15

Graph Problems: Polynomial-Time

Algorithmic graph problems constitute approximately one third of the problems in this catalog. Problems from other sections could have been formulated equally well in terms of graphs, such as bandwidth minimization and finite-state automata optimization. Identifying the name of a graph-theoretic invariant or problem is one of the primary skills of a good algorist. Indeed, the catalog will tell you exactly how to proceed as soon as you figure out your particular problem's name.

In this section, we deal only with problems for which there are efficient algorithms to solve them. As there is often more than one way to model a given application, it makes sense to look here before proceeding on to the harder formulations.

The algorithms presented here have running times that grow polynomially with the size of the graph. We adopt throughout the convention that n refers to the number of vertices in a graph, while m is the number of edges.

Graphs are often best understood as drawings. Many interesting graph properties follow from the nature of a particular type of drawing, such as planar graphs. Thus, we also discuss algorithms for drawing graphs, trees, and planar graphs.

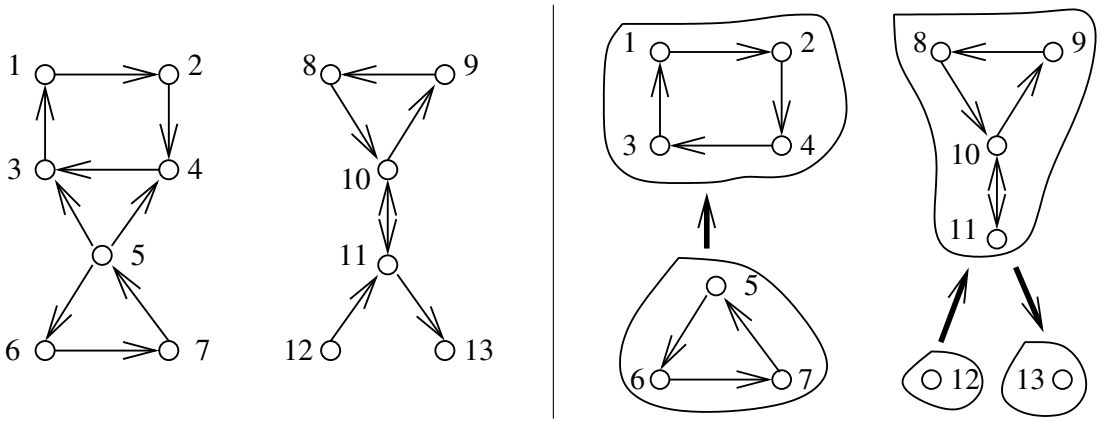
Most advanced graph algorithms are difficult to program. However, good implementations are available if you know where to look. The best general sources include LEDA [MN99] and the Boost Graph Library [SLL02]. However, better special-purpose codes exist for many problems.

See the *Handbook of Graph Algorithms* [TNX08] for up-to-date surveys on all areas of graph algorithms. Other excellent surveys include van Leeuwen [vL90a], and several chapters in Atallah [Ata98]. Books specializing in graph algorithms include:

- *Sedgewick* [Sed98] – The graph algorithms volume of this algorithms text provides a comprehensive but gentle introduction to the field.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_15,

- *Ahuja, Magnanti, and Orlin* [AMO93] – While purporting to be a book on network flows, it covers the gamut of graph algorithms with an emphasis on operations research. Strongly recommended.
- *Gibbons* [Gib85] – A good book on graph algorithms, including planarity testing, matching, and Eulerian/Hamiltonian cycles, as well as more elementary topics.
- *Even* [Eve79a] – An older but still respected advanced text on graph algorithms, with a particularly thorough treatment of planarity-testing algorithms.



INPUT OUTPUT

15.1 Connected Components

Input description: A directed or undirected graph G.

Problem description: Identify the different pieces or components of G, where vertices x and y are members of different components if no path exists from x to y in G.

Discussion: The connected components of a graph represent, in grossest terms, the pieces of the graph. Two vertices are in the same component of G if and only if there exists some path between them.

Finding connected components is at the heart of many graph applications. For example, consider the problem of identifying natural clusters in a set of items. We represent each item by a vertex and add an edge between each pair of items deemed "similar." The connected components of this graph correspond to different classes of items.

Testing whether a graph is connected is an essential preprocessing step for every graph algorithm. Subtle, hard-to-detect bugs often result when an algorithm is run only on one component of a disconnected graph. Connectivity tests are so quick and easy that you should always verify the integrity of your input graph, even when you know for certain that it *has* to be connected.

Testing the connectivity of any undirected graph is a job for either depth-first or breadth-first search, as discussed in Section 5 (page 145). Which one you choose doesn't really matter. Both traversals initialize the *component-number* field for each vertex to 0, and then start the search for component 1 from vertex v_1. As each vertex is discovered, the value of this field is set to the current component

number. When the initial traversal ends, the component number is incremented, and the search begins again from the first vertex whose *component-number* remains 0. Properly implemented using adjacency lists (as is done in Section 5.7.1 (page 166)) this runs in $O(n+m)$ time.

Other notions of connectivity also arise in practice:

- *What if my graph is directed?* – There are two distinct notions of connected components for directed graphs. A directed graph is *weakly connected* if it would be connected by ignoring the direction of edges. Thus, a weakly connected graph consists of a single piece. A directed graph is *strongly connected* if there is a directed path between every pair of vertices. This distinction is best made clear by considering the network of one- and two-way streets in any city. The network is strongly connected if it is possible to drive legally between every two positions. The network is weakly connected when it is possible to drive legally or *illegally* between every two positions. The network is disconnected if there is no possible way to drive from a to b.

  Weakly and strongly connected components define unique partitions of the vertices. The output figure at the beginning of this section illustrates a directed graph consisting of two weakly or five strongly-connected components (also called *blocks* of G).

  Testing whether a directed graph is weakly connected can be done easily in linear time. Simply turn all edges of G into undirected edges and use the DFS-based connected components algorithm described previously. Tests for strong connectivity are somewhat more complicated. The simplest linear-time algorithm performs a search from some vertex v to demonstrate that the entire graph is reachable from v. We then construct a graph G' where we reverse all the edges of G. A traversal of G' from v suffices to decide whether all vertices of G can reach v. Graph G is strongly connected iff all vertices can reach, and are reachable, from v.

  All the strongly connected components of G can be extracted in linear time using more sophisticated DFS-based algorithms. A generalization of the above "two-DFS" idea is deceptively easy to program but somewhat subtle to understand exactly why it works:

  1. Perform a DFS, starting from an arbitrary vertex in G, and labeling each vertex in order of its completion (not discovery).
  2. Reverse the direction of each edge in G, yielding G'.
  3. Perform a DFS of G', starting from the highest numbered vertex in G. If this search does not completely traverse G', continue with the highest numbered unvisited vertex.
  4. Each DFS tree created in Step 3 is a strongly connected component.

My implementation of a single-pass algorithm appears in Section 5.10.2 (page 181). In either case, it is probably easier to start from an existing implementation than a textbook description.

- *What is the weakest point in my graph/network?* – A chain is only as strong as its weakest link. Losing one or more internal links causes a chain to become disconnected. The *connectivity* of graphs measures their strength—how many edges or vertices must be removed to disconnect it. Connectivity is an essential invariant for network design and other structural problems.

  Algorithmic connectivity problems are discussed in Section 15.8 (page 505). In particular, *biconnected components* are pieces of the graph that result from cutting the edges incident on a single vertex. All biconnected components can be found in linear time using DFS. See Section 5.9.2 (page 173) for an implementation of this algorithm. Vertices whose deletion disconnects the graph belong to more than one biconnected component, whose edges are uniquely partitioned by components.

- *Is the graph a tree? How can I find a cycle if one exists?* – The problem of cycle identification often arises, particularly with respect to directed graphs. For example, testing if a sequence of conditions can deadlock often reduces to cycle detection. If I am waiting for Fred, and Fred is waiting for Mary, and Mary is waiting for me, there is a cycle and we are all deadlocked.

  For undirected graphs, the analogous problem is tree identification. A tree is, by definition, an undirected, connected graph without any cycles. As described above, a depth-first search can be used to test whether it is connected. If the graph is connected and has $n - 1$ edges for n vertices, it is a tree.

  Depth-first search can be used to find cycles in both directed and undirected graphs. Whenever we encounter a back edge in our DFS—i.e. , an edge to an ancestor vertex in the DFS tree—the back edge and the tree together define a directed cycle. No other such cycle can exist in a directed graph. Directed graphs without cycles are called DAGs (directed acyclic graphs). Topological sorting (see Section 15.2 (page 481)) is the fundamental operation on DAGs.

Implementations: The graph data structure implementations of Section 12.4 (page 381) all include implementations of BFS/DFS, and hence connectivity testing to at least some degree. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) provides implementations of connected components and strongly connected components. LEDA (see Section 19.1.1 (page 658)) provides these plus biconnected and triconnected components, breadth-first and depth-first search, connected components and strongly connected components, all in C++.

With respect to Java, *JUNG* (*http://jung.sourceforge.net/*) also provides biconnected component algorithms, while JGraphT (*http://jgrapht.sourceforge.net/*) does strongly connected components.

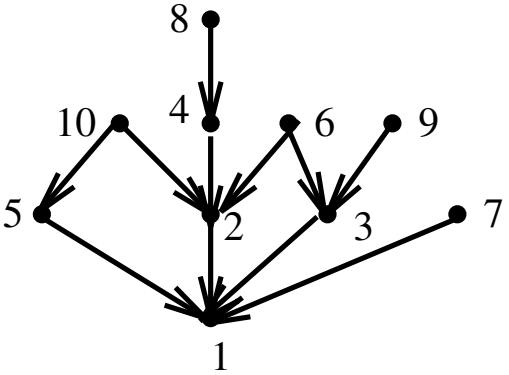
My (biased) preference for C language implementations of all basic graph connectivity algorithms. including strongly connected components and biconnected components, is the library associated with this book. See Section 19.1.10 (page 661) for details.

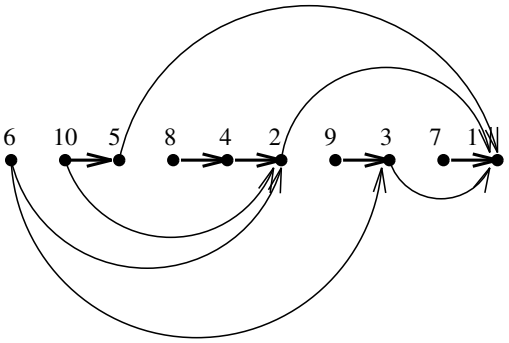
Notes: Depth-first search was first used to find paths out of mazes, and dates back to the nineteenth century [Luc91, Tar95]. Breadth-first search was first reported to find the shortest path by Moore in 1957 [Moo59].

Hopcroft and Tarjan [HT73b, Tar72] established depth-first search as a fundamental technique for efficient graph algorithms. Expositions on depth-first and breadth-first search appear in every book discussing graph algorithms, with [CLRS01] perhaps the most thorough description available.

The first linear-time algorithm for strongly connected components is due to Tarjan [Tar72], with expositions including [BvG99, Eve79a, Man89]. Another algorithm—simpler to program and slicker—for finding strongly connected components is due to Sharir and Kosaraju. Good expositions of this algorithm appear in [AHU83, CLRS01]. Cheriyan and Mehlhorn [CM96] propose improved algorithms for certain problems on dense graphs, including strongly connected components.

Related Problems: Edge-vertex connectivity (see page 505), shortest path (see page 489).





INPUT OUTPUT

15.2 Topological Sorting

Input description: A directed acyclic graph $G = (V, E)$, also known as a *partial order* or *poset*.

Problem description: Find a linear ordering of the vertices of V such that for each edge $(i, j) \in E$, vertex i is to the left of vertex j.

Discussion: Topological sorting arises as a subproblem in most algorithms on directed acyclic graphs. Topological sorting orders the vertices and edges of a DAG in a simple and consistent way and hence plays the same role for DAGs that a depth-first search does for general graphs.

Topological sorting can be used to schedule tasks under precedence constraints. Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. These precedence constraints form a directed acyclic graph, and any topological sort (also known as a *linear extension*) defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.

Three important facts about topological sorting are

1. *Only* DAGs can be topologically sorted, since any directed cycle provides an inherent contradiction to a linear order of tasks.

2. *Every* DAG can be topologically sorted, so there must always be at least one schedule for any reasonable precedence constraints among jobs.

3. DAGs can often be topologically sorted in many different ways, especially when there are few constraints. Consider n unconstrained jobs. Any of the $n!$ permutations of the jobs constitutes a valid topological ordering.

The conceptually simplest linear-time algorithm for topological sorting performs a depth-first search of the DAG to identify the complete set of *source vertices*, where source vertices are vertices of in-degree zero. At least one such source must exist in any DAG. Source vertices can appear at the start of any schedule without violating any constraints. Deleting all the outgoing edges of these source vertices will create new source vertices, which can then sit comfortably to the immediate right of the first set. We repeat until all vertices are accounted for. A modest amount of care with data structures (adjacency lists and queues) is sufficient to make this run in $O(n + m)$ time.

An alternate algorithm makes use of the observation that ordering the vertices in terms of decreasing DFS finishing time yields a linear extension. An implementation of this algorithm with an argument for correctness is given in Section 5.10.1 (page 179).

Two special considerations with respect to topological sorting are:

- *What if I need all the linear extensions, instead of just one of them?* – In certain applications, it is important to construct *all* linear extensions of a DAG. Beware, because the number of linear extensions can grow exponentially in the size of the graph. Even the problem of counting the number of linear extensions is NP-hard.

  Algorithms for listing all linear extensions in a DAG are based on backtracking. They build all possible orderings from left to right, where each of the in-degree zero vertices are candidates for the next vertex. The outgoing edges from the selected vertex are deleted before moving on. An optimal algorithm for listing (or counting) linear extensions is discussed in the notes.

  Algorithms for constructing random linear extensions start from an arbitrary linear extension. We then repeatedly sample pairs of vertices. These are exchanged if the resulting permutation remains a topological ordering. This results in a random linear extension given enough random samples. See the Notes section for details.

- *What if your graph is not acyclic?* – When a set of constraints contains inherent contradictions, the natural problem becomes removing the smallest set of items that eliminates all inconsistencies. The sets of offending jobs (vertices) or constraints (edges) whose deletion leaves a DAG are known as the *feedback vertex set* and the *feedback arc set*, respectively. They are discussed in Section 16.11 (page 559). Unfortunately, both problems are NP-complete.

  Since the topological sorting algorithm gets stuck as soon as it identifies a vertex on a directed cycle, we can delete the offending edge or vertex and continue. This quick-and-dirty heuristic will eventually leave a DAG, but might delete more things than necessary. Section 9.10.3 (page 348) describes a better approach to the problem.

Implementations: Essentially all the graph data structure implementations of Section 12.4 (page 381) include implementations of topological sorting. This means the Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) and LEDA (see Section 19.1.1 (page 658)) for C++. For Java, the *Data Structures Library in Java* (JDSL) (*http://www.jdsl.org/*) includes a special routine to compute a unit-weighted topological numbering. Also check out JGraphT (*http://jgrapht.sourceforge.net/*).

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) provides C language programs to generate linear extensions in both lexicographic and Gray code orders, as well as count them. An interactive interface is also provided.

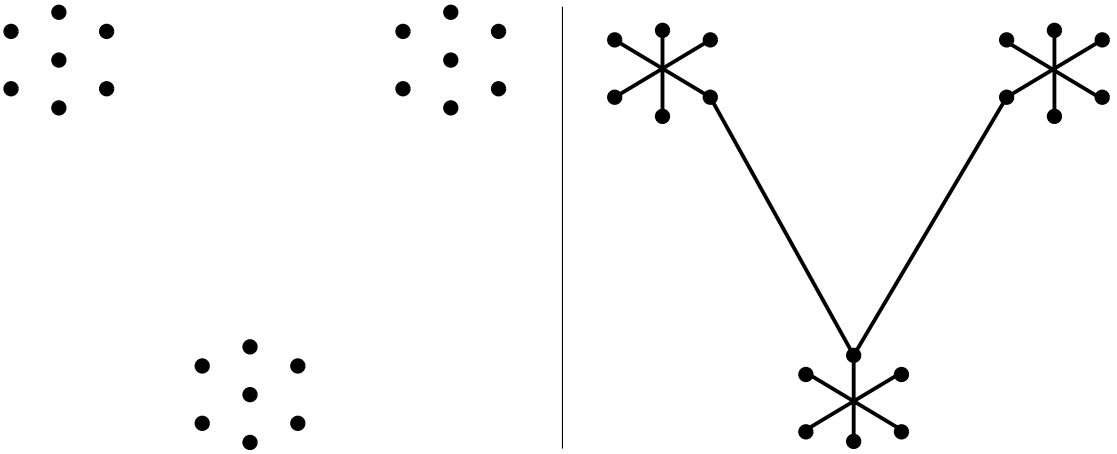
My (biased) preference for C language implementations of all basic graph algorithms, including topological sorting, is the library associated with this book. See Section 19.1.10 (page 661) for details.

Notes: Good expositions on topological sorting include [CLRS01, Man89]. Brightwell and Winkler [BW91] prove that it is #P-complete to count the number of linear extensions of a partial order. The complexity class #P includes NP, so any #P-complete problem is at least NP-hard.

Pruesse and Ruskey [PR86] give an algorithm that generates linear extensions of a DAG in constant amortized time. Further, each extension differs from its predecessor by either one or two adjacent transpositions. This algorithm can be used to count the number of linear extensions $e(G)$ of an n-vertex DAG G in $O(n^2 + e(G))$. Alternately, the reverse search technique of Avis and Fukuda [AF96] can be employed to list linear extensions. A backtracking program to generate all linear extensions is described in [KS74].

Huber [Hub06] gives an algorithm to sample linear extensions uniformly at random from an arbitrary partial order in expected $O(n^3 \lg n)$ time, improving the result of [BD99].

Related Problems: Sorting (see page 436), feedback edge/vertex set (see page 559).



INPUT OUTPUT

15.3 Minimum Spanning Tree

Input description: A graph $G = (V, E)$ with weighted edges.

Problem description: The minimum weight subset of edges $E' \subset E$ that form a tree on V.

Discussion: The minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component. Telephone companies are interested in minimum spanning trees, because the MST of a set of locations defines the wiring scheme that connects the sites using as little wire as possible. MST is the mother of all network design problems.

Minimum spanning trees prove important for several reasons:

- They can be computed quickly and easily, and create a sparse subgraph that reflects a lot about the original graph.
- They provide a way to identify clusters in sets of points. Deleting the long edges from an MST leaves connected components that define natural clusters in the data set, as shown in the output figure above.
- They can be used to give approximate solutions to hard problems such as Steiner tree and traveling salesman.
- As an educational tool, MST algorithms provide graphic evidence that greedy algorithms can give provably optimal solutions.

Three classical algorithms efficiently construct MSTs. Detailed implementations of two of them (Prim's and Kruskal's) are given with correctness arguments in Section 6.1 (page 192). The third somehow manages to be less well known despite being invented first and (arguably) being both easier to implement and more efficient.

The contenders are

- *Kruskal's algorithm* – Each vertex starts as a separate tree and these trees merges together by repeatedly adding the lowest cost edge that spans two distinct subtrees (i.e. , does not create a cycle).

```
Kruskal(G)
    Sort the edges in order of increasing weight
    count = 0
    while (count < n − 1) do
        get next edge (v, w)
        if (component (v) ≠ component(w))
            add to T
            component(v) = component(w)
```

  The "which component?" tests can be efficiently implemented using the union-find data structure (Section 12.5 (page 385)) to yield an $O(m \lg m)$ algorithm.

- *Prim's algorithm* – Starts with an arbitrary vertex v and "grows" a tree from it, repeatedly finding the lowest-cost edge that links some new vertex into this tree. During execution, we label each vertex as either in the tree, in the *fringe* (meaning there exists an edge from a tree vertex), or *unseen* (meaning the vertex is still more than one edge away from the tree).

```
Prim(G)
    Select an arbitrary vertex to start
    While (there are fringe vertices)
        select minimum-weight edge between tree and fringe
        add the selected edge and vertex to the tree
        update the cost to all affected fringe vertices
```

  This creates a spanning tree for any connected graph, since no cycle can be introduced via edges between tree and fringe vertices. That it is in fact a tree of minimum weight can be proven by contradiction. With simple data structures, Prim's algorithm can be implemented in $O(n^2)$ time.

- *Boruvka's algorithm* – Rests on the observation that the lowest-weight edge incident on each vertex must be in the minimum spanning tree. The union of these edges will result in a spanning forest of at most $n/2$ trees. Now for each of these trees T, select the edge (x, y) of lowest weight such that $x \in T$ and

$y \notin T$. Each of these edges must again be in an MST, and the union again results in a spanning forest with at most half as many trees as before:

```
Boruvka(G)
    Initialize spanning forest F to n single-vertex trees
        While (F has more than one tree)
            for each T in F, find the smallest edge from T to G − T
            add all selected edges to F, thus merging pairs of trees
```

The number of trees are at least halved in each round, so we get the MST after at most $\lg n$ iterations, each of which takes linear time. This gives an $O(m \log n)$ algorithm without using any fancy data structures.

MST is only one of several spanning tree problems that arise in practice. The following questions will help you sort your way through them:

- *Are the weights of all edges of your graph identical?* – Every spanning tree on n points contains exactly $n-1$ edges. Thus, if your graph is unweighted, *any* spanning tree will be a minimum spanning tree. Either breadth-first or depth-first search can be used to find a rooted spanning tree in linear time. DFS trees tend to be long and thin, while BFS trees better reflect the distance structure of the graph, as discussed in Section 5 (page 145).

- *Should I use Prim's or Kruskal's algorithm?* – As implemented in Section 6.1 (page 192), Prim's algorithm runs in $O(n^2)$, while Kruskal's algorithm takes $O(m \log m)$ time. Thus Prim's algorithm is faster on dense graphs, while Kruskal's is faster on sparse graphs.

  That said, Prim's algorithm can be implemented in $O(m + n \lg n)$ time using more advanced data structures, and a Prim's implementation using pairing heaps would be the fastest practical choice for both sparse and dense graphs.

- *What if my input is points in the plane, instead of a graph?* – Geometric instances, comprising n points in d-dimensions, can be solved by constructing the complete distance graph in $O(n^2)$ and then finding the MST of this complete graph. However, for points in two dimensions, it is more efficient to solve the geometric version of the problem directly. To find the minimum spanning tree of n points, first construct the Delaunay triangulation of these points (see Sections 17.3 and 17.4). In two dimensions, this gives a graph with $O(n)$ edges that contains all the edges of the minimum spanning tree of the point set. Running Kruskal's algorithm on this sparse graph finishes the job in $O(n \lg n)$ time.

- *How do I find a spanning tree that avoids vertices of high degree?* – Another common goal of spanning tree problems is to minimize the maximum degree,

typically to minimize the fan out in an interconnection network. Unfortunately, finding a spanning tree of maximum degree 2 is NP-complete, since this is identical to the Hamiltonian path problem. However, efficient algorithms are known that construct spanning trees whose maximum degree is at most one more than required. This is likely to suffice in practice. See the references below.

Implementations: All the graph data structure implementations of Section 12.4 (page 381) *should* include implementations of Prim's and/or Kruskal's algorithms. This includes the Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) and LEDA (see Section 19.1.1 (page 658)) for C++. For some reason it does not seem to include the Java graph libraries oriented around social networks, but Prim and Kruskal are present in the *Data Structures Library in Java* (JDSL) (*http://www.jdsl.org/*).

Timing experiments on MST algorithms produce contradicting results, suggesting the stakes are really too low to matter. Pascal implementations of Prim's, Kruskal's, and the Cheriton-Tarjan algorithm are provided in [MS91], along with extensive empirical analysis proving that Prim's algorithm with the appropriate priority queue is fastest on most graphs. The programs in [MS91] are available by anonymous FTP from *cs.unm.edu* in directory */pub/moret_shapiro*. Kruskal's algorithm proved the fastest of four different MST algorithms in the Stanford GraphBase (see Section 19.1.8 (page 660)).

Combinatorica [PS03] provides Mathematica implementations of Kruskal's MST algorithm and quickly counting the number of spanning trees of a graph. See Section 19.1.9 (page 661).

My (biased) preference for C language implementations of all basic graph algorithms, including minimum spanning trees, is the library associated with this book. See Section 19.1.10 (page 661) for details.

Notes: The MST problem dates back to Boruvka's algorithm in 1926. Prim's [Pri57] and Kruskal's [Kru56] algorithms did not appear until the mid-1950's. Prim's algorithm was then rediscovered by Dijkstra [Dij59]. See [GH85] for more on the interesting history of MST algorithms. Wu and Chao [WC04b] have written a monograph on MSTs and related problems.

The fastest implementations of Prim's and Kruskal's algorithms use Fibonacci heaps [FT87]. However, pairing heaps have been proposed to realize the same bounds with less overhead. Experiments with pairing heaps are reported in [SV87].

A simple combination of Boruvka's algorithm with Prim's algorithm yields an $O(m \lg \lg n)$ algorithm. Run Borukva's algorithm for $\lg \lg n$ iterations, yielding a forest of at most $n/\lg n$ trees. Now create a graph G' with one vertex for each tree in this forest, with the weight of the edge between trees T_i and T_j set to the lightest edge (x, y), where $x \in T_i$ and $y \in T_j$. The MST of G' coupled with the edges selected by Boruvka's algorithm yields the MST of G. Prim's algorithm (implemented with Fibonacci heaps) will take $O(n + m)$ time on this $n/\lg n$ vertex, m edge graph.

The best theoretical bounds on finding MSTs tell a complicated story. Karger, Klein, and Tarjan [KKT95] give a linear-time randomized algorithm for MSTs, based again on Borukva's algorithm. Chazelle [Cha00] gave a deterministic $O(n\alpha(m,n))$ algorithm, where $\alpha(m,n)$ is the inverse Ackerman function. Pettie and Ramachandran [PR02] give an provably optimal algorithm whose exact running time is (paradoxically) unknown, but lies between $\Omega(n+m)$ and $O(n\alpha(m,n))$.

A *spanner* $S(G)$ of a given graph G is a subgraph that offers an effective compromise between two competing network objectives. To be precise, they have total weight close to the MST of G, while guaranteeing that the shortest path between vertices x and y in $S(G)$ approaches the shortest path in the full graph G. The monograph of Narasimhan and Smid [NS07] provides a complete, up-to-date survey on spanner networks.

The $O(n \log n)$ algorithm for Euclidean MSTs is due to Shamos, and discussed in computational geometry texts such as [dBvKOS00, PS85].

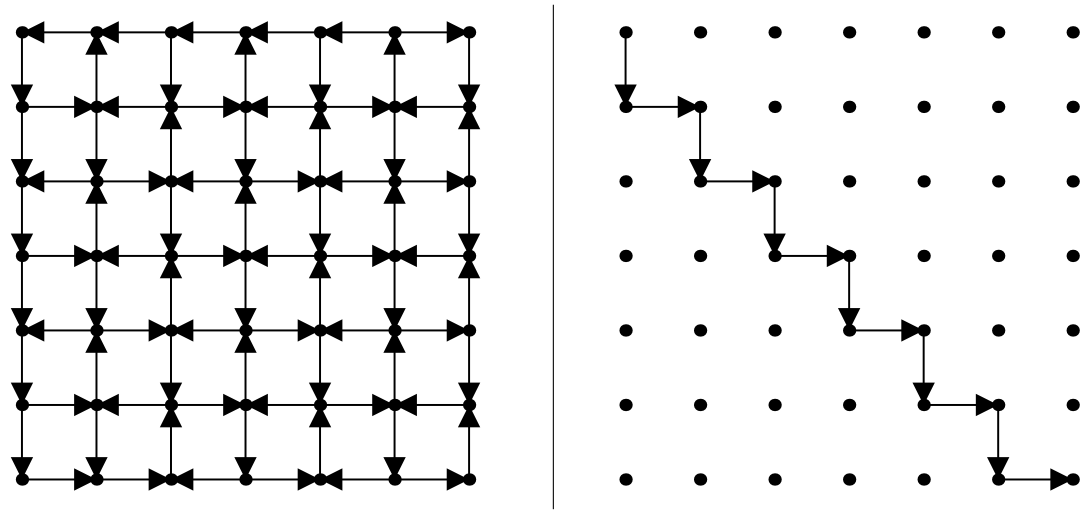
Fürer and Raghavachari [FR94] give an algorithm that constructs a spanning tree whose maximum degree is almost minimized—indeed is at most one more than the lowest-degree spanning tree. The situation is analogous to Vizing's theorem for edge coloring, which also gives an approximation algorithm to within additive factor one. A recent generalization [SL07] gives a polynomial-time algorithm for finding a spanning tree of maximum degree $\leq k+1$ whose cost is no more than that of the optimal minimum spanning tree of maximum degree $\leq k$.

Minimum spanning tree algorithms have an interpretation in terms of *matroids*, which are systems of subsets closed under inclusion. The maximum weighted independent set in matroids can be found using a greedy algorithm. The connection between greedy algorithms and matroids was established by Edmonds [Edm71]. Expositions on the theory of matroids include [Law76, PS98].

Dynamic graph algorithms seek to maintain an graph invariant (such as the MST) efficiently under edge insertion or deletion operations. Holm et al. [HdlT01] gives an efficient, deterministic algorithm to maintain MSTs (and several other invariants) in amortized polylogarithmic time per update.

Algorithms for generating spanning trees in order from minimum to maximum weight are presented in [Gab77]. The complete set of spanning trees of an unweighted graph can be generated in constant amortized time. See Ruskey [Rus03] for an overview of algorithms for generating, ranking, and unranking spanning trees.

Related Problems: Steiner tree (see page 555), traveling salesman (see page 533).



INPUT OUTPUT

15.4 Shortest Path

Input description: An edge-weighted graph G, with vertices s and t.

Problem description: Find the shortest path from s to t in G.

Discussion: The problem of finding shortest paths in a graph has several applications, some quite surprising:

- The most obvious applications arise in transportation or communications, such as finding the best route to drive between Chicago and Phoenix or figuring how to direct packets to a destination across a network.
- Consider the task of partitioning a digitized image into regions containing distinct objects—a problem known as *image segmentation*. Separating lines are needed to carve the space into regions, but what path should these lines take through the grid? We may want a line that relatively directly goes from x to y, but avoids cutting through object pixels as much as possible. This grid of pixels can be modeled as a graph, with the cost of an edge reflecting the color transitions between neighboring pixels. The shortest path from x to y in this weighted graph defines the best separating line.
- *Homophones* are words that sound alike, such as *to*, *two*, and *too*. Distinguishing between homophones is a major problem in speech recognition systems.

The key is to bring some notion of grammatical constraints into the interpretation. We map each string of phonemes (recognized sounds) into words they might possibly match. We construct a graph whose vertices correspond to these possible word interpretations, with edges between neighboring word-interpretations. If we set the weight of each edge to reflect the likelihood of transition, the shortest path across this graph defines the best interpretation of the sentence. See Section 6.4 (page 212) for a more detailed account of a similar application.

- Suppose we want to draw an informative visualization of a graph. The "center" of the graph should appear near the center of the page. A good definition of the graph center is the vertex that minimizes the maximum distance to any other vertex in the graph. Identifying this center point requires knowing the distance (i.e. , shortest path) between all pairs of vertices.

The primary algorithm for finding shortest paths is *Dijkstra's algorithm*, which efficiently computes the shortest path from a given starting vertex x to all $n-1$ other vertices. In each iteration, it identifies a new vertex v for which the shortest path from x to v is known. We maintain a set of vertices S to which we currently know the shortest path from v, and this set grows by one vertex in each iteration. In each iteration, we identify the edge (u, v) where $u, u' \in S$ and $v, v' \in V - S$ such that

$$dist(x, u) + weight(u, v) = \min_{(u', v') \in E} dist(x, u') + weight(u', v')$$

This edge (u, v) gets added to a *shortest path tree*, whose root is x and describes all the shortest paths from x.

An $O(n^2)$ implementation of Dijkstra's algorithm appears in in Section 6.3.1 (page 206). Theoretically faster times can be achieved using significantly more complicated data structures, as described below. If we just need to know the shortest path from x to y, terminate the algorithm as soon as y enters S.

Dijkstra's algorithm is the right choice for single-source shortest path on positively weighted graphs. However, special circumstances dictate different choices:

- *Is your graph weighted or unweighted?* – If your graph is unweighted, a simple breadth-first search starting from the source vertex will find the shortest path to all other vertices in linear time. Only when edges have different weights do you need more sophisticated algorithms. A breadth-first search is both simpler and faster than Dijkstra's algorithm.

- *Does your graph have negative cost weights?* – Dijkstra's algorithm assumes that all edges have positive cost. For graphs with edges of negative weight, you must use the more general, but less efficient, Bellman-Ford algorithm. Graphs with negative cost cycles are an even bigger problem. Observe that the shortest x to y path in such a graph is not defined because we can detour

from x to the negative cost cycle and repeatedly loop around it, making the total cost arbitrarily small.

Note that adding a fixed amount of weight to make each edge positive *does not* solve the problem. Dijkstra's algorithm will then favor paths using a fewer number of edges, even if those were not the shortest weighted paths in the original graph.

- *Is your input a set of geometric obstacles instead of a graph?* – Many applications seek the shortest path between two points in a geometric setting, such as an obstacle-filled room. The most straightforward solution is to convert your problem into a graph of distances to feed to Dijkstra's algorithm. Vertices will correspond the vertices on the boundaries of the obstacles, with edges defined only between pairs of vertices that "see" each other.

  Alternately, there are more efficient geometric algorithms that compute the shortest path directly from the arrangement of obstacles. See Section 17.14 (page 610) on motion planning and the Notes section for pointers to such geometric algorithms.

- *Is your graph acyclic—i.e. , a DAG?* – Shortest paths in directed acyclic graphs can be found in linear time. Perform a topological sort to order the vertices such that all edges go from left to right starting from source s. The distance from s to itself, $d(s, s)$, clearly equals 0. We now process the vertices from left to right. Observe that

$$d(s, j) = \min_{(x,i) \in E} d(s, i) + w(i, j)$$

  since we already know the shortest path $d(s, i)$ for all vertices to the left of j. Indeed, most dynamic programming problems can be formulated as shortest paths on specific DAGs. Note that the same algorithm (replacing min with max) also suffices to find the *longest path* in a DAG, which is useful in many applications like scheduling (see Section 14.9 (page 468)).

- *Do you need the shortest path between all pairs of points?* – If you are interested in the shortest path between all pairs of vertices, one solution is to run Dijkstra n times, once with each vertex as the source. The Floyd-Warshall algorithm is a slick $O(n^3)$ dynamic programming algorithm for all-pairs shortest path, which is faster and easier to program than Dijkstra. It works with negative cost edges but not cycles, and is presented with an implementation in Section 6.3.2 (page 210). Let M denote the distance matrix, where $M_{ij} = \infty$ if there is no edge (i, j):

$$D^0 = M$$
$$\text{for } k = 1 \text{ to } n \text{ do}$$
$$\quad \text{for } i = 1 \text{ to } n \text{ do}$$

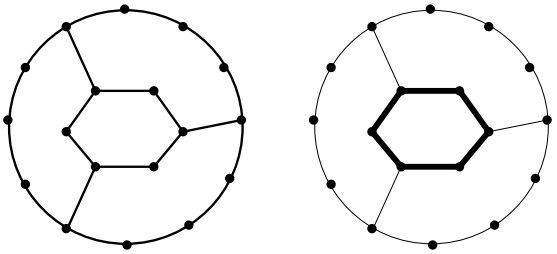


Figure 15.1: The girth, or shortest cycle, in a graph

---

for $j = 1$ to n do
$D_{ij}^k = \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$
Return D^n

The key to understanding Floyd's algorithm is that D_{ij}^k denotes "the length of the shortest path from i to j that goes through vertices $1, \ldots, k$ as possible intermediate vertices." Note that $O(n^2)$ space suffices, since we only must keep D^k and D^{k-1} around at time k.

- *How do I find the shortest cycle in a graph?* – One application of all-pairs shortest path is to find the shortest cycle in a graph, called its *girth*. Floyd's algorithm can be used to compute d_{ii} for $1 \leq i \leq n$, which is the length of the shortest way to get from vertex i to i—in other words, the shortest cycle through i.

  This *might* be what you want. The shortest cycle through x is likely to go from x to y back to x, using the same edge twice. A *simple* cycle is one that visits no edge or vertex twice. To find the shortest simple cycle, the easiest approach is to compute the lengths of the shortest paths from i to all other vertices, and then explicitly check whether there is an acceptable edge from each vertex back to i.

  Finding the *longest* cycle in a graph includes Hamiltonian cycle as a special case (see Section 16.5), so it is NP-complete.

The all-pairs shortest path matrix can be used to compute several useful invariants related to the center of graph G. The *eccentricity* of vertex v in a graph is the shortest-path distance to the farthest vertex from v. From the eccentricity come other graph invariants. The *radius* of a graph is the smallest eccentricity of any vertex, while the *center* is the set of vertices whose eccentricity is the radius. The *diameter* of a graph is the maximum eccentricity of any vertex.

Implementations: The highest performance shortest path codes are due to Andrew Goldberg and his collaborators, at *http://www.avglab.com/andrew/soft.html*.

In particular, MLB is a C++ short path implementation for non-negative, integer-weighted edges. See [Gol01] for details of the algorithm and its implementation. Its running time is typically only 4-5 times that of a breadth-first search, and it is capable of handling graphs with millions of vertices. High-performance C implementations of both Dijkstra and Bellman-Ford are also available [CGR99].

All the C++ and Java graph libraries discussed in Section 12.4 (page 381) include at least an implementation of Dijkstra's algorithm. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) has a particularly broad collection, including Bellman-Ford and Johnson's all-pairs shortest-path algorithm. LEDA (see Section 19.1.1 (page 658)) provides good implementations in C++ for all of the shortest-path algorithms we have discussed, including Dijkstra, Bellman-Ford, and Floyd's algorithms. JGraphT (*http://jgrapht.sourceforge.net/*) provides both Dijkstra and Bellman-Ford in Java. C language implementations of Dijkstra and Floyd's algorithms are provided in the library from this book. See Section 19.1.10 (page 661) for details.

Shortest-path algorithms was the subject of the 9th DIMACS Implementation Challenge, held in October 2006. Implementations of efficient algorithms for several aspects of finding shortest paths were discussed. The papers, instances, and implementations are available at *http://dimacs.rutgers.edu/Challenges/*.

Notes: Good expositions on Dijkstra's algorithm [Dij59], the Bellman-Ford algorithm [Bel58, FF62], and Floyd's all-pairs-shortest-path algorithm [Flo62] include [CLRS01]. Zwick [Zwi01] provides an up-to-date survey on shortest path algorithms. Geometric shortest-path algorithms are surveyed by Mitchell [PN04].

The fastest algorithm known for single-source shortest-path for positive edge weight graphs is Dijkstra's algorithm with Fibonacci heaps, running in $O(m + n \log n)$ time [FT87]. Experimental studies of shortest-path algorithms include [DF79, DGKK79]. However, these experiments were done before Fibonacci heaps were developed. See [CGR99] for a more recent study. Heuristics can be used to enhance the performance of Dijkstra's algorithm in practice. Holzer, et al. [HSWW05] provide a careful experimental study of how four such heuristics interact together.

Online services like Mapquest quickly find at least an approximate shortest path between two points in enormous road networks. This problem differs somewhat from the shortest-path problems here in that (1) preprocessing costs can be amortized over many point-to-point queries, (2) the backbone of high-speed, long-distance highways can reduce the path problem to identifying the best place to get on and off this backbone, and (3) approximate or heuristic solutions suffice in practice.

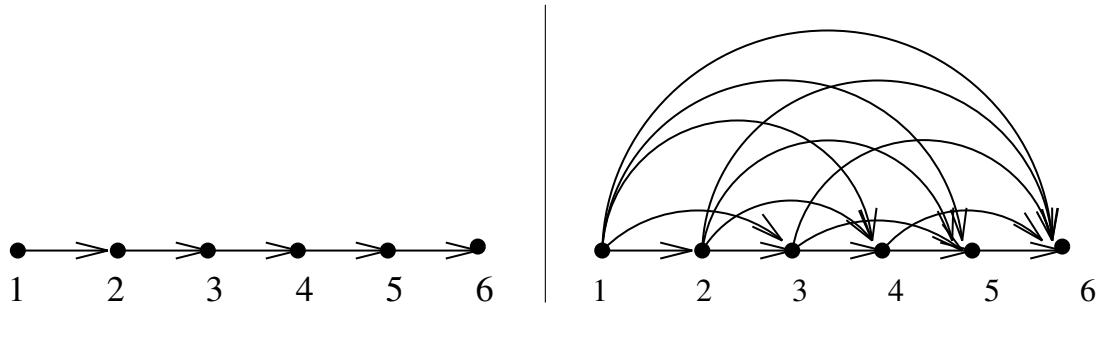
The A^*-algorithm performs a best-first search for the shortest path coupled with a lower-bound analysis to establish when the best path we have seen is indeed the shortest-path in the graph. Goldberg, Kaplan, and Werneck [GKW06, GKW07] describe an implementation of A^* capable of answering point-to-point queries in one millisecond on national-scale road networks after two hours of preprocessing.

Many applications demand multiple alternative short paths in addition to the optimal path. This motivates the problem of finding the k shortest paths. Variants exist depending upon whether the paths must be simple, or can contain cycles. Eppstein [Epp98] generates an implicit representation of these paths in $O(m+n \log n+k)$ time, from which each path

can be reconstructed in $O(n)$ time. Hershberger, et al. [HMS03] present a new algorithm and experimental results.

Fast algorithms for computing the girth are known for both general [IR78] and planar graphs [Dji00].

Related Problems: Network flow (see page 509), motion planning (see page 610).



INPUT OUTPUT

15.5 Transitive Closure and Reduction

Input description: A directed graph $G = (V, E)$.

Problem description: For *transitive closure*, construct a graph $G' = (V, E')$ with edge $(i, j) \in E'$ iff there is a directed path from i to j in G. For *transitive reduction*, construct a small graph $G' = (V, E')$ with a directed path from i to j in G' iff there is a directed path from i to j in G.

Discussion: Transitive closure can be thought of as establishing a data structure that makes it possible to solve reachability questions (can I get to x from y?) efficiently. After constructing the transitive closure, all reachability queries can be answered in constant time by simply reporting the appropriate matrix entry.

Transitive closure is fundamental in propagating the consequences of modified attributes of a graph G. For example, consider the graph underlying any spreadsheet model whose vertices are cells and have an edge from cell i to cell j if the result of cell j depends on cell i. When the value of a given cell is modified, the values of all reachable cells must also be updated. The identity of these cells is revealed by the transitive closure of G. Many database problems reduce to computing transitive closures, for analogous reasons.

There are three basic algorithms for computing transitive closure:

- The simplest algorithm just performs a breadth-first or depth-first search from each vertex and keeps track of all vertices encountered. Doing n such traversals gives an $O(n(n+m))$ algorithm, which degenerates to cubic time if the graph is dense. This algorithm is easily implemented, runs well on sparse graphs, and is likely the right answer for your application.

- Warshall's algorithm constructs transitive closures in $O(n^3)$ using a simple, slick algorithm that is identical to Floyd's all-pairs shortest-path algorithm

of Section 15.4 (page 489). If we are interested only in the transitive closure, and not the length of the resulting paths, we can reduce storage by retaining only one bit for each matrix element. Thus, $D_{ij}^k = 1$ iff j is reachable from i using only vertices $1, \ldots, k$ as intermediates.

- Matrix multiplication can also be used to solve transitive closure. Let M^1 be the adjacency matrix of graph G. The non-zero matrix entries of $M^2 = M \times M$ identify all length-2 paths in G. Observe that $M^2[i,j] = \sum_x M[i,x] \cdot M[x,j]$, so path (i,x,j) contributes to $M^2[i,j]$. Thus, the union $\cup_i^n M^i$ yields the transitive closure T. Furthermore, this union can be computed using only $O(\lg n)$ matrix operations using the fast exponentiation algorithm in Section 13.9 (page 423).

  You might conceivably win for large n by using Strassen's fast matrix multiplication algorithm, although I for one wouldn't bother trying. Since transitive closure is provably as hard as matrix multiplication, there is little hope for a significantly faster algorithm.

The running time of all three of these procedures can be substantially improved on many graphs. Recall that a strongly connected component is a set of vertices for which all pairs are mutually reachable. For example, any cycle defines a strongly connected subgraph. All the vertices in any strongly connected component must reach exactly the same subset of G. Thus, we can reduce our problem finding the transitive closure on a graph of strongly connected components that should have considerably fewer edges and vertices than G. The strongly connected components of G can be computed in linear time (see Section 15.1 (page 477)).

Transitive reduction (also known as *minimum equivalent digraph*) is the inverse operation of transitive closure, namely reducing the number of edges while maintaining identical reachability properties. The transitive closure of G is identical to the transitive closure of the transitive reduction of G. The primary application of transitive reduction is space minimization, by eliminating redundant edges from G that do not effect reachability. Transitive reduction also arises in graph drawing, where it is important to eliminate as many unnecessary edges as possible to reduce the visual clutter.

Although the transitive closure of G is uniquely defined, a graph may have many different transitive reductions, including G itself. We want the smallest such reduction, but there are multiple formulations of the problem:

- A linear-time, quick-and-dirty transitive reduction algorithm identifies the strongly connected components of G, replaces each by a simple directed cycle, and adds these edges to those bridging the different components. Although this reduction is not provably minimal, it is likely to be pretty close on typical graphs.

  One catch with this heuristic is that it might add edges to the transitive reduction of G that are not in G. This may or may not be a problem depending on your application.

- If all edges of our transitive reduction must exist in G, we have to abandon hope of finding the minimum size reduction. To see why, consider a directed graph consisting of one strongly connected component so that every vertex can reach every other vertex. The smallest possible transitive reduction will be a simple directed cycle, consisting of exactly n edges. This is possible if and only if G is Hamiltonian, thus proving that finding the smallest subset of edges is NP-complete.

  A heuristic for finding such a transitive reduction is to consider each edge successively and delete it if its removal does not change the transitive reduction. Implementing this efficiently means minimizing the time spent on reachability tests. Observe that a directed edge (i, j) can be eliminated whenever there is another path from i to j avoiding this edge.

- The minimum size reduction where we are allowed arbitrary pairs of vertices as edges can be found in $O(n^3)$ time. See the references below for details. However, the quick-and-dirty heuristic above will likely suffice for most applications, being easier to program as well as more efficient.

Implementations: The Boost implementation of transitive closure appears particularly well engineered, and relies on algorithms from [Nuu95]. LEDA (see Section 19.1.1 (page 658)) provides implementations of both transitive closure and reduction in C++ [MN99].

None of our usual Java libraries appear to contain implementations of either transitive closure or reduction. However, *Graphlib* contains a Java `Transitivity` library with both of them. See *http://www-verimag.imag.fr/∼cotton/* for details.

Combinatorica [PS03] provides Mathematica implementations of transitive closure and reduction, as well as the display of partial orders requiring transitive reduction. See Section 19.1.9 (page 661).

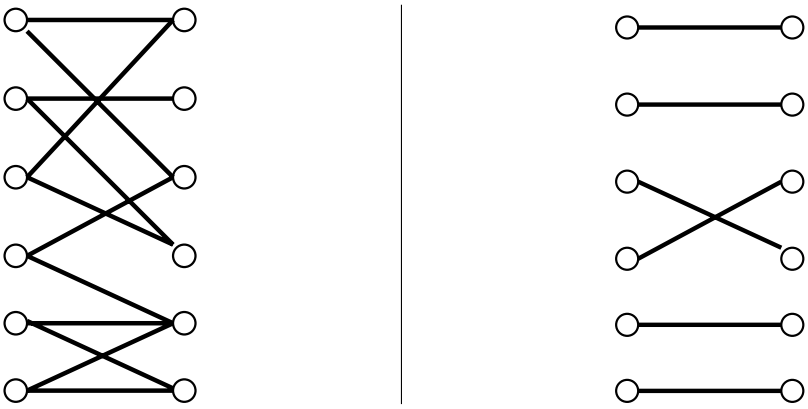
Notes: Van Leeuwen [vL90a] provides an excellent survey on transitive closure and reduction. The equivalence between matrix multiplication and transitive closure was proven by Fischer and Meyer [FM71], with expositions including [AHU74].

There is a surprising amount of recent activity on transitive closure, much of it captured by Nuutila [Nuu95]. Penner and Prasanna [PP06] improved the performance of Warshall's algorithm [War62] by roughly a factor of two through a cache-friendly implementation.

The equivalence between transitive closure and reduction, as well as the $O(n^3)$ reduction algorithm, was established in [AGU72]. Empirical studies of transitive closure algorithms include [Nuu95, PP06, SD75].

Estimating the size of the transitive closure is important in database query optimization. A linear-time algorithm for estimating the size of the closure is given by Cohen [Coh94].

Related Problems: Connected components (see page 477), shortest path (see page 489).



INPUT OUTPUT

15.6 Matching

Input description: A (weighted) graph $G = (V, E)$.

Problem description: Find the largest set of edges E' from E such that each vertex in V is incident to at most one edge of E'.

Discussion: Suppose we manage a group of workers, each of whom is capable of performing a subset of the tasks needed to complete a job. Construct a graph with vertices representing both the set of workers and the set of tasks. Edges link workers to the tasks they can perform. We must assign each task to a different worker so that no worker is overloaded. The desired assignment is the largest possible set of edges where no employee or job is repeated—i.e. , a matching.

Matching is a very powerful piece of algorithmic magic, so powerful that it is surprising that optimal matchings can be found efficiently. Applications arise often once you know to look for them.

Marrying off a set of boys to a set of girls such that each couple is happy is another bipartite matching problem, on a graph with an edge between any compatible boy and girl. For a synthetic biology application [MPC+06], I need to shuffle the characters in a string S to maximize the number of characters that move. For example, *aaabc* can be shuffled to *bcaaa* so that only one character stays fixed. This is yet another bipartite matching problem, where the boys represent the multiset of alphabet symbols and the girls are the positions in the string (1 to $|S|$). Edges link symbols to all the string positions that originally contained a different symbol.

This basic matching framework can be enhanced in several ways, while remaining essentially the same *assignment* problem:

- *Is your graph bipartite?* – Most matching problems involve bipartite graphs, as in the classic assignment problem of jobs to workers. This is fortunate because faster and simpler algorithms exist for bipartite matching.

- *What if certain employees can be given multiple jobs?* – Natural generalizations of the assignment problem include assigning certain employees more than one task to do, or (equivalently) seeking multiple workers for a given job. Here, we do not seek a matching so much as a covering with small "stars." Such desires can be modeled by replicating an employee vertex by as many times as we want her to be matched. Indeed, we employed this trick in the string permutation example above.

- *Is your graph weighted or unweighted?* – Many matching applications are based on unweighted graphs. Perhaps we seek to maximize the total number of tasks performed, where one task is as good as another. Here we seek a maximum *cardinality* matching—ideally a *perfect* matching where every vertex is matched to another in the matching.

  For other applications, however, we need to augment each edge with a weight, perhaps reflecting the suitability of an employee for a given task. The problem now becomes constructing a maximum *weight* matching—i.e. , the set of independent edges of maximum total cost.

Efficient algorithms for constructing matchings work by constructing *augmenting paths* in graphs. Given a (partial) matching M in a graph G, an augmenting path is a path of edges P that alternate (out-of-M, in-M, ..., out-of-M). We can enlarge the matching by one edge given such an augmenting path, replacing the even-numbered edges of P from M with the odd-numbered edges of P. Berge's theorem states that a matching is maximum if and only if it does not contain any augmenting path. Therefore, we can construct maximum-cardinality matchings by searching for augmenting paths and stopping when none exist.

General graphs prove trickier because it is possible to have augmenting paths that are odd-length cycles (i.e. , the first and last vertices are the same). Such cycles (or blossoms) are impossible in bipartite graphs, which by definition do not contain odd-length cycles.

The standard algorithms for bipartite matching are based on network flow, using a simple transformation to convert a bipartite graph into an equivalent flow graph. Indeed, an implementation of this is given in Section 6.5 (page 217).

Be warned that different approaches are needed to solve weighted matching problems, most notably the matrix-oriented "Hungarian algorithm."

Implementations: High-performance codes for both weighted and unweighted bipartite matching have been developed by Andrew Goldberg and his collaborators. `CSA` is a weighted bipartite matching code in C based on cost-scaling network flow, developed by Goldberg and Kennedy [GK95]. `BIM` is a faster unweighted bipartite matching code based on augmenting path methods, developed

by Cherkassky, et al. [CGM+98]. Both are available for noncommercial use from *http://www.avglab.com/andrew/soft.html*.

The First DIMACS Implementation Challenge [JM93] focused on network flows and matching. Several instance generators and implementations for maximum weight and maximum cardinality matching were collected, and can be obtained by anonymous FTP from *dimacs.rutgers.edu* in the directory *pub/netflow/matching*. These include

- A maximum-cardinality matching solver in Fortran 77 by R. Bruce Mattingly and Nathan P. Ritchey.
- A maximum-cardinality matching solver in C by Edward Rothberg, that implements Gabow's $O(n^3)$ algorithm.
- A maximum-weighted matching solver in C by Edward Rothberg. This is slower but more general than his unweighted solver just described.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including weighted bipartite matching. LEDA (see Section 19.1.1 (page 658)) provides efficient implementations in C++ for both maximum-cardinality and maximum-weighted matching, on both bipartite and general graphs.

Blossum IV [CR99] is an efficient code in C for minimum-weight perfect matching available at *http://www2.isye.gatech.edu/~wcook/software.html*. An $O(mn\alpha(m,n))$ implementation of maximum-cardinality matching in general graphs (*http://www.cs.arizona.edu/~kece/Research/software.html*) is due to Kececioglu and Pecqueur [KP98].

The Stanford GraphBase (see Section 19.1.8 (page 660)) contains an implementation of the Hungarian algorithm for bipartite matching. To provide readily visualized weighted bipartite graphs, Knuth uses a digitized version of the Mona Lisa and seeks row/column disjoint pixels of maximum brightness. Matching is also used to construct clever, resampled "domino portraits".

Notes: Lovász and Plummer [LP86] is the definitive reference on matching theory and algorithms. Survey articles on matching algorithms include [Gal86]. Good expositions on network flow algorithms for bipartite matching include [CLRS01, Eve79a, Man89], and those on the Hungarian method include [Law76, PS98]. The best algorithm for maximum bipartite matching, due to Hopcroft and Karp [HK73], repeatedly finds the shortest augmenting paths instead of using network flow, and runs in $O(\sqrt{n}m)$. The Hungarian algorithm runs in $O(n(m + n \log n))$ time.

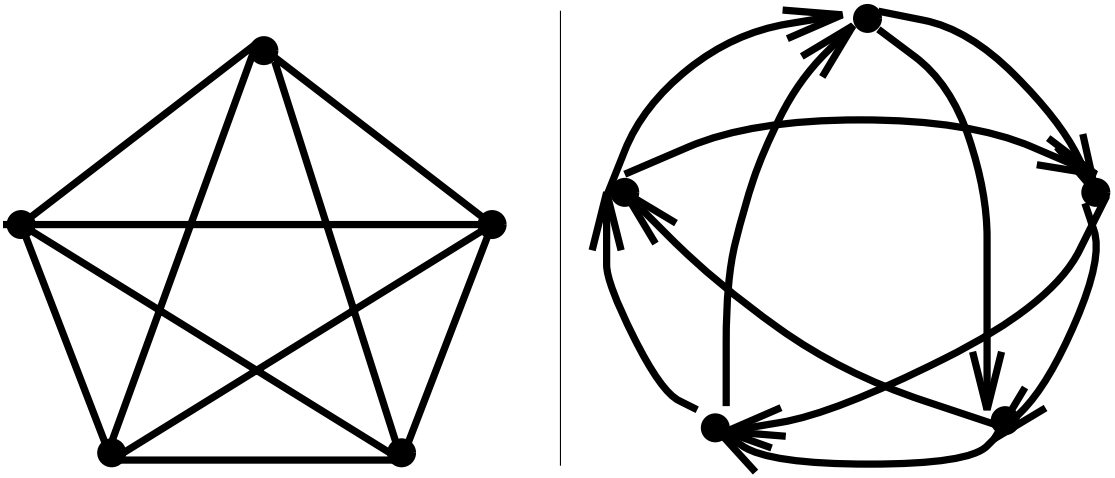
Edmond's algorithm [Edm65] for maximum-cardinality matching is of great historical interest for provoking questions on what problems can be solved in polynomial time. Expositions on Edmond's algorithm include [Law76, PS98, Tar83]. Gabow's [Gab76] implementation of Edmond's algorithm runs in $O(n^3)$ time. The best algorithm known for general matching runs in $O(\sqrt{n}m)$ [MV80].

Consider a matching of boys to girls containing edges (B_1, G_1) and (B_2, G_2), where B_1 and G_2 in fact prefer each other to their own spouses. In real life, these two would

run off with each other, breaking the marriages. A marriage without any such couples is said to be *stable*. The theory of stable matching is thoroughly treated in [GI89]. It is a surprising fact that no matter how the boys and girls rate each other, there is always at least one stable marriage. Further, such a marriage can be found in $O(n^2)$ time [GS62]. An important application of stable marriage occurs in the annual matching of medical residents to hospitals.

The maximum matching is equal in size to the minimum vertex cover in bipartite graphs. This implies that both the minimum vertex cover problem and maximum independent set problems can be solved in polynomial time on bipartite graphs.

Related Problems: Eulerian cycle (see page 502), network flow (see page 509).



INPUT OUTPUT

15.7 Eulerian Cycle/Chinese Postman

Input description: A graph $G = (V, E)$.

Problem description: Find the shortest tour visiting each edge of G at least once.

Discussion: Suppose you are given the map of a city and charged with designing the routes for garbage trucks, snow plows, or postmen. In each of these applications, every road in the city must be completely traversed at least once in order to ensure that all deliveries or pickups are made. For efficiency, you seek to minimize total drive time, or (equivalently) the total distance or number of edges traversed.

Alternately, consider a human-factors validation of telephone menu systems. Each "Press 4 for more information" option is properly interpreted as an edge between two vertices in a graph. Our tester seeks the most efficient way to walk over this graph and visit every link in the system at least once.

Such applications are variants of the *Eulerian cycle* problem, best characterized by the puzzle that asks children to draw a given figure completely without (1) without repeating any edges, or (2) lifting their pencil off the paper. They seek a path or cycle through a graph that visits each edge exactly once.

Well-known conditions exist for determining whether a graph contains an Eulerian cycle or path:

- An *undirected* graph contains an Eulerian *cycle* iff (1) it is connected, and (2) each vertex is of even degree.

- An *undirected* graph contains an Eulerian *path* iff (1) it is connected, and (2) all but two vertices are of even degree. These two vertices will be the start and end points of any path.

- A *directed* graph contains an Eulerian *cycle* iff (1) it is strongly-connected, and (2) each vertex has the same in-degree as out-degree.

- Finally, a *directed* graph contains an Eulerian *path* from x to y iff (1) it is connected, and (2) all other vertices have the same in-degree as out-degree, with x and y being vertices with in-degree one less and one more than their out-degrees, respectively.

This characterization of Eulerian graphs makes it is easy to test whether such a cycle exists: verify that the graph is connected using DFS or BFS, and then count the number of odd-degree vertices. Explicitly constructing the cycle also takes linear time. Use DFS to find an arbitrary cycle in the graph. Delete this cycle and repeat until the entire set of edges has been partitioned into a set of edge-disjoint cycles. Since deleting a cycle reduces each vertex degree by an even number, the remaining graph will continue to satisfy the same Eulerian degree-bound conditions. These cycles will have common vertices (since the graph is connected), and so can be spliced together in a "figure eight" at a shared vertex. By so splicing all the extracted cycles together, we construct a single circuit containing all of the edges.

An Eulerian cycle, if one exists, solves the motivating snowplow problem, since any tour that visits each edge only once must have minimum length. However, it is unlikely that your road network happens to satisfy the Eulerian degree conditions. Instead, we need to solve the more general *Chinese postman problem*, which minimizes the length of a cycle that traverses every edge at least once. This minimum cycle will never visit any edge more than twice, so good tours exist for any road network.

The optimal postman tour can be constructed by adding the appropriate edges to the graph G to make it Eulerian. Specifically, we find the shortest path between each pair of odd-degree vertices in G. Adding a path between two odd-degree vertices in G turns both of them to even-degree, moving G closer to becoming an Eulerian graph. Finding the best set of shortest paths to add to G reduces to identifying a minimum-weight perfect matching in a special graph G'.

For undirected graphs, the vertices of G' correspond the odd-degree vertices of G, with the weight of edge (i, j) defined to be the length of the shortest path from i to j in G. For directed graphs, the vertices of G' correspond to the degree-imbalanced vertices from G, with the bonus that all edges in G' go from out-degree deficient vertices to in-degree deficient ones. Thus, bipartite matching algorithms suffice when G is directed. Once the graph is Eulerian, the actual cycle can be extracted in linear time using the procedure just described.

Implementations: Several graph libraries provide implementations of Eulerian cycles, but Chinese postman implementations are rarer. We recommend the imple-

mentation of directed Chinese postman by Thimbleby [Thi03]. This Java implementation is available at *http://www.cs.swan.ac.uk/~csharold/cpp/index.html*.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including Chinese postman for both directed and undirected graphs. LEDA (see Section 19.1.1 (page 658)) provides all the tools for an efficient implementation: Eulerian cycles, matching, and shortest-paths bipartite and general graphs.

Combinatorica [PS03] provides Mathematica implementations of Eulerian cycles and de Bruijn sequences. See Section 19.1.9 (page 661).

Notes: The history of graph theory began in 1736, when Euler [Eul36] first solved the seven bridges of Königsberg problem. Königsberg (now Kaliningrad) is a city on the banks of the Pregel river. In Euler's day there were seven bridges linking the banks and two islands, which can be modeled as a multigraph with seven edges and four vertices. Euler sought a way to walk over each of the bridges exactly once and return home—i.e. , an Eulerian cycle. Euler proved that such a tour is impossible, since all four of the vertices had odd degrees. The bridges were destroyed in World War II. See [BLW76] for a translation of Euler's original paper and a history of the problem.

Expositions on linear-time algorithms for constructing Eulerian cycles [Ebe88] include [Eve79a, Man89]. Fleury's algorithm [Luc91] is a direct and elegant approach to constructing Eulerian cycles. Start walking from any vertex, and erase any edge that has been traversed. The only criterion in picking the next edge is that we avoid using a bridge (an edge whose deletion disconnects the graph) until no other alternative remains.

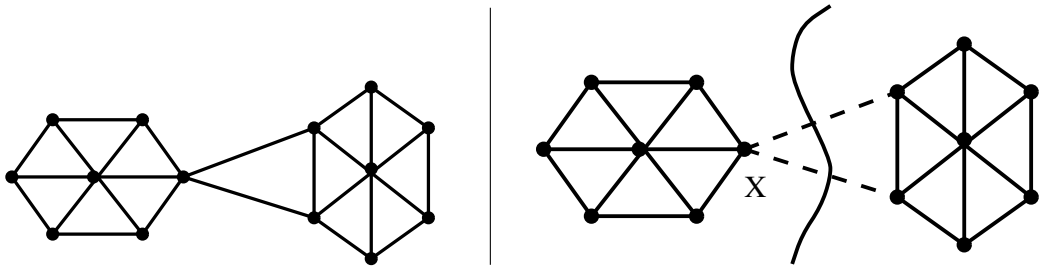
The Euler's tour technique is an important paradigm in parallel graph algorithms. Many parallel graph algorithms start by finding a spanning tree and then rooting the tree, where the rooting is done using the Euler tour technique. See parallel algorithms texts (e.g., [JÓ2]) for an exposition, and [CB04] for recent experience in practice. Efficient algorithms exist to count the number of Eulerian cycles in a graph [HP73].

The problem of finding the shortest tour traversing all edges in a graph was introduced by Kwan [Kwa62], hence the name *Chinese* postman. The bipartite matching algorithm for solving Chinese postman is due to Edmonds and Johnson [EJ73]. This algorithm works for both directed and undirected graphs, although the problem is NP-complete for mixed graphs [Pap76a]. Mixed graphs contain both directed and undirected edges. Expositions of the Chinese postman algorithm include [Law76].

A *de Bruijn* sequence S of span n on an alphabet Σ of size α is a circular string of length α^n containing all strings of length n as substrings of S, each exactly once. For example, for $n = 3$ and $\Sigma = \{0, 1\}$, the circular string 00011101 contains the following substrings in order: 000, 001, 011, 111, 110, 101, 010, 100. De Bruijn sequences can be thought of as "safe cracker" sequences, describing the shortest sequence of dial turns with α positions sufficient to try out all combinations of length n.

De Bruijn sequences can be constructed by building a directed graph whose vertices are all α^{n-1} strings of length $n-1$, where there is an edge (u, v) iff $u = s_1 s_2 \ldots s_{n-1}$ and $v = s_2 \ldots s_{n-1} s_n$. Any Eulerian cycle on this graph describes a de Bruijn sequence. Expositions on de Bruijn sequences and their construction include [Eve79a, PS03].

Related Problems: Matching (see page 498), Hamiltonian cycle (see page 538).



INPUT OUTPUT

15.8 Edge and Vertex Connectivity

Input description: A graph G. Optionally, a pair of vertices s and t.

Problem description: What is the smallest subset of vertices (or edges) whose deletion will disconnect G? Or which will separate s from t?

Discussion: Graph connectivity often arises in problems related to network reliability. In the context of telephone networks, the vertex connectivity is the smallest number of switching stations that a terrorist must bomb in order to separate the network—i.e. , prevent two unbombed stations from talking to each other. The edge connectivity is the smallest number of wires that need to be cut to accomplish the same objective. One well-placed bomb or snipping the right pair of cables suffices to disconnect the above network.

The edge (vertex) connectivity of a graph G is the smallest number of edge (vertex) deletions sufficient to disconnect G. There is a close relationship between the two quantities. The vertex connectivity is always less than or equal to the edge connectivity, since deleting one vertex from each edge in a cut set succeeds in disconnecting the graph. But smaller vertex subsets may be possible. The minimum vertex degree is an upper bound for both edge and vertex connectivity, since deleting all its neighbors (or cutting the edges to all its neighbors) disconnects the graph into one big and one single-vertex component.

Several connectivity problems prove to be of interest:

- *Is the graph already disconnected?* – The simplest connectivity problem is testing whether the graph is in fact connected. A simple depth-first or breadth-first search suffices to identify all connected components in linear time, as discussed in Section 15.1 (page 477). For directed graphs, the issue is whether the graph is *strongly connected*, meaning there is a directed path between any pair of vertices. In a *weakly connected* graph, there may exist paths to nodes from which there is no way to return.

- *Is there one weak link in my graph?* – We say that G is *biconnected* if no single vertex deletion is sufficient to disconnect G. Any vertex that is such a weak point is called an *articulation vertex*. A *bridge* is the analogous concept for edges, meaning a single edge whose deletion disconnects the graph.

  The simplest algorithms for identifying articulation vertices (or bridges) try deleting vertices (or edges) one by one, and then use DFS or BFS to test whether the resulting graph is still connected. More sophisticated linear-time algorithms exist for both problems, based on depth-first search. Indeed, a full implementation is given in Section 5.9.2 (page 173).

- *What if I want to split the graph into equal-sized pieces?* – What is often sought is a small cut set that breaks the graph into roughly equal-sized pieces. For example, suppose we want to split a big computer program into two maintainable units. We can construct a graph whose the vertices represent subroutines. Edges can be added between any two subroutines that interact, namely where one calls the other. We now seek to partition the subroutines into roughly equal-sized sets so that few pairs of interacting routines span the divide.

  This is the *graph partition* problem, further discussed in Section 16.6 (page 541). Although the problem is NP-complete, reasonable heuristics exist to solve it.

- *Are arbitrary cuts OK, or must I separate a given pair of vertices?* – There are two flavors of the general connectivity problem. One asks for the smallest cut-set for the entire graph, the other for the smallest set to separate s from t. Any algorithm for $(s-t)$ connectivity can be used with each of the $n(n-1)/2$ possible pairs of vertices to give an algorithm for general connectivity. Less obviously, $n-1$ runs suffice for testing edge connectivity, since we know that vertex v_1 must end up in a different component from at least one of the other $n-1$ vertices after deleting any cut set.

Edge and vertex connectivity can both be found using network-flow techniques. Network flow, discussed in Section 15.9 (page 509), interprets a weighted graph as a network of pipes where each edge has a maximum capacity and we seek to maximize the flow between two given vertices of the graph. The maximum flow between v_i, v_j in G is exactly the weight of the smallest set of edges to disconnect v_i from v_j. Thus the edge connectivity can be found by minimizing the flow between v_i and each of the $n-1$ other vertices in an unweighted graph G. Why? After deleting the minimum-edge cut set, v_i will be separated from some other vertex.

Vertex connectivity is characterized by *Menger's theorem*, which states that a graph is k-connected if and only if every pair of vertices is joined by at least k vertex-disjoint paths. Network flow can again be used to perform this calculation, since a flow of k between a pair of vertices implies k *edge*-disjoint paths. To exploit Menger's theorem, we construct a graph G' such that any set of edge-disjoint paths

in G' corresponds to vertex-disjoint paths in G. This is done by replacing each vertex v_i of G with two vertices $v_{i,1}$ and $v_{i,2}$, such that edge $(v_{i,1}, v_{i,2}) \in G'$ for all $v_i \in G$, and by replacing every edge $(v_i, x) \in G$ by edges $(v_{i,j}, x_k)$, $j \neq k \in \{0, 1\}$ in G'. Thus two edge-disjoint paths in G' correspond to each vertex-disjoint path in G.

Implementations: `MINCUTLIB` is a collection of high-performance codes for several different cut algorithms, including both flow and contraction-based methods. They were implemented by Chekuri, et al. as part of a substantial experimental study [CGK+97]. The codes are available for noncommercial use at *http://www.avglab.com/andrew/soft.html*. Also included is the full version of [CGK+97]—an excellent presentation of these algorithms and the heuristics needed to make them run fast.

Most of the graph data structure libraries of Section 15.1 (page 477) include routines for connectivity and biconnectivity testing. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) is distinguished by also including an implementation of edge connectivity testing.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including both edge and vertex connectivity.

LEDA (see Section 19.1.1 (page 658)) contains extensive support for both low-level connectivity testing (both biconnected and triconnected components) and edge connectivity/minimum-cut in C++.

Combinatorica [PS03] provides Mathematica implementations of edge and vertex connectivity, as well as connected, biconnected, and strongly connected components with bridges and articulation vertices. See Section 19.1.9 (page 661).

Notes: Good expositions on the network-flow approach to edge and vertex connectivity include [Eve79a, PS03]. The correctness of these algorithms is based on Menger's theorem [Men27] that connectivity is determined by the number of edge and vertex disjoint paths separating a pair of vertices. The maximum-flow, minimum-cut theorem is due to Ford and Fulkerson [FF62].

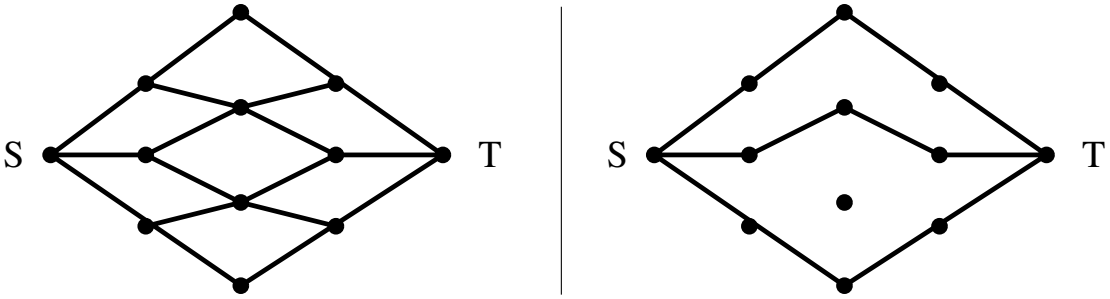
The theoretically fastest algorithms for minimum-cut/edge connectivity are based on graph contraction, not network flows. Contracting an edge (x, y) in a graph G merges the two incident vertices into one, removing self-loops but leaving multiedges. Any sequence of such contractions can raise (but not lower) the minimum cut in G, and leaves the cut unchanged if no edge of the cut is contracted. Karger gave a beautiful randomized algorithm for minimum cut, observing that the minimum cut is left unchanged with nontrivial probability over the course of any random series of deletions. The fastest version of Karger's algorithm runs in $(m \lg^3 n)$ expected time [Kar00]. See Motwani and Raghavan [MR95] for an excellent treatment of randomized algorithms, including a presentation of Karger's algorithm.

Nagamouchi and Ibaraki [NI92] give a deterministic contraction-based algorithm to find the minimum cut in $O(n(m + n \log n))$. In each round, this algorithm identifies and contracts an edge that is provably not in the minimum cut. See [CGK+97, NOI94] for experimental comparisons of algorithms for finding minimum cuts.

Minimum-cut methods have found many applications in computer vision, including image segmentation. Boykov and Kolmogorov [BK04] report on an experimental evaluation of minimum-cut algorithms in this context.

A nonflow-based algorithm for edge k-connectivity in $O(kn^2)$ is due to Matula [Mat87]. Faster k-connectivity algorithms are known for certain small values of k. All three-connected components of a graph can be generated in linear time [HT73a], while $O(n^2)$ suffices to test 4-connectivity [KR91].

Related Problems: Connected components (see page 477), network flow (see page 509), graph partition (see page 541).



INPUT OUTPUT

15.9 Network Flow

Input description: A directed graph G, where each edge $e = (i, j)$ has a capacity c_e. A source node s and sink node t.

Problem description: What is the maximum flow you can route from s to t while respecting the capacity constraint of each edge?

Discussion: Applications of network flow go far beyond plumbing. Finding the most cost-effective way to ship goods between a set of factories and a set of stores defines a network-flow problem, as do many resource-allocation problems in communications networks.

The real power of network flow is (1) that a surprising variety of linear programming problems arising in practice can be modeled as network-flow problems, and (2) that network-flow algorithms can solve these problems much faster than general-purpose linear programming methods. Several graph problems we have discussed in this book can be solved using network flow, including bipartite matching, shortest path, and edge/vertex connectivity.

The key to exploiting this power is recognizing that your problem can be modeled as network flow. This requires experience and study. My recommendation is that you first construct a linear programming model for your problem and then compare it with linear programs for the two primary classes of network flow problems: *maximum flow* and *minimum-cost flow*:

- *Maximum Flow* – Here we seek the heaviest possible flow from s to t, given the edge capacity constraints of G. Let x_{ij} be a variable accounting for the flow from vertex i through directed edge (i, j). The flow through this edge is constrained by its capacity c_{ij}, so

$$0 \leq x_{ij} \leq c_{ij} \text{ for } 1 \leq i, j \leq n$$

Furthermore, an equal flow comes in as goes out at each nonsource or sink vertex, so

$$\sum_{j=1}^{n} x_{ij} - \sum_{j=1}^{n} x_{ji} = 0 \text{ for } 1 \leq i \leq n$$

We seek the assignment that maximizes the flow into sink t, namely $\sum_{i=1}^{n} x_{it}$

- *Minimum Cost Flow* – Here we have an extra parameter for each edge (i, j), namely the cost (d_{ij}) of sending one unit of flow from i to j. We also have a targeted amount of flow f we want to send from s to t at minimum total cost. Hence, we seek the assignment that minimizes

$$\sum_{j=1}^{n} d_{ij} \cdot x_{ij}$$

subject to the edge and vertex capacity constraints of maximum flow, plus the additional restriction that $\sum_{i=1}^{n} x_{it} = f$.

Special considerations include:

- *What if I have multiple sources and/or sinks?* – No problem. We can handle this by modifying the network to create a vertex to serve as a super-source that feeds all the sources, and a super-sink that drains all the sinks.

- *What if all arc capacities are identical, either 0 or 1?* – Faster algorithms exist for 0-1 network flows. See the Notes section for details.

- *What if all my edge costs are identical?* – Use the simpler and faster algorithms for solving maximum flow as opposed to minimum-cost flow. Max-flow without edge costs arises in many applications, including edge/vertex connectivity and bipartite matching.

- *What if I have multiple types of material moving through the network?* – In a telecommunications network, every message has a given source and destination. Each destination needs to receive *exactly* those calls sent to it, not an equal amount of communication from arbitrary places. This can be modeled as a *multicommodity flow* problem, where each call defines a different commodity and we seek to satisfy all demands without exceeding the total capacity of any edge.

  Linear programming will suffice for multicommodity flow if fractional flows are permitted. Unfortunately, integral multicommodity flow is NP-complete, even for only two commodities.

Network flow algorithms can be complicated, and significant engineering is required to optimize performance. Thus, we strongly recommend that you use an existing code rather than implement your own. Excellent codes are available and described below. The two primary classes of algorithms are:

- *Augmenting path methods* – These algorithms repeatedly find a path of positive capacity from source to sink and add it to the flow. It can be shown that the flow through a network is optimal if and only if it contains no augmenting path. Since each augmentation adds something to the flow, we eventually find the maximum. The difference between network-flow algorithms is in *how* they select the augmenting path. If we are not careful, each augmenting path will add but a little to the total flow, and so the algorithm might take a long time to converge.
- *Preflow-push methods* – These algorithms push flows from one vertex to another, initially ignoring the constraint that in-flow must equal out-flow at each vertex. Preflow-push methods prove faster than augmenting-path methods, essentially because multiple paths can be augmented simultaneously. These algorithms are the method of choice and are implemented in the best codes described in the next section.

Implementations: High-performance codes for both maximum flow and minimum cost flow were developed by Andrew Goldberg and his collaborators. The codes `HIPR` and `PRF` [CG94] are provided for maximum flow, with the proviso that `HIPR` is recommended in most cases. For minimum-cost flow, the code of choice is `CS` [Gol97]. Both are written in C and available for noncommercial use from *http://www.avglab.com/andrew/soft.html*.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including several different algorithms for both maximum flow and minimum-cost flow. The same holds true for LEDA, if a commercial-strength solution is needed. See Section 19.1.1 (page 658).

The First DIMACS Implementation Challenge on Network Flows and Matching [JM93] collected several implementations and generators for network flow, which can be obtained by anonymous FTP from *dimacs.rutgers.edu* in the directory *pub/netflow/maxflow*. These include: (1) a preflow-push network flow implementation in C by Edward Rothberg, and (2) an implementation of eleven network flow variants in C, including the older Dinic and Karzanov algorithms by Richard Anderson and Joao Setubal.

Notes: The primary book on network flows and its applications is [AMO93]. Good expositions on network flow algorithms old and new include [CCPS98, CLRS01, PS98]. Expositions on the hardness of multicommodity flow [Ita78] include [Eve79a].

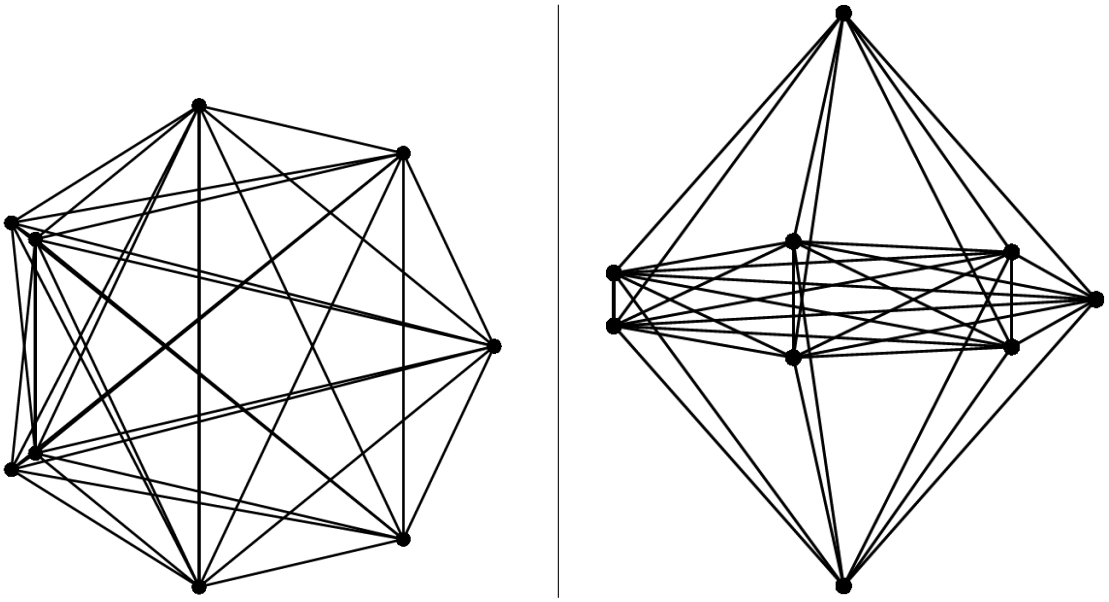
There is a very close connection to maximum flow and edge connectivity in graphs. The fundamental maximum-flow, minimum-cut theorem is due to Ford and Fulkerson

[FF62]. See Section 15.8 (page 505) for simpler and more efficient algorithms to compute the minimum cut in a graph.

Conventional wisdom holds that network flow should be computable in $O(nm)$ time, and there has been steady progress in lowering the time complexity. See [AMO93] for a history of algorithms for the problem. The fastest known general network flow algorithm runs in $O(nm \lg(n^2/m))$ time [GT88]. Empirical studies of minimum-cost flow algorithms include [GKK74, Gol97].

Information flows through a network can be modeled as multicommodity flows through a network, with the observation that replicating and manipulating information at internal nodes can eliminate the need for distinct sources to sink paths when multiple sinks are interested in the same information. The field of *network coding* [YLCZ05] uses such ideas to achieve information flows through networks at the theoretical limits of the max-flow, min-cut theorem.

Related Problems: Linear programming (see page 411), matching (see page 498), connectivity (see page 505).



INPUT OUTPUT

15.10 Drawing Graphs Nicely

Input description: A graph G.

Problem description: Draw a graph G so as to accurately reflect its structure.

Discussion: Graph drawing is a problem that constantly arises in applications, yet it is inherently ill-defined. What exactly does a nice drawing mean? We seek an algorithm that shows off the structure of the graph so the viewer can best understand it. Simultaneously, we seek a drawing that looks aesthetically pleasing.

Unfortunately, these are "soft" criteria for which it is impossible to design an optimization algorithm. Indeed, it is easy to come up with radically different drawings of a given graph, each of which is most appropriate in a certain context. Three different drawings of the Petersen graph are given on page 550. Which of these is the "right" one?

Several "hard" criteria can partially measure the quality of a drawing:

- *Crossings* – We seek a drawing with as few pairs of crossing edges as possible, since they are distracting.

- *Area* – We seek a drawing that uses as little paper as possible, while ensuring that no pair of vertices gets placed too close to each other.
- *Edge length* – We seek a drawing that avoids long edges, since they tend to obscure other features of the drawing.
- *Angular resolution* – We seek a drawing avoiding small angles between two edges incident on a given vertex, as the resulting lines tend to partially or fully overlap.
- *Aspect ratio* – We seek a drawing whose aspect ratio (width/height) reflects the desired output medium (typically a computer screen at 4/3) as closely as possible.

Unfortunately, these goals are mutually contradictory, and the problem of finding the best drawing under any nonempty subset of them is likely to be NP-complete.

Two final warnings before getting down to business. For graphs without inherent symmetries or structure, it is likely that no really nice drawing exists. This is especially for true for graphs with more than 10 to 15 vertices. The shear amount of ink needed to draw any large, dense graph will overwhelm any display. A drawing of the complete graph on 100 vertices (K_{100}) contains approximately 5,000 edges. On a 1000×1000 pixel display, this works out to 200 pixels per edge. What can you hope to see except a black blob in the center of the screen?

Once all this is understood, it must be admitted that graph-drawing algorithms can be quite effective and fun to play with. To help choose the right one, ask yourself the following questions:

- *Must the edges be straight, or can I have curves and/or bends?* – Straight-line drawing algorithms are relatively simple, but have their limitations. Orthogonal polyline drawings seem to work best to visualize complicated graphs such as circuit designs. *Orthogonal* means that all lines must be drawn either horizontal or vertical, with no intermediate slopes. *Polyline* means that each graph edge is represented by a chain of straight-line segments, connected by vertices or bends.
- *Is there a natural, application-specific drawing?* – If your graph represents a network of cities and roads, you are unlikely to find a better drawing than placing the vertices in the same position as the cities on a map. This same principle holds for many different applications.
- *Is your graph either planar or a tree?* – If so, use one of the special planar graph or tree drawing algorithms of Sections 15.11 and 15.12.
- *Is your graph directed?* – Edge direction has a significant impact on the nature of the desired drawing. When drawing directed acyclic graphs (DAGs), it is often important that all edges flow in a logical direction—perhaps left-right or top-down.

- *How fast must your algorithm be?* – Your graph drawing algorithm had better be very fast if it will be used for interactive update and display. You are presumably limited to using incremental algorithms, which change the vertex positions only in the immediate neighborhood of the edited vertex. You can afford more time for optimization if instead you are printing a pretty picture for extended study.

- *Does your graph contain symmetries?* – The output drawing above is attractive because the graph contains symmetries—namely two vertices identically connected to a core K_5. The inherent symmetries in a graph can be identified by computing its *automorphisms*, or self-isomorphisms. Graph isomorphism codes (see Section 16.9 (page 550)) can be readily used to find all automorphisms.

As a first quick and dirty drawing, I recommend simply spacing the vertices evenly on a circle, and then drawing the edges as straight lines between vertices. Such drawings are easy to program and fast to construct. They have the substantial advantage that no two edges will obscure each other, since no three vertices will be collinear. Such artifacts can be hard to avoid as soon as you allow internal vertices into your drawing. An unexpected pleasure with circular drawings is the symmetry sometimes revealed because vertices appear in the order they were inserted into the graph. Simulated annealing can be used to permute the circular vertex order to minimize crossings or edge length, and thus significantly improve the drawing.

A good, general purpose graph-drawing heuristic models the graph as a system of springs and then uses energy minimization to space the vertices. Let adjacent vertices attract each other with a force proportional to (say) the logarithm of their separation, while all nonadjacent vertices repel each other with a force proportional to their separation distance. These weights provide incentive for all edges to be as short as possible, while spreading the vertices apart. The behavior of such a system can be approximated by determining the force acting on each vertex at a particular time and then moving each vertex a small amount in the appropriate direction. After several such iterations, the system should stabilize on a reasonable drawing. The input and output figures above demonstrate the effectiveness of the spring embedding on a particular small graph.

If you need a polyline graph-drawing algorithm, my recommendation is that you study the systems presented next or described in [JM03] to decide whether one of them can do the job. You will have to do a significant amount of work before you can hope to develop a better algorithm.

Drawing your graph opens another can of worms, namely where to place the edge/vertex labels. We seek to position labels very close to the edges or vertices they identify, and yet to place them such that they do not overlap each other or other important graph features. Optimizing label placement can be shown to be an NP-complete problem, but heuristics related to bin packing (see Section 17.9 (page 595)) can be effectively used.

Implementations: GraphViz (*http://www.graphviz.org*) is a popular and well-supported graph-drawing program developed by Stephen North of Bell Laboratories. It represents edges as splines and can construct useful drawings of quite large and complicated graphs. It has sufficed for all of my professional graph-drawing needs over the years.

All of the graph data structure libraries of Section 12.4 (page 381) devote some effort to visualizing graphs. The Boost Graph Library provides an interface to GraphViz instead of reinventing the wheel. The Java graph libraries, most notably JGraphT (*http://jgrapht.sourceforge.net/*), are particularly suitable for interactive applications.

Graph drawing is a problem where very good commercial products exist, including those from Tom Sawyer Software (*www.tomsawyer.com*), yFiles (*www.yworks.com*), and iLOG's JViews (*www.ilog.com/products/jviews/*). Pajek [NMB05] is a package particularly designed for drawing social networks, and available at *http://vlado.fmf.uni-lj.si/pub/networks/pajek/*. All of these have free trial or noncommercial use downloads.

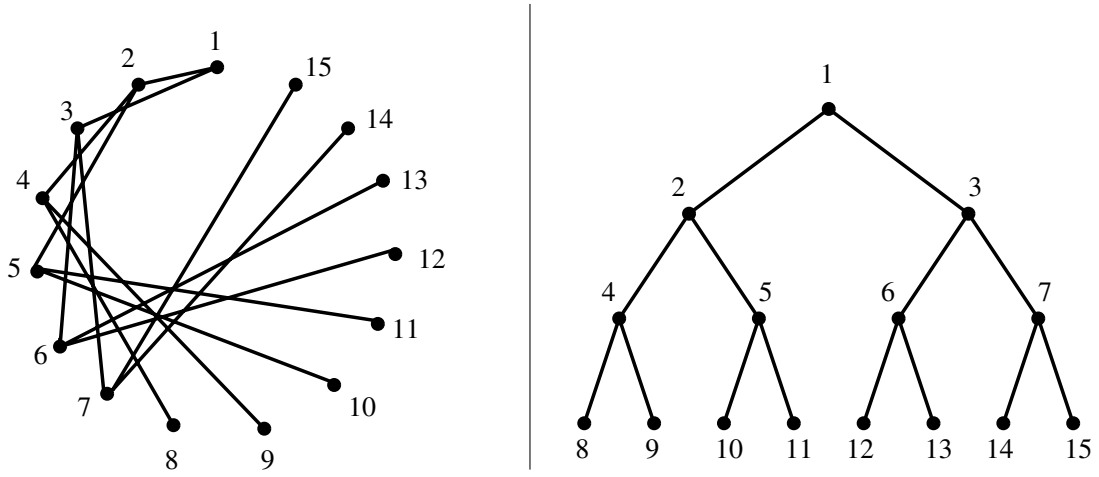
Combinatorica [PS03] provides Mathematica implementations of several graph-drawing algorithms, including circular, spring, and ranked embeddings. See Section 19.1.9 (page 661) for further information on Combinatorica.

Notes: A significant community of researchers in graph drawing exists, fueled by or fueling an annual conference on graph drawing. The proceedings of this conference are published by Springer-Verlag's Lecture Notes in Computer Science series. Perusing a volume of the proceedings will provide a good view of the state-of-the-art and of what kinds of ideas people are thinking about. The forthcoming *Handbook of Graph Drawing and Visualization* [Tam08] promises to be the most comprehensive review of the field.

Two excellent books on graph-drawing algorithms are Battista, et al. [BETT99] and Kaufmann and Wagner [KW01]. A third book by Jünger and Mutzel [JM03] is organized around systems instead of algorithms, but provides technical details about the drawing methods each system employs. Map-labeling heuristics are described in [BDY06, WW95].

It is trivial to space n points evenly along the boundary of a circle. However, the problem is considerably more difficult on the surface of a sphere. Extensive tables of such spherical codes for $n \leq 130$ have been constructed by Hardin, Sloane, and Smith [HSS07].

Related Problems: Drawing trees (see page 517), planarity testing (see page 520).



INPUT OUTPUT

15.11 Drawing Trees

Input description: A tree T, which is a graph without any cycles.

Problem description: Create a nice drawing of the tree T.

Discussion: Many applications require drawing pictures of trees. Tree diagrams are commonly used to display and traverse the hierarchical structure of file system directories. My attempts to Google "tree drawing software" revealed special-purpose applications for visualizing family trees, syntax trees (sentence diagrams), and evolutionary phylogenic trees all in the top twenty links.

Different aesthetics follow from each application. That said, the primary issue in tree drawing is establishing whether you are drawing free or rooted trees:

- *Rooted trees* define a hierarchical order, emanating from a single source node identified as the root. Any drawing should reflect this hierarchical structure, as well as any additional application-dependent constraints on the order in which children must appear. For example, family trees are rooted, with sibling nodes typically drawn from left to right in the order of birth.

- *Free trees* do not encode any structure beyond their connection topology. There is no root associated with the minimum spanning tree (MST) of a graph, so a hierarchical drawing will be misleading. Such free trees might well inherit their drawing from that of the full underlying graph, such as the map of the cities whose distances define the MST.

Trees are always planar graphs, and hence can and should be drawn so no two edges cross. Any of the planar drawing algorithms of Section 15.12 (page 520) could be used to do so. However, such algorithms are overkill, because much simpler algorithms can be used to construct planar drawings of trees. The spring-embedding heuristics of Section 15.10 (page 513) work well on free trees, although they may be too slow for certain applications.

The most natural tree-drawing algorithms assume rooted trees. However, they can be used equally well with free trees, after selecting one vertex to serve as the root of the drawing. This faux-root can be selected arbitrarily, or, even better, by using a *center* vertex of the tree. A center vertex minimizes the maximum distance to other vertices. For trees, the center always consists of either one vertex or two adjacent vertices. This tree center can be identified in linear time by repeatedly trimming all the leaves until only the center remains.

Your two primary options for drawing rooted trees are *ranked* and *radial* embeddings:

- *Ranked embeddings* – Place the root in the top center of your page, and then partition the page into the root-degree number of top-down strips. Deleting the root creates the root-degree number of subtrees, each of which is assigned to its own strip. Draw each subtree recursively, by placing its new root (the vertex adjacent to the old root) in the center of its strip a fixed distance down from the top, with a line from old root to new root. The output figure above is a nicely ranked embedding of a balanced binary tree.

  Such ranked embeddings are particularly effective for rooted trees used to represent a hierarchy—be it a family tree, data structure, or corporate ladder. The top-down distance illustrates how far each node is from the root. Unfortunately, such repeated subdivision eventually produces very narrow strips, until most of the vertices are crammed into a small region of the page. Try to adjust the width of each strip to reflect the total number of nodes it will contain, and don't be afraid of expanding into neighboring region's turf once their shorter subtrees have been completed.

- *Radial embeddings* – Free trees are better drawn using a radial embedding, where the root/center of the tree is placed in the center of the drawing. The space around this center vertex is divided into angular sectors for each subtree. Although the same problem of cramping will eventually occur, radial embeddings make better use of space than ranked embeddings and appear considerably more natural for free trees. The rankings of vertices in terms of distance from the center is illustrated by the concentric circles of vertices.

Implementations: GraphViz (*http://www.graphviz.org*) is a popular and well-supported graph-drawing program developed by Stephen North of Bell Laboratories. It represents edges as splines and can construct useful drawings of quite large

and complicated graphs. It has sufficed for all of my professional graph drawing needs over the years.

Graph/tree drawing is a problem where very good commercial products exist, including those from Tom Sawyer Software (*www.tomsawyer.com*), yFiles (*www.yworks.com*), and iLOG's JViews (*www.ilog.com/products/jviews/*). All of these have free trial or noncommercial use downloads.

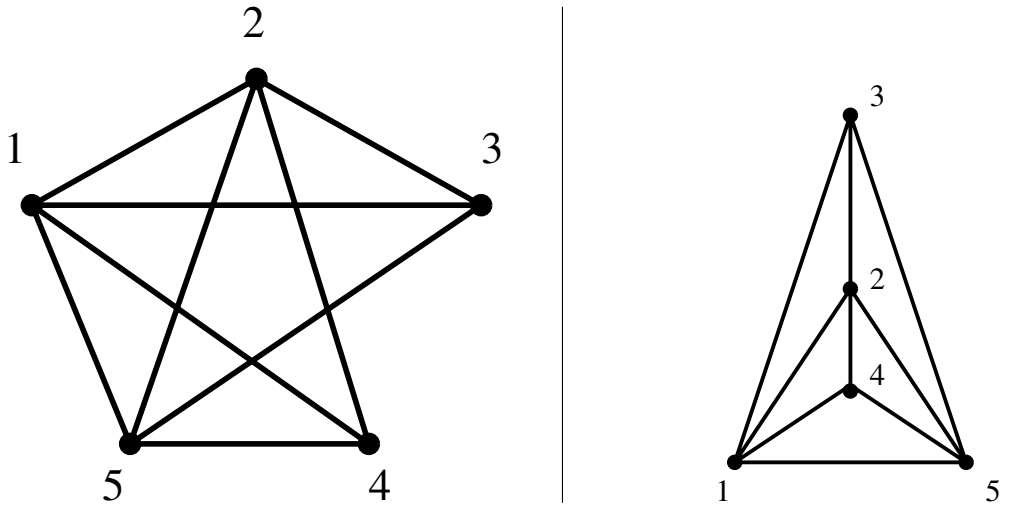
Combinatorica [PS03] provides Mathematica implementations of several tree drawing algorithms, including radial and rooted embeddings. See Section 19.1.9 (page 661) for further information on Combinatorica.

Notes: All books and surveys on graph drawing include discussions of specific tree-drawing algorithms. The forthcoming *Handbook of Graph Drawing and Visualization* [Tam08] promises to be the most comprehensive review of the field. Two excellent books on graph drawing algorithms are Battista, et al. [BETT99] and Kaufmann and Wagner [KW01]. A third book by Jünger and Mutzel [JM03] is organized around systems instead of algorithms, but provides technical details about the drawing methods each system employs.

Heuristics for tree layout have been studied by several researchers [RT81, Moe90], with Buchheim, et al. [BJL06] reflective of the state-of-the-art. Under certain aesthetic criteria, the problem is NP-complete [SR83].

Certain tree layout algorithms arise from non-drawing applications. The Van Emde Boas layout of a binary tree offers better external memory performance than conventional binary search, at a cost of greater complexity. See the survey of Arge, et al. [ABF05] for more on this and other cache-oblivious data structures.

Related Problems: Drawing graphs (see page 513), planar drawings (see page 520).



INPUT OUTPUT

15.12 Planarity Detection and Embedding

Input description: A graph G.

Problem description: Can G be drawn in the plane such that no two edges cross? If so, produce such a drawing.

Discussion: Planar drawings (or *embeddings*) make clear the structure of a given graph by eliminating crossing edges, which can be confused as additional vertices. Graphs defined by road networks, printed circuit board layouts, and the like are inherently planar because they are completely defined by surface structures.

Planar graphs have a variety of nice properties that can be exploited to yield faster algorithms for many problems. The most important fact to know is that every planar graph is *sparse*. Euler's formula shows that $|E| \leq 3|V| - 6$ for every nontrivial planar graph $G = (V, E)$. This means that every planar graph contains a linear number of edges, and further that every planar graph contains a vertex of degree ≤ 5. Every subgraph of a planar graph is planar, so there must always a sequence of low-degree vertices to delete from G, reducing it to the empty graph.

To gain a better appreciation of the subtleties of planar drawings, I encourage the reader to construct a planar (noncrossing) embedding for the graph $K_5 - e$, shown on the input figure above. Then try to construct such an embedding where all the edges are straight. Finally, add the missing edge to the graph and try to do the same for K_5 itself.

The study of planarity has motivated much of the development of graph theory. It must be confessed, however, that the need for planarity testing arises relatively infrequently in applications. Most graph-drawing systems do not explicitly seek planar embeddings. *"Planarity Detection"* proved to be among the least frequently hit pages of the Algorithm Repository (*http://www.cs.sunysb.edu/∼algorith*) [Ski99]. That said, it is still very useful to know how to deal with planar graphs when you encounter them.

Thus, it pays to distinguish the problem of planarity testing (does a graph have a planar drawing?) from constructing planar embeddings (actually finding the drawing), although both can be done in linear time. Many efficient planar graph algorithms do not make any use of the drawing, but instead exploit the low-degree deletion sequence described above.

Algorithms for planarity testing begin by embedding an arbitrary cycle from the graph in the plane and then considering additional paths in G, connecting vertices on this cycle. Whenever two such paths cross, one must be drawn outside the cycle and one inside. When three such paths mutually cross, there is no way to resolve the problem, so the graph cannot be planar. Linear-time algorithms for planarity detection are based on depth-first search, but they are subtle and complicated enough that you are wise to seek an existing implementation.

Such path-crossing algorithms can be used to construct a planar embedding by inserting the paths into the drawing one by one. Unfortunately, because they work in an incremental manner, nothing prevents them from inserting many vertices and edges into a relatively small area of the drawing. Such cramping is a major problem, for it leads to ugly drawings that are hard to understand. Better algorithms have been devised that construct *planar-grid embeddings*, where each vertex lies on a $(2n - 4) \times (n - 2)$ grid. Thus, no region can get too cramped and no edge can get too long. Still, the resulting drawings tend not to look as natural as one might hope.

For nonplanar graphs, what is often sought is a drawing that minimizes the number of crossings. Unfortunately, computing the crossing number of a graph is NP-complete. A useful heuristic extracts a large planar subgraph of G, embeds this subgraph, and then inserts the remaining edges one by one to minimize the number of crossings. This won't do much for dense graphs, which are doomed to have a large number of crossings, but it will work well for graphs that are almost planar, such as road networks with overpasses or printed circuit boards with multiple layers. Large planar subgraphs can be found by modifying planarity-testing algorithms to delete troublemaking edges when encountered.

Implementations: LEDA (see Section 19.1.1 (page 658)) includes linear-time algorithms for both planarity testing and constructing straight-line planar-grid embeddings. Their planarity tester returns an obstructing Kuratowski subgraph (see notes) for any graph deemed nonplanar, yielding concrete proof of its nonplanarity.

JGraphEd (*http://www.jharris.ca/JGraphEd/*) is a Java graph-drawing framework that includes several planarity testing/embedding algorithms, including both the Booth-Lueker PQ-tree algorithm and the modern straight-line grid embedding.

PIGALE (*http://pigale.sourceforge.net/*) is a C++ graph editor/algorithm library focusing on planar graphs. It contains a variety of algorithms for constructing planar drawings as well as efficient algorithms to test planarity and identify an obstructing subgraph ($K_{3,3}$ or K_5), if one exists.

Greedy randomized adaptive search (GRASP) heuristics for finding the largest planar subgraph have been implemented by Ribeiro and Resende [RR99] as Algorithm 797 of the *Collected Algorithms of the ACM* (see Section 19.1.6 (page 659)). These Fortran codes are also available from *http://www.research.att.com/∼mgcr/src/*.

Notes: Kuratowski [Kur30] gave the first characterization of planar graphs, namely that they do not contain a subgraph homeomorphic to $K_{3,3}$ or K_5. Thus, if you are still working on the exercise to embed K_5, now is an appropriate time to give it up. Fary's theorem [FÁ48] states that every planar graph can be drawn in such a way that each edge is straight.

Hopcroft and Tarjan [HT74] gave the first linear-time algorithm for drawing graphs. Booth and Lueker [BL76] developed an alternate planarity-testing algorithm based on PQ-trees. Simplified planarity-testing algorithms include [BCPB04, MM96, SH99]. Efficient $2n \times n$ planar grid embeddings were first developed by [dFPP90]. The book by Nishizeki and Rahman [NR04] provide a good overview of the spectrum of planar drawing algorithms.

Outerplanar graphs are those that can be drawn so all vertices lie on the outer face of the drawing. Such graphs can be characterized as having no subgraph homeomorphic to $K_{2,3}$ and can be recognized and embedded in linear time.

Related Problems: Graph partition (see page 541), drawing trees (see page 517).