GRAPH THEORY

GRAPH THEORY FUNDAMENTALS

A *graph* $G(V, E)$ consists of a set V of *vertices* (or *nodes*) and a set E of pairs of vertices from V, referred to as *edges.* An edge may have associated with it a direction, in which case the graph is called *directed* (as opposed to *undirected*), or a *weight,* in which case the graph is called *weighted.* Two vertices $u, v \in V$ for which an edge $e = (u, v)$ exists in E are said to be *adjacent* and edge e is said to be *incident* on them. The *degree* of a vertex is the number of edges adjacent to it. A (*simple*) *path* is a sequence of distinct vertices $(a_0, a_1, \ldots, a_k)$ of V such that every two vertices in the sequence are adjacent. A (*simple*) *cycle* is a sequence of vertices $(a_0, a_1, \ldots, a_k, a_0)$ such that $(a_0, a_1, \ldots, a_k)$ is a path and a_k, a_0 are adjacent. A graph is *connected* if there is a path between every pair of vertices. A graph $G'(V', E')$ is a *subgraph* of graph

$G(V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A *spanning tree* of a connected graph $G(V, E)$ is a subgraph of G that comprises all vertices of G and has no cycles. Given a subset $V' \subseteq V$, the *induced subgraph* of $G(V, E)$ *by* V' is a subgraph $G'(V', E')$, where E' comprises all edges (u, v) in E with $u, v \in V'$.

In a directed graph, each edge (sometimes referred to as *arc*) is an ordered pair of vertices and the graph is denoted by $G(V, A)$. For an edge $(u, v) \in A$, v is called the *head* and u the *tail* of the edge. The number of edges for which u is a tail is called the *out-degree* of u and the number of edges for which u is a head is called the *in-degree* of u. A (*simple*) *directed path* is a sequence of distinct vertices $(a_0, a_1, \ldots, a_k)$ of V such that (a_i, a_{i+1}), $\forall i$, $0 \leq i \leq k - 1$, is an edge of the graph. A (*simple*) *directed cycle* is a sequence of vertices $(a_0, a_1, \ldots, a_k, a_0)$ such that $(a_0, a_1, \ldots, a_k)$ is a directed path and $(a_k, a_0) \in A$. A directed graph is *strongly connected* if there is a directed path between every pair of vertices. A directed graph is *weakly connected* if there is an undirected path between every pair of vertices.

Graphs are a very important modeling tool that can be used to model a great variety of problems in areas such as operations research [e.g., see (1,2,3)], very large scale integration (VLSI) computer-aided design (CAD) for digital circuits [e.g., see (4)], computer and communications networks [e.g., see (5)]. For example, in operations research a graph can be used to model the assignment of workers to tasks, the distribution of goods from warehouses to customers, etc. In VLSI CAD a graph can be used to represent a digital circuit at any abstract level of representation (gate level, module level, etc.). Each vertex in this case corresponds to a gate or module and each edge corresponds to a circuit line that connects the respective components. In computer and communications networks, a graph can be used to represent any given interconnection, with vertices representing host computers or routers and edges representing communication links.

There are many special cases of graphs. Some of the most common ones are listed below. A *tree* is a connected graph that contains no cycles. A *bipartite graph* is a graph with the property that its vertex set V can be partitioned into two disjoint subsets V_1 and V_2, $V_1 \cup V_2 = v$, such that every edge in E comprises one vertex from V_1 and one vertex from V_2. A *directed acyclic graph* is a directed graph that contains no directed cycles. Directed acyclic graphs can be used to represent combinational circuits in VLSI CAD. A *transitive graph* is a directed graph with the property that for any vertices u, v, $w \in V$ for which there exist edges (u, v), $(v, w) \in A$, edge (u, w) also belongs to A. A *planar graph* is a graph with the property that its edges can be drawn on the plane so as not to cross each other. A typical application of planar graphs is in VLSI, where the requirement is for all the circuit lines to be routed on a single layer.

A *cycle graph* is a graph that is obtained by a cycle with chords as follows: For every chord (a, b) of the cycle, there is a vertex $v_{(a,b)}$ in the cycle graph. There is an edge $(v_{(a,b)}, v_{(c,d)})$ in the cycle graph if and only if the respective chords (a, b) and (c, d) intersect. Cycle graphs find application in VLSI CAD as a channel with two terminal nets, or a switchbox with two terminal nets can be represented as a cycle graph. Then the problem of finding the maximum number of nets in the channel (or switchbox) that can be routed on the plane amounts to finding a maximum independent set in the respective cycle graph.

A *permutation graph* is a special case of a cycle graph. It is based on the notion of a *permutation diagram.* A permutation diagram is simply a sequence of N integers in the range from 1 to N (but not necessarily ordered). Given an ordering, there is a vertex for every integer in the diagram, and there is an edge (u, v) if and only if the integers u, v are not in the correct order in the permutation diagram. A permutation diagram can be used to represent a special case of a *permutable channel* in VLSI, where all nets have two terminals that belong to opposite channel sides. The problem of finding the maximum number of nets in the permutable channel that can be routed on the plane amounts to finding a maximum independent set in the respective permutation graph. This, in turn, amounts to finding the maximum increasing (or decreasing) subsequence of integers in the permutation diagram.

ALGORITHMS AND TIME COMPLEXITY

An *algorithm* is an unambiguous description of a finite set of operations for solving a computational problem in a finite amount of time. The set of allowable operations corresponds to the operations supported by a specific computing machine (computer) or to a model of that machine.

A *computational problem* comprises a set of parameters that have to satisfy a set of well-defined mathematical constraints. A specific assignment of values to these parameters constitutes an *instance* of the problem. For some computational problems there is no algorithm as defined above to find a solution. For example, the problem of determining whether an arbitrary computer program terminates in a finite amount of time given a set of input data cannot be solved (it is "undecidable") (6). For the computational problems for which there does exist an algorithm, the point of concern is how "efficient" that algorithm is. The *efficiency* of an algorithm is primarily defined in terms of how much time the algorithm takes to terminate. (Sometimes, other considerations such as the space requirement in terms of the physical information storage capacity of the computing machine are also taken into account, but in this exposition we concentrate exclusively on time.)

In order to formally define the efficiency of an algorithm, the following notions are introduced:

The *size* of an instance of a problem is defined as the total number of symbols for the complete specification of the instance under a finite set of symbols and a "succinct" encoding scheme. A "succinct" encoding scheme is considered to be a logarithmic encoding scheme, in contrast to a unary encoding scheme. The time requirement (*time complexity*) of an algorithm is expressed then as a function $f(n)$ of the size n of an instance of the problem and gives the total number of "basic" steps that the algorithm needs to go through to solve that instance. Most of the time, the number of steps is taken with regard to the worst case, although alternative measures like the average number of steps can also be considered. What constitutes a "basic" step is purposely left unspecified, provided that the time the basic step takes to be completed is bounded from above by a *constant,* that is, a value not dependent on the instance. This hides implementation details and machine-dependent timings and provides the required degree of general applicability.

An algorithm with time complexity $f(n)$ is said to be of the *order of* $g(n)$ [denoted as $O(g(n))$], where $g(n)$ is another function, if there is a constant c such that $f(n) \leq c \cdot g(n)$ for all $n \geq 0$. For example, an algorithm for finding the minimum element of a list of size n takes time $O(n)$, an algorithm for finding a given element in a sorted list takes time $O(\log n)$, algorithms for sorting a list of elements can take time $O(n^2)$, $O(n \log n)$, or $O(n)$ (the latter when additional information about the range of the elements is known). If moreover there are constants c_L and c_H such that $c_L \cdot g(n) \leq f(n) \leq c_H \cdot g(n)$ for all $n \geq 0$, then $f(n)$ is said to be $\Theta(g(n))$.

The smaller the "order-of" function, the more efficient an algorithm is generally taken to be, but in the analysis of algorithms, the term "efficient" is applied liberally to any algorithm whose "order-of" function is a polynomial $p(n)$. The latter includes time complexities like $O(n \log n)$ or $O(n\sqrt{n})$, which are clearly bounded by a polynomial. Any algorithm with a nonpolynomial time complexity is not considered to be efficient. All nonpolynomial-time algorithms are referred to as *exponential* and include algorithms with such time complexities as $O(2^n)$, $O(n!)$, $O(n^n)$, $O(n^{\log n})$ (the latter is sometimes referred to as *subexponential*). Of course, in practice, for an algorithm of polynomial time complexity $O(p(n))$ to be actually efficient, the degree of polynomial $p(n)$ as well as the constant of proportionality in the expression $O(p(n))$ should rather be small. In addition, because of the worst-case nature of the $O(\ )$ formulation, an "exponential" algorithm might exhibit exponential behavior in practice only in rare cases (the latter seems to be the case with the simplex method for linear programming). However, the fact on the one hand that most of the polynomial-time algorithms for the problems that occur in practice tend indeed to have small polynomial degrees and small constants of proportionality, and on the other that most nonpolynomial algorithms for the problems that occur in practice eventually resort to the trivial approach of exhaustively searching (enumerating) all candidate solutions, justifies the use of the term "efficient" for only the polynomial-time algorithms.

Given a new computational problem to be solved, it is of course desirable to find a polynomial-time algorithm to solve it. The determination of whether such a polynomial-time algorithm actually exists for that problem is a subject of primary importance. To this end, a whole discipline dealing with the classification of the computational problems and their interrelations has been developed.

P, NP, and *NP*-Complete Problems

The classification starts technically with a special class of computational problems known as decision problems. A computational problem is a *decision problem* if its solution can actually take the form of a yes or no answer. For example, the problem of determining whether a given graph contains a simple cycle that passes through every vertex is a decision problem (known as the Hamiltonian Cycle problem). In contrast, if the graph has weights on the edges and the goal is to find a simple cycle that passes through every vertex and has minimum sum of edge weights is not a decision problem, but an *optimization* problem (the latter problem is known as the Traveling Salesman problem). An optimization problem (sometimes referred to also as *combinatorial optimization problem*) seeks to find the best solution, in terms of a well-defined objective function $Q(\ )$, over a set of feasible solutions. Interestingly, every optimization problem has a "decision" version in which the goal of minimizing (or maximizing) the objective function $Q(\ )$ in the optimization problem corresponds to the question of whether there exists a solution with $Q(\ ) \leq k$ (or $Q(\ ) \geq k$) in the decision problem, where k is now an additional input parameter to the decision problem. For example, the decision version of the Traveling Salesman problem is, given a graph and an integer K, to find a simple cycle that passes through every vertex and whose sum of edge weights is no more than K.

All decision problems that can be solved in polynomial time comprise the so-called class P (for polynomial). Another established class of decision problems is the *NP* class which consists of all decision problems for which a polynomial-time algorithm can *verify* if a candidate solution (which has polynomial size with respect to the original instance) yields a yes or no answer. The initials *NP* stand for nondeterministic polynomial, in that if a yes answer exists for an instance of an *NP* problem, that answer can be obtained *nondeterministically* (in effect, guessed) and then verified in polynomial time. (The requirement for polynomial-time verification is readily met for most of the common problems, but there are problems, like the minimum equivalent expression (see below), for which this seems not to be the case.) Every problem in class P belongs clearly to *NP*, but the question of whether class *NP* strictly contains P or not is a famous unresolved problem. It is conjectured that $NP \neq P$, but there is no actual proof up to now. Notice that in order to simulate the nondeterministic guess in the statement above, an obvious deterministic algorithm would have to enumerate all possible cases, which is an exponential-time task. It is in fact the question of whether such an "obvious" algorithm is actually the best one can do that has not been resolved.

Showing that an *NP* decision problem actually belongs to P is equivalent to establishing a polynomial-time algorithm to solve that problem. In the investigation of the relations between problems in P and in *NP*, the notion of *polynomial reducibility* plays a fundamental role. A problem R is said to be *polynomially reducible* to another problem S if the existence of a polynomial-time algorithm for S implies the existence of a polynomial-time algorithm for R. That is, in more practical terms, if the assumed polynomial-time algorithm for problem S is viewed as a subroutine, then an algorithm that solves R by making a polynomially bounded number of calls to that subroutine and taking a polynomial amount of time for some extra work would constitute a polynomial-time algorithm for R.

There is a special class of *NP* problems with the property that if and only if *any one* of those problems could be solved polynomially, then so would *all* of the *NP* problems (i.e., *NP* would be equal to P). These *NP* problems are known as NP-*complete.* An *NP*-complete problem is an *NP* problem to which every other *NP* problem reduces polynomially. The first problem to be shown *NP*-complete was the Satisfiability problem (6). This problem concerns the existence of a truth assignment to a given set of boolean variables so that the conjunction of a given set of disjunctive clauses formed from these variables and their complements becomes true. The proof (given by Stephen Cook in 1971) was done by showing that every *NP* problem reduces polynomially to the Satisfiability problem. After the establishment of the first *NP*-complete case, an extensive

and on-going list of *NP*-complete problems has been established [see (6)]. The interest in showing that a particular problem R is *NP*-complete lies exactly in the fact that if it finally turns out that *NP* strictly contains P, then R cannot be solved polynomially (or, from another point of view, if a polynomial-time algorithm happens to be discovered for R, then $NP = P$). The process of showing that a decision problem is *NP*-complete involves showing that the problem belongs to *NP* and that some known *NP*-complete problem reduces polynomially to it. The difficulty of this task lies in the choice of an appropriate *NP*-complete problem to reduce from as well as in the mechanics of the polynomial reduction. An example of an *NP*-complete proof is given below.

We consider a problem that occurs in the testing of digital circuits (7): We are given a collection C of subsets of a set U with maximum subset size w, and an integer bound $k \leq w$. We want to determine whether there exists a mapping $f: U \rightarrow [1 \ldots |U|]$, so that for each set s in C, the corresponding set $s_f = \{f(a) \bmod k : a \in s\}$ has size $\min(|s|, k)$. This problem (referred to as the SETMOD problem) is *NP*-complete and this is established as follows (8):

Theorem 1 The SETMOD problem is *NP*-complete.

Proof. The problem belongs clearly to *NP*, since once a satisfying mapping f has been guessed, the verification can be done in polynomial (actually linear) time. We make the reduction from a known *NP*-complete problem that is called Not-All-Equal-3SAT (NAE) (6). The latter problem is a special case of the Satisfiability problem in which each disjunctive clause comprises exactly three literals and the goal is to find a truth assignment so that each clause has at least one true and at least one false literal. Let φ be a NAE instance with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. For each variable x_i, we consider the set $\{x_i, \bar{x}_i\}$, and for each clause $C_j = x_{j1} \vee x_{j2} \vee x_{j3}$, we consider the set $\{x_{j1}, x_{j2}, x_{j3}\}$ (each element in each clause is actually identical with some variable or its complement.) Set U is the set of all literals, that is, $|U| = 2n$.

Let $k = 2$. Suppose first that φ is satisfiable. For variable x_i, $1 \leq i \leq n$, we assign $f(x_i) = 2i - 1$, $f(\bar{x}_i) = 2i$ if the variable is true and $f(x_i) = 2i$, $f(\bar{x}_i) = 2i - 1$ if the variable is false. Since each clause has at least one true and one false literal, we have that each set s has at least one element with remainder 0 and at least one element with remainder 1 modulo $k = 2$, namely $|s'| = k = \min(|s|, k)$. Conversely, suppose that there is a solution for an instance of the problem in question that consists of the above collection of sets and $k = 2$. Then, since the three elements in each clause set cannot all be labeled odd or even, and also the two elements in each literal set cannot both be labeled odd or even, we have a satisfying assignment for the NAE instance.

Some representative *NP*-complete problems that occur in various areas in operations research, digital design automation, and computer networks are listed below.

- *3-Satisfiability.* Given a set of boolean variables and a set of disjunctive clauses over the variables, each one comprising exactly three literals, is there a satisfying assignment for the conjunction of all the clauses?
- *Hamiltonian Cycle.* Given a graph $G(V, E)$, does G contain a simple cycle that includes all the vertices of G?
- *Longest Path.* Given a graph $G(V, E)$ and an integer $K \leq |V|$, is there a simple path in G with at least K edges?
- *Vertex Cover.* Given a graph $G(V, E)$ and an integer $K \leq |V|$, is there a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge $(u, v) \in E$, at least one of u, v belongs to V'?
- *Independent Set.* Given a graph $G(V, E)$ and an integer $K \leq |V|$, is there a subset $V' \subseteq V$ such that $|V'| \geq K$ and no two vertices in V' are joined by an edge in E?
- *Feedback Vertex Set.* Given a directed graph $G(V, A)$ and an integer $K \leq |V|$, is there a subset $V' \subseteq V$ such that $|V'| \leq K$ and every directed cycle in G has at least one vertex in V'?
- *Graph Colorability.* Given a graph $G(V, E)$ and an integer $K \leq |V|$, is there a "coloring" function $f: V \rightarrow 1, 2, \ldots, K$ such that for every edge $(u, v) \in E$, $f(u) \neq f(v)$?
- *Graph Bandwidth.* Given a graph $G(V, E)$ and an integer $K \leq |V|$, is there a one-to-one function $f: V \rightarrow 1, 2, \ldots, |V|$ such that for every edge $(u, v) \in E$, $|f(u) - f(v)| \leq K$?
- *Graph Isomorphism.* Given two graphs $G(V_1, E_1)$ and $G(V_2, E_2)$, is there a one-to-one function $f: V_1 \rightarrow V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$?
- *Induced Bipartite Subgraph.* Given a graph $G(V, E)$ and an integer $K \leq |V|$, is there a subset $V' \subseteq V$ such that $|V'| \geq K$ and the subgraph induced by V' is bipartite?
- *Planar Subgraph.* Given a graph $G(V, E)$ and an integer $K \leq |E|$, is there a subset $E' \subseteq E$ such that $|E'| \geq K$ and the subgraph $G'(V, E')$ is planar?
- *Steiner Tree.* Given a weighted graph $G(V, E)$, a subset $V' \subseteq V$, and a positive integer bound B, is there a subgraph of G that is a tree, comprises at least all vertices in V', and has a total sum of weights no more than B?
- *Graph Partitioning.* Given a graph $G(V, E)$ and two positive integers K and J, is there a partition of V into disjoint subsets $V_1, V_2, \ldots, V_m$ such that each subset contains no more than K vertices and the total number of edges that are incident on vertices in two different subsets is no more than J?
- *Traveling Salesman.* Given a set of cities, a distance between every pair of cities, and a bound K, is there a tour that visits each city exactly once and has total distance no more than K?
- *Bin Packing.* Given a finite set S of positive integers, and two positive integers B and K, is there a partition of S into K disjoint subsets $S_1, S_2, \ldots, S_K$ such that the sum of the elements in each S_i is no more than B?
- *Subset Sum.* Given a finite set S of positive integers and a positive integer B, is there a subset $S' \subseteq S$ whose sum of elements is exactly B?
- *Knapsack.* Given a finite set S of items, each with an integer size and an integer value, and two integer bounds B and P, is there a subset $S' \subseteq S$ whose total sum of sizes is at most B, and the total sum of values is at least P?
- *One-Processor Scheduling with Release Times and Deadlines.* Given a set T of tasks, each task $t \in T$ having a duration $l(t) \in Z^+$, a release time $r(t) \in Z_0^+$, and a deadline $d(t) \in Z+$, is there a function $q: T \rightarrow Z_0^+$ such that for all $t \in T$, $q(t) \geq r(t)$, $q(t) + l(t) \leq d(t)$, and for any pair t, t' with $q(t) > q(t')$, $q(t) \geq q(t') + l(t')$?

- *Multiprocessor Scheduling.* Given a set T of tasks, each task $t \in T$ having a duration $l(t) \in Z^+$, and two positive integers m and D, is there a function $q : T \to Z_0^+$ such that the number of tasks that have overlapping intervals $[q(t), q(t) + l(t)]$ is no more than m and $\max_{t \in T} \{(q(t) + l(t))\} \leq D$?
- *Integer Linear Programming.* Given K integer vectors x_i and K integers b_i, $1 \leq i \leq K$, as well as an integer vector c and an integer B, is there an integer vector y such that $x_i \cdot y \leq b_i$, $1 \leq i \leq K$, and $c \cdot y \geq B$?

NP-Hard Problems

A generalization of the *NP*-complete class is the *NP*-hard class. The *NP*-hard class is extended to comprise optimization problems, as well as decision problems that do not seem to belong to *NP*. All that is required for a problem to be *NP*-hard is that some *NP*-complete problem reduce polynomially to it. For example, the optimization version of the Traveling Salesman problem is an *NP*-hard problem, since if it were polynomially solvable, the decision version of the problem (which is *NP*-complete) would be trivially solved polynomially. An example of a decision problem that is *NP*-hard but not known to be *NP*-complete is the *K*th Heaviest Subset problem: Given a finite set S of integers and two integers K and B, are there K distinct subsets $S_1, S_2, \ldots, S_K \subseteq S$, each of which has a sum of elements at least B? This problem has been shown to be *NP*-hard by a reduction from the Partition problem (6), but it is not known to be in *NP* since the obvious candidate solution to be verified (i.e., the list of the K subsets) does not have polynomial size with respect to the original instance (as K can be as large as $2^{|S|}$). Another problem that is *NP*-hard but not known to belong to *NP* is the Minimum Equivalent Expression problem: Given a well-formed boolean expression E involving a set of variables, the constants "true" and "false," and the logical connectives "and," "or," "not," and "implies," and a positive integer K, is there another expression E' that is equivalent to E and contains no more than K literals? Minimum Equivalent Expression is *NP*-hard since the Satisfiability problem reduces polynomially to it. But it is not known to be in *NP* because, although a candidate solution E' can be described polynomially with respect to the original instance size, the obvious verification of that solution involves the use of an algorithm for the Satisfiability of Boolean Expressions problem, which is itself *NP*-complete (as a generalization of the Satisfiability problem).

If the decision version of an optimization problem is *NP*-complete, then the optimization problem is *NP*-hard, since the yes or no answer sought in the decision version can readily be given in polynomial time once the optimum solution in the optimization version has been obtained. But it is also the case that for most *NP*-hard optimization problems, a reverse relation holds: That is, these *NP*-hard optimization problems can reduce polynomially to their *NP*-complete decision versions. The strategy is to use a binary search procedure that establishes the optimal value after a logarithmically bounded number of calls to the decision version subroutine. Such *NP*-hard problems are sometimes referred to as *NP-equivalent.* The latter fact is another motivation for the study of *NP*-complete problems: a polynomial-time algorithm for any *NP*-complete (decision) problem would provide actually a polynomial-time algorithm for all such *NP*-equivalent optimization problems.

Algorithms for *NP*-Hard Problems

Once a new problem for which an algorithm is sought is proved to be *NP*-complete or *NP*-hard, the search for a polynomial-time algorithm is abandoned (unless one seeks to prove that $NP = P$), and the following basic four approaches remain to be followed:

1. Try to improve as much as possible over the straightforward exhaustive (exponential) search by using techniques like branch-and-bound, dynamic programming, cutting plane methods, or Lagrangian techniques.
2. For optimization problems, try to obtain a polynomial-time algorithm that finds a solution that is probably close to the optimal. Such an algorithm is known as *approximation algorithm* and is generally the next best thing one can hope for to solve the problem.
3. For problems that involve numerical bounds, try to obtain an algorithm that is polynomial in terms of the instance size *and* the size of the maximum number occuring in the encoding of the instance. Such an algorithm is known as *pseudopolynomial-time algorithm* and becomes practical if the numbers involved in a particular instance are not too large. An *NP*-complete problem for which a pseudopolynomial-time algorithm exists is referred to as *weakly* NP-*complete* (as opposed to *strongly* NP-*complete*).
4. Use a polynomial-time algorithm to find a "good" solution based on rules of thumb and insight. Such an algorithm is known as a heuristic. No proof is provided about how good the solution is, but well-justified arguments and empirical studies justify the use of these algorithms in practice.

In addition, before any of these approaches is examined, one should check whether the problem of concern is actually a special case of an *NP*-complete or an *NP*-hard problem, since many special cases can be solved polynomially or pseudopolynomially. Examples include polynomial algorithms for finding a longest path in a directed acyclic graph, finding a maximum independent set in a transitive graph, finding a schedule in the One-Processor Scheduling with Release Times and Deadlines problem when all tasks have unit length; pseudopolynomial-time algorithms for solving the Bin Packing problem when the number K of bins is fixed, finding a schedule in the One-Processor Scheduling with Release Times and Deadlines problem when the release times and deadlines are bounded by a constant, and so on.

In the next section we give some more information about approximation and pseudopolynomial-time algorithms.

POLYNOMIAL-TIME ALGORITHMS

Graph Representations and Traversals

There are two basic schemes for representing graphs in a computer program. Without loss of generality, we assume that the graph is directed [represented by $G(V, A)$]. Undirected graphs can always be considered as bi-directed. In the first scheme, known as the *adjacency matrix* representation, a $|V| \times |V|$ matrix M is used, where every row and column of the matrix corresponds to a vertex of the graph, and entry

$M(a, b)$ is 1 if and only if $(a, b) \in A$. This simple representation requires $O(|V|^2)$ time and space.

In the second scheme, known as the *adjacency list* representation, an array $L[1 \ldots |V|]$ of linked lists is used. The linked list starting at entry $L[i]$ contains the set of all vertices that are the heads of all edges with tail vertex i. The time and space complexity of this scheme is $(|V| + |E|)$.

Both schemes are widely used as part of polynomial-time algorithms for working with graphs [e.g., see (9)]. The adjacency list representation is more economical to construct, but locating an edge using the adjacency matrix representation is very fast (takes $O(1)$ time compared to the $O(|V|)$ time required using the adjacency list representation). The choice between the two depends on the way the algorithm needs to access the information on the graph.

A basic operation on graphs is the *graph traversal,* where the goal is to visit systematically all the vertices of the graph. There are three graph traversal methods: *Depth-first search* (*DFS*), *breadth-first search* (*BFS*), and *topological search*. The last applies only to directed acyclic graphs. Assume that all vertices are marked initially as unvisited, and that the graph is represented using an adjacency list L.

Depth-first search traverses a graph following the deepest (forward) direction possible. The algorithm starts by selecting the lowest numbered vertex v and marking it as visited. DFS selects an edge (v, u), where u is still unvisited, marks u as visited, and starts a new search from vertex u. After completing the search along all paths starting at u, DFS returns to v. The process is continued until all vertices reachable from v have been marked as visited. If there are still unvisited vertices, the next unvisited vertex w is selected and the same process is repeated until all vertices of the graph are visited.

The following is a recursive implementation of subroutine of DFS(v) that determines all the vertices reachable from a selected vertex v. $L[v]$ represents the list of all vertices that are the heads of edges with tail v, and array $M[u]$ contains the visited or unvisited status of every vertex u.

```
Procedure DFS(v)
M[v] := visited;
FOR each vertex u ∈ L[v] DO
    IF M[u] = unvisited THEN Call DFS(u);
END DFS
```

The time complexity of DFS(v) is $O(|V_v| + |E_v|)$ where $|V_v|$, $|E_v|$ are all the number of vertices and edges that have been visited by DFS(v). The total time for traversing the graph using DFS is $O(|E| + |V|) = O(|E|)$.

Breadth-first search visits all vertices at distance k from the lowest numbered vertex v before visiting any vertices at distance $k + 1$. Breadth-first search constructs a breadth-first search tree, initially containing only the lowest numbered vertex. Whenever an unvisited vertex w is visited in the course of scanning the adjacency list of an already visited vertex u, vertex w and edge (u, w) are added to the tree. The traversal terminates when all vertices have been visited. The approach can be implemented using queues so that it terminates in $O(|E|)$ time.

The final graph traversal method is the topological search which applies only to directed acyclic graphs. In directed acyclic graphs there are vertices that have no incoming edges and vertices that have no outgoing edges. Topological search visits a vertex only if it has no incoming edges or all its incoming edges have been explored. The approach can also be implemented to run in $O(|E|)$ time.

Design Techniques for Polynomial-Time Algorithms

There are three frameworks that can be used to obtain polynomial-time algorithms for combinatorial optimization (or decision) problems: (1) greedy algorithms, (2) divide-and-conquer algorithms, and (3) dynamic programming algorithms.

Greedy Algorithms. These are algorithms that use a greedy (straightforward) approach to solve a combinatorial optimization problem. Consider the following problem, known as the *Program Storage problem.* The instance consists of a set of n programs that are to be stored on a tape of length L. Every program has an integer length l_i, $1 \le i \le n$. When a program is to be retrieved, the tape is positioned at the start. Thus, if the order of the programs in the tape is $p_1, p_2, \ldots, p_n$, the time to retrieve program $p_m = \sum_{k=1}^{m} l_{p_k}$. The goal is to find the best possible way of storing the programs on the tape so that the total program retrieval time $\sum_{j=1}^{n} \sum_{k=1}^{m} l_{p_k}$ is minimized.

The greedy algorithm for this problem is to store the programs on the tape in increasing order of their lengths. The time complexity of this algorithm is $O(n \log n)$ and is determined by the procedure that sorts the programs according to their lengths. We refer the reader to Ref. (9) for sorting algorithms.

This greedy algorithm is very simple, and we avoid a more formal description. This is normally the case for all greedy algorithms. Despite the simplicity in their description, the proof of the optimality of a greedy solution is not a trivial task. In general, the proof is based on a systematic sequence of contradiction arguments. The following theorem proves that the greedy algorithm for the program storage problem is optimal (10). Similar methodology to the one in the proof of the theorem can be used to show the optimality of greedy algorithms.

Theorem 2 If $l_1 \le l_2 \le \ldots \le l_n$, the ordering $i_j = j$, $1 \le j \le n$ minimizes $\sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j}$ over all ordering permutations.

Proof. Let $I = i_1, i_2, \ldots i_n$ be the optimal permutation of the index set $\{1, 2, \ldots, n\}$. Then the retrieval time $R(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j}$ is equal to $\sum_{k=1}^{n} (n - k + 1) l_{i_k}$.

Assume that the opposite holds and that in the optimal ordering there exist programs a, b such that $a < b$ and $l_{i_a} > l_{i_b}$ (here a and b denote the relative positions of the programs in the permutation). In particular, let a and b be the two leftmost programs for which the above condition holds. If this is the case, interchanging the order of i_a and i_b results in a permutation I' for which it is shown that the total program retrieval time $R(I')$ is less than $R(I)$. This is enough to show that this greedy algorithm is optimal, and for any two programs a and b (from left to right in the current program permutation) for which $l_{i_a} > l_{i_b}$, the same argument can be applied to reduce the retrieval cost $R(\ )$. That is, the cost of the greedy permutation is no more than the cost of the optimal permutation I, and thus it is optimal.

The fact that the total program retrieval time $R(I')$ is less than $R(I)$ is shown as follows: Observe that $R(I') = \sum_{k, \neq a, b} ((n - k + 1) l_{i_k}) + (n - a + 1) l_{i_b} + (n - b + 1) l_{i_a}$. Then $R(I') - R(I) = (n - a + 1)(l_{i_a} - l_{i_b}) + (n - b + 1)(l_{i_b} - l_{i_a}) = (b - a)(l_{i_a} - l_{i_b}) > 0$.

Divide-and-Conquer Algorithms. This methodology is based on a systematic partition of the input instance in a top-down manner into smaller instances until small enough instances are obtained for which the problem degenerates to trivial computations. The overall optimal solution, which is the optimal solution on the input instance, is then calculated by appropriately working on the already calculated results on the subinstances. The recursive nature of the methodology necessitates the solution of one or more recurrence relations for determining the execution time.

As an example, we show how divide-and-conquer can be applied to find the maximum and the minimum integer in an array $A[1\ .\ .\ n]$. More explicitly, the goal is to assign to variables *max* and *min* the largest and smallest integers in the array. This problem is central in many areas of computer science, computer engineering, and operations research, among others.

The idea is to recursively find the maximum and the minimum element in subarrays $A[i\ .\ .\ j]$ that have at least two elements by first locating the midpoint m, and recursively solve the problem in the two smaller arrays $A[i\ .\ .\ m]$ and $A[m+1\ .\ .\ j]$. Once the maximum and minimum on the latter two arrays are computed, we take the maximum of the two calculated maxima and the minimum of the two calculated minima and we assign them as the maximum and the minimum of $A[i\ .\ .\ j]$, respectively.

```
Procedure MaxMin(i, j)
IF i = j THEN rmax = A[i]; rmin = A[i];
ELSE IF i = j − 1 THEN IF A[i] < A[j] THEN
  rmax = A[j]; rmin = A[i];
  ELSE rmax = A[i]; rmin = A[j];
ELSE
    m = ⌊(i + j)/2⌋;
    (m1, m2) := MaxMin(i, m); (m3, m4) :=
      MaxMin(m + 1, j);
    rmax = max(m1,m3); rmin = min(m2,m4);
Return(rmax, rmin);
END MaxMin
```

We analyze the time complexity of the approach by counting element comparisons. Let $T(n)$ be this number. The recursive nature of the MaxMin procedure allows us to express $T(n)$ by the following recurrence relation:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2 \text{ when } n > 2$$

$$T(n) = 1 \text{ when } n = 2, \quad T(n) = 0 \text{ when } n = 1$$

When n is a power of 2, exactly $3n/2 - 2$ number of comparisons are needed in the average, worst, and best case. In general, the worst case on the number of comparisons is $O(n)$.

Dynamic Programming Algorithms. In dynamic programming the optimal solution is calculated by starting from the simplest subinstances and combining the solutions of the smaller subinstances to solve larger subinstances, in a bottom-up manner. In order to guarantee a polynomial-time algorithm, the total number of subinstances that have to be solved must be polynomially bounded. Once a subinstance has been solved, any larger subinstance that needs that subinstance's solution, does not recompute it, but rather looks it up in table where it has been stored. Dynamic programming is applicable only to problems which obey the *principle of optimality.* This principle holds whenever in an optimal sequence of choices, each subsequence is also optimal. The difficulty in this approach is to come up with a decomposition of the problem into a sequence of subproblems for which the principle of optimality holds and can be applied in polynomial time.

We illustrate the approach by finding the maximum independent set in a cycle graph in $O(n^2)$ time, where n is the number of chordal endpoints in the cycle [see (4)]. Note that the maximum independent set is *NP*-hard on general graphs.

Let $G(V, E)$ be a cycle graph, and that $v_{ab} \in V$ corresponds to a chord in the cycle. We assume that no two of the n chords share an end point, and that the end points are labeled from 0 to $2n - 1$ clockwise around the cycle. Let G_{ij} be the subgraph induced by the set of vertices $v_{ab} \in V$ such that $i \le a$, $b \le j$.

Let $M(i, j)$ denote a maximum independent set of $G_{i,j}$. $M(i, j)$ is computed for every pair of chords, but $M(i, a)$ must be computed before $M(i, b)$ if $a < b$. Observe that if $i \ge j$, $M(i, j) = 0$ because $G_{i,j}$ has no chords. In general, in order to compute $M(i, j)$, the end point k of the chord with end point j must be found. If k is not in the range $[i, j - 1]$, then $M(i, j) = M(i, j - 1)$ because graph $G_{i,j}$ is identical to graph $G_{i,j-1}$. Otherwise, we consider two cases: First, if $v_{kj} \in M(i, j)$, then $M(i, j)$ does not have any vertex v_{ab} where $a \in [i, k - 1]$ and $b \in [k + 1, j]$. In this case $M(i, j) = M(i, k - 1) \cup M(K = 1, j - 1) \cup v_{kj}$; second, if $v_{kj} \notin M(i, j)$, then $M(i, j) = M(i, j - 1)$. Either of these two cases may apply, but the largest of the two maximum independent sets will be allocated to $M(i, j)$. The flowchart of the algorithm is given below:

```
Procedure MIS(V)
FOR j = 0 TO 2N − 1 DO
    Let (j, k) be the chord whose one end point
      is j;
    FOR i = 0 TO j − 1 DO
       IF i ≤ k ≤ j − 1 AND |M(i, k − 1)| + 1
         + |M(k + 1, j − 1)| > |M(i, j − 1)| THEN
         M(i, j) = M(i, k − 1) ∪ v_kj ∪ M(k + 1,
           j − 1);
       ELSE M(i, j) = M(i, j − 1);
END MIS
```

Three Basic Graph Problems

In this section we define formally and present polynomial time algorithms for three problems that are widely used in operations research, networking, and VLSI CAD, in the sense that many problems are reduced to solving these basic graph problems. They are the *shortest path problem,* the *flow problem,* and the *matching problem.*

Shortest Paths. The instance consists of a graph $G(V, E)$ with lengths $l(u, v)$ on its edges (u, v), a given source $s \in V$ and a target $t \in V$. We assume without loss of generality that the graph is directed. The goal is to find a shortest length path from s to t. The weights can be positive or negative numbers but there is no cycle for which the sum of the weights on its edges is negative. (If negative length cycles are allowed the problem is *NP*-hard.) Variations of the problem include

the all-pair of vertices shortest paths, and the m shortest path calculation in a graph.

We present here a dynamic programming algorithm for the shortest path problem which is known as the Bellman-Ford algorithm. The algorithm has $O(n^3)$ time complexity, but faster algorithms exist when all the weights are positive [e.g., the Dijkstra algorithm with complexity $O(n \cdot \min\{\log |E|, |V|)\}$] or when the graph is acyclic (based on topological search and with linear time complexity). All existing algorithms for the shortest path problem are based on dynamic programming. The Bellman–Ford algorithm works as follows:

Let $l(i, k)$ be the length of edge (i, j) if directed edge (i, j) exists and ∞ otherwise. Let $s(j)$ denote the length of the shortest path from the source s to vertex j. Assume that the source has label 1 and that the target has label $n = |V|$. We have that $s(1) = 0$. We also know that in a shortest path to any vertex j there must exist a vertex k, $k \neq j$, such that $s(j) = s(k) + l(k, j)$. Therefore

$$s(j) = \min_{k \neq j}\{s(k) + l(k, j)\}, \qquad j \geq 2$$

Bellman–Ford's algorithm, which eventually computes all $s(j)$, $1 \leq j \leq n$, calculates optimally the quantity $s(j)^{m+1}$ defined as the length of the shortest path to vertex j subject to the condition that the path does not contain more than $m + 1$ edges, $0 \leq m \leq |V| - 2$. In order to be able to calculate quantity $s(j)^{m+1}$ for some value $m + 1$, the $s(j)^m$ values for all vertices j have to be calculated.

Given the initialization $s(1)^1 = 0$, $s(j)^1 = l(1, j)$, $j \neq 1$, the computation of $s(j)^{m+1}$ for any values of j and m can be recursively computed using the formula

$$s(j)^{m+1} = \min\{s(j)^m\}, \quad \min\{s(k)^m + l(k, j)\}$$

The computation terminates when $m = |V| - 1$, because no shortest path has more than $|V| - 1$ edges.

Flows. All flow problem formulations consider a directed or undirected graph $G + (V, E)$, a designated source s, a designated target t, and a nonnegative integer capacity $c(i, j)$ on every edge (i, j). Such a graph is sometimes referred to as a *network*. We assume that the graph is directed. A *flow* from s to t is an assignment F of numbers $f(i, j)$ on the edges, called the amount of flow through edge (i, j), subject to the following conditions:

$$0 \leq f(i, j) \leq c(i, j) \tag{1}$$

Besides s and t, any vertex i must satisfy the *conservation of flow*. That is,

$$\sum_j f(j, i) - \sum_j f(i, j) = 0 \tag{2}$$

Let $v = \sum_j f(s, j)$. Then clearly $\sum_j f(s, j) = v = \sum_j f(j, t)$. v is called the value of the flow. From Eq. (2) we have

$$\begin{aligned} \sum_j f(j, i) - \sum_j f(i, j) &= -v \qquad \text{if } i = 2 \\ \sum_j f(i, j) &= 0 \qquad \text{if } i \neq s, t \\ \sum_j f(i, j) &= v \qquad \text{if } i = t \end{aligned} \tag{3}$$

A flow F that satisfies Eqs. (1) and (3) is called *feasible. In the max flow problem* the goal is to find a feasible flow F for which v is maximized. Such a flow is called a *maximum flow.* There is a problem variation, called the *minimum flow problem,* where condition (1) is substituted by $f(i, j) \geq c(i, j)$ and the goal is to find a flow F for which v is minimized. The minimum flow problem can be solved by modifying algorithms that compute the maximum flow in a graph.

Finally, another flow problem formulation is the *minimum cost flow problem.* Here each edge has, in addition to its capacity $c(i, j)$, a *cost* $p(i, j)$. If $f(i, j)$ is the flow through the edge, then the cost of the flow through the edge is $p(i, j) \cdot f(i, j)$ and the overall cost C for a flow F of a value v is $\sum_{i,j} p(i, j) \cdot f(i, j)$. The problem is to find a *minimum cost flow* F for a given value v.

Many problems in operations research, networking, scheduling, and VLSI CAD can be modeled or reduced to one of these three flow problem variations. All three problems can be solved in polynomial time using as subroutines shortest path calculations. Here we describe an $O(|V^3|)$ algorithm for the maximum flow problem. However, faster algorithms exist in the literature. We first give some definitions and theorems.

Let P be an *undirected* path from s to t, i.e., the direction of the edges is ignored. An edge $(i, j) \in P$ is said to be a *forward* edge if it is directed from s to t and *backward* otherwise. P is said to be an *augmenting path* with respect to a given flow F if $f(i, j) < c(i, j)$ for each forward edge, and $f(i, j) > 0$ for each backward edge in P.

Observe that if the flow in each forward edge of the augmenting path is increased by one unit and the flow in each backward edge is decreased by one unit, the flow is feasible and its value has been increased by one unit. We will show that a flow has maximum value if and only if there is no augmenting path in the graph. Then the maximum flow algorithm is simply a series of calls to a subroutine that finds an augmenting path and increments the value of the flow as described earlier.

Let $S \subset V$ be a subset of the vertices. The pair (S, T) is called a *cutset* if $T = V - S$. If $s \in S$ and $t \in T$, the (S, T) is called an (s, t) *cutset.* The *capacity of the cutset* (S, T) is defined as $c(S, T) = \sum_{i \in S} \sum_{j \in T} c(i, j)$, which is the sum of the capacities of all edges from S to T. We note that many problems in networking, operations research, scheduling, and VLSI CAD (physical design, synthesis, and testing) are formulated as minimum capacity (s, t) cutset problems. We show below that the minimum capacity (s, t) problem can be solved with a maximum flow algorithm.

Lemma 3.1 The value of any (s, t) flow cannot exceed the capacity of any (s, t) cutset.

Proof. Let F be an (s, t) flow with value v. Let (S, T) be an (s, t) cutset. From Eq. (3) the value of the flow v is also $v = \sum_{i \in S}(\sum_j f(i, j) - \sum_j f(j, i)) = \sum_{i \in S} \sum_{j \in S}(f(i, j) - f(j, i)) + \sum_{i \in S} \sum_{j \in T}(f(i, j) - f(j, i)) = \sum_{i \in S} \sum_{j \in T}(f(i, j) - f(j, i))$, since $\sum_{i \in S} \sum_{j \in S}(f(i, j) - f(j, i))$ is 0. But $f(i, j) \leq c(i, j)$ and $f(j, i) \geq 0$. Therefore $v \leq \sum_{i \in S} \sum_{j \in S} c(i, j) = c(S, T)$.

Theorem 3 A flow F has maximum value v if and only if there is no augmenting path from s to t.

Proof. If there is an augmenting path then we can modify the flow to get a larger value flow. This contradicts the assumption that the original flow has a maximum value.

Suppose, on the other hand, that F is a flow such that there is no augmenting path from s to t. We want to show that F has the maximum flow value. Let S be the set of all the vertices j (including s) for which there is an augmenting path from s to j. By the assumption that there is an augmenting path from s to t, we must have that $t \notin S$. Let $T = V - S$ (recall that $t \in T$). From the definition of S and T, it follows that $f(i, j) = c(i, j)$ and $f(j, i) = 0$, $\forall i \in S, j \in T$.

Now $v = \sum_{i \in S}(\sum_j f(i, j) - \sum_j f(j, i)) = \sum_{i \in S} \sum_{j \in S}(f(i, j) - f(j, i)) + \sum_{i \in S} \sum_{j \in T}(f(i, j) - f(j, i)) = \sum_{i \in S} \sum_{j \in S} c(i, j)$, since $c(i, j) = f(i, j)$ and $f(j, i) = 0$, $\forall i, j$. By Lemma 3.1 the flow has the maximum value.

Next we state two theorems whose proof is rather straightforward.

Theorem 4 If all the capacities are integers, then there exists a maximum flow F, where all $f(i, j)$ are integers.

Theorem 5 The maximum value of an (s, t) flow is equal to the minimum capacity of an (s, t) cutset.

Finding an augmenting path in a graph can be done by a systematic graph traversal in linear time. Thus a straightforward implementation of the maximum flow algorithm repeatedly finds an augmenting path and increments the amount of the (s, t) flow. This is a pseudopolynomial-time algorithm (see the next section), whose worst-case time complexity is $O(v \cdot |E|)$. In many cases such an algorithm may turn out to be very efficient. For example, when all capacities are uniform, then the overall complexity becomes $O(|E|^2)$.

In general, the approach needs to be modified using the Edmonds–Karp modification (1) so that each flow augmentation is made along an augmenting path with a minimum number of edges. With this modification, it has been proven that a maximum flow is obtained after no more than $|E| \cdot |V|/2$ augmentations, and the approach becomes fully polynomial. Faster algorithms for maximum flow computation rely on capacity scaling techniques and are described in (9,3), among others.

Matchings. Matching problems are defined on undirected graphs $G(V, E)$. A *matching* in a graph is a set $M \subset E$ such that no two edges in M are incident to the same vertex.

The *maximum cardinality matching* problem is the most common version of the matching problem. Here the goal is to obtain a matching so that the size (cardinality) of M is maximized.

In the *maximum weighted matching* version, each edge $(i, j) \in V$ has a nonnegative integer weight, and the goal is to find a matching M so that $\sum_{e \in M} w(e)$ is maximized.

In the *min-max matching problem* the goal is to find a maximum cardinality matching M where the minimum weight on an edge in M is maximized. The *max-min matching problem* is defined in an analogous manner.

All the above matching variations are solvable in polynomial time and find important applications. For example, a variation of the min-cut graph partitioning problem that is central in physical design automation for VLSI asks for partitioning the vertices of a graph into sets of size at most two so that the sum of the weights on all edges with end points in different sets is minimized. It is easy to see that this partitioning problem reduces to the maximum weighted matching problem.

Matching problems often occur on bipartite $G(V_1 \cup V_2, E)$ graphs. The maximum cardinality matching problem amounts to the *maximum assignment* of elements in V_1 ("workers") on to the elements of V_2 ("tasks") so that no worker in V_1 is assigned more than one task. This finds various applications in operations research.

The maximum cardinality matching problem on a bipartite graph $G(V_1 \cup V_2, E)$ can be solved by a maximum flow formulation. Simply, each vertex $v \in V_1$ is connected to a new vertex s by an edge (s, v) and each vertex $u \in V_2$ to a new vertex t by an edge (u, t). In the resulting graph, every edge is assigned unit capacity. The maximum flow value v corresponds to the cardinality of the maximum matching in the original bipartite graph G.

Although the matching problem variations on bipartite graphs are amenable to easily described polynomial-time algorithms, such as the one given above, the existing polynomial-time algorithms for matchings on general graphs are more complex [see (1)].

Approximation and Pseudopolynomial Algorithms

Approximation and pseudopolynomial-time algorithms concern mainly the solution of problems that are proved to be *NP*-hard (although they can sometimes be used on problems that are solvable in polynomial time) but for which the corresponding polynomial-time algorithm involves large constants.

An α-approximation algorithm A for an optimization problem R is a polynomial-time algorithm such that for any instance I of R, $|S_A(I) - S_{\text{OPT}}(I)|/S_{\text{OPT}}(I) \leq \alpha + c$, where $S_{\text{OPT}}(I)$ is the cost of the optimal solution for instance I, $S_A(I)$ is the cost of the solution found by algorithm A, and c is a constant.

Two examples of approximation algorithms are given below: For the optimization version of the Bin Packing problem, an approximation algorithm is the following: Sort the items into decreasing order of sizes. Assign each item in this order into the first bin that has room for it. If no such bin exists, introduce a new bin. This simple algorithm has been shown (6) to always give a solution that is no more than 11/9 times the optimal plus 4, namely $|S_A(I) - S_{\text{OPT}}(I)|/S_{\text{OPT}}(I) \leq 2/9 + c$.

The second example concerns a special but practical version of the Traveling Salesman problem that obeys the triangular inequality for all city distances. Given a weighted graph $G(V, E)$ of the cities, the algorithm first finds a minimum spanning tree T of G (i.e., a spanning tree that has minimum sum of edge weights). Then it finds a minimum weight matching M among all vertices that have odd degree in T. Lastly, it forms the subgraph $G'(V, E')$, where E' is the set of all edges in T and M and finds a path that starts from and terminates to the same vertex and passes through every edge exactly once (such a path is known as Eulerian tour). Every step in this algorithm takes polynomial time. It has been shown that $|S_A(I) - S_{\text{OPT}}(I)|/S_{\text{OPT}}(I) \leq 1/2$.

Unfortunately, obtaining a polynomial-time approximation algorithm for an *NP*-hard optimization problem can be very difficult. In fact it has been shown for some cases that this may be impossible. For example, it has been shown that un-

less $NP = P$ there is no α-approximation algorithm for the general Traveling Salesman problem for any $\alpha > 0$.

A pseudopolynomial-time algorithm for a problem R is an algorithm with time complexity $O(p(n, m))$, where $p(\ )$ is a polynomial of two variables, n is the size of the instance, and m is the magnitude of the largest number occuring in the instance. Only problems involving numbers that are not bounded by a polynomial on the size of the instance are applicable for solution by a pseudopolynomial-time algorithm. In principle, a pseudopolynomial-time algorithm is exponential given that the magnitude of a number is exponential to the size of its logarithmic encoding in the problem instance, but in practice, such an algorithm may be useful in cases where the numbers involved are not large.

NP-complete problems for which a pseudopolynomial-time algorithm exists are referred to as *weakly NP-complete,* whereas *NP*-complete problems for which no pseudopolynomial-time algorithm exists (unless $NP = P$) are referred to as *strongly NP-complete.* Examples of weakly *NP*-complete problems are the Knapsack and Subset Sum problems. A pseudopolynomial-time algorithm for the latter is the following: Given a set of positive integers $S = \{s_1, s_2, \ldots, s_n\}$ and an integer B, let $T[i, j]$, $1 \le i \le n$, $1 \le j \le B$, be 1 whenever it is true that there is a subset of the first i integers $s_1, s_2, \ldots, s_i$ with sum exactly B (otherwise, $T[i, j] = 0$). The entries of matrix T are systematically assigned as: $T[1, j] = 1$ if $j = 0$ or $j = s_1$; and, for $2 \le i \le n$, $1 \le j \le B$, $T[i, j] = 1$ if $T[i - 1, j] = 1$, or $s_i \le j$ and $T[i - 1, j - s_i] = 1$. This algorithm (a case of dynamic programming) takes time $O(nB)$, and the answer to the Subset Sum problem is given by the value of $T[n, B]$.

Examples of problems that are strongly *NP*-complete are the Traveling Salesman and the Bin Packing problems.

Probabilistic Algorithms

Probabilistic algorithms are a class of algorithms that do not depend exclusively on their input to carry out the computation. Instead, at one or more points in the course of the algorithm where a choice has to be made, they use a pseudorandom number generator to select "randomly" one out of a finite set of alternatives for arriving at a solution. Probabilistic algorithms are fully programmable (i.e., they are not like the nondeterministic algorithms), but in contrast with the deterministic algorithms, they may give different results each time for the same input instance (assuming that the initial state of the pseudorandom number generator is different each time).

Probabilistic algorithms trade off the certainty of a correct solution with a reduction in the computation time. As a simple example, in order to find a number in a list of n numbers that is greater than or equal to the median, $n/2$ elements have to be examined. However, assuming that the numbers are equally distributed, by examining only k numbers and keeping the maximum, the probability that the number is greater than or equal to the median is $1 - (\frac{1}{2})^k$, which is very practical if, for example, $k = 20$ while $n = 1{,}000{,}000$. Two major classes of probabilistic algorithms are the Monte Carlo type and the Las Vegas type. In the former, the algorithm guarantees that the solution it gives has a high probability of being correct for *every* instance. In the latter, the algorithm guarantees that any solution it reports is correct with certainty, but occasionally it may terminate by reporting failure. A representative case of probabilistic algorithms is testing whether a given very large integer is prime (this has applications to cryptography).

There is also a class of algorithms that resemble the probabilistic in that they use a pseudorandom number generator, but the solution they give is always correct. Such algorithms make certain choices that are not determined by the input instance in order to obtain the solution faster (the solution itself does not depend on these choices). These algorithms are known as *randomized.* A typical example is the Quicksort algorithm, which takes $O(n^2)$ time in the worst case for sorting a list of n elements, but $O(n \log n)$ time on average if the order of the elements in the list is randomized by the algorithm (the latter average time complexity is not conditioned on any distribution for the elements of the list).

BIBLIOGRAPHY

1. E. L. Lawler, *Combinatorial Optimization: Algorithms and Matroids,* New York: Holt, Rinehart and Winston, 1976.
2. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Englewood Cliffs, NJ: Prentice-Hall, 1982.
3. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows,* Englewood Cliffs, NJ: Prentice-Hall, 1993.
4. N. A. Sherwani, *Algorithms for VLSI Physical Design Automation,* Norwell, MA: Kluwer, 1993.
5. D. Bertsekas and R. Gallagher, *Data Networks,* Upper Saddle River, NJ: Prentice-Hall, 1992.
6. M. R. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness,* San Francisco, CA: W. H. Freeman, 1979.
7. M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design,* Rockville, MD: Computer Science Press, 1990.
8. D. Kagaris and S. Tragoudas, Avoiding linear dependencies in LFSR test pattern generators, *J. Electron. Test. Theor. Appl.,* **6**: 229–241, 1995.
9. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms,* Cambridge, MA: MIT Press, 1990.
10. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Rockville, MD: Computer Science Press, 1984.

DIMITRIOS KAGARIS
Southern Illinois University

SPYROS TRAGOUDAS
The University of Arizona

GRAPH THEORY. See GEOMETRY.