16

Graph Problems: Hard Problems

A cynical view of graph algorithms is that "everything we want to do is hard." Indeed, no polynomial-time algorithms are known for any of the problems in this section. All of them are provably NP-complete with the exception of graph isomorphism—whose complexity status remains an open question.

The theory of NP-completeness demonstrates that *all* NP-complete problems must have polynomial-time algorithms if *any* one of them does. This prospect is sufficiently preposterous that an NP-completeness reduction suffices as de facto proof that no efficient algorithm exists to solve the given problem.

Still, do not abandon hope if your problem resides in this chapter. We provide a recommended line of attack for each problem, be it combinatorial search, heuristics, approximation algorithms, or algorithms for restricted instances. Hard problems require a different methodology to work with than polynomial-time problems, but with care they can usually be dealt with successfully.

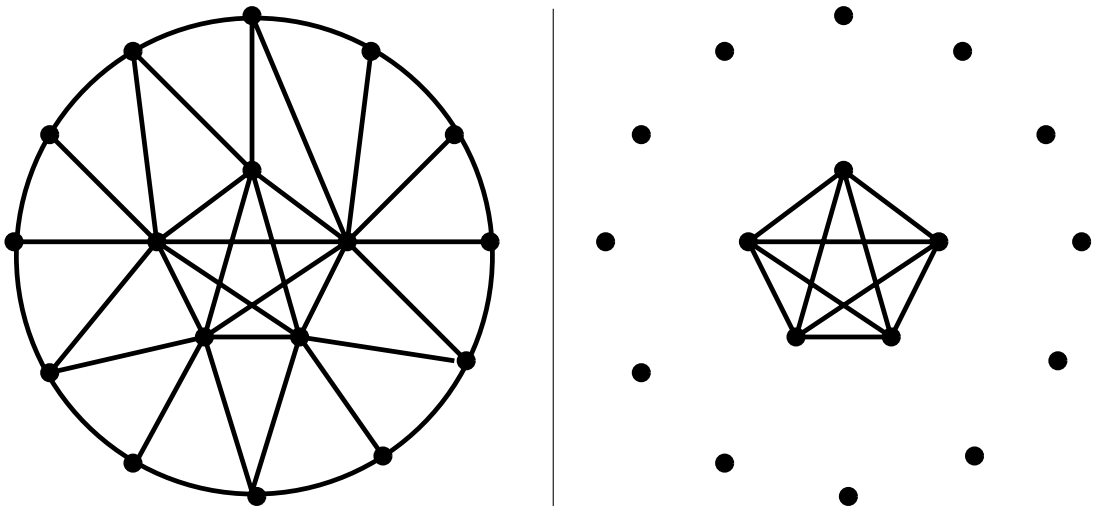
The following books will help you deal with NP-complete problems:

- *Garey and Johnson* [GJ79] – This is the classic reference on the theory of NP-completeness. Most notably, it contains a concise catalog of over 400 NP-complete problems, with associated references and comments. Browse through the catalog as soon as you question the existence of an efficient algorithm for your problem. Indeed, this is the single book in my library that I reach for most often.

- *Crescenzi and Kann* [ACG+03] – This book serves as the "Garey and Johnson" for the world of approximation algorithms. Its reference section, *The Compendium of NP Optimization Problems*, is maintained online at *www.nada.kth.se/∼viggo/problemlist/* and should be the first place to look for a provably good heuristic for any given problem.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_16,

- *Vazirani* [Vaz04] – A complete treatment of the theory of approximation algorithms by a highly regarded researcher in the field.
- *Hochbaum* [Hoc96] – This nice book was the first survey of approximation algorithms for NP-complete problems, but rapid developments have left it somewhat dated.
- *Gonzalez* [Gon07] – This *Handbook of Approximation Algorithms and Metaheuristics* contains current surveys on a variety of techniques for dealing with hard problems, both applied and theoretical.



INPUT OUTPUT

16.1 Clique

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

Problem description: What is the largest $S \subset V$ such that for all $x, y \in S$, $(x, y) \in E$?

Discussion: In high school, everybody complained about the "clique,"—a group of friends who all hung around together and seemed to dominate everything social. Consider a graph representing the school's social network. Vertices correspond to people, with edges between any pair of people who are friends. Thus, the high school clique defines a (complete subgraph) clique in this friendship graph.

Identifying "clusters" of related objects often reduces to finding large cliques in graphs. An interesting example arose in a program the Internal Revenue Service (IRS) developed to detect organized tax fraud. In this scam, large groups of phony tax returns are submitted in the hopes of getting undeserved refunds. But generating large numbers of *different* phony tax returns is hard work. The IRS constructs graphs with vertices corresponding to submitted tax forms and edges between any two forms that appear suspiciously similar. Any large clique in this graph points to fraud.

Since any edge in a graph represents a clique of two vertices, the challenge lies not in finding a clique, but in finding a large clique. And it is indeed a challenge, for finding a maximum clique is NP-complete. To make matters worse, it is provably

hard to approximate even to within a factor of $n^{1/2-\epsilon}$. Theoretically, clique is about as hard as a problem in this book can get. So what can we hope to do about it?

- *Will a maximal clique suffice?* – A *maximal* clique is a clique that cannot be enlarged by adding any additional vertex. This doesn't mean that it has to be large relative to the largest possible clique, but it might be. To find a nice maximal (and hopefully large) clique, sort the vertices from highest degree to lowest degree, put the first vertex in the clique, and then test each of the other vertices to see whether it is adjacent to all the clique vertices added thus far. If so, add it; if not, continue down the list. By using a bit vector to mark which vertices are currently in the clique, this can be made to run in $O(n+m)$ time. An alternative approach might incorporate some randomness into the vertex ordering, and accept the largest maximal clique you find after a certain number of trials.

- *What if I will settle for a large, dense subgraph?* – Insisting on cliques to define clusters in a graph can be risky, since a single missing edge will eliminate a vertex from consideration. Instead, we should seek large *dense* subgraphs—i.e. , subsets of vertices that contain a large number of edges between them. Cliques are, by definition, the densest subgraphs possible.

  The largest set of vertices whose induced (defined) subgraph has minimum vertex degree $\geq k$ can be found with a simple linear-time algorithm. Begin by deleting all the vertices whose degree is less than k. This may reduce the degree of other vertices below k, if they were adjacent to sufficiently deleted low-degree vertices. Repeating this process until all remaining vertices have degree $\geq k$ constructs the largest high-degree subgraph. This algorithm can be implemented in $O(n+m)$ time by using adjacency lists and the constant-width priority queue of Section 12.2 (page 373). If we continue to delete the lowest-degree vertices, we eventually end up with a clique or set of cliques, – but they may be as small as two vertices.

- *What if the graph is planar?* – Planar graphs cannot have cliques of a size larger than four, or else they cease to be planar. Since each edge defines a clique of size 2, the only interesting cases are cliques of three and four vertices. Efficient algorithms to find such small cliques consider the vertices from lowest to highest degree. Any planar graph must contain a vertex of at most 5 degrees (see Section 15.12 (page 520)), so there is only a constant-sized neighborhood to check exhaustively for a clique containing it. We then delete this vertex to leave a smaller planar graph, containing another low-degree vertex. Repeat this check and delete processes until the graph is empty.

If you *really* need to find the largest clique in a graph, an exhaustive search via backtracking provides the only real solution. We search through all k-subsets of the vertices, pruning a subset as soon as it contains a vertex that is not adjacent to all the rest. A simple upper bound on the maximum clique in G is the highest vertex

degree plus 1. A better upper bound comes from sorting the vertices in order of decreasing degree. Let j be the largest index such that degree of the jth vertex is at least $j-1$. The largest clique in the graph contains no more than j vertices, since no vertex of degree $< (j-1)$ can appear in a clique of size j. To speed our search, we should delete all such useless vertices from G.

Heuristics for finding large cliques based on randomized techniques such as simulated annealing are likely to work reasonably well.

Implementations: `Cliquer` is a set of C routines for finding cliques in arbitrary weighted graphs by Patric Östergard. It uses an exact branch-and-bound algorithm, and is available at *http://users.tkk.fi/~pat/cliquer.html*.

Programs for finding cliques and independent sets were sought for the Second DIMACS Implementation Challenge [JT96]. Programs and data from the challenge are available by anonymous FTP from *dimacs.rutgers.edu*. Source codes are available under *pub/challenge/graph* and test data under *pub/djs*. `dfmax.c` implements a simple-minded branch-and-bound algorithm similar to [CP90]. `dmclique.c` uses a "semi-exhaustive greedy" scheme for finding large independent sets from [JAMS91].

Kreher and Stinson [KS99] provide branch-and-bound programs in C for finding the maximum clique using a variety of lower-bounds, available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements branch-and-bound algorithms for finding large cliques. They claim to be able to work with graphs as large as 150 to 200 vertices.

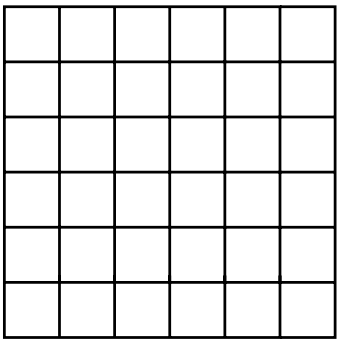
Notes: Bomze, et al. [BBPP99] give the most comprehensive survey on the problem of finding maximum cliques. Particularly interesting is the work from the operations research community on branch-and-bound algorithms for finding cliques effectively. More recent experimental results are reported in [JS01].

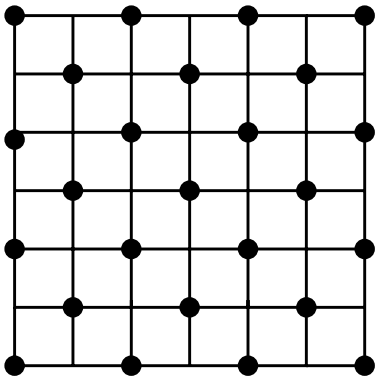
The proof that clique is NP-complete is due to Karp [Kar72]. His reduction (given in Section 9.3.3 (page 327)) established that clique, vertex cover, and independent set are very closely related problems, so heuristics and programs that solve one of them should also produce reasonable solutions for the other two.

The *densest subgraph problem* seeks the subset of vertices whose induced subgraph has the highest average vertex degree. A clique of k vertices is clearly the densest subgraph of its size, but larger, noncomplete subgraphs may achieve higher average degree. The problem is NP-complete, but simple heuristics based on repeatedly deleting the lowest-degree vertex achieve reasonable approximation ratios [AITT00]. See [GKT05] for an interesting application of densest subgraph, namely detecting link spam on the Web.

That clique cannot be approximated to within a factor of $n^{1/2-\epsilon}$ unless $P = NP$ (and $n^{1-\epsilon}$ under weaker assumptions) is shown in [Has82].

Related Problems: Independent set (see page 528), vertex cover (see page 530).





INPUT OUTPUT

16.2 Independent Set

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

Problem description: What is the largest subset S of vertices of V such that for each edge $(x, y) \in E$, either $x \notin E$ or $y \notin E$?

Discussion: The need to find large independent sets arises in facility dispersion problems, where we seek a set of mutually separated locations. It is important that no two locations of our new "McAlgorithm" franchise service be placed close enough to compete with each other. We can construct a graph where the vertices are the set of possible locations, and then add edges between any two locations deemed close enough to interfere. The maximum independent set gives the largest number of franchises we can sell without cannibalizing sales.

Independent sets (also known as *stable sets*) avoid conflicts between elements, and hence arise often in coding theory and scheduling problems. Define a graph whose vertices represent the set of possible code words, and add edges between any two code words sufficiently similar to be confused due to noise. The maximum independent set of this graph defines the highest capacity code for the given communication channel.

Independent set is closely related to two other NP-complete problems:

- *Clique* – Watch what you say, for a clique is what you get if you give an independent set a complement. The *complement* of $G = (V, E)$ is a graph $G' = (V, E')$ where $(i, j) \in E'$ iff (i, j) is not in E. In other words, we replace each edge by a non-edge and vice versa. The maximum independent set in G is exactly the maximum clique in G', so the two problems are algorithmically

identical. Thus, the algorithms and implementations in Section 16.1 (page 525) can easily be used for independent set.

- *Vertex coloring* – The vertex coloring of a graph $G = (V, E)$ is a partition of V into a small number of sets (colors), where no two vertices of the same color can have an edge between them. Each color class defines an independent set. Many scheduling applications of independent set are really coloring problems, since all tasks eventually must be completed.

  Indeed, one heuristic to find a large independent set is to use any vertex coloring algorithm/heuristic, and take the largest color class. One consequence of this observation is that all graphs with small chromatic numbers (such as planar and bipartite graphs) have large independent sets.

The simplest reasonable heuristic is to find the lowest-degree vertex, add it to the independent set, and then delete it and all vertices adjacent to it. Repeating this process until the graph is empty gives a *maximal* independent set, in that it can't be made larger by just adding vertices. Using randomization or perhaps some degree of exhaustive search might result in somewhat larger independent sets.

The independent set problem is in some sense dual to the graph-matching problem. The former asks for a large set of vertices with no edge in common, while the latter asks for a large set of edges with no vertex in common. This suggests trying to rephrase your problem as an efficiently-computable matching problem instead of maximum independent set problem, which is NP-complete.

The maximum independent set of a tree can be found in linear time by (1) stripping off the leaf nodes, (2) adding them to the independent set, (3) deleting all adjacent nodes, and then (4) repeating from the first step on the resulting trees until it is empty.

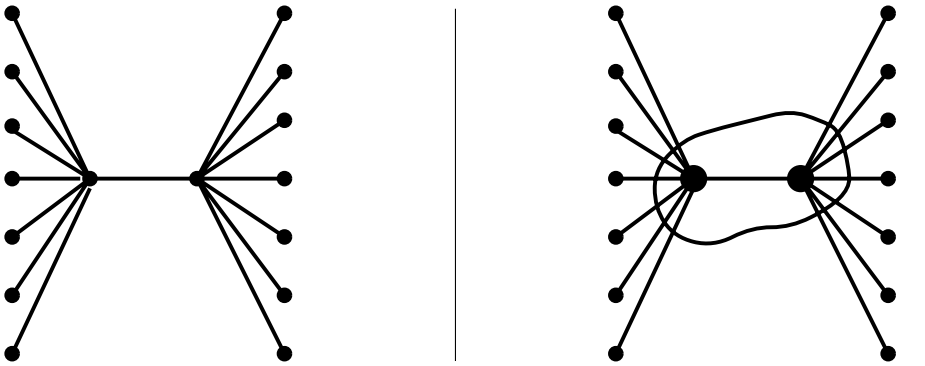
Implementations: Any program for computing the maximum clique in a graph can find maximum independent sets by just complementing the input graph. Therefore, we refer the reader to the clique-finding programs of Section 16.1 (page 525).

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements a branch-and-bound algorithm for finding independent sets (called stable sets in the manual).

Greedy randomized adaptive search (GRASP) heuristics for independent set set have been implemented by Resende, et al. [RFS98] as Algorithm 787 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: The proof that independent set is NP-complete is due to Karp [Kar72]. It remains NP-complete for planar cubic graphs [GJ79]. Independent set can be solved efficiently for bipartite graphs [Law76]. This is not trivial—indeed the larger of the "part" of a bipartite graph is not necessarily its maximum independent set.

Related Problems: Clique (see page 525), vertex coloring (see page 544), vertex cover (see page 530).



INPUT OUTPUT

16.3 Vertex Cover

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

Problem description: What is the smallest subset of $S \subset V$ such that each edge $(x, y) \in E$ contains at least one vertex of S?

Discussion: Vertex cover is a special case of the more general *set cover* problem, which takes as input an arbitrary collection of subsets $S = (S_1, \ldots, S_n)$ of the universal set $U = \{1, \ldots, m\}$. We seek the smallest subset of subsets from S whose union is U. Set cover arises in many applications associated with buying things sold in fixed lots or assortments. See Section 18.1 (page 621) for a discussion of set cover.

To turn vertex cover into a set cover problem, let universal set U represent the set E of edges from G, and define S_i to be the set of edges incident on vertex i. A set of vertices defines a vertex cover in graph G iff the corresponding subsets define a set cover in this particular instance. However, since each edge can be in only two different subsets, vertex cover instances are simpler than general set cover. Vertex cover is a relative lightweight among NP-complete problems, and can be more effectively solved than general set cover.

Vertex cover and independent set are very closely related graph problems. Since every edge in E is (by definition) incident on a vertex in any cover S, there can be no edge both endpoints are in $V - S$. Thus, $V - S$ must be an independent set. Since minimizing S is the same as maximizing $V - S$, the problems are equivalent. This means that any independent set solver can be applied to vertex cover as well. Having two ways of looking at your problem is helpful, since one may appear easier in a given context.

The simplest heuristic for vertex cover selects the vertex with highest degree, adds it to the cover, deletes all adjacent edges, and then repeats until the graph is empty. With the right data structures, this can be done in linear time, and should "usually" produce a "pretty good" cover. However, this cover might be $\lg n$ times worse than the optimal cover for certain input graphs.

Fortunately, we can always find a vertex cover whose size is at most twice as large as optimal. Find a *maximal* matching M in the graph—i.e. , a set of edges no two of which share a vertex in common and which cannot be enlarged by adding additional edges. Such a maximal matching can be constructed incrementally, by picking an arbitrary edge e in the graph, deleting any edge sharing a vertex with e, and repeating until the graph is out of edges. Taking *both* of the vertices for each edge in a maximal matching gives us a vertex cover. Why? Because *any* vertex cover must contain *at least* one of the two vertices in each matching edge just to cover the edges of M, this cover is at most twice as large as the minimum cover.

This heuristic can be tweaked to perform somewhat better in practice, if not in theory. We can select the matching edges to "kill off" as many other edges as possible, which should reduce the size of the maximal matching and hence the number of pairs of vertices in the vertex cover. Also, some of the vertices from M may in fact not be necessary, since all of their incident edges might also have been covered using other selected vertices. We can identify and delete these losers by making a second pass through our cover.

The vertex cover problem seeks to cover all edges using few vertices. Two other important problems have similar sounding objectives:

- *Cover all vertices using few vertices* – The *dominating set* problem seeks the smallest set of vertices D such that every vertex in $V - D$ is adjacent to at least one vertex in the dominating set D. Every vertex cover of a nontrivial connected graph is also a dominating set, but dominating sets can be much smaller. Any single vertex represents the minimum dominating set of complete graph K_n, while $n-1$ vertices are needed for a vertex cover. Dominating sets tend to arise in communications problems, since they represent the hubs or broadcast centers sufficient to communicate with all sites/users.

  Dominating set problems can be easily expressed as instances of set cover (see Section 18.1 (page 621)). Each vertex v_i defines a subset of vertices consisting of itself plus all the vertices it is adjacent to. The greedy set cover heuristic running on this instance yields a $\Theta(\lg n)$ approximation to the optimal dominating set.

- *Cover all vertices using few edges* – The *edge cover* problem seeks the smallest set of edges such that each vertex is included in one of the edges. In fact, edge cover can be solved efficiently by finding a maximum cardinality matching (see Section 15.6 (page 498)) and then selecting arbitrary edges to account for the unmatched vertices.

Implementations: Any program for computing the maximum clique in a graph can be applied to vertex cover by complementing the input graph and selecting the vertices which do not appear in the clique. Therefore, we refer the reader to check out the clique-finding programs of Section 16.1 (page 525).

`COVER` [RHG07] is a very effective vertex cover solver based on a stochastic local search algorithm. It is available at *http://www.nicta.com.au/people/richters/*.

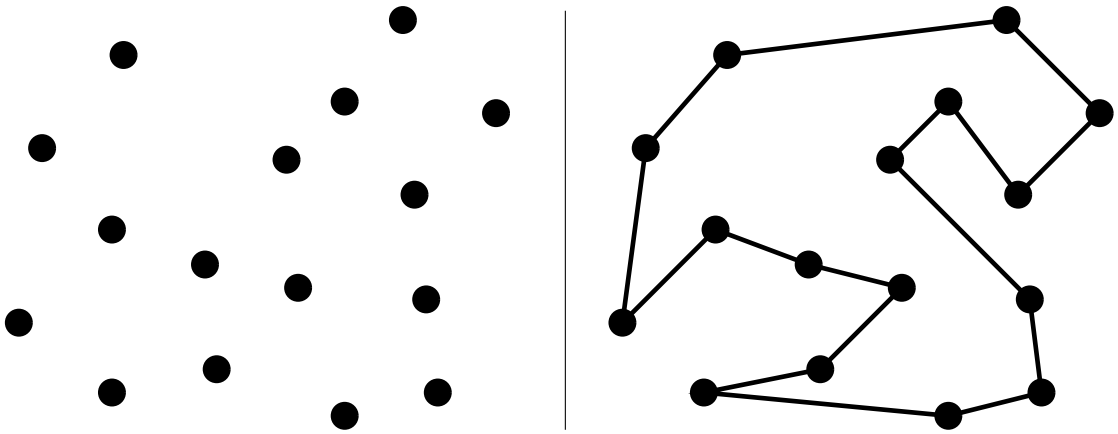
JGraphT (*http://jgrapht.sourceforge.net/*) is a Java graph library that contains greedy and 2-approximate heuristics for vertex cover.

Notes: Karp [Kar72] first proved that vertex-cover is NP-complete. Several different heuristics yield 2-approximation algorithms for vertex cover, including randomized rounding. Good expositions on these 2-approximation algorithms include [CLRS01, Hoc96, Pas97, Vaz04]. The example that the greedy algorithm can be as bad as $\lg n$ times optimal is due to [Joh74] and presented in [PS98]. Experimental studies of vertex cover heuristics include [GMPV06, GW97, RHG07].

Whether there exists a better than 2-factor approximation for vertex cover is one of the major open problems in approximation algorithms. Hastad [Has97] proved there does not exist a better than 1.1666-factor approximation algorithm for vertex cover.

The primary reference on dominating sets is the monograph of Haynes et al. [HHS98]. Heuristics for the connected dominating set problem are presented in [GK98]. Dominating set cannot be approximated to better than the $\Omega(\lg n)$ factor [ACG+03] of set cover.

Related Problems: Independent set (see page 528), set cover (see page 621).



INPUT OUTPUT

16.4 Traveling Salesman Problem

Input description: A weighted graph G.

Problem description: Find the cycle of minimum cost, visiting each vertex of G exactly once.

Discussion: The traveling salesman problem is the most notorious NP-complete problem. This is a function both of its general usefulness and the ease with which it can be explained to the public at large. Imagine a traveling salesman planning a car trip to visit a set of cities. What is the shortest route that will enable him to do so and return home, thus minimizing his total driving?

The traveling salesman problem arises in many transportation and routing problems. Other important applications involve optimizing tool paths for manufacturing equipment. For example, consider a robot arm assigned to solder all the connections on a printed circuit board. The shortest tour that visits each solder point exactly once defines the most efficient route for the robot.

Several issues arise in solving TSPs:

- *Is the graph unweighted?* – If the graph is unweighted, or all the edges have one of two possible cost values, the problem reduces to finding a *Hamiltonian cycle*. See Section 16.5 (page 538) for a discussion of this problem.

- *Does your input satisfy the triangle inequality?* – Our sense of how proper distance measures behave is captured by the *triangle inequality*. This property states that $d(i, j) \leq d(i, k) + d(k, j)$ for all vertices $i, j, k \in V$. Geometric

distances all satisfy the triangle inequality because the shortest distance between two points is as the crow flies. Commercial air fares do *not* satisfy the triangle inequality, which is why it is so hard to find the cheapest airfare between two points. TSP heuristics work much better on sensible graphs that do obey the triangle inequality.

- *Are you given n points as input or a weighted graph?* – Geometric instances are often easier to work with than a graph representation. Since pair of points define a complete graph, there is never an issue of finding a feasible tour. We can save space by computing these distances on demand, thus eliminating the need to store an $n \times n$ distance matrix. Geometric instances inherently satisfy the triangle inequality, so they can exploit performance guarantees from certain heuristics. Finally, one can take advantage of geometric data structures like kd-trees to quickly identify close unvisited sites.

- *Can you visit a vertex more than once?* – The restriction that the tour not revisit any vertex is irrelevant in many applications. In air travel, the cheapest way to visit all vertices might involve repeatedly visiting an airport hub. Note that this issue does not arise when the input observes the triangle inequality.

  TSP with repeated vertices is easily solved by using any conventional TSP code on a new cost matrix D, where $D(i, j)$ is the shortest path distance from i to j. This matrix can be constructed by solving an all-pairs shortest path (see Section 15.4 (page 489)) and satisfies the triangle inequality.

- *Is your distance function symmetric?* – A distance function is *asymmetric* when there exists x, y such that $d(x, y) \neq d(y, x)$. The asymmetric traveling salesman problem (ATSP) is much harder to solve in practice than symmetric (STSP) instances. Try to avoid such pathological distance functions. Be aware that there is a reduction converting ATSP instances to symmetric instances containing twice as many vertices [GP07], that may be useful because symmetric solvers are so much better.

- *How important is it to find the optimal tour?* – Usually heuristic solutions will suffice for applications. There are two different approaches if you insist on solving your TSP to optimality, however. *Cutting plane methods* model the problem as an integer program, then solve the linear programming relaxation of it. Additional constraints designed to force integrality are added if the optimal solution is not at an integer point. *Branch-and-bound algorithms* perform a combinatorial search while maintaining careful upper and lower bounds on the cost of a tour. In the hands of professionals, problems with thousands of vertices can be solved. Maybe you can too, if you use the best solver available.

Almost any flavor of TSP is going to be NP-complete, so the right way to proceed is with heuristics. These typically come within a few percent of the optimal

solution, which is close enough for engineering work. Unfortunately, literally dozens of heuristics have been proposed for TSP, so the situation can be confusing. Empirical results in the literature are sometime contradictory. However, we recommend choosing from among the following heuristics:

- *Minimum spanning trees* – This heuristic starts by finding the minimum spanning tree (MST) of the sites, and then does a depth-first search of the resulting tree. In the course of DFS, we walk over each of the $n-1$ edges exactly twice: once going down to discover a new vertex, and once going up when we backtrack. Now define a tour by ordering the vertices by when they were discovered. If the graph obeys the triangle inequality, the resulting tour is at most twice the length of the optimal TSP tour. In practice, it is usually better, typically 15% to 20% over optimal. Furthermore, the running time is bounded by that of computing the MST, which is only $O(n \lg n)$ in the case of points in the plane (see Section 15.3 (page 484)).

- *Incremental insertion methods* – A different class of heuristics starts from a single vertex, and then inserts new points into this partial tour one at a time until the tour is complete. The version of this heuristic that seems to work best is *furthest point* insertion: of all remaining points, insert the point v into a partial tour T such that

  $$\max_{v \in V} \min_{i=1}^{|T|} (d(v, v_i) + d(v, v_{i+1}))$$

  The "min" ensures that we insert the vertex in the position that adds the smallest amount of distance to the tour, while the "max" ensures that we pick the worst such vertex first. This seems to work well because it "roughs out" a partial tour first before filling in details. Such tours are typically only 5% to 10% longer than optimal.

- *K-optimal tours* – Substantially more powerful are the Kernighan-Lin, or *k-opt*, class of heuristics. The method applies local refinements to an initially arbitrary tour in the hopes of improving it. In particular, subsets of k edges are deleted from the tour and the k remaining subchains rewired to form a different tour with hopefully a better cost. A tour is k-optimal when no subset of k edges can be deleted and rewired to reduce the cost of the tour. Two-opting a tour is a fast and effective way to improve any other heuristic. Extensive experiments suggest that 3-optimal tours are usually within a few percent of the cost of optimal tours. For $k > 3$, the computation time increases considerably faster than the solution quality. Simulated annealing provides an alternate mechanism to employ edge flips to improve heuristic tours.

Implementations: Concorde is a program for the symmetric traveling salesman problem and related network optimization problems, written in ANSI C. This

world record-setting program by Applegate, Bixby, Chvatal, and Cook [ABCC07] has obtained the optimal solutions to 106 of TSPLIB's 110 instances; the largest of which has 15,112 cities. Concorde is available for academic research use from *http://www.tsp.gatech.edu/concorde*. It is the clear choice among available TSP codes. Their *http://www.tsp.gatech.edu/* website features very interesting material on the history and applications of TSP.

Lodi and Punnen [LP07] put together an excellent survey of available software for solving TSP. Current links to all programs mentioned are maintained at *http://www.or.deis.unibo.it/research_pages/tspsoft.html*.

TSPLIB [Rei91] provides the standard collection of hard instances of TSPs that arise in practice. The best-supported version of TSPLIB is available from *http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/*, although the instances are also available from Netlib (see Section 19.1.5 (page 659)).

Tsp_solve is a C++ code by Chad Hurwitz and Robert Craig that provides both heuristic and optimal solutions. Geometric problems of size up to 100 points are manageable. It is available from *http://www.cs.sunysb.edu/∼algorith* or by e-mailing Chad Hurrwitz at churritz@cts.com. GOBLIN (*http://www.math.uni-augsburg.de/∼fremuth/goblin.html*) implements branch-and-bound algorithms for both symmetric and asymmetric TSP, as well as a variety of heuristics.

Algorithm 608 [Wes83] of the *Collected Algorithms of the ACM* is a Fortran implementation of a heuristic for the quadratic assignment problem—a more general problem that includes the traveling salesman as a special case. Algorithm 750 [CDT95] is a Fortran code for the exact solution of asymmetric TSP instances. See Section 19.1.5 (page 659) for details.

Notes: The book by Applegate, Bixby, Chvatal, and Cook [ABCC07] documents the techniques they used in their record-setting TSP solvers, as well as the theory and history behind the problem. Gutin and Punnen [GP07] now offer the best reference on all aspects and variations of the traveling salesman problem, displacing an older but much beloved book by Lawler et al. [LLKS85].

Experimental results on heuristic methods for solving large TSPs include [Ben92a, GBDS80, Rei94]. Typically, it is possible to get within a few percent of optimal with such methods.

The Christofides heuristic [Chr76] is an improvement over the minimum-spanning tree heuristic and guarantees a tour whose cost is at most 3/2 times optimal on Euclidean graphs. It runs in $O(n^3)$, where the bottleneck is the time it takes to find a minimum-weight perfect matching (see Section 15.6 (page 498)). The minimum spanning tree heuristic is due to [RSL77].

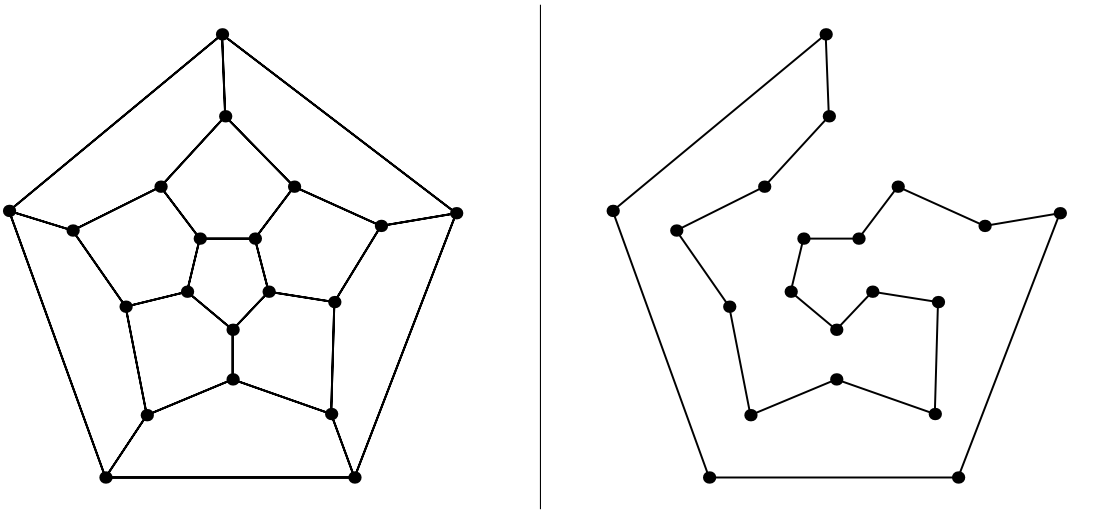
Polynomial-time approximation schemes for Euclidean TSP have been developed by Arora [Aro98] and Mitchell [Mit99], which offer $1 + \epsilon$ factor approximations in polynomial time for any $\epsilon > 0$. They are of great theoretical interest, although any practical consequences remain to be determined.

The history of progress on optimal TSP solutions is inspiring. In 1954, Dantzig, Fulkerson, and Johnson solved a symmetric TSP instance of 42 United States cities [DFJ54]. In 1980, Padberg and Hong solved an instance on 318 vertices [PH80]. Applegate et al. [ABCC07] have recently solved problems that are twenty times larger than this. Some of

this increase is due to improved hardware, but most is due to better algorithms. The rate of growth demonstrates that exact solutions to NP-complete problems can be obtained for large instances if the stakes are high enough. Fortunately or unfortunately, they seldom are.

Size is not the only criterion for hard instances. One can easily construct an enormous graph consisting of one cheap cycle, for which it would be easy to find the optimal solution. For sets of points in convex position in the plane, the minimum TSP tour is described by its convex hull (see Section 17.2 (page 568)), which can be computed in $O(n \lg n)$ time. Other easy special cases are known.

Related Problems: Hamiltonian cycle (see page 538), minimum spanning tree (see page 484), convex hull (see page 568).



INPUT OUTPUT

16.5 Hamiltonian Cycle

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

Problem description: Find a tour of the vertices using only edges from G, such that each vertex is visited exactly once.

Discussion: Finding a Hamiltonian cycle or path in a graph G is a special case of the traveling salesman problem G'—one where each edge in G has distance 1 in $'G$. Non-edge vertex pairs are separated by a greater distance, say 2. Such a weighted graph has TSP tour of cost n in G' iff G is Hamiltonian.

Closely related is the problem of finding the longest path or cycle in a graph. This arises often in pattern recognition problems. Let the vertices in the graph correspond to possible symbols, with edges linking pairs of symbols that might occur next to each other. The longest path through this graph is a good candidate for the proper interpretation.

The problems of finding longest cycles and paths are both NP-complete, even on very restrictive classes of unweighted graphs. There are several possible lines of attack, however:

- *Is there a serious penalty for visiting vertices more than once?* – Reformulating the Hamiltonian cycle problem instead of minimizing the total number of vertices visited on a complete tour turns it into an optimization problem.

This allows possibilities for heuristics and approximation algorithms. Finding a spanning tree of the graph and doing a depth-first search, as discussed in Section 16.4 (page 533), yields a tour with at most $2n$ vertices. Using randomization or simulated annealing might bring the size of this down considerably.

- *Am I seeking the longest path in a directed acyclic graph (DAG)?* – The problem of finding the longest path in a DAG can be solved in linear time using dynamic programming. Conveniently, the algorithm for finding the *shortest* path in a DAG (presented in Section 15.4 (page 489)) does the job if we replace min with max. DAGs are the most interesting case of longest path for which efficient algorithms exist.

- *Is my graph dense?* – Sufficiently dense graphs always contain Hamiltonian cycles. Further, the cycles implied by such sufficiency conditions can be efficiently constructed. In particular, any graph where all vertices have degree $\geq n/2$ must be Hamiltonian. Stronger sufficient conditions also hold; see the Notes section.

- *Are you visiting all the vertices or all the edges?* – Verify that you really have a vertex-tour problem and not an edge-tour problem. With a little cleverness it is sometimes possible to reformulate a Hamiltonian cycle problem in terms of Eulerian cycles, which instead visit every edge of a graph. Perhaps the most famous such instance is the problem of constructing de Bruijn sequences, discussed in Section 15.7 (page 502). The benefit is that fast algorithms exist for Eulerian cycles and many related variants, while Hamiltonian cycle is NP-complete.

If you *really* must know whether your graph is Hamiltonian, backtracking with pruning is your only possible solution. Certainly check whether your graph is biconnected (see Section 15.8 (page 505)). If not, this means that the graph has an articulation vertex whose deletion will disconnect the graph and so cannot be Hamiltonian.

Implementations: The construction described above (weight 1 for an edge and 2 for a non-edge) reduces Hamiltonian cycles to a symmetric TSP problem that obeys the triangle inequality. Thus we refer the reader to the TSP solvers discussed in Section 16.4 (page 533). Foremost among them is `Concorde`, a program for the symmetric traveling salesman problem and related network optimization problems, written in ANSI C. `Concorde` is available for academic research use from *http://www.tsp.gatech.edu/concorde*. It is the clear choice among available TSP codes.

An effective program for solving Hamiltonian cycle problems resulted from the masters thesis of Vandegriend [Van98]. Both the code and the thesis are available from *http://web.cs.ualberta.ca/∼joe/Theses/vandegriend.html*.

Lodi and Punnen [LP07] put together an excellent survey of available TSP software, including the special case of Hamiltonian cycle. Current links to all programs are maintained at *http://www.or.deis.unibo.it/research_pages/tspsoft.html*.

The football program of the Stanford GraphBase (see Section 19.1.8 (page 660)) uses a stratified greedy algorithm to solve the asymmetric longest-path problem. The goal is to derive a chain of football scores to establish the superiority of one football team over another. After all, if Virginia beat Illinois by 30 points, and Illinois beat Stony Brook by 14 points, then by transitivity Virginia would beat Stony Brook by 44 points if they played, right? We seek the longest simple path in a graph where the weight of edge (x, y) denotes the number of points by which x beat y.

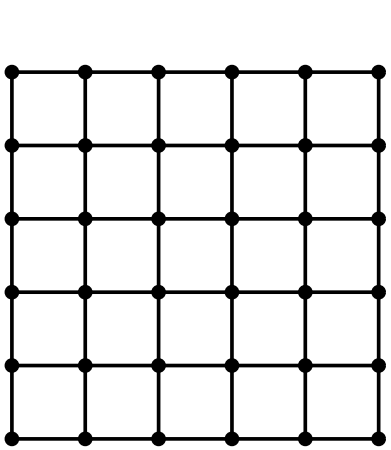
Nijenhuis and Wilf [NW78] provide an efficient routine to enumerate all Hamiltonian cycles of a graph by backtracking. See Section 19.1.10 (page 661). Algorithm 595 [Mar83] of the *Collected Algorithms of the ACM* is a similar Fortran code that can be used as either an exact procedure or a heuristic by controlling the amount of backtracking. See Section 19.1.5 (page 659).

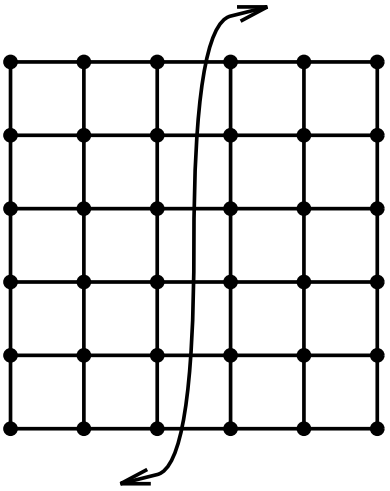
Notes: Hamiltonian cycles apparently first arose in Euler's study of the knight's tour problem, although they were popularized by Hamilton's "Around the World" game in 1839. See [ABCC07, GP07, LLKS85] for comprehensive references on the traveling salesman problem, including discussions on Hamiltonian cycle.

Most good texts in graph theory review sufficiency conditions for graphs to be Hamiltonian. My favorite is West [Wes00].

Techniques for solving optimization problems in the laboratory using biological processes have attracted considerable attention. In the original application of these "biocomputing" techniques, Adleman [Adl94] solved a seven-vertex instance of the directed Hamiltonian path problem. Unfortunately, this approach requires an exponential number of molecules, and Avogadro's number implies that such experiments are inconceivable for graphs beyond $n \approx 70$.

Related Problems: Eulerian cycle (see page 502), traveling salesman (see page 533).





INPUT OUTPUT

16.6 Graph Partition

Input description: A (weighted) graph $G = (V, E)$ and integers k and m.

Problem description: Partition the vertices into m roughly equal-sized subsets such that the total edge cost spanning the subsets is at most k.

Discussion: Graph partitioning arises in many divide-and-conquer algorithms, which gain their efficiency by breaking problems into equal-sized pieces such that the respective solutions can easily be reassembled. Minimizing the number of edges cut in the partition usually simplifies the task of merging.

Graph partition also arises when we need to cluster the vertices into logical components. If edges link "similar" pairs of objects, the clusters remaining after partition should reflect coherent groupings. Large graphs are often partitioned into reasonable-sized pieces to improve data locality or make less cluttered drawings.

Finally, graph partition is a critical step in many parallel algorithms. Consider the finite element method, which is used to compute the physical properties (such as stress and heat transfer) of geometric models. Parallelizing such calculations requires partitioning the models into equal-sized pieces whose interface is small. This is a graph-partitioning problem, since the topology of a geometric model is usually represented using a graph.

Several different flavors of graph partitioning arise depending on the desired objective function:

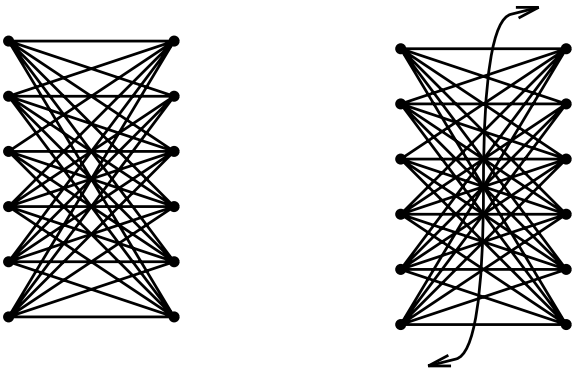


Figure 16.1: The maximum cut of a graph

- *Minimum cut set* – The *smallest* set of edges to cut that will disconnect a graph can be efficiently found using network flow or randomized algorithms. See Section 15.8 (page 505) for more on connectivity algorithms. The smallest cutset might split off only a single vertex, so the resulting partition could be very unbalanced.

- *Graph partition* – A better partition criterion seeks a small cut that partitions the vertices into roughly equal-sized pieces. Unfortunately, this problem is NP-complete. Fortunately, the heuristics discussed below work well in practice.

  Certain special graphs always have small *separators*, that partition the vertices into balanced pieces. For any tree, there always exists a single vertex whose deletion partitions the tree so that no component contains more than $n/2$ of the original n vertices. These components need not always be connected; consider the separating vertex of a star-shaped tree. This separating vertex can be found in linear time using depth first-search. Every planar graph has a set of $O(\sqrt{n})$ vertices whose deletion leaves no component with more than $2n/3$ vertices. These separators provide a useful way to decompose geometric models, which are often defined by planar graphs.

- *Maximum cut* – Given an electronic circuit specified by a graph, the *maximum cut* defines the largest amount of data communication that can simultaneously occur in the circuit. The highest-speed communications channel should thus span the vertex partition defined by the maximum edge cut. Finding the maximum cut in a graph is NP-complete [Kar72], however heuristics similar to those of graph partitioning work well.

The basic approach for dealing with graph partitioning or max-cut problems is to construct an initial partition of the vertices (either randomly or according

to some problem-specific strategy) and then sweep through the vertices, deciding whether the size of the cut would improve if we moved this vertex over to the other side. The decision to move vertex v can be made in time proportional to its degree, by identifying which side of the partition contains more of v's neighbors. Of course, the desirable side for v may change after its neighbors jump, so several iterations are likely to be needed before the process converges on a local optimum. Even so, such a local optimum can be arbitrarily far away from the global max-cut.

There are many variations of this basic procedure, by changing the order we test the vertices in or moving clusters of vertices simultaneously. Using some form of randomization, particularly simulated annealing, is almost certain to be a good idea. When more than two components are desired, the partitioning heuristic should be applied recursively.

Spectral partitioning methods use sophisticated linear algebra techniques to obtain a good partitioning. The spectral bisection method uses the second-lowest eigenvector of the *Laplacian matrix* of the graph to partition it into two pieces. Spectral methods tend to do a good job of identifying the general area to partition, but the results can be improved by cleaning up with a local optimization method.

Implementations: Chaco is a widely-used graph partitioning code designed to partition graphs for parallel computing applications. It employs several different partitioning algorithms, including both Kernighan-Lin and spectral methods. Chaco is available at *http://www.cs.sandia.gov/~bahendr/chaco.html*.

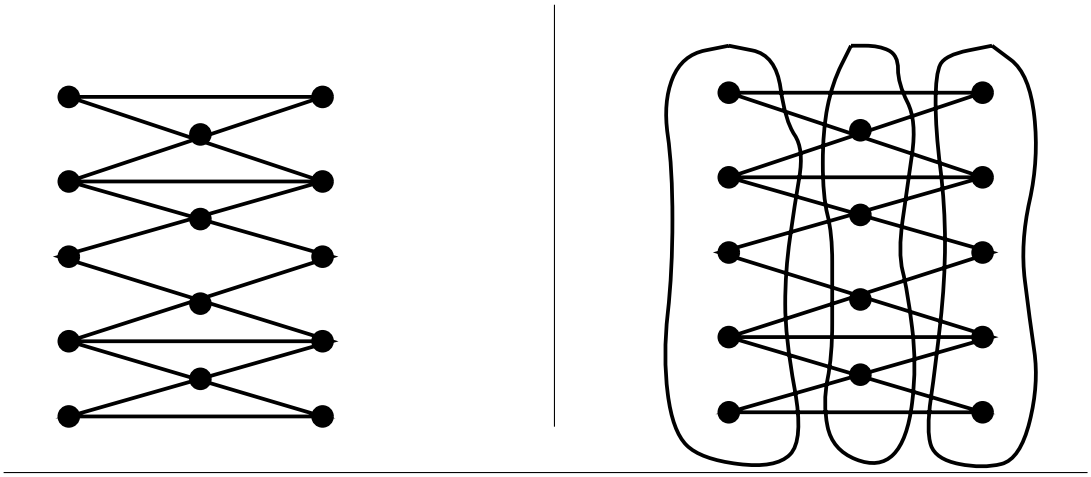
METIS (*http://glaros.dtc.umn.edu/gkhome/views/metis*) is another well-regarded code for graph partitioning. It has successfully partitioned graphs with over 1,000,000 vertices. Available versions include one variant designed to run on parallel machines and another suitable for partitioning hypergraphs. Other respected codes include Scotch (*http://www.labri.fr/perso/pelegrin/scotch/*) and JOSTLE (*http://staffweb.cms.gre.ac.uk/~wc06/jostle/*).

Notes: The fundamental local improvement heuristics for graph partitioning are due to Kernighan-Lin [KL70] and Fiduccia-Mattheyses [FM82]. Spectral methods for graph partition are discussed in [Chu97, PSL90]. Empirical results on graph partitioning heuristics include [BG95, LR93].

The planar separator theorem and an efficient algorithm for finding such a separator are due to Lipton and Tarjan [LT79, LT80]. For experiences in implementing planar separator algorithms, see [ADGM04, HPS+05].

Any random vertex partition will expect to cut half of the edges in the graph, since the probability that the two vertices defining an edge end up on different sides of the partition is 1/2. Goemans and Williamson [GW95] gave an 0.878-factor approximation algorithm for maximum-cut, based on semi-definite programming techniques. Tighter analysis of this algorithm was followed by Karloff [Kar96].

Related Problems: Edge/vertex connectivity (see page 505), network flow (see page 509).



INPUT OUTPUT

16.7 Vertex Coloring

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

Problem description: Color the vertices of V using the minimum number of colors such that i and j have different colors for all $(i, j) \in E$.

Discussion: Vertex coloring arises in many scheduling and clustering applications. Register allocation in compiler optimization is a canonical application of coloring. Each variable in a given program fragment has a range of times during which its value must be kept intact, in particular after it is initialized and before its final use. Any two variables whose life spans intersect cannot be placed in the same register. Construct a graph where each vertex corresponds to a variable, with an edge between any two vertices whose variable life spans intersect. Since none of the variables assigned the same color clash, they all can be assigned to the same register.

No conflicts will occur if each vertex is colored using a distinct color. But computers have a limited number of registers, so we seek a coloring using the fewest colors. The smallest number of colors sufficient to vertex-color a graph is its *chromatic number*.

Several special cases of interest arise in practice:

- *Can I color the graph using only two colors?* – An important special case is testing whether a graph is *bipartite*, meaning it can be colored using only two different colors. Bipartite graphs arise naturally in such applications as mapping workers to possible jobs. Fast, simple algorithms exist for problems

such as matching (see Section 15.6 (page 498)) when restricted to bipartite graphs.

Testing whether a graph is bipartite is easy. Color the first vertex blue, and then do a depth-first search of the graph. Whenever we discover a new, uncolored vertex, color it opposite of its parent, since the same color would cause a clash. The graph cannot be bipartite if we ever find an edge (x, y) where both x and y have been colored identically. Otherwise, the final coloring will be a 2-coloring, constructed in $O(n + m)$ time. An implementation of this algorithm is given in Section 5.7.2 (page 167).

- *Is the graph planar, or are all vertices of low degree?* – The famous four-color theorem states that every planar graph can be vertex colored using at most four distinct colors. Efficient algorithms for finding a four-coloring on planar graphs are known, although it is NP-complete to decide whether a given planar graph is three-colorable.

  There is a very simple algorithm to find a vertex coloring of a planar graph using at most six colors. In any planar graph, there exists a vertex of at most five degree. Delete this vertex and recursively color the graph. This vertex has at most five neighbors, which means that it can always be colored using one of the six colors that does not appear as a neighbor. This works because deleting a vertex from a planar graph leaves a planar graph, meaning that it must also have a low-degree vertex to delete. The same idea can be used to color any graph of maximum degree Δ using $\leq \Delta + 1$ colors in $O(n\Delta)$ time.

- *Is this an edge-coloring problem?* – Certain vertex coloring problems can be modeled as *edge coloring*, where we seek to color the edges of a graph G such that no two edges are colored the same if they have a vertex in common. The payoff is that there is an efficient algorithm that always returns a near-optimal edge coloring. Algorithms for edge coloring are the focus of Section 16.8 (page 548).

Computing the chromatic number of a graph is NP-complete, so if you need an exact solution you must resort to backtracking, which can be surprisingly effective in coloring certain random graphs. It remains hard to compute a good approximation to the optimal coloring, so expect no guarantees.

Incremental methods are the heuristic of choice for vertex coloring. As in the previously-mentioned algorithm for planar graphs, vertices are colored sequentially, with the colors chosen in response to colors already assigned in the vertex's neighborhood. These methods vary in how the next vertex is selected and how it is assigned a color. Experience suggests inserting the vertices in nonincreasing order of degree, since high-degree vertices have more color constraints and so are most likely to require an additional color if inserted late. Brèlaz's heuristic [Brè79] dynamically selected the uncolored vertex of highest *color degree* (i.e. , adjacent to the most different colors), and colors it with the lowest-numbered unused color.

Incremental methods can be further improved by using *color interchange*. Taking a properly colored graph and exchanging two of the colors (painting the red vertices blue and the blue vertices red) leaves a proper vertex coloring. Now suppose we take a properly colored graph and delete all but the red and blue vertices. We can repaint one or more of the resulting connected components, again leaving a proper coloring. After such a recoloring, some vertex v previously adjacent to both red and blue vertices might now be only adjacent to blue vertices, thus freeing v to be colored red.

Color interchange is a win in terms of producing better colorings, at a cost of increased time and implementation complexity. Implementations are described next. Simulated annealing algorithms that incorporate color interchange to move from state to state are likely to be even more effective.

Implementations: Graph coloring has been blessed with two useful Web resources. Culberson's graph coloring page, *http://web.cs.ualberta.ca/~joe/Coloring/*, provides an extensive bibliography and programs to generate and solve hard graph coloring instances. Trick's page, *http://mat.gsia.cmu.edu/COLOR/color.html*, provides a nice overview of graph coloring applications, an annotated bibliography, and a collection of over 70 graph-coloring instances arising in applications such as register allocation and printed circuit board testing. Both contain a C language implementation of the DSATUR coloring algorithm.

Programs for the closely related problems of finding cliques and vertex coloring graphs were sought for at the Second DIMACS Implementation Challenge [JT96], held in October 1993. Programs and data from the challenge are available by anonymous FTP from *dimacs.rutgers.edu*. Source codes are available under *pub/challenge/graph* and test data under *pub/djs*, including a simple "semi-exhaustive greedy" scheme used in the graph-coloring algorithm XRLF [JAMS91].

GraphCol (*http://code.google.com/p/graphcol/*) contains tabu search and simulated annealing heuristics for constructing colorings in C.

The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) contains an implementation of greedy incremental vertex coloring heuristics. GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements a branch-and-bound algorithm for vertex coloring.

Pascal implementations of backtracking algorithms for vertex coloring and several heuristics, including largest-first and smallest-last incremental orderings and color interchange, appear in [SDK83]. See Section 19.1.10 (page 662).

Nijenhuis and Wilf [NW78] provide an efficient Fortran implementation of chromatic polynomials and vertex coloring by backtracking. See Section 19.1.10 (page 661).

Combinatorica [PS03] provides Mathematica implementations of bipartite graph testing, heuristic colorings, chromatic polynomials, and vertex coloring by backtracking. See Section 19.1.9 (page 661).

Notes: An old but excellent source on vertex coloring heuristics is Syslo, Deo, and Kowalik [SDK83], which includes experimental results. Classical heuristics for vertex coloring include [Brè79, MMI72, Tur88]; see [GH06, HDD03] for more recent results.

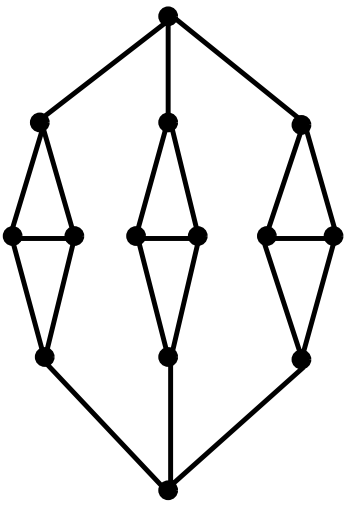
Wilf [Wil84] proved that backtracking to test whether a random graph has chromatic number k runs in *constant time*, dependent on k but independent of n. This is not as interesting as it sounds, because only a vanishingly small fraction of such graphs are indeed k-colorable. A number of provably efficient (but still exponential) algorithms for vertex coloring are known. See [Woe03] for a survey.

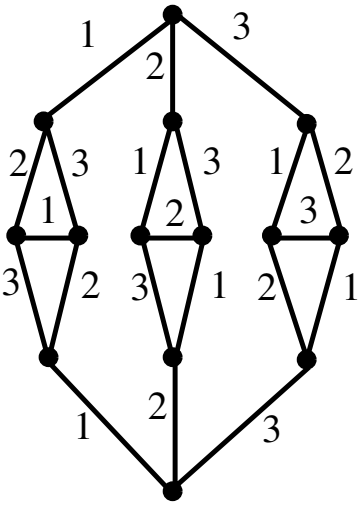
Paschos [Pas03] reviews what is known about provably good approximation algorithms for vertex coloring. On one hand, it is provably hard to approximate within a polynomial factor [BGS95]. On the other hand, there are heuristics that offer some nontrivial guarantees in terms of various parameters, such as Wigderson's [Wig83] factor of $n^{1-1/(\chi(G)-1)}$ approximation algorithm, where $\chi(G)$ is the chromatic number of G.

Brook's theorem states that the chromatic number $\chi(G) \leq \Delta(G) + 1$, where $\Delta(G)$ is the maximum degree of a vertex of G. Equality holds only for odd-length cycles (which have chromatic number 3) and complete graphs.

The most famous problem in the history of graph theory is the four-color problem, first posed in 1852 and finally settled in 1976 by Appel and Haken using a proof involving extensive computation. Any planar graph can be five-colored using a variation of the color interchange heuristic. Despite the four-color theorem, it is NP-complete to test whether a particular planar graph requires four colors or if three suffice. See [SK86] for an exposition on the history of the four-color problem and the proof. An efficient algorithm to four-color a graph is presented in [RSST96].

Related Problems: Independent set (see page 528), edge coloring (see page 548).





INPUT OUTPUT

16.8 Edge Coloring

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

Problem description: What is the smallest set of colors needed to color the edges of G such that no two same-color edges share a common vertex?

Discussion: The edge coloring of graphs arises in scheduling applications, typically associated with minimizing the number of noninterfering rounds needed to complete a given set of tasks. For example, consider a situation where we must schedule a given set of two-person interviews, where each interview takes one hour. All meetings could be scheduled to occur at distinct times to avoid conflicts, but it is less wasteful to schedule nonconflicting events simultaneously. We construct a graph whose vertices are people and whose edges represent the pairs of people who need to meet. An edge coloring of this graph defines the schedule. The color classes represent the different time periods in the schedule, with all meetings of the same color happening simultaneously.

The National Football League solves such an edge-coloring problem each season to make up its schedule. Each team's opponents are determined by the records of the previous season. Assigning the opponents to weeks of the season is an edge-coloring problem, complicated by extra constraints of spacing out rematches and making sure that there is a good game every Monday night.

The minimum number of colors needed to edge color a graph is called its *edge-chromatic number* by some and its *chromatic index* by others. Note that an even-length cycle can be edge-colored with 2 colors, while odd-length cycles have an edge-chromatic number of 3.

Edge coloring has a better (if less famous) theorem associated with it than vertex coloring. Vizing's theorem states that any graph with a maximum vertex degree of Δ can be edge colored using at most $\Delta + 1$ colors. To put this in perspective, note that *any* edge coloring must have at least Δ colors, since all the edges incident on any vertex must be distinct colors.

The proof of Vizing's theorem is constructive, meaning it can be turned into an $O(nm\Delta)$ algorithm to find an edge-coloring with $\Delta + 1$ colors. Since deciding whether we can get away using one less color than this is NP-complete, it hardly seems worth the effort to worry about it. An implementation of Vizing's theorem is described below.

Any edge-coloring problem on G can be converted to the problem of finding a vertex coloring on the *line graph* $L(G)$, which has a vertex of $L(G)$ for each edge of G and an edge of $L(G)$ if and only if the two edges of G share a common vertex. Line graphs can be constructed in time linear to their size, and any vertex-coloring code can be employed to color them. That said, it is disappointing to go the vertex coloring route. Vizing's theorem is our reward for the extra thought needed to see that we have an edge-coloring problem.

Implementations: Yan Dong produced an implementation of Vizing's theorem in C++ as a course project for my algorithms course while a student at Stony Brook. It can be found on the algorithm repository site at *http://www.cs.sunysb.edu/~algorith*.

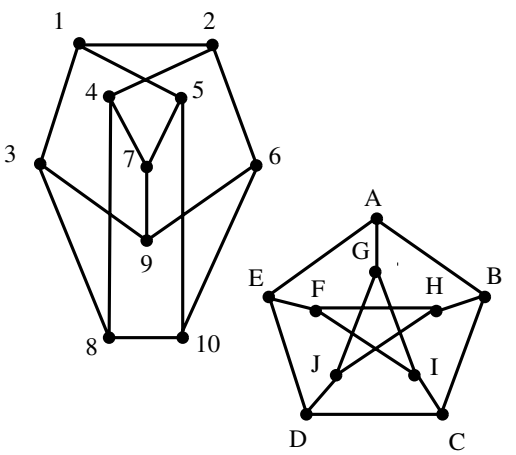
GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements a branch-and-bound algorithm for edge coloring.

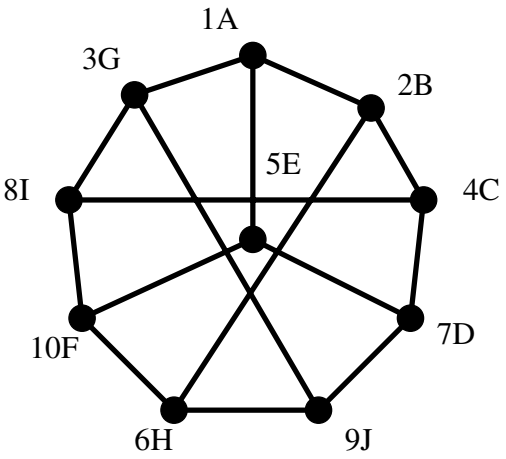
See Section 16.7 (page 544) for a larger collection of vertex-coloring codes and heuristics, which can be applied to the line graph of your target graph. Combinatorica [PS03] provides Mathematica implementations of edge coloring in this fashion, via the line graph transformation and vertex coloring routines. See Section 19.1.9 (page 661) for more information on Combinatorica.

Notes: Graph-theoretic results on edge coloring are surveyed in [FW77, GT94]. Vizing [Viz64] and Gupta [Gup66] independently proved that any graph can be edge colored using at most $\Delta + 1$ colors. Misra and Gries give a simple constructive proof of this result [MG92]. Despite these tight bounds, it is NP-complete to compute the edge-chromatic number [Hol81]. Bipartite graphs can be edge-colored in polynomial time [Sch98].

Whitney, in introducing line graphs [Whi32], showed that with the exception of K_3 and $K_{1,3}$, any two connected graphs with isomorphic line graphs are isomorphic. It is an interesting exercise to show that the line graph of an Eulerian graph is both Eulerian and Hamiltonian, while the line graph of a Hamiltonian graph is always Hamiltonian.

Related Problems: Vertex coloring (see page 544), scheduling (see page 468).





INPUT OUTPUT

16.9 Graph Isomorphism

Input description: Two graphs, G and H.

Problem description: Find a (or all) mapping f from the vertices of G to the vertices of H such that G and H are identical; i.e. , (x, y) is an edge of G iff $(f(x), f(y))$ is an edge of H.

Discussion: Isomorphism is the problem of testing whether two graphs are really the same. Suppose we are given a collection of graphs and must perform some operation on each of them. If we can identify which of the graphs are duplicates, we can discard copies to avoid redundant work.

Certain pattern recognition problems are readily mapped to graph or subgraph isomorphism detection. The structure of chemical compounds are naturally described by labeled graphs, with each atom represented by a vertex. Identifying all molecules in a structure database containing a particular functional group is an instance of subgraph isomorphism testing.

We need some terminology to settle what is meant when we say two graphs are the same. Two labeled graphs $G = (V_g, E_g)$ and $H = (V_h, E_h)$ are *identical* when $(x, y) \in E_g$ iff $(x, y) \in E_h$. The isomorphism problem consists of finding a mapping from the vertices of G to H such that they are identical. Such a mapping is called an *isomorphism*; the problem of finding the mapping is sometimes called *graph matching*.

Identifying symmetries is another important application of graph isomorphism. A mapping of a graph to itself is called an *automorphism*, and the collection of

automorphisms (the automorphism *group*) provides a great deal of information about symmetries in the graph. For example, the complete graph K_n has $n!$ automorphisms (any mapping will do), while an arbitrary random graph is likely to have few or perhaps only one, since G is always identical to itself.

Several variations of graph isomorphism arise in practice:

- *Is graph G contained in graph H?* – Instead of testing equality, we are often interested in knowing whether a small pattern graph G is a *subgraph* of H. Such problems as clique, independent set, and Hamiltonian cycle are important special cases of subgraph isomorphism.

  There are two distinct graph-theoretic notions of "contained in." *Subgraph isomorphism* asks whether there is a subset of edges and vertices of H that is isomorphic to a smaller graph G. *Induced subgraph isomorphism* asks whether there is a subset of vertices of H whose deletion leaves a subgraph isomorphic to a smaller graph G. For induced subgraph isomorphism, (1) all edges of G must be present in H, and (2) no *non-edges* of G can be present in H. Clique happens to be an instance of both subgraph isomorphism problems, while Hamiltonian cycle is only an example of vanilla subgraph isomorphism.

  Be aware of this distinction in your application. Subgraph isomorphism problems tend to be harder than graph isomorphism, while induced subgraph problems tend to be even harder than subgraph isomorphism. Some flavor of backtracking is your only viable approach.

- *Are your graphs labeled or unlabeled?* – In many applications, vertices or edges of the graphs are *labeled* with some attribute that must be respected in determining isomorphisms. For example, in comparing two bipartite graphs, each with "worker" vertices and "job" vertices, any isomorphism that equated a job with a worker would make no sense.

  Labels and related constraints can be factored into any backtracking algorithm. Further, such constraints significantly speed up the search, by creating many more opportunities for pruning whenever two vertex labels do not match up.

- *Are you testing whether two trees are isomorphic?* – Faster algorithms exist for certain special cases of graph isomorphism, such as trees and planar graphs. Perhaps the most important case is detecting isomorphisms among trees, a problem that arises in language pattern matching and parsing applications. A parse tree is often used to describe the structure of a text; two parse trees will be isomorphic if the underlying pair of texts have the same structure.

  Efficient algorithms for tree isomorphism begin with the leaves of both trees and work inward toward the center. Each vertex in one tree is assigned a label representing the set of vertices in the second tree that might possibly

be isomorphic to it, based on the constraints of labels and vertex degrees. For example, all the leaves in tree T_1 are initially potentially equivalent to all leaves of T_2. Now, working inward, we can partition the vertices adjacent to leaves in T_1 into classes based on how many leaves and non-leaves they are adjacent to. By carefully keeping track of the labels of the subtrees, we can make sure that we have the same distribution of labeled subtrees for T_1 and T_2. Any mismatch means $T_1 \neq T_2$, while completing the process partitions the vertices into equivalence classes defining all isomorphisms. See the references below for more details.

- *How many graphs do you have?* – Many data mining applications involve searching for all instances of a particular pattern graph in a big database of graphs. The chemical structure mapping application described above falls into this family. Such databases typically contain a large number of relatively small graphs. This puts an onus on indexing the graph database by small substructures (say five to ten vertex each), and doing expensive isomorphism tests only against those containing the same substructures as the query graph.

No polynomial-time algorithm is known for graph isomorphism, but neither is it known to be NP-complete. Along with integer factorization (see Section 13.8 (page 420)), it is one of the few important algorithmic problems whose rough computational complexity is still not known. The conventional wisdom is that isomorphism is a problem that lies between P and NP-complete if P $\neq$ NP.

Although no worst-case polynomial-time algorithm is known, testing isomorphism is *usually* not very hard in practice. The basic algorithm backtracks through all $n!$ possible relabelings of the vertices of graph h with the names of vertices of graph g, and then tests whether the graphs are identical. Of course, we can prune the search of all permutations with a given prefix as soon as we detect any mismatch between edges whose vertices are both in the prefix.

However, the real key to efficient isomorphism testing is to preprocess the vertices into "equivalence classes," partitioning them into sets of vertices so that two vertices in different sets cannot possibly be mistaken for each other. All vertices in each equivalence class must share the same value of some invariant that is independent of labeling. Possibilities include:

- *Vertex degree* – This simplest way to partition vertices is based on their degree—the number of edges incident on the vertex. Two vertices of different degrees cannot be identical. This simple partition can be a big win, but won't do much for regular (equal degree) graphs.

- *Shortest path matrix* – For each vertex v, the all-pairs shortest path matrix (see Section 15.4 (page 489)) defines a multiset of $n-1$ distances representing the distances between v and each of the other vertices. Any two identical vertices must define the exact same multiset of distances, so we can partition the vertices into equivalence classes defining identical distance multisets.

- *Counting length-k paths* – Taking the adjacency matrix of G and raising it to the kth power gives a matrix where $G^k[i,j]$ counts the number of (nonsimple) paths from i to j. For each vertex and each k, this matrix defines a multiset of path-counts, which can be used for partitioning as with distances above. You could try all $1 \leq k \leq n$ or beyond, and use any single deviation as an excuse to partition.

Using these invariants, you should be able to partition the vertices of most graphs into a large number of small equivalence classes. Finishing the job off with backtracking should then be short work. We assign each vertex the name of its equivalence class as a label, and treat it as a labeled matching problem. It is harder to detect isomorphisms between highly-symmetric graphs than it is with random graphs because of the reduced effectiveness of these equivalence-class partitioning heuristics.

Implementations: The best known isomorphism testing program is `nauty` (No AUTomorphisms, Yes?)—a set of very efficient C language procedures for determining the automorphism group of a vertex-colored graph. Nauty is also able to produce a canonically-labeled isomorph of the graph, to assist in isomorphism testing. It was the basis of the first program to generate all 11-vertex graphs without isomorphs, and can test most graphs with fewer than 100 vertices in well under a second. Nauty has been ported to a variety of operating systems and C compilers. It is available at *http://cs.anu.edu.au/∼bdm/nauty/*. The theory behind `nauty` is described in [McK81].

The `VFLib` graph-matching library contains implementations for several different algorithms for *both* graph and subgraph isomorphism testing. This library has been widely used and very carefully benchmarked [FSV01]. It is available at *http://amalfi.dis.unina.it/graph/*.

`GraphGrep` [GS02] (*http://www.cs.nyu.edu/shasha/papers/graphgrep/*) is a representative data mining tool for querying large databases of graphs.

Valiente [Val02] has made available the implementations of graph/subgraph isomorphism algorithms for both trees and graphs in his book [Val02]. These C++ implementations run on top of LEDA (see Section 19.1.1 (page 658)), and are available at *http://www.lsi.upc.edu/∼valiente/algorithm/*.

Kreher and Stinson [KS99] compute isomorphisms of graphs in addition to more general group-theoretic operations. These implementations in C are available at *http://www.math.mtu.edu/∼kreher/cages/Src.html*.

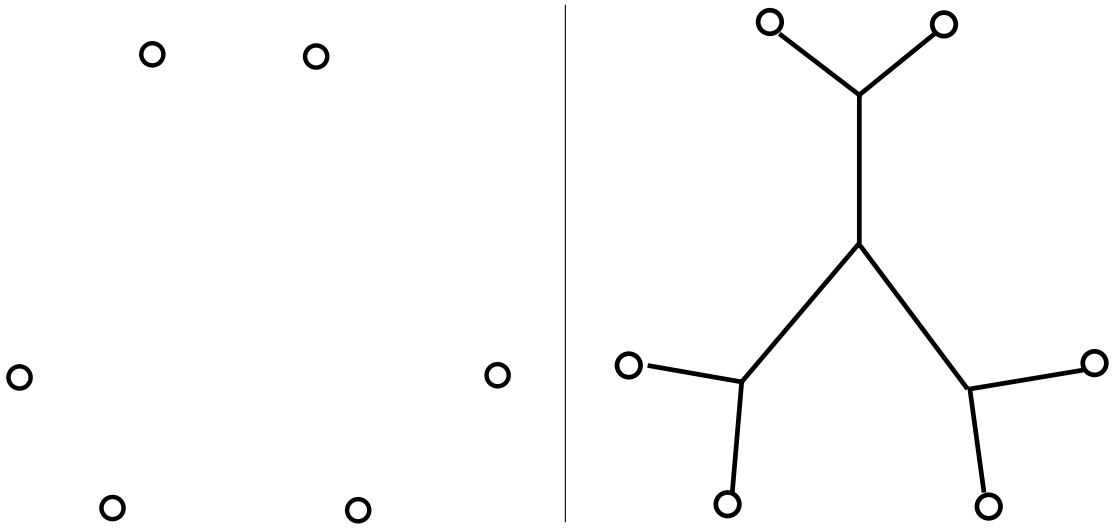
Notes: Graph isomorphism is an important problem in complexity theory. Monographs on isomorphism detection include [Hof82, KST93]. Valiente [Val02] focuses on algorithms for tree and subgraph isomorphism. Kreher and Stinson [KS99] take a more group-theoretic approach to isomorphism testing. Graph mining systems and algorithms are surveyed in [CH06]. See [FSV01] for performance comparisons between different graph and subgraph isomorphism algorithms.

Polynomial-time algorithms are known for planar graph isomorphism [HW74] and for graphs where the maximum vertex degree is bounded by a constant [Luk80]. The all-pairs

shortest path heuristic is due to [SD76], although there exist nonisomorphic graphs that realize the exact same set of distances [BH90]. A linear-time tree isomorphism algorithm for both labeled and unlabeled trees is presented in [AHU74].

A problem is said to be *isomorphism-complete* if it is provably as hard as isomorphism. Testing the isomorphism of bipartite graphs is isomorphism-complete, since any graph can be made bipartite by replacing each edge by two edges connected with a new vertex. Clearly, the original graphs are isomorphic if and only if the transformed graphs are.

Related Problems: Shortest path (see page 489), string matching (see page 628).



INPUT OUTPUT

16.10 Steiner Tree

Input description: A graph $G = (V, E)$. A subset of vertices $T \in V$.

Problem description: Find the smallest tree connecting all the vertices of T.

Discussion: Steiner trees arise often in network design problems, since the minimum Steiner tree describes how to connect a given set of sites using the smallest amount of wire. Analogous problems occur when designing networks of water pipes or heating ducts and in VLSI circuit design. Typical Steiner tree problems in VLSI are to connect a set of sites to (say) ground under constraints such as material cost, signal propagation time, or reducing capacitance.

The Steiner tree problem is distinguished from the minimum spanning tree (MST) problem (see Section 15.3 (page 484)) in that we are permitted to construct or select intermediate connection points to reduce the cost of the tree. Issues in Steiner tree construction include:

- *How many points do you have to connect?* – The Steiner tree of a pair of vertices is simply the shortest path between them (see Section 15.4 (page 489)). The Steiner tree of all the vertices, when $S = V$, simply defines the MST of G. The general minimum Steiner tree problem is NP-hard despite these special cases, and remains so under a broad range of restrictions.

- *Is the input a set of geometric points or a distance graph?* – Geometric versions of Steiner tree take a set of points as input, typically in the plane, and seek the smallest tree connecting the points. A complication is that the set of possible intermediate points is not given as part of the input but must be deduced from the set of points. These possible Steiner points must satisfy several geometric properties, which can be used to reduce the set of candidates down to a finite number. For example, every Steiner point will have a degree of exactly three in a minimum Steiner tree, and the angles formed between any two of these edges must be exactly 120 degrees.

- *Are there constraints on the edges we can use?* – Many wiring problems correspond to geometric versions of the problem, where all edges are restricted to being either horizontal or vertical. This is the so-called *rectilinear Steiner problem*. A different set of angular and degree conditions apply for rectilinear Steiner trees than for Euclidean trees. In particular, all angles must be multiples of 90 degrees, and each vertex is of a degree up to four.

- *Do I really need an optimal tree?* – Certain Steiner tree applications (e.g., circuit design and communications networks) justify investing large amounts of computation to find the best possible Steiner tree. This implies an exhaustive search technique such as backtracking or branch-and-bound. There are many opportunities for pruning search based on geometric and graph-theoretic constraints.

  Still, Steiner tree remains a hard problem. We recommend experimenting with the implementations described below before attempting your own.

- *How can I reconstruct Steiner vertices I never knew about?* – A very special type of Steiner tree arises in classification and evolution. A *phylogenic tree* illustrates the relative similarity between different objects or species. Each object represents (typically) a leaf/terminal vertex of the tree, with intermediate vertices representing branching points between classes of objects. For example, an evolutionary tree of animal species might have leaf nodes of *human, dog, snake* and internal nodes corresponding to taxa *(animal, mammal, reptile)*. A tree rooted at *animal* with *dog* and *human* classified under *mammal* implies that humans are closer to dogs than to snakes.

  Many different phylogenic tree construction algorithms have been developed that vary in (1) the data they attempt to model, and (2) the desired optimization criterion. Each combination of reconstruction algorithm and distance measure is likely to give a different answer, so identifying the "right" method for any given application is somewhat a question of faith. A reasonable procedure is to acquire a standard package of implementations, discussed below, and then see what happens to your data under all of them.

Fortunately, there is a good, efficient heuristic for finding Steiner trees that works well on all versions of the problem. Construct a graph modeling your input,

setting the weight of edge (i, j) equal to the distance from point i to point j. Find an MST of this graph. You are guaranteed a provably good approximation for both Euclidean and rectilinear Steiner trees.

The worst case for a MST approximation of the Euclidean Steiner tree is three points forming an equilateral triangle. The MST will contain two of the sides (for a length of 2), whereas the minimum Steiner tree will connect the three points using an interior point, for a total length of $\sqrt{3}$. This ratio of $\sqrt{3}/2 \approx 0.866$ is always achieved, and in practice the easily-computed MST is usually within a few percent of the optimal Steiner tree. The rectilinear Steiner tree / MST ratio is always $\geq 2/3 \approx 0.667$.

Such an MST can be refined by inserting a Steiner point whenever the edges of the minimum spanning tree incident on a vertex form an angle of less than 120 degrees between them. Inserting these points and locally readjusting the tree edges can move the solution a few more percent towards the optimum. Similar optimizations are possible for rectilinear spanning trees.

Note that we are only interested in the subtree connecting the terminal vertices. We may need to trim the MST if we add nonterminal vertices to the input of the problem. We retain only the tree edges which lie on the (unique) path between some pair of terminal nodes. The complete set of these can be found in $O(n)$ time by performing a BFS on the full tree starting from any single terminal node.

An alternative heuristic for graphs is based on shortest path. Start with a tree consisting of the shortest path between two terminals. For each remaining terminal t, find the shortest path to a vertex v in the tree and add this path to the tree. The time complexity and quality of this heuristic depend upon the insertion order of the terminals and how the shortest-path computations are performed, but something simple and fairly effective is likely to result.

Implementations: GeoSteiner is a package for solving both Euclidean and rectilinear Steiner tree problems in the plane by Warme, Winter, and Zachariasen [WWZ00]. It also solves the related problem of MSTs in hypergraphs, and claims to have solved problems as large as 10,000 points to optimality. It is available from *http://www.diku.dk/geosteiner/*. This is almost certainly the best code for geometric instances of Steiner tree.

FLUTE (*http://home.eng.iastate.edu/~cnchu/flute.html*) computes rectilinear Steiner trees, emphasizing speed. It contains a user-defined parameter to control the tradeoff between solution quality and run time.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) includes both heuristics and search methods for finding Steiner trees in graphs.

The programs PHYLIP (*http://evolution.genetics.washington.edu/phylip.html*) and PAUP (*http://paup.csit.fsu.edu/*) are widely-used packages for inferring phylogenic trees. Both contain over 20 different algorithms for constructing phylogenic trees from data. Although many of them are designed to work with molecular sequence data, several general methods accept arbitrary distance matrices as input.

Notes: Recent monographs on the Steiner tree problem include Hwang, Richards, and Winter [HRW92] and Prömel and Steger [PS02]. Du, et al. [DSR00] is a collection of recent surveys on all aspects of Steiner trees. Older surveys on the problem include [Kuh75]. Empirical results on Steiner tree heuristics include [SFG82, Vos92].

The Euclidean Steiner problem dates back to Fermat, who asked how to find a point p in the plane minimizing the sum of the distances to three given points. This was solved by Torricelli before 1640. Steiner was apparently one of several mathematicians who worked the general problem for n points, and was mistakenly credited with the problem. An interesting, more detailed history appears in [HRW92].

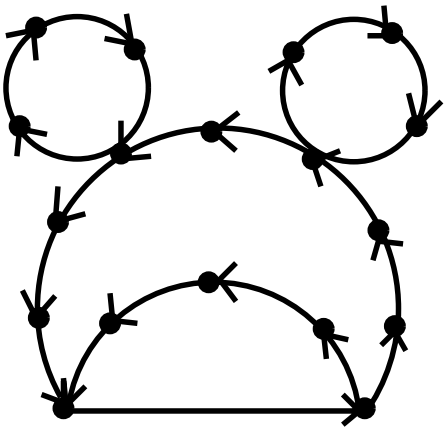
Gilbert and Pollak [GP68] first conjectured that the ratio of the length of the minimum Steiner tree over the MST is always $\geq \sqrt{3}/2 \approx 0.866$. After twenty years of active research, the Gilbert-Pollak ratio was finally proven by Du and Hwang [DH92]. The Euclidean MST for n points in the plane can be constructed in $O(n \lg n)$ time [PS85].

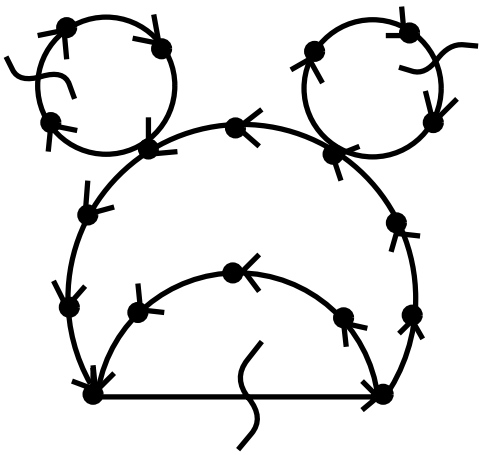
Arora [Aro98] gave a polynomial-time approximation scheme (PTAS) for Steiner trees in k-dimensional Euclidian space. A 1.55-factor approximation for Steiner trees on graphs is due to Robins and Zelikovsky [RZ05].

Expositions on the proof that the Steiner tree problem for graphs is hard [Kar72] include [Eve79a]. Expositions on exact algorithms for Steiner trees in graphs include [Law76]. The hardness of Steiner tree for Euclidean and rectilinear metrics was established in [GGJ77, GJ77]. Euclidean Steiner tree is not known to be in NP, because of numerical issues in representing distances.

Analogies can be drawn between minimum Steiner trees and minimum energy configurations in certain physical systems. The case that such analog systems—including the behavior of soap films over wire frames—"solve" the Steiner tree problem is discussed in [Mie58].

Related Problems: Minimum spanning tree (see page 484), shortest path (see page 489).





INPUT OUTPUT

16.11 Feedback Edge/Vertex Set

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

Problem description: What is the smallest set of edges E' or vertices V' whose deletion leaves an acyclic graph?

Discussion: Feedback set problems arise because many things are easier to do on directed acyclic graphs (DAGs) than general digraphs. Consider the problem of scheduling jobs with precedence constraints (i.e. , job A must come before job B). When the constraints are all consistent, the resulting graph is a DAG, and topological sort (see Section 15.2 (page 481)) can be used to order the vertices to respect them. But how can you design a schedule when there are cyclic constraints, such as A must be done before B, which must be done before C, which must be done before A?

By identifying a feedback set, we identify the smallest number of constraints that must be dropped to permit a valid schedule. In the *feedback edge* (or arc) set problem, we drop individual precedence constraints. In the *feedback vertex set* problem, we drop entire jobs and all constraints associated with them.

Similar considerations are involved in eliminating race conditions from electronic circuits. This explains why the problem is called "feedback" set. It is also known as the *maximum acyclic subgraph problem.*

One final application has to do with ranking tournaments. Suppose we want to rank order the skills of players at some two-player game, such as chess or tennis. We can construct a directed graph where there is an arc from x to y if x beats y in a game. The higher-ranked player *should* be at the lower-ranked player, although upsets often occur. A natural ranking is the topological sort resulting after deleting the minimum set of feedback edges (upsets) from the graph.

Issues in feedback set problems include:

- *Do* any *constraints have to be dropped?* – No changes are needed if the graph is already a DAG, which can be determined via topological sort. One way to find a feedback set modifies the topological sorting algorithm to delete whatever edge or vertex is causing the trouble whenever a contradiction is found. This feedback set might be much larger than needed, however, since feedback edge set and feedback vertex set are NP-complete on directed graphs.

- *How can I find a good feedback edge set?* – An effective linear-time heuristic constructs a vertex ordering and then deletes any arc going in the wrong direction. At least half the arcs must go either left-to-right or right-to-left for any vertex order, so take the smaller partition as your feedback set.

  But what is the right vertex order to start from? One natural order is to sort the vertices in terms of edge-imbalance, namely in-degree minus out-degree. Another approach starts by picking an arbitrary vertex v. Any vertex x defined by an in-going edge (x, v) will be placed to the left of v. Any x defined by out-going edge (v, x) will analogously be placed to the right of v. We can now recur on the left and right subsets to complete the vertex order.

- *How can I find a good feedback vertex set?* – The heuristics above yield vertex orders defining (hopefully) few back edges. We seek a small set of vertices that together cover these backedges. This is exactly a vertex cover problem, the heuristics for which are discussed in Section 16.3 (page 530).

- *What if I want to break all cycles in an undirected graph?* – The problem of finding feedback sets in undirected graphs is quite different from digraphs. Trees are undirected graphs without cycles, and every tree on n vertices contains exactly $n - 1$ edges. Thus the smallest feedback edge set of any undirected graph G is $|E| - (n - c)$, where c is the number of connected components of G. The back edges encountered during a depth-first search of G qualified as a minimum feedback edge set.

  The feedback vertex set problem remains NP-complete for undirected graphs, however. A reasonable heuristic uses breadth-first search to identify the shortest cycle in G. The vertices in this cycle are all deleted from G, and the shortest remaining cycle identified. This find-and-delete procedure is employed until the graph is acyclic. The optimal feedback vertex set must contain at least one vertex from each of these vertex-disjoint cycles, so the average deleted-cycle length determines just how good our approximation is.

It may pay to refine any of these heuristic solutions using randomization or simulated annealing. To move between states, we can modify the vertex permutation by swapping pairs in order or insert/delete vertices to/from the candidate feedback set.

Implementations: Greedy randomized adaptive search (GRASP) heuristics for both feedback vertex and feedback edge set problems have been implemented by Festa, et al. [FPR01] as Algorithm 815 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/*.

GOBLIN (*http://www.math.uni-augsburg.de/∼fremuth/goblin.html*) includes an approximation heuristic for minimum feedback arc set.

The econ_order program of the Stanford GraphBase (see Section 19.1.8 (page 660)) permutes the rows and columns of a matrix so as to minimize the sum of the numbers below the main diagonal. Using an adjacency matrix as the input and deleting all edges below the main diagonal leaves an acyclic graph.

Notes: See [FPR99] for a survey on the feedback set problem. Expositions of the proofs that feedback minimization is hard [Kar72] include [AHU74, Eve79a]. Both feedback vertex and edge set remain hard even if no vertex has in-degree or out-degree greater than two [GJ79].

Bafna, et al. [BBF99] gives a 2-factor approximation for feedback vertex set in undirected graphs. Feedback edge sets in directed graphs can be approximated to within a factor of $O(\log n \log \log n)$ [ENSS98]. Heuristics for ranking tournaments are discussed in [CFR06]. Experiments with heuristics are reported in [Koe05].

An interesting application of feedback arc set to economics is presented in [Knu94]. For each pair A, B of sectors of the economy, we are given how much money flows from A to B. We seek to order the sectors to determine which sectors are primarily producers to other sectors, and which deliver primarily to consumers.

Related Problems: Bandwidth reduction (see page 398), topological sorting (see page 481), scheduling (see page 468).