flapir, Fmccown, Fmrstatter, Fredrik, French Tourist, Gaeddal, Galoubet, Gdsssy, Gdr, Geofftech, Giftlite, Gnomz007, Gogo Dodo, Graham87, Greenmatter, GregorB, H, Hairy Dude, Hans Adler, Happyuk, Haterade111, Hawk777, Herbee, Hike395, Honnza, Hyperneural, Ironholds, Irrevenant, Isaac, IstvanWolf, IvR, Ixfd64, J.delanoy, JackH, Jackbars, Jamesd9007, Jasmje2, Jasonb05, Jeffhoy, Jim.Callahan, Orlando, John of Reading, Johngouf85, Johnleach, Jok2000, JonathanFreed, Jsamarziya, Jugander, Justin W Smith, KGV, Kane5187, Karada, Kenneth M Burke, Kenyon, Kf4bdy, Kiefer.Wolfowitz, Kjells, Klausikm, Kotasik, Kri, Ksana, Kvamsi82, Kyokpae, LFaraone, LOL, Lambiam, Lanthanum-138, Laudaka, Lingwanjae, MSGJ, MagicMatt1021, Male1979, Manitupula, MarSch, Marj Tiefert, Mark91, Martynas Patasius, Materialscientist, MathMartin, Meld, Mdd, Mellum, Melsaran, Mhahsler, Michael Hardy, Michael Slone, Mild Bill Hiccup, Miym, Mojoworker, Monstergurk, MoraSique, Mormegil, Musiphil, Mzamora2, Naff89, Nethrig, Nguyen Thanh Quang, Ninjagecko, Nobbie, Nr9, Obradovic Goran, Orfest, Ozziev, Paul Silverman, Pauli133, Pegasusbupt, PeterC, Petrus, Pgr94, Phcho8, Piano non troppo, PierreSelim, Pleasantville, Pmdboi, Pschaus, Qaramazov, Qorilla, Quadrell, R3m0t, Random contributor, Ratfox, Raul654, Reconsider the static, RedLyons, Requestion, Rheun, Richmeister, Rjwlmnsi, RobinK, Rocarvaj, Ronaldo, Rror, Ruakh, Ruud Koot, Rya Roos, STGM, SaedVeradi, Sahaugin, Sarkar12, Scravy, See82, Seraphimblade, Sergey539, Shadowjams, Shaefat, Sharcho, ShelfSkewed, Shoujun, Shubhrasanakar, Siddhart, Simetrical, Sladen, Smmurphy, Smremde, Smyth, Some standardized rigour, Soupz, South Texas Waterboy, SpNeo, Spock of Vulcan, SpuriousQ, Stemonitis, Stevertigo, Stimpy, Stochastix, StradivariusTV, Superm401, Supermjina, Tamfang, Teamtho, Tedder, That Guy, From That Show!, The Anome, The Thing That Should Not Be, The stuart, Theodore Kloba, Thisisbossi, Thore Husfeldt, Tigerqin, Tinman, Tobias Bergemann, Tom Duff, Tom3118, Tomgally, Tomhubbard, Tommy2010, Tsplog, Twas Now, Vasil, Vgy7ujm, WhatIsFeelings?, Wizard191, Wumpus3000, Wwwwolf, Xiaojeng, Xnn, Yixin.cao, Ynhockey, Zaphraud, Zeno Gantner, ZeroOne, Zyqqh, 551 anonymous edits

Bottleneck traveling salesman problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509363651> *Contributors:* Altenmann, Andreas Kaufmann, Ansell, David Eppstein, Dcoetze, Dysprosia, Gdr, IRRelayer, Oleg Alexandrov, Pkrekcer, Silverfish, TRauMa, Vespristiano, 6 anonymous edits

Christofides' heuristic for the TSP *Source:* <http://en.wikipedia.org/w/index.php?oldid=486433260> *Contributors:* A3 nm, Andreas Kaufmann, Apanag, Cerebellum, Daniel Dandrade, Edward, Fkoenig, H.ehsaan, Headbomb, Lanthanum-138, Lynxoid84, Michael Hardy, Nobbie, Omnipaedita, Ruud Koot, Vaclav.brozek, Zyqqh, 29 anonymous edits

Route inspection problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=506044524> *Contributors:* AbcXyz, Arthur Rubin, Aviados, Bender235, Bryan Derksen, C. lorenz, Da Joe, David Eppstein, Dysprosia, ErikvanB, FvdP, Giftlite, Headbomb, Honnza, Justin W Smith, Kedwar5, Mangledorf, Mariozzyx, Michael Hardy, Mikhail Dvorkin, Mwtwoes, PV=nRT, Petermgiles, PolyGnome, RobinK, Rockstone35, SD5, Sam Hocevar, Shuroo, Softest123, Stern, Taejo, VictorAnyakin, WhiteCrane, Wik, Zaslav, 33 anonymous edits

Matching *Source:* <http://en.wikipedia.org/w/index.php?oldid=508637692> *Contributors:* Adking80, Aednichols, Altenmann, Ankushshah89, Aranel, Azer Red, Bender2k14, CRGreathouse, Calle, Cyhawk, Daqu, David Eppstein, Dcoetze, Debamf, DonDiego, Edemaine, El C, Excirial, Faisalsyn, Favonian, Fghtngthfght, Giftlite, Green Snake, Gutza, H@r@ld, Hermel, Itamarro, Jammydodger, Jdpipe, Kainaw, Kamyar1, Kerry Raymond, Kilom691, Kraymer, LOL, Lambiam, Mac, Madheros88, Mamling, Mangarah, MarSch, Markoid, Mastergreg82, MathMartin, Maxal, Mental Blank, Michael Hardy, Michael Slone, MickPurcell, Mintleaf, Miym, Mmarci111, Movado73, Nils Grimsimo, Non64, Oliphant, Omegatron, Oudmatie, Pparys, RamanTheGreat, Rentier, RobertBorgersen, Sahba ezami, Smartnut007, Squizzz, Surfingpete, TheEternalVortex, Tim32, Tomo, Yllo, Zaslav, 90 anonymous edits

Hopcroft–Karp algorithm for maximum matching in bipartite graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=507216698> *Contributors:* Altenmann, Anks9b, David Eppstein, Edward, Enobrev, Exile oi, Gnagnoyil, GregorB, Headbomb, Justin W Smith, Kathmandu guy, Mange01, MarSukiasyan, Michael Veksler, Miym, Mogers, Nils Grimsimo, Nitay, Oliphant, Pazabo, Shkoorah, Thanabhat.jo, Tim32, Twanvl, X7q, Zhacob, ۳۳ anonymous edits

Edmonds's algorithm for maximum matching in non-bipartite graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=498616874> *Contributors:* A3 nm, Bender2k14, Bkell, Daiyuda, Daveagp, Edemaine, Favonian, Girlwithgreeneyes, Headbomb, Mangarah, Markoid, Michael Hardy, Miym, Ruud Koot, Wismon, 13 anonymous edits

Assignment problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509363764> *Contributors:* \$Mathe94\$, A.A.Graff, AZhnaZg, Altenmann, Andrewa, Anonymous, Arthena, Benbest, Bender2k14, Charles Matthews, David Eppstein, Evercat, Fnielsen, Garbodor, Infrogmation, Jklm, Justin W Smith, Karipuf, Klahnako, Lage, Lambiam, Lawsonsj, Materialscientist, Michael Hardy, Michael Slone, MichaelGensheimer, Miym, Nils Grimsimo, Oleg Alexandrov, Onebyone, Ophiccius, PAS, Paul Stansifer, Pearle, RJFJR, Rjpb, Rogerb67, Scuirinæ, SimonP, Teknic, The Anome, Tribaal, Vikrams jammal, WikHead, Wikid77, Wshun, 52 anonymous edits

Hungarian algorithm for the assignment problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=506439233> *Contributors:* \$Mathe94\$, Ahmadabdolkader, Andreas Kaufmann, Aostrander, Arichnad, Balajiivish, Bender2k14, Bkkbrad, Buehrenn, Culipan, David Eppstein, Delaszk, Delta 51, Discospinster, DrJunge, Egnever, Fnielsen, Giftlite, Immunize, Incredible Shrinking Dandy, Jjsosoos, Jokes Free4Me, Justin W Smith, Klahnako, Kope, LoveEncounterFlow, LutzL, Macrakis, Mahue, Mboverload, Mekeor, Michael Hardy, Mike0001, Miskin, Miym, NSiiN, Nils Grimsimo, Nkarthiks, NoldorinElf, 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, Vacio, Watson Ladd, Werdna, Zweije, 96 anonymous edits

FKT algorithm for counting matchings in planar graphs *Source:* <http://en.wikipedia.org/w/index.php?oldid=492834087> *Contributors:* A3 nm, Bearcat, Bender2k14, Extra999, Headbomb, Michael Hardy, R'n'B, StAnselm, 3 anonymous edits

Stable marriage problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=506551590> *Contributors:* Alroth, Andrewaskew, Araqsy Manukyan, Arthur Frayn, Augur, BlckKnght, Brusegadi, Burn, Cachedio, Dalhamir, Danrah, David Eppstein, Dcoetze, Dr. Universe, Ecb29, Evercat, Felagund, Geoffrey, Giftlite, Headbomb, Henning Makholm, Hughdbrown, Jayamohan, Justin W Smith, Koczy, Lantonov, Lingust, Mahahahaneapneap, Mcstrother, Michael Hardy, Miym, Naojoj, Nichtich, Not-just-yet, Oleg Alexandrov, PV=nRT, PerryTachett, Richwales, Rponamgi, Shreevatsa, Teimu.tm, The Anome, Timb, Undsoeweiter, User A1, Vrenator, Wshun, Zarvak, 64 anonymous edits

Stable roommates problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509631558> *Contributors:* David Eppstein, GregorB, Headbomb, JackSchmidt, Mcherm, Mindspin311, Miym, PipZandahalf, RobWIrving, Skier Dude, Squeaky201, TakuyaMurata, 15 anonymous edits

Permanent *Source:* <http://en.wikipedia.org/w/index.php?oldid=508568991> *Contributors:* AdjustShift, Alansohn, Altenmann, Andris, Army1987, AxelBoldt, Backslash Forwardslash, Bender2k14, BradHD, Bte99, Charles Matthews, Chris the speller, Creidieki, Cybercobra, Dcoetze, Draco flavus, El C, Emerson7, Esprit15d, EverettYou, Fuzzy, Galoubet, Garfl, Gdr, Giftlite, Gustavb, Headbomb, Irchans, Jim1138, Jmath666, Joachim Selke, Karmastan, Kilom691, Laudak, Lazylaces, Masnevets, Melelaswe, Michael Hardy, Mikhail Dvorkin, Mukadderat, Oleg Alexandrov, Peap, Populus, Pschemp, Rjwilmsi, Sbyrnes321, Shreevatsa, Sodin, Stemonitis, Thore Husfeldt, Twri, Vanish2, Wcherowi, Xev lexx, Xnn, 55 anonymous edits

Computing the permanent *Source:* <http://en.wikipedia.org/w/index.php?oldid=490884362> *Contributors:* Altenmann, Bender2k14, Circularargument, David Eppstein, Giftlite, Headbomb, Hermel, Kilom691, Laudak, Michael Hardy, Mukadderat, Ruziklan, Shreevatsa, Thore Husfeldt, Twri, Xezbeth, 11 anonymous edits

Maximum flow problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=493315152> *Contributors:* Alansohn, Altenmann, BenFrantzDale, Charles Matthews, Crystallina, CsfpWaiting, Cyhawk, Davepape, David Eppstein, Debamf, Giftlite, GregorB, Headbomb, Huttarl, Jackfork, Jleedev, Joao Meidanis, Joeplnl, LOL, Lambertch, Majestic-chimp, Mandarinaclementina, Materialscientist, Meld, Milad621, Minimac, Mtvc, Nick Number, Nils Grimsimo, Nrawat, Parthasarathy.kr, Paul August, Polisher of Cobwebs, Pugget, Rizzolo, Rkleckner, Svick, Teshasaposse, TempeteCoeur, Urod, Wikiklrs, WuTheFWasThat, X7q, ZeroOne, Zuphilip, 47 anonymous edits

Max-flow min-cut theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=505781776> *Contributors:* Aaron Schulz, Aashcrahn, Adam Zivner, Addshore, Aindeev, Ali Esfandiari, AxelBoldt, Booyabazooka, C S, Daiyuda, Darren Strash, David Eppstein, Dcoetzee, Delaszk, Douglas R. White, Dysprosia, ESkog, Eric119, Finfobia, Floridi, Gabn1, Gaius Cornelius, Geometry guy, Gifflite, Guof, Huttari, Jamelan, Jokes Free4Me, Kawaa, LOL, Luc4, MathMartin, Meekohi, Mgreenbe, Mhym, Michael Hardy, Nils Grimsmo, Oleg Alexandrov, Parthasarathy.kr, Paul August, Petter Strandmark, Qwertyus, Ratfox, Rizzolo, Rshin, Ruud Koot, Sergio01, Simon Lacoste-Julien, Snoyes, Spoon!, Stdazi, SteveJothen, Tchashasposse, Tomo, Ycl6, Yewang315, Zfeinst, 70 anonymous edits

Ford–Fulkerson algorithm for maximum flows *Source:* <http://en.wikipedia.org/w/index.php?oldid=493831978> *Contributors:* Alex Selby, Almi, Andreas Kaufmann, Arauzo, Babbage, Benbread, Beta16, BrotherE, Cburnett, Colossus13, CryptoDerk, Cyhawk, DFRussia, Dcoetzee, Dionyziz, Dopehead9, Ecb29, EdG1, Gandalf61, Gerel, Gifflite, Heineman, Hephaestos, Island Monkey, JanKuipers, Jaredwf, Knowledge-is-power, Leland McInnes, Liorma, Mad728, Mamling, Mange01, Manuel Anastacio, Mark1421, Michael Hardy, Nausher, Nils Grimsmo, Petter Strandmark, Poor Yorick, Prara, Queeg, Quiark, Randywombat, Riedel, Rrusin, Shashwat986, SiobhanHansa, Svick, Toncek, Treyphonuff, Tuna027, Zero0000, ملکی زادگان, 81 anonymous edits

Edmonds–Karp algorithm for maximum flows *Source:* <http://en.wikipedia.org/w/index.php?oldid=498074969> *Contributors:* Aaron Hurst, Amelio Vázquez, Balloonguy, Cburnett, Chopchopwhitey, Cosmi, Cquan, Darth Panda, Gifflite, Glrx, Hashproduct, Headbomb, Htmnssn, Jamelan, John of Reading, Katie5584, Kristjan Wager, Kubek15, LiuZhaoliang, Martani, Michael Hardy, Mihai Capotă, Niloofer pirooz, Nils Grimsmo, Nixdorff, Nkojuharov, Paradox, Pkrilin, Poor Yorick, Pt, Pugget, RJFJR, Sesse, Simonfl, TPReal, Zero0000, قلچی زادگان, 65 anonymous edits

Dinic's algorithm for maximum flows *Source:* <http://en.wikipedia.org/w/index.php?oldid=498637717> *Contributors:* Andreas Kaufmann, Bbi5291, Evergrey, Gawi, Gifflite, Headbomb, Michael Hardy, NuclearWarfare, Octahedron80, Omnipediadista, R'n'B, Rjwilmsi, Sun Creator, Tchashasposse, Urod, X7q, 10 anonymous edits

Push-relabel maximum flow algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=497225060> *Contributors:* Andreas Kaufmann, Babak.barati, Damian Yerrick, Debamf, Dylan Thurston, Headbomb, JanKuipers, Nils Grimsmo, Rich Farmbrough, RussBlau, Slon02, Stefan Knauf, Sunderland06, Sverigekullen, Svick, Wmayer, 33 anonymous edits

Closure problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=509364847> *Contributors:* Ak1990, Andreas Kaufmann, ChrisGaultieri, Connor mezza, David Eppstein, Erkan Yilmaz, Faridani, Headbomb, Juanpabloaj, LilHelpa, Malcolma, Michael Hardy, Nihola, RJFJR, Wilhelmina Will, 9 anonymous edits

Minimum cost flow problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=417264342> *Contributors:* Adamarthurryan, Alexandersachawolff, Bwsulliv, Headbomb, Hs1771, Kovacspter84, Michael Hardy, Miym, Nils Grimsmo, Ruud Koot, 16 anonymous edits

Planar graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=50879848> *Contributors:* A3 nm, ABCD, Afil, Ahoerstemeier, Alberto da Calvairate, AlexCornejo, Alexb@cut-the-knot.com, Altenmann, Arunachalam.t, AstroHurricane001, AxelBoldt, Bdesham, Bender2k14, Birurlobet, Bobo192, Booyabazooka, Brian0918, Burn, Cameron.walsh, Charles Matthews, Chris Pressey, Conversion script, Corti, Cutelyaware, DFRussia, DHN, David Eppstein, Dbenbenn, Dcoetzee, Debamf, Dilip rajeev, Disavian, Dominus, Don4of4, Doradus, Douglike, Dpv, Drebbien, Dysprosia, Edemaine, El Roih, Eric119, Everyking, FreplySpang, Ged.R, Gifflite, Graph Theory page blanker, Guillaume Damiand, Hadaso, Hat600, Headbomb, Heron, HorsePunchKid, Jamie King, Jaredwf, Jason Quinn, Jfraser, Jillbones, JohnBoyerPhd, Joriki, Jorunn, Joyous!, Justin W Smith, Kok090, Kv75, LC, Lenoxus, Libertyernie2, Life of Riley, Linas, Lyonsam, M4gnoun0, Magister Mathematicae, ManRabe, Marozols, Mastergreg82, MathMartin, Mav, McCart42, McKay, Mellum, Michael Hardy, NatusRoma, Nine Tail Fox, Nonenmac, Nsevs, Oleg Alexandrov, Omnipediadista, Paul August, Paul Pogonyshhev, Phs, R'n'B, Rjwilmsi, Romann, Rp, Scarboy, Shellreef, Snigbrook, Some jerk on the Internet, Sun Creator, Svdb, Tamfang, Taral, Taxipom, Taxman, Thewayforward, Timwi, Tomo, Trevor MacInnis, Urdutext, Vinayak Pathak, Wantnot, Wavelength, Yecril, Zanetu, Zero0000, گل، 95 anonymous edits

Dual graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=497840347> *Contributors:* Altenmann, Andreas Kaufmann, Arthena, Berland, Charles Matthews, David Eppstein, Doradus, Ewlyahooocom, Futurebird, Gifflite, Hadaso, Headbomb, Jearroll, Kirelagin, Kmote, Magister Mathematicae, McKay, Michael Slone, MichaelMcGuffin, Mormegil, PEJO89, Prabhakar97, R'n'B, Rp, Sangak, Sasquatch, Suffusion of Yellow, Taxipom, Tompw, Tomruen, Vadmium, Wangzhaoguang, 23 anonymous edits

Fáry's theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=507625645> *Contributors:* Altenmann, Andreas Kaufmann, Brad7777, David Eppstein, El Roih, Gene Nygaard, Gifflite, Headbomb, Jsondow, Mhym, Michael Hardy, Miym, OdedSchramm, Oleg Alexandrov, RDBury, RobinK, SaschaWolff, Taxipom, VladimirReshetnikov, 3 anonymous edits

Steinitz's theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=505778986> *Contributors:* Agaffss, Akriasas, Andreschulz, Brad7777, David Eppstein, Gifflite, Headbomb, Kbuchin, Kiefer.Wolfowitz, Michael Hardy, Terrek, Twri, 2 anonymous edits

Planarity testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=484886736> *Contributors:* A5b, DFRussia, David Eppstein, Dcoetze, Edward, HairyFotr, Michael Hardy, R'n'B, RobinK, SeniorInt, Svick, Taxipom, 14 anonymous edits

Fraysséix–Rosenstiehl's planarity criterion *Source:* <http://en.wikipedia.org/w/index.php?oldid=504633492> *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=508073646> *Contributors:* A.bit, Aaronzat, Almit39, Altenmann, AnakngAraw, Andreas Kaufmann, Anonymous Dissident, ArthurDenture, Bobcat64, Charles Matthews, Concubine119, Dardasvata, David Eppstein, Dbenbenn, Dysprosia, Euphilos, Experiment123, FirefoxRocks, Foo123, Fred Bradstadt, Fredrik, Gbrose85, Gifflite, Gregbard, HaeB, Harrigan, Headbomb, Hintss, Jaredwf, Jldwig, Jon Awbrey, JonHarder, Justin W Smith, Karada, Kuru, Liberatus, MForster, Manuscript, Martarius, Mdd, Mgreenbe, MichaelMcGuffin, Michele.dallachiesa, Osric, Pintman, Rcaawsey, Rholton, Roland wiese, Rp, Shwoodside, ShelfSkewed, Some jerk on the Internet, Stephan Leeds, StephenFerg, Sverdrup, Taxipom, Thoughtpuzzle, Tomo, Vonkje, Wavelength, ZweiOhren, 96 anonymous edits

Force-based graph drawing algorithms *Source:* <http://en.wikipedia.org/w/index.php?oldid=509659561> *Contributors:* Adam Schloss, Altenmann, Andreas Kaufmann, AnnGabrieli, ArthurDenture, Bkell, Boneill, Charles Matthews, Classicaelecon, David Eppstein, Dcoetze, Denkar, Drphilharmonic, Dtunkelang, Efficacious, Fryed-peach, G. Moore, Gak, Gareth McCaughran, Headbomb, Hypotroph, Ivan Kickir, Ixfd64, Jldwig, JonHarder, Justin W Smith, Kku, Kostmo, Mahak library, McGeddon, Michael Hardy, MrBlueSky, Nickaschenbach, O1ive, Oleg Alexandrov, Parakkum, Resprinter123, RodrigoAiEs, Roenbaeck, Ronz, Rswarbrick, Ruud Koot, Samwass, Scftn, SteveCoast, Stevecooperorg, Tgdwyer, Thorwald, Thue, Yifanhu, 58 anonymous edits

Graph embedding *Source:* <http://en.wikipedia.org/w/index.php?oldid=490889197> *Contributors:* A.A.Graff, Altenmann, Bender2k14, Bentsm, Birutorul, CBM, CRGreathouse, Chris the speller, DFRussia, David Eppstein, Deltaway, Eassin, Ferritecore, Flammifer, Genusfour, Gifflite, Headbomb, JaGa, PieMan.EXE, Taxipom, Twri, Zaslav, 13 anonymous edits

Application: Sociograms *Source:* <http://en.wikipedia.org/w/index.php?oldid=488831288> *Contributors:* Alansohn, AndreniW, Bendistraw, David Eppstein, Doczilla, Egmontaz, HaeB, Isarra, J.delanoy, Joekoz451, Marc Bernardin, Mdd, Mlaboria, Mysisd, Plrk, Tgr, TheGroble, Wykis, 29 anonymous edits

Application: Concept maps *Source:* <http://en.wikipedia.org/w/index.php?oldid=503191708> *Contributors:* A. B., AP Shinobi, Alan Pascoe, Alexander.stohr, Alfredotifi, Allan McInnes, Altenmann, Andyjsmith, AntiVan, Argey, Arthena, AubreyEllenShomo, AugPi, B9 hummingbird hovering, BenFrantzDale, Bender235, Bertolt, Blbowen108, Bobo192, Brian.m.moon, Bspierre, Buddhipriya, Carbo1200, Chan Bok, Chance74, Charlettehamilton, ChemGardener, Chowbok, Closedmouth, Cmdrjameson, Conradwki, Cornellrockey, Crusoe2, D.Right, DVdm, DanMS, David Eppstein, Deflago, Derek Andrews, Dezignr, Digitalis3, Disavian, Discospinster, Dmccreary, DrDooBig, Dstringham, EBlack, EbenezerLePage, El C, Eliazar, Enzo Aquarius, Epbr123, ErikCincinati, Escape Orbit, Estevaoeai, Excial, Flemina, Floorschtein, FreplySpang, Friviere, Gaius Cornelius, Gdm, Gecko, George100, Gregbard, Ground, Hallenrm, Hbent, Headbomb, Henk Daalder, Heron, Hilen, IceCreamAntisocial, Iridescent, J.delanoy, Jamelan, Jensgb, Jm34harvey, Jmabel, Jtnell, Jupersthr, KYPark, Karl-Henner, Khalid hassani, Kku, Klimov, KnightRider, Kollision, Konetidy, Kuru, LarryQ, Lheuer, Ligulem, Lindsay658, MBSaddoris, Majorly, Marybjt, Mdd, Meena74, Mel Etitis, Michael Hardy, Michig, Micru, Modify, Mwanner, NRaja, Nagarjunag, Nagy, Nesbit, NextD Journal, Nigholith, Nikevich, Oceans and oceans, Ohnoitsjamie, Orionus, Oxymoron83, Pashilkar, Pavel Vozenilek, Perigean, Pharaoh of the Wizards, Piano non troppo, Picksg, Pwilkins, Quercusrobur, Quiddity, RJBurkhart, Ray Oaks, Rcarvajal, Reinyday, RichardF, Ronz, Rurus, SEWilco, Saxifrage, Sbwoodside, Sepreece, Sherbrooke, Spdegabrielle, Spookfish, Spot2112, Stephan Leeds, Stephen0928, Tedickey, Tennismenace88, The Anome, The ed17, The web, Tide rolls, Tobias Bergemann, Tommy2010, Tomo, Truman Burbank, Umer992, Viewwood40, Vladimir B., Vrenator, William Avery, WindOwl, Wxidea, 284 anonymous edits

Interval graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=492075847> *Contributors:* Arvind, David Eppstein, Dysprosia, Headbomb, Jaredwf, Justin W Smith, Kiefer.Wolfowitz, Lyonsam, M11101, Mastergreg82, Mgreenbe, Miym, Odwl, Ott2, PaulTanenbaum, Polymedes, Rich Farmbrough, Ruvald, Smyle 6, Terrykel, Tgr, TonyW, Twri, 19 anonymous edits

Chordal graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=505683633> *Contributors:* Altenmann, Bkell, Danielx, David Eppstein, Domination989, Ged.R, Headbomb, Kilom691, Landon1980, LokiClock, Lyonsam, Martygo, Michael Slone, MichaelMcGuffin, Oliphant, Peter Kwok, Rich Farmbrough, Someguy1221, Tizio, Zaslav, 29 anonymous edits

Perfect graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=506812725> *Contributors:* Adking80, After Midnight, BeteNoir, CXCV, Chaney, Charles Matthews, Claygate, Colorajo, David Eppstein, Dcoetze, Freiert, Gifflite, Columbic, Headbomb, Igorpak, Jamie King, Jaredwf, Michael Hardy, Mon4, Odwl, Peter Kwok, Phs, Qutezuce, Sundar, Svick, Taxipom, Teorth, Tizio, Vince Vatter, Xnn, 23 anonymous edits

Intersection graph Source: <http://en.wikipedia.org/w/index.php?oldid=434655468> Contributors: Altenmann, David Eppstein, Dcoetze, Gandalf61, Gato ocioso, Justin W Smith, Kope, Mcoury, Michael Hardy, OdedSchramm, PaulTanenbaum, Rocchini, Salix alba, Tangi-tamma, Twri, Zaslav, 11 anonymous edits

Unit disk graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=460714794> *Contributors:* Altenmann, Avb, Davepape, David Eppstein, Igorpak, Ifxld64, Lyonsam, Pbrandao, Rb82, TalesHeimfarth, Twri, 3 anonymous edits

Line graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=505685088> *Contributors:* 2D, Abdull, Ahruman, Ale_jrb, Allens, Altenmann, Archdukefranz, Arman Cagle, At-par, Avraham, Booyabazooka, Braintrain0000, CBM, Calle, Capicorn42, Confession0791, DVD_R_W, Danski14, David_Eppstein, DavidCary, Dbenbenn, Docetzeer, Discospinster, Dominus, DropZone, Dysprosia, El_Roith, Ebpr123, Gelingvistoj, Giflste, Graham5571, Gscshoyru, Guardian of Light, Hayhway0147, Headbomb, Hydrogen Iodide, Iain99, Ilmari_Karonen, J.delanoy, Jaredwf, Kitresaiba, LiLiHelpa, Lostinareforest, Lyonsanon, MathMartin, Meaghan, Melaen, Merlion444, Miym, Oatmealcookiemon, Oleg_Alexandrov, OllieFury, Omnipaediista, Onecanadasquarebishopsgate, PMajer, Paul_August, Peter_Kwok, Pinethicket, Qef, Reyk, SJP, Seba5618, Tangi-tamma, TedPavlic, Twri, Vary, Wiki13, YouWillBeAssimilated, Zaslav, 109 anonymous edits

Claw-free graph Source: <http://en.wikipedia.org/w/index.php?oldid=50613763> Contributors: Adking80, Cdills, David Eppstein, Drange net, GregorB, Headbomb, 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=507935511> *Contributors:* BD2412, Brewbrewer, CharlotteWebb, David Eppstein, Fratrep, Gambette, Gifflite, Howard McCay, Igorpak, Justin W Smith, Kilom691, Philippe Giabbanelli, Rjwilmsi, 5 anonymous edits

Graph isomorphism Source: <http://en.wikipedia.org/w/index.php?oldid=507794812> Contributors: AdamAtlas, Aeons, Aerosprite, Altenmann, Ansatz, Arthur Rubin, AxelBoldt, Bkell, Blaisorblade, Blake-, Booyabazooka, CBM, Citrus538, Davepape, David Eppstein, Dcoetzee, El Roih, EmilJ, Giftlite, Headbomb, Igor Yalovecky, Iotatau, Itub, J, Finkelstein, Jamelan, Jason Quinn, Jitse Niesen, Jldwight, Justin W Smith, KSmrq, Kingfishr, Linas, Loopology, MathMartin, Maxal, Michael Hardy, Michael Sloane, MoraSique, Nemnkim, Nibbio84, Oliphaunt, PV=nRT, Paul August, PierreCA22, Puzne, Ricardo Ferreira de Oliveira, Rich Farmbrough, Robert Illes, Shreevatsa, Sleepinj, SuperMidget, ThomasP1985, Tim32, Trjumpet, Twri, Vegasprof, Verbal, 50 anonymous edits

Graph isomorphism problem Source: <http://en.wikipedia.org/w/index.php?oldid=499957178> Contributors: A.A.Graff, Akiezun, Bender2k14, Benja, Birutorul, Charles Matthews, David Eppstein, Dcoetzee, Dekart, Delirium, Dylan Thurston, Emilij, EsfirK, Headbomb, Iotatau, J_Finkelstein, Jamelan, Jam67, JoergenB, Joriki, Justin W Smith, Laurusnobilis, LouScheffer, Mart071, Maxal, Melchoir, Myhm, Michael Hardy, Miym, Neilc, Ott2, Pascal.Tesson, RobinK, Root4(one), Rsarge, Rued Koot, Shreevatsa, Slawekb, Stux, Tim32, Tomo, Twri, Verbal, Kanevsky, 23 anonymous edits

Graph canonization Source: <http://en.wikipedia.org/w/index.php?oldid=491735587> Contributors: Altenmann, BenFrantzDale, Blaisorblade, David Eppstein, Gifflite, Headbomb, Michael Hardy, Oudit, Tim32, 3 anonymous edits

Subgraph isomorphism problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=495965222> *Contributors:* A3 nm, Andreas Kaufmann, Brian0918, BrotherE, Centrx, Citrus538, Cowsmilkd, D6, DaveTheRed, David Eppstein, Dcoetelet, Decay7, Dysprosia, EmijUl, Fabior1984, Graph Theory page blanker, Graue, HanielBarbosaa, Hasanjamili, Joerg Kurt Wegner, Justin W Smith, MathMartin, Mazi, Mellum, Orgueil, PieFlu Oink, R'n'B, RobinK, That Guv, From That Show!, Thv, Tim Goedwijn, 23 anonymous edits

Color-coding Source: <http://en.wikipedia.org/w/index.php?oldid=482228147> Contributors: Bwilkins, David Eppstein, Dhwuang, FairFare, Frap, Hermel, Jenny Lam, Michael Hardy, Pullnull123, R'n'B, SwisterTwister, Swnb, 7 anonymous edits

Induced subgraph isomorphism problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=508894153> *Contributors:* 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=495965676> Contributors: A3 nm, Andreas Kaufmann, DVanDyck, Joerg Kurt Warner, Juander, Michael Hardy, Modify, P's'B, RobinK, Srm, 6 anonymous edits

Graph partition *Source:* <http://en.wikipedia.org/w/index.php?oldid=504784669> *Contributors:* Aydin.Khatamnejad, David Eppstein, Eboman, Headbomb, Jheiv, Jitse Niesen, JonDePlum, LutzSchaffter, Matsuura, 82, MichaelHudec, Mm, Neelambari, Pacholski, Pimiloumi, RehikK, RuudKoet, Shreevatsa, Suhiraj, Thibaut, Tigris, Wouter, Zeta, Zorgatous, 12 anonymous users

Kernighan–Lin algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=491681572> *Contributors:* CBM, Headbomb, Johlo, Luoq, Marmale, Nanobear, Poochy, RobinK, Ruud Koot,

Tree decomposition *Source:* <http://en.wikipedia.org/w/index.php?oldid=505220045> *Contributors:* Alexdow, Andreas Kaufmann, Bender2k14, Bodlaender, Brutaldeluxe, Dagh, David Eppstein

Branch-decomposition Source: <http://en.wikipedia.org/w/index.php?oldid=509695116> Contributors: Andreas Kaufmann, David Eppstein, Hiiiiiiiiiiiiiiiiiiiiiiii, Jitse Niesen, LeadSongDog, Tizito, Toonim, Trepleeks, Twri, Wpl103, Wzha0555, Xmn, 22 anonymous edits

Path decomposition *Source:* <http://en.wikipedia.org/w/index.php?oldid=505293786> *Contributors:* David Eppstein, Eran.a, Headbomb, Hermel, Jitse Niesen, Justin W Smith, R'n'B, Rjw

Planar separator theorem *Source:* <http://en.wikipedia.org/w/index.php?oldid=505781940> *Contributors:* Andreas Kaufmann, AntoniosAntoniadis, David Eppstein, EoGuy, Giftlite, Headbomb,

Graph minors. Source: <http://en.wikipedia.org/w/index.php?oldid=507866406>. Contributors: A1kmm, Adoarns, AxelBoldt, Bethnim, Cmcq, David Eppstein, Dbenbenn, Dcoetzee, Dkostic

Caenouelle's theorem. Survey: https://en.wikipedia.org/w/index.php?title=Caenouelle%27s_theorem&oldid=402218090. Consulted on David Eppstein, Giftfile, History 2007.

Cyan, David Eppstein, Dbenbenn, Dcoetzee, Dfeldmann, Dominus, FvdP, Giftlite, Glasser, Headbomb, Jon Awbrey, Justin W Smith, MSGJ, Mandarax, Melchoir, Michael Hardy, PhS, Populus, Psychonaut, Quixplusone, R.e.b., R.e.s., RDBury, RobinK, Szftang, Svick, Tizio, Twri, Udufruduhu, Unzerlegbarkeit, Wzhao553, Zaslav, 16 anonymous edits

Biunidimensionality. Source: <http://en.wikipedia.org/w/index.php?oldid=500629052>. Contributors: David Eppstein, Diwas, Gosha Igosha, Headbotino, Joaquim000, Rudd Root, Thon, 5 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

Image:Konigsberg bridges.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Konigsberg_bridges.png *License:* GNU Free Documentation License *Contributors:* Bogdan Giușă

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, 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:6n-graf.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:6n-graf.svg> *License:* Public Domain *Contributors:* User:AzaToth

File:Multigraph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Multigraph.svg> *License:* Public Domain *Contributors:* Arthena

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: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: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:DirectedDegrees.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:DirectedDegrees.svg> *License:* GNU Free Documentation License *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)

Image:Speakerlink.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Speakerlink.svg> *License:* Creative Commons Attribution 3.0 *Contributors:* Woodstone. Original uploader was Woodstone at en.wikipedia

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:6n-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:* Lepida

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:* Lepida

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-\$4-Plan.svg; Fool (talk) derivative work: Lepida (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:* Lepida

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:* User:Stimp

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:* BenRG

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 Bmatheeny 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, Regnar0n

Image:GermanyBFS.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:GermanyBFS.svg> *License:* Public Domain *Contributors:* Regnar0n

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, EugeneZelenko, 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.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:SPQR_tree.svg *License:* Public Domain *Contributors:* David Eppstein

File:Min cut.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Min_cut.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Daiyuda

File:The process of contraction.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:The_process_of_contraction.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Daiyuda

File:Execution of inner while loop.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Execution_of_inner_while_loop.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Daiyuda

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: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 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:Dijksta Anim.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Dijksta_Anim.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: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:First_graph.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:First_graph.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Robertoocfc1

File:Speaker Icon.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Speaker_Icon.svg *License:* Public Domain *Contributors:* Blast, G.Hagedorn, Mobius, Tehdog, 2 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:AstarExample.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:AstarExample.gif> *License:* Public Domain *Contributors:* Naugtrur

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:Longest path example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Longest_path_example.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Magog the Ogre, Vacio

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

Image:Graph betweenness.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_betweenness.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *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.

Image: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:Msp1.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Msp1.jpg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* fungzewai

File:Msp-the-cut.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Msp-the-cut.jpg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* fungzewai

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: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:* Purpy Purple

Image:Prim Algorithm 0.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_0.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel

Image:Prim Algorithm 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_1.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_2.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_3.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_4.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_5.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 6.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_6.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 7.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_7.svg *License:* Creative Commons Attribution-Sharealike 3.0,2,5,2,0,1,0 *Contributors:* Alexander Drichel, Stefan Birkner

File:Prim-algorithm-animation-2.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Prim-algorithm-animation-2.gif> *License:* Creative Commons Attribution-Sharealike 3.0
Contributors: fungzewai

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

Image:Chamber.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Chamber.png> *License:* Public Domain *Contributors:* Simplex

Image:Chamber division.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_division.png *License:* Public Domain *Contributors:* Simplex

Image:Chamber divided.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_divided.png *License:* Public Domain *Contributors:* Simplex

Image:Chamber subdivision.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_subdivision.png *License:* Public Domain *Contributors:* Simplex

Image:Chamber finished.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Chamber_finished.png *License:* Public Domain *Contributors:* Simplex

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:* Dcoetze, 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, Dcoetze, Lipedia

Image:Graph with all three-colourings.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Graph_with_all_three-colourings.svg *License:* GNU Free Documentation License *Contributors:* User:Arbor, User:Booyabazooka

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, Dcoetze

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: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, 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

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:Nojhan

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:* Csfypwaiting

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: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

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

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

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, 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:* -

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

Image:Dualgraphs.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Dualgraphs.png> *License:* Creative Commons Attribution-Sharealike 2.0 *Contributors:* Dcoetzee, Maksim, Perhelion, Petr Dlouhý, 1 anonymous edits

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:* User:Chris 73

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

Image:Social-network.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Social-network.svg> *License:* Public Domain *Contributors:* User:Wykis

Image:Conceptmap.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Conceptmap.gif> *License:* GNU Free Documentation License *Contributors:* Vicwood40

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-ShareAlike 3.0 Unported *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: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 Braintrain0000 at en.wikipedia

File:LineGraphExampleA.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:LineGraphExampleA.svg> *License:* Public Domain *Contributors:* Ilmari Karonen ()

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:Summer claw-free matching.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Summer_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:3x3 grid graph haven.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:3x3_grid_graph_haven.svg *License:* Creative Commons Zero *Contributors:* User:David Eppstein

Image:Partial 3-tree forbidden minors.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Partial_3-tree_forbidden_minors.svg *License:* Public Domain *Contributors:* David Eppstein

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 Unported
[//creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)