Chapter 18 Approximation Algorithms

Lily pads, 2006. Michael T. Goodrich. Used with permission.

Contents

Astronomers can determine the composition and distance of galaxies and stars by performing a spectrographic analysis of the light coming from these objects. Doing such an analysis involves collecting light from one of these objects over a relatively long period of time, and transmitting this light through a fiber-optic cable to a spectroscope. The spectrograph then splits this light into its various frequencies and measures the intensities of these light frequencies. By matching the patterns of high and low light frequencies coming from such an astronomical object to known patterns for the light emitted when various elements are burned, the astronomers can determine the elements that are present in the object. In addition, by observing the amount that these patterns are shifted to the red end of the spectrum, astronomers can also determine the distance of this object from earth. This distance can be determined using estimates for the speed at which the universe is expanding, because of the Doppler effect, where light wavelengths increase as an object is moving away from us.

As an optimization problem, one of the most time-consuming parts of this process is the first step—collecting the light from the galaxy or star over a given period of time. To do this with a telescope, a large aluminum disk the size of the diameter of the telescope is used. This disk is placed in the focal plane of the telescope, so that the light from each stellar objects in an observation falls in a specific spot on the disk. The astronomers know where these spots are located and, before the disk is placed in the focal plane, they use robotic drilling equipment to drill a hole in each spot of interest and they insert a fiber-optic cable into each such hole and connect it to a spectrograph. (See Figure 18.1.)

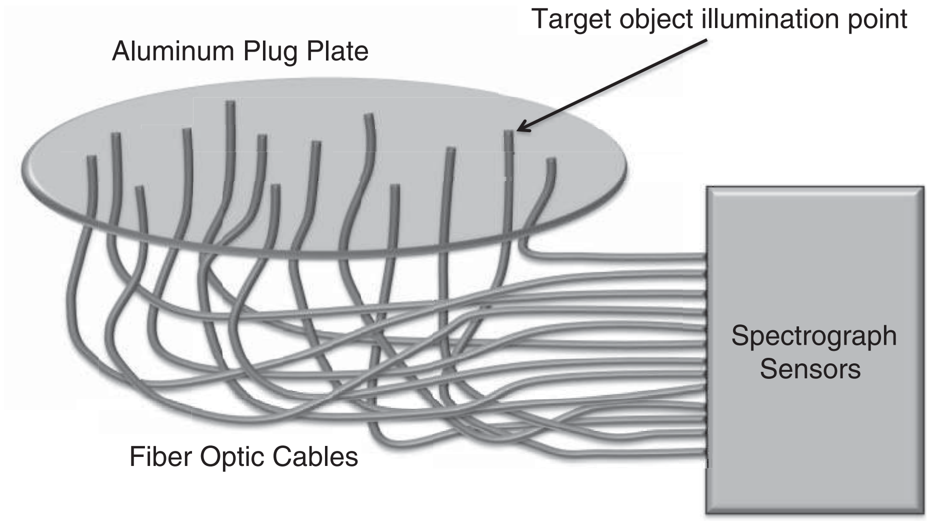

Figure 18.1: Aparatus for spectrographic analysis of stellar objects using an aluminum plug plate and fiber-optic cables. The aluminum plug plate is placed in the focal plane of a telescope.

As discussed at the beginning of Chapter 17, the problem of finding a fastest way to drill all these holes is an instance of the traveling salesperson problem (TSP). According to the abstract formulation of TSP, each of the hole locations is a "city" and the time it takes to move a robot drill from one hole to another corresponds to the distance between the cities corresponding to these two holes. Thus, a minimum-distance tour of the cities that starts and ends at the resting position for the robot drill is one that will drill the holes the fastest. Unfortunately, as we discuss in the previous chapter, the decision version of this optimization problem is NP-complete. Nevertheless, even though this is a difficult problem, it still needs to be solved in order to do the spectrographic analysis. The astronomers doing spectrographic analyses might not require the absolutely best solution, however. They might be happy with a solution that comes close to the optimum, especially if one can prove that it won't be too far from this optimum.

In addition to the problem of quickly drilling the holes in a plug plate for a spectrographic analysis, another optimization problem arises in this application as well. This problem is to minimize the number of observations needed in order to collect the spectra of all the stellar objects of interest. In this case, the astronomers have a map of all the stellar objects of interest and they want to cover it with the minimum number of disks having the same diameter as the telescope. (See Figure 18.2.) This optimization problem is an instance of the ***set cover*** problem. Each of the distinct sets of objects that can be included in a single observation is given as an input set and the optimization problem is to minimize the number of sets whose union includes all the objects of interest. As with TSP, the decision version of this problem is also NP-complete, but it is a problem for which an approximation to the optimum might be sufficient.

Approximation Ratios

In general, many optimization problems whose decision versions are NP-complete correspond to problems whose solution in the real world can oftentimes save money, time, or other resources. Thus, the main topic of this chapter is on approximate ways of dealing with ***NP***-completeness. One of the most effective methods is to construct polynomial-time algorithms that come close to solving difficult problems. Although such algorithms do not always produce optimal solutions, in some cases we can guarantee how close such an approximation algorithm will come to an optimal solution. Indeed, we explore several such approximations in this chapter, including algorithms for the knapsack, vertex cover, traveling salesperson, and set cover problems.

The general situation is that we have some problem instance x, which could be an encoding of a set of numbers, a graph, etc. In addition, for the problem we are interested in solving for x, there will often be a large set, $\mathcal{F}$, of ***feasible*** solutions for x. We also have a cost function, c, that determines a numeric cost $c(S)$ for any solution $S \in \mathcal{F}$. In the general optimization problem, we are interested in finding

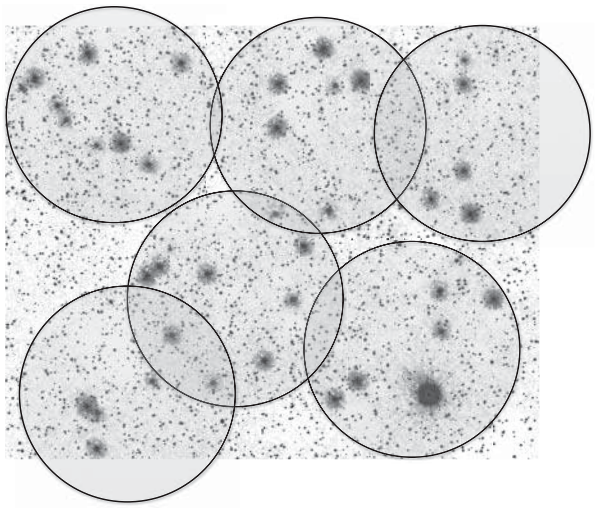

Figure 18.2: An example disk cover for a set of significant stellar objects (smaller objects are not included). Background image is from Omega Centauri, 2009. U.S. government image. Credit: NASA, ESA, and the Hubble SM4 ERO team.

a solution S in $\mathcal{F}$, such that

$$c(S) = \mathit{OPT} = \min\{c(T) \colon T \in \mathcal{F}\}.$$

That is, we want a solution with minimum cost. We could also formulate a maximization version of the optimization problem, as well, which would simply involve replacing the above "$\min$" with "$\max$." To keep the discussion in this section simple, however, we will typically take the view that, unless otherwise stated, our optimization goal is minimization.

The goal of an approximation algorithm is to come as close to the optimum value as possible in a reasonable amount of time. As we have been doing for this entire chapter, we take the view in this section that a reasonable amount of time is at most polynomial time.

Ideally, we would like to provide a guarantee of how close an approximation algorithm comes to the optimal value, OPT. We say that a ***δ-approximation*** algorithm for a particular optimization problem is an algorithm that returns a feasible solution S (that is, $S \in \mathcal{F}$), such that

$$c(S) \leq \delta\, \mathit{OPT},$$

for a minimization problem. For a maximization problem, a δ-approximation algorithm would guarantee $\mathit{OPT} \leq \delta\, c(S)$. Or, in general, we have

$$\delta \geq \max\{c(S)/\mathit{OPT},\ \mathit{OPT}/c(S)\}.$$

In this chapter, we study problems for which we can construct δ-approximation algorithms for various values of δ.

18.1 The Metric Traveling Salesperson Problem

In the optimization version of the traveling salesperson problem, or TSP, we are given a weighted graph, G, such that each edge e in G has an integer weight $c(e)$, and we are asked to find a minimum-weight cycle in G that visits all the vertices in G. In this section we study approximation algorithms for a special case of TSP.

Consider a ***metric*** version of TSP such that the edge weights satisfy the ***triangle inequality***. That is, for any three edges (u, v), (v, w), and (u, w) in G,

$$c((u, v)) + c((v, w)) \geq c((u, w)).$$

Also, suppose that every pair of vertices in G is connected by an edge, that is, G is a complete graph. This instance of TSP is called METRIC-TSP. The above properties, which hold for any distance metric, and which arise in lots of TSP applications, imply that the optimal tour of G will visit each vertex exactly once.

18.1.1 A 2-Approximation for METRIC-TSP

Our first approximation algorithm takes advantage of the above properties of G to design a very simple 2-approximation algorithm for METRIC-TSP. The algorithm has three steps. In the first step we construct a minimum spanning tree, M, of G (Section 15.1). In the second step we construct an Euler-tour traversal, E, of M, that is, a traversal of M that starts and ends at the same vertex and traverses each edge of M exactly once in each direction (Section 2.3.3). In the third step we construct a tour T from E by marching through the edges of E, but each time we have two edges (u, v) and (v, w) in E, such that v has already been visited, we replace these two edges by the edge (u, w) and continue. In essence, we are constructing T as a preorder traversal of M. This three-step algorithm clearly runs in polynomial time. (See Figure 18.3.)

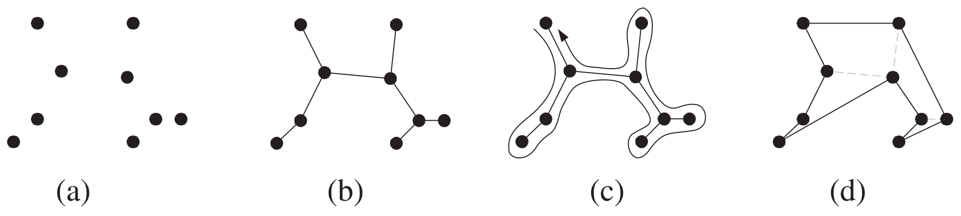

Figure 18.3: Example run of the 2-approximation algorithm for METRIC-TSP: (a) a set S of points in the plane, with Euclidean distance defining the costs of the edges (not shown); (b) the minimum spanning tree, M, for S; (c) an Euler tour, E, of M; (d) the approximate TSP tour, T.

Analysis of the 2-Approximation METRIC-TSP Algorithm

The analysis of why this algorithm achieves an approximation factor of 2 is also simple. Let us extend our notation so that $c(H)$ denotes the total weight of the edges in a subgraph H of G. Let S be a solution to METRIC-TSP, that is, an optimal TSP tour for the graph G. If we delete any edge from S, we get a path, which is, of course, also a spanning tree. Thus,

$$c(M) \leq c(S).$$

We can also easily relate the cost of E to that of M, as

$$c(E) = 2c(M),$$

since the Euler tour E visits each edge of M exactly once in each direction. Finally, note that, by the triangle inequality, when we construct our tour T, each time we replace two edges (u, v) and (v, w) with the edge (u, w), we do not increase the cost of the tour. That is,

$$c(T) \leq c(E).$$

Therefore, we have

$$c(T) \leq 2c(S).$$

(See Figure 18.4.)

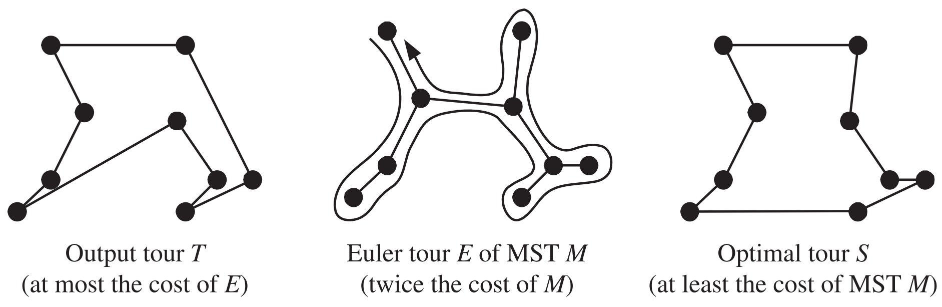
Output tour *T*
(at most the cost of *E*)
Euler tour *E* of MST *M*
(twice the cost of *M*)
Optimal tour *S*
(at least the cost of MST *M*)

Figure 18.4: Illustrating the proof that MST-based algorithm is a 2-approximation for the TSP optimization problem.

We may summarize this discussion as follows.

Theorem 18.1: *There is a 2-approximation algorithm for the* METRIC-TSP *optimization problem that runs in polynomial time.*

This theorem depends heavily on the fact that the cost function on the graph G satisfies the triangle inequality. In fact, without this assumption, no constant-factor approximation algorithm for the optimization version of TSP exists that runs in polynomial time, unless $\boldsymbol{P} = \boldsymbol{NP}$. (See Exercise C-18.1.)

18.1.2 The Christofides Approximation Algorithm

There is a somewhat more complex algorithm, which is known as the ***Christofides approximation algorithm***, that can achieve an even better approximation ratio than the above method. Like the above 2-approximation algorithm, the Christofides approximation algorithm has just a few steps that are applications of other algorithms. The most difficult step involves computing a minimum-cost ***perfect matching*** in an undirected weighted graph, H, having $2N$ vertices, that is, a set of N edges in H that has minimum total weight and such that no two edges are incident on the same vertex. This is a problem that can be solved in polynomial time, albeit using an algorithm that is somewhat complicated and thus not included in this book.

Suppose we are given an instance of METRIC-TSP specified as a complete graph G with weights on its edges satisfying the triangle inequality. The Christofides approximation algorithm is as follows (See Figure 18.5):

1. Construct a minimum spanning tree, M, for G.
2. Let W be the set of vertices of G that have odd degree in M and let H be the subgraph of G induced by the vertices in W. That is, H is the graph that has W as its vertices and all the edges from G that join such vertices. By a simple argument, we can show that the number of vertices in W is even (see Exercise R-18.12). Compute a minimum-cost perfect matching, P, in H.
3. Combine the graphs M and P to create a graph, G', but don't combine parallel edges into single edges. That is, if an edge e is in both M and P, then we create two copies of e in the combined graph, G'.
4. Create an Eulerian circuit, C, in G', which visits each edge exactly once (unlike in the 2-approximation algorithm, here the edges of G' are undirected).
5. Convert C into a tour, T, by skipping over previously visited vertices.

The running time of this algorithm is dominated by Step 2, which takes $O(n^3)$ time. Thus, although it is not as fast as the above 2-approximation algorithm, the Christofides approximation algorithm achieves a better approximation ratio.

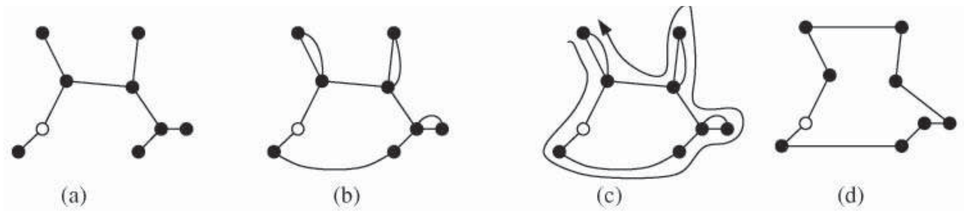

Figure 18.5: Illustrating the Christofides approximation algorithm: (a) a minimum spanning tree, M, for G; (b) a minimum-cost perfect matching P on the vertices in W (the vertices in W are shown solid and the edges in P are shown as curved arcs); (c) an Eulerian circuit, C, of G'; (d) the approximate TSP tour, T.

Analyzing the Christofides Approximation Algorithm

To begin our analysis of the Christofides approximation algorithm, let S be an optimal solution to this instance of METRIC-TSP and let T be the tour that is produced by the Christofides approximation algorithm. Because S includes a spanning tree and M is a minimum spanning tree in G,

$$c(M) \leq c(S).$$

In addition, let R denote a solution to the traveling salesperson problem on H. Since the edges in G (and, hence, H) satisfy the triangle inequality, and all the edges of H are also in G,

$$c(R) \leq c(S).$$

That is, visiting more vertices than in the tour R cannot reduce its total cost. Consider now the cost of a perfect matching, P, of H, and how it relates to R, an optimal traveling salesperson tour of H. Number the edges of R, and ignore the last edge (which returns to the start vertex). Note that the costs of the set of odd-numbered edges and the set of even-numbered edges in R sum to $c(R)$; hence, one of these two sets has total cost at most half of that of R, that is, cost at most $c(R)/2$. In addition, the set of odd-numbered edges and the set of even-numbered edges in R are both perfect matchings; hence, the cost of P, a minimum-weight perfect matching on the edges of H, will be at most the smaller of these two. That is,

$$c(P) \leq c(R)/2.$$

Therefore,

$$c(M) + c(P) \leq c(S) + c(R)/2 \leq 3c(S)/2.$$

Since the edges in G satisfy the triangle inequality, we can only improve the cost of a tour by making shortcuts that avoid previously visited vertices. Thus,

$$c(T) \leq c(M) + c(P),$$

which implies that

$$c(T) \leq 3c(S)/2.$$

In other words, the Christofides approximation algorithm gives us the following.

Theorem 18.2: *There is a $(3/2)$-approximation algorithm for the* METRIC-TSP *optimization problem that runs in polynomial time.*

18.2 Approximations for Covering Problems

In this section, we describe greedy approximation algorithms for the VERTEX-COVER and SET-COVER problems (Section 17.4).

18.2.1 A 2-Approximation for VERTEX-COVER

We begin with a 2-approximation for the VERTEX-COVER problem. In the optimization version of this problem, we are given a graph G and we are asked to produce the smallest set C that is a vertex cover for G, that is, every edge in G is incident on some vertex in C.

This approximation algorithm is based on the greedy method, as mentioned above, and is rather simple. It involves picking an edge in the graph, adding both its endpoints to the cover, and then deleting this edge and its incident edges from the graph. The algorithm repeats this process until no edges are left. We give the details for this approach in Algorithm 18.6.

```
Algorithm VertexCoverApprox($G$):
    Input: A graph $G$
    Output: A small vertex cover $C$ for $G$
    $C \leftarrow \emptyset$
    while $G$ still has edges do
        select an edge $e = (v, w)$ of $G$
        add vertices $v$ and $w$ to $C$
        for each edge $f$ incident to $v$ or $w$ do
            remove $f$ from $G$
    return $C$
```

Algorithm 18.6: A 2-approximation algorithm for VERTEX-COVER.

We leave the details of how to perform this algorithm in $O(n + m)$ time as a simple exercise (R-18.1). For the analysis, first observe that each edge $e = (v, w)$ selected by the algorithm, and used to add v and w to C, must be covered in any vertex cover. That is, any vertex cover for G must contain v or w (possibly both). The approximation algorithm adds both v and w to C in such a case. When the approximation algorithm completes, there are no uncovered edges left in G, for we remove all the edges covered by the vertices v and w when we add them to C. Thus, C forms a vertex cover of G. Moreover, the size of C is at most twice that of an optimal vertex cover for G, since, for every two vertices we add to C, one of these vertices must belong to the optimal cover. Therefore, we have the following.

Theorem 18.3: *There is a 2-approximation algorithm for the* VERTEX-COVER *problem that runs in $O(n + m)$ time on a graph with n vertices and m edges.*

18.2.2 A Logarithmic Approximation for SET-COVER

There are some cases when achieving even a constant-factor approximation in polynomial time is difficult. In this section, we study one of the best known of such problems, the SET-COVER problem (Section 17.4). In the optimization version of this problem, we are given a collection of sets $S_1, S_2, \ldots, S_m$, whose union is a universe U of size n, and we are asked to find the smallest integer k, such that there is a subcollection of k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ with

$$U = \bigcup_{i=1}^{m} S_i = \bigcup_{j=1}^{k} S_{i_j}.$$

Although it is difficult to find a constant-factor approximation algorithm that runs in polynomial time for this problem, we can design an efficient algorithm that has an approximation factor of $O(\log n)$. As with several other approximation algorithms for hard problems, this algorithm is based on the greedy method (Section 10).

A Greedy Approach

Our algorithm selects sets one at a time, each time selecting the set that has the most uncovered elements. When every element in U is covered, we are done. We give a simple pseudocode description in Algorithm 18.7.

```
Algorithm SetCoverApprox($S$):
    Input: A collection $S$ of sets $S_1, S_2, \ldots, S_m$ whose union is $U$
    Output: A small set cover $C$ for $S$
    $C \leftarrow \emptyset$     // The set cover built so far
    $E \leftarrow \emptyset$     // The elements from $U$ currently covered by $C$
    while $E \neq U$ do
        select a set $S_i$ that has the maximum number of uncovered elements
        add $S_i$ to $C$
        $E \leftarrow E \cup S_i$
    Return $C$.
```

Algorithm 18.7: An approximation algorithm for SET-COVER.

This algorithm runs in polynomial time. (See Exercise R-18.2.)

To analyze the approximation factor of the above greedy SET-COVER algorithm, we will use an amortization argument based on a charging scheme (Section 1.4). Namely, each time our approximation algorithm selects a set S_j, we will charge the elements of S_j for its selection.

Specifically, consider the moment in our algorithm when a set S_j is added to C, and let k be the number of previously uncovered elements in S_j. We must pay a total charge of 1 to add this set to C, so we charge each previously uncovered element i of S_j a charge of

$$c(i) = 1/k.$$

Thus, the total size of our cover is equal to the total charges made. That is,

$$|C| = \sum_{i \in U} c(i).$$

To prove an approximation bound, we will consider the charges made to the elements in each subset S_j that belongs to an optimal cover, C'. So, suppose that S_j belongs to C'. Let us write $S_j = \{x_1, x_2, \ldots, x_{n_j}\}$ so that S_j's elements are listed in the order in which they are covered by our algorithm (we break ties arbitrarily). Now, consider the iteration in which x_1 is first covered. At that moment, S_j has not yet been selected; hence, whichever set is selected must have at least n_j uncovered elements. Thus, x_1 is charged at most $1/n_j$. So let us consider, then, the moment our algorithm charges an element x_l of S_j. In the worst case, we will have not yet chosen S_j (indeed, our algorithm may never choose this S_j). Whichever set is chosen in this iteration has, in the worst case, at least $n_j - l + 1$ uncovered elements; hence, x_l is charged at most $1/(n_j - l + 1)$. Therefore, the total amount charged to all the elements of S_j is at most

$$\sum_{l=1}^{n_j} \frac{1}{n_l - l + 1} = \sum_{l=1}^{n_j} \frac{1}{l},$$

which is the familiar ***harmonic number***, H_{n_i}. It is well known (for example, see the Appendix) that H_{n_j} is $O(\log n_j)$. Let $c(S_j)$ denote the total charges given to all the elements of a set S_j that belongs to the optimal cover C'. Our charging scheme implies that $c(S_j)$ is $O(\log n_j)$. Thus, summing over the sets of C', we obtain

$$\begin{aligned} \sum_{S_j \in C'} c(S_j) &\le \sum_{S_j \in C'} b \log n_j \\ &\le b|C'| \log n, \end{aligned}$$

for some constant $b \ge 1$. But, since C' is a set cover,

$$\sum_{i \in U} c(i) \le \sum_{S_j \in C'} c(S_j).$$

Therefore,

$$|C| \le b|C'| \log n.$$

This fact gives us the following result.

Theorem 18.4: *The optimization version of the* SET-COVER *problem has an $O(\log n)$-approximation polynomial-time algorithm for finding a cover of a collection of sets whose union is a universe of size n.*

18.3 Polynomial-Time Approximation Schemes

There are some problems for which we can construct δ-approximation algorithms that run in polynomial time with $\delta = 1 + \epsilon$, for any fixed value $\epsilon > 0$. The running time of such a collection of algorithms depends both on n, the size of the input, and also on the fixed value ϵ. We refer to such a collection of algorithms as a ***polynomial-time approximation scheme***, or ***PTAS***. When we have a polynomial-time approximation scheme for a given optimization problem, we can tune our performance guarantee based on how much time we can afford to spend. Ideally, the running time is polynomial in both n and $1/\epsilon$, in which case we have a ***fully polynomial-time approximation scheme***.

Polynomial-time approximation schemes take advantage of a property that some hard problems possess, namely, that they are rescalable. A problem is said to be ***rescalable*** if an instance x of the problem can be transformed into an equivalent instance x' (that is, one with the same optimal solution) by scaling the cost function, c. For example, TSP is rescalable. Given an instance G of TSP, we can construct an equivalent instance G' by multiplying the distance between every pair of vertices by a scaling factor s. The traveling salesperson tour in G' will be the same as in G, although its cost will now be multiplied by s.

A Fully Polynomial-Time Approximation Scheme for Knapsack

To be more concrete, let us give a fully polynomial approximation scheme for the optimization version of a well-known problem, Knapsack (Sections 10.1 and 17.5). In the optimization version of this problem, we are given a set S of items, numbered 1 to n, together with a size constraint, s. Each item i in S is given an integer size, s_i, and worth, w_i, and we are asked to find a subset, T, of S, such that T maximizes the worth

$$w = \sum_{i \in T} w_i \quad \text{while satisfying} \quad \sum_{i \in T} s_i \leq s.$$

We desire a PTAS that produces a $(1 + \epsilon)$-approximation, for any given fixed constant ϵ. That is, such an algorithm should find a subset T' satisfying the size constraint such that if we define $w' = \sum_{i \in T'} w_i$, then

$$OPT \leq (1 + \epsilon)w',$$

where *OPT* is the optimal worth summation, taken over all possible subsets satisfying the total size constraint. To prove that this inequality holds, we will actually prove that

$$w' \geq (1 - \epsilon/2)OPT,$$

for $0 < \epsilon < 1$. This will be sufficient, however, since, for any fixed $0 < \epsilon < 1$,

$$\frac{1}{1 - \epsilon/2} < 1 + \epsilon.$$

To derive a PTAS for KNAPSACK, we take advantage of the fact that this problem is rescalable. Suppose we are given a value of ϵ with $0 < \epsilon < 1$. Let $w_{\max}$ denote the maximum worth of any item in S. Without loss of generality, we assume that the size of each item is at most s (for an item larger than this could not fit in the knapsack). Thus, the item with worth $w_{\max}$ defines a lower bound for the optimal value. That is, $w_{\max} \leq \mathit{OPT}$. Likewise, we can define an upper bound on the optimal solution by noting that the knapsack can at most contain all n items in S, each of which has worth at most $w_{\max}$. Thus, $\mathit{OPT} \leq n\, w_{\max}$. To take advantage of the rescalability of KNAPSACK, we round each worth value w_i to $w_i{}'$, the nearest smaller multiple of $M = \epsilon w_{\max}/2n$. Let us denote the rounded version of S as S', and let us also use OPT' to denote the solution for this rounded version S'. Note that, by simple substitution, $\mathit{OPT} \leq 2n^2 M/\epsilon$. Moreover, $\mathit{OPT}' \leq \mathit{OPT}$, since we rounded every worth value in S down to form S'. Thus, $\mathit{OPT}' \leq 2n^2 M/\epsilon$.

Therefore, let us turn to our solution for the rounded version S' of the KNAPSACK problem for S. Since every worth value in S' is a multiple of M, any achievable worth of a collection of items taken from S' is also a multiple of M. Moreover, there are just $N = \lceil 2n^2/\epsilon \rceil$ such multiples that need to be considered, because of the upper bound on OPT'. We can use dynamic programming to construct an efficient algorithm for finding the optimal worth for S'. In particular, let us define the parameter

$s[i,j] =$ the size of the smallest set of items in $\{1, 2, \ldots, j\}$ with worth iM.

The key insight to the design of a dynamic programming algorithm for solving the rounded KNAPSACK problem is the observation that we can write

$$s[i,j] = \min\{s[i,j-1],\ s_j + s[i - (w_j{}'/M), j-1]\},$$

for $i = 1, 2, \ldots, N$, and $j = 1, 2, \ldots, n$. (See Figure 18.8.)

The above equation for $s[i,j]$ follows from the fact that item j will either contribute or not contribute to the smallest way of achieving worth iM from the items in $\{1, 2, \ldots, j\}$. In addition, note that for the base case, $j = 0$, when no items at all are included in the knapsack, then

$$s[i,0] = +\infty,$$

for $i = 1, 2, \ldots, N$. That is, such a size is undefined. In addition,

$$s[0,j] = 0,$$

for $j = 1, 2, \ldots, n$, since we can always achieve worth 0 by including no items in the knapsack. The optimal value is defined by

$$\mathit{OPT}' = \max\{iM \colon s[i,n] \leq s\}.$$

This is the value that is output by our PTAS algorithm.

Analysis of the PTAS for KNAPSACK

We can easily convert the above description into a dynamic programming algorithm that computes OPT' in $O(n^3/\epsilon)$ time. Such an algorithm gives us the value of an optimal solution, but we can easily translate the dynamic programming algorithm for computing the size into one for the actual set of items.

Let us consider, then, how good an approximation OPT' is for OPT. Recall that we reduced the worth w_i of each item i by at most $M = \epsilon w_{\max}/2n$. Thus,

$$OPT' \geq OPT - \epsilon w_{\max}/2,$$

since the optimal solution can contain at most n items. Since $OPT \geq w_{\max}$, this in turn implies that

$$OPT' \geq OPT - \epsilon OPT/2 = (1 - \epsilon/2)OPT.$$

Thus, $OPT \leq (1+\epsilon)OPT'$, which was what we wished to prove. The running time of our approximation algorithm is $O(n^3/\epsilon)$. Our scheme of designing an efficient algorithm for any given $\epsilon > 0$ gives rise to a fully polynomial approximation scheme, since the running time is polynomial in both n and $1/\epsilon$. This fact gives us the following.

Theorem 18.5: *The* KNAPSACK *optimization problem has a fully polynomial approximation scheme that achieves a $(1+\epsilon)$-approximation factor in $O(n^3/\epsilon)$ time, where n is the number of items in the* KNAPSACK *instance and $0 < \epsilon < 1$ is a given fixed constant.*

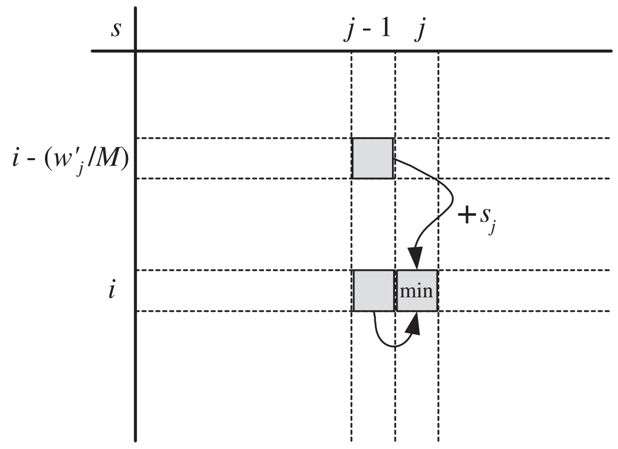

Figure 18.8: Illustration of the equation for $s[i,j]$ used in the dynamic program for the scaled version of KNAPSACK.

18.4 Backtracking and Branch-and-Bound

At this point, we know that there are many problems to be ***NP***-complete. Thus, unless ***P*** $=$ ***NP***, which the majority of computer scientists believes is not true, it is impossible to solve any of these problems in polynomial time. Nevertheless, many of these problems arise in real-life applications where solutions to them need to be found, even if finding these solutions may take a long time. Thus, in this section, we address techniques for dealing with ***NP***-completeness that have shown much promise in practice. These techniques allow us to design algorithms that can find solutions to hard problems, often in a reasonable amount of time. In this section, we study the methods of ***backtracking*** and ***branch-and-bound***.

18.4.1 Backtracking

The backtracking technique is a way to build an algorithm for some hard problem L. Such an algorithm searches through a large, possibly even exponential-size, set of possibilities in a systematic way. The search strategy is typically optimized to avoid symmetries in problem instances for L and to traverse the search space so as to find an "easy" solution for L if such a solution exists.

The ***backtracking*** technique takes advantage of the inherent structure that many ***NP***-complete problems possess. Recall that acceptance for an instance x in a problem in ***NP*** can be verified in polynomial time given a polynomial-sized certificate. Oftentimes, this certificate consists of a set of "choices," such as the values assigned to a collection of Boolean variables, a subset of vertices in a graph to include in a special set, or a set of objects to include in a knapsack. Likewise, the verification for a certificate often involves a simple test of whether the certificate demonstrates a successful configuration for x, such as satisfying a formula, covering all the edges in a graph, or conforming to certain performance criteria. In such cases, we can use the ***backtracking*** algorithm, given in Algorithm 18.9, to systematically search for a solution to our problem, if such a problem exists.

The backtracking algorithm traverses through possible "search paths" to locate solutions or "dead ends." The configuration at the end of such a path consists of a pair (x, y), where x is the remaining subproblem to be solved and y is the set of choices that have been made to get to this subproblem from the original problem instance. Initially, we give the backtracking algorithm the pair $(x, \emptyset)$, where x is our original problem instance. Anytime the backtracking algorithm discovers that a configuration (x, y) cannot lead to a valid solution no matter how additional choices are made, then it cuts off all future searches from this configuration and "backtracks" to another configuration. In fact, this approach gives the backtracking algorithm its name.

```
Algorithm Backtrack($x$):
  Input: A problem instance $x$ for a hard problem
  Output: A solution for $x$ or “no solution” if none exists
  $F \leftarrow \{(x, \emptyset)\}$.    // $F$ is the “frontier” set of subproblem configurations
  while $F \neq \emptyset$ do
      Select from $F$ the most “promising” configuration $(x, y)$.
      Expand $(x, y)$ by making a small set of additional choices.
      Let $(x_1, y_1)$, $(x_2, y_2)$, ..., $(x_k, y_k)$ be the set of new configurations.
      for each new configuration $(x_i, y_i)$ do
          Perform a simple consistency check on $(x_i, y_i)$.
          if the check returns “solution found” then
              return the solution derived from $(x_i, y_i)$
          if the check returns “dead end” then
              Discard the configuration $(x_i, y_i)$.    // Backtrack
          else
              $F \leftarrow F \cup \{(x_i, y_i)\}$    // $(x_i, y_i)$ starts a promising search path
  return “no solution”
```

Algorithm 18.9: The template for a backtracking algorithm.

Filling in the Details

In order to turn the backtracking strategy into an actual algorithm, we need only fill in the following details:

1. Define a way of selecting the most “promising” candidate configuration from the frontier set F.
2. Specify the way of expanding a configuration (x, y) into subproblem configurations. This expansion process should, in principle, be able to generate all feasible configurations, starting from the initial configuration, $(x, \emptyset)$.
3. Describe how to perform a simple consistency check for a configuration (x, y) that returns “solution found,” “dead end,” or “continue.”

If F is a stack, then we get a depth-first search of the configuration space. In fact, in this case we could even use recursion to implement F automatically as a stack. Alternatively, if F is a queue, then we get a breadth-first search of the configuration space. We can also imagine other data structures to implement F, but as long as we have an intuitive notion of how to select the most “promising” configuration from F with each iteration, then we have a backtracking algorithm.

So as to make this approach more concrete, let us work through an application of the backtracking technique to the CNF-SAT problem.

A Backtracking Algorithm for CNF-SAT

Recall that in the CNF-SAT problem we are given a Boolean formula S in conjunctive normal form (CNF) and are asked whether S is satisfiable. To design a backtracking algorithm for CNF-SAT, we will systematically make tentative assignments to the variables in S and see if such assignments make S evaluate immediately to 1 or 0, or yield a new formula S' for which we could continue making tentative value assignments. Thus, a configuration in our algorithm will consist of a pair (S', y), where S' is a Boolean formula in CNF, and y is an assignment of values to Boolean variables not in S' such that making these assignments in S results in the formula S'.

To formulate our backtracking algorithm, then, we need to give the details of each of the three components to the backtracking algorithm. Given a frontier F of configurations, we make our most "promising" choice, which is the subformula S' with the smallest clause. Such a formula is the most constrained of all the formulas in F; hence, we would expect it to hit a dead end most quickly if that is indeed its destiny.

Let us consider, then, how to generate subproblems from a subformula S'. We do this by locating a smallest clause C in S', and picking a variable x_i that appears in C. We then create two new subproblems that are associated with our assigning $x_i = 1$ and $x_i = 0$, respectively.

Finally, we must say how to process S' to perform a consistency check for an assignment of a variable x_i in S'. We begin this processing by reducing any clauses containing x_i based on the assigned 0 or 1 value of x_i (depending on the choice we made). If this reduction results in a new clause with a single literal, x_j or $\overline{x_j}$, we also perform the appropriate value assignment to x_j to make this single-literal clause satisfied. We then process the resulting formula to propagate the assigned value of x_j. If this new assignment in turn results in a new single-literal clause, we repeat this process until we have no more single-literal clauses. If at any point we discover a contradiction (that is, clauses x_i and $\overline{x_i}$, or an empty clause), then we return "dead end." If we reduce the subformula S' all the way to the constant 1, then we return "solution found," along with all the variable assignments we made to reach this point. Otherwise, we derive a new subformula, S'', such that each clause has at least two literals, along with the value assignments that lead from the original formula S to S''. We call this operation the ***reduce*** operation for propagating an assignment of value to x_i in S'.

Fitting all of these pieces into the template for the backtracking algorithm results in an algorithm that solves the CNF-SAT problem in about as fast a time as we can expect. In general, the worst-case running time for this algorithm is still exponential, but the backtracking can often speed things up. Indeed, if every clause in the given formula has at most two literals, then this algorithm runs in polynomial time. (See Exercise C-17.6.)

18.4.2 Branch-and-Bound

The backtracking algorithm works for decision problems, but it is not designed for optimization problems, where, in addition to having some feasibility condition be satisfied for a certificate y associated with an instance x, we also have a cost function $f(x)$ that we wish to minimize or maximize (without loss of generality, let us assume the cost function should be minimized). Nevertheless, we can extend the backtracking algorithm to work for such optimization problems, and in so doing derive the algorithmic technique known as ***branch-and-bound***.

The branch-and-bound technique has all the elements of backtracking, except that rather than simply stopping the entire search process any time a solution is found, we continue processing until the best solution is found. In addition, the algorithm has a scoring mechanism to always choose the most promising configuration to explore in each iteration.

To provide for the optimization criterion of always selecting the "most promising" configuration, we extend the three assumptions for a backtracking algorithm to add one more condition:

- For any configuration, (x, y), we assume we have a function, $lb(x, y)$, which is a lower bound on the cost of any solution derived from (x, y).

To make the branch-and-bound approach more concrete, let us consider how it can be applied to solve the optimization version of the traveling salesperson (TSP) problem. In the optimization version of this problem, we are given a graph G with a cost function $c(e)$ defined for each edge e in G, and we wish to find the smallest total-cost tour that visits every vertex in G, returning back to its starting vertex.

We can design an algorithm for TSP by computing for each edge $e = (v, w)$, the minimum-cost path that begins at v and ends at w while visiting all other vertices in G along the way. To find such a path, we apply the branch-and-bound technique. We generate the path from v to w in $G - \{e\}$ by augmenting a current path by one vertex in each loop of the branch-and-bound algorithm.

- After we have built a partial path P, starting, say, at v, we only consider augmenting P with vertices in not in P.
- We can classify a partial path P as a "dead end" if the vertices not in P are disconnected in $G - \{e\}$.
- To define the lower-bound function, lb, we can use the total cost of the edges in P plus $c(e)$. This will certainly be a lower bound for any tour that will be built from e and P.

In addition, after we have run the algorithm to completion for one edge e in G, we can use the best path found so far over all tested edges, rather than restarting the current best solution b at $+\infty$. The running time of the resulting algorithm will still be exponential in the worst case, but it will avoid a considerable amount of unnecessary computation in practice.

18.5 Exercises

Reinforcement

R-18.1 Describe in detail how to implement Algorithm 18.6 in $O(n+m)$ time on an n-vertex graph with m edges. You may use the traditional operation-count measure of running time in this case.

R-18.2 Describe the details of an efficient implementation of Algorithm 18.7 and analyze its running time.

R-18.3 Give an example of a graph G with at least 10 vertices such that the greedy 2-approximation algorithm for VERTEX-COVER given above is guaranteed to produce a suboptimal vertex cover.

R-18.4 Give a complete, weighted graph G, such that its edge weights satisfy the triangle inequality but the MST-based approximation algorithm for TSP does not find an optimal solution.

R-18.5 Give a pseudocode description of the backtracking algorithm for CNF-SAT.

R-18.6 Give a recursive pseudocode description of the backtracking algorithm, assuming the search strategy should visit configurations in a depth-first fashion.

R-18.7 Give a pseudocode description of the branch-and-bound algorithm for TSP.

R-18.8 The branch-and-bound program in Section 18.4.2, for solving the KNAPSACK problem, uses a Boolean flag to determine when an item is included in a solution or not. Show that this flag is redundant. That is, even if we remove this field, there is a way (using no additional fields) to tell if an item is included in a solution or not.

R-18.9 Suppose G is a complete undirected graph such that every edge has weight 1 or 2. Show that the weights in G satisfy the triangle inequality.

R-18.10 Suppose G is an undirected weighted graph such that G is not the complete graph but every edge in G has positive weight. Create a complete graph, H, having the same vertex set as G, such that if (v, u) is an edge in G, then (v, u) has the same weight in H as in G, and if (v, u) is not an edge in G, then (v, u) has weight in H equal to the length of a shortest path from v to u in G. Show that the edge weights in H satisfy the triangle inequality.

R-18.11 Suppose we are given the following collection of sets:

$$S_1 = \{1,2,3,4,5,6\},\ S_2 = \{5,6,8,9\},\ S_3 = \{1,4,7,10\},$$

$$S_4 = \{2,5,7,8,11\},\ S_5 = \{3,6,9,12\},\ S_6 = \{10,11\}.$$

What is the optimal solution to this instance of the SET-COVER problem and what is the solution produced by the greedy algorithm?

R-18.12 Show that the number of vertices of odd degree in a tree is even.

Creativity

C-18.1 Consider the general optimization version of the TSP problem, where the underlying graph need not satisfy the triangle inequality. Show that, for any fixed value $\delta \geq 1$, there is no polynomial-time δ-approximation algorithm for the general TSP problem unless $\boldsymbol{P} = \boldsymbol{NP}$.

Hint: Reduce HAMILTONIAN-CYCLE to this problem by defining a cost function for a complete graph H for the n-vertex input graph G so that edges of H also in G have cost 1 but edges of H not in G have cost δn more than 1.

C-18.2 Derive an efficient backtracking algorithm for the HAMILTONIAN-CYCLE problem.

C-18.3 Derive an efficient backtracking algorithm for the KNAPSACK decision problem.

C-18.4 Derive an efficient branch-and-bound algorithm for the KNAPSACK optimization problem.

C-18.5 Derive a new lower-bound function, lb, for a branch-and-bound algorithm for solving the TSP optimization problem. Your function should always be greater than or equal to the lb function used in Section 18.4.2, but still be a valid lower-bound function. Describe an example where your lb is strictly greater than the lb function used in Section 18.4.2.

C-18.6 In the ***bottleneck traveling salesperson problem*** (***TSP***), we are given an undirected graph G with weights on its edges and asked to find a tour that visits the vertices of G exactly once and returns to the start so as to minimize the cost of the maximum-weight edge in the tour. Assuming that the weights in G satisfy the triangle inequality, design a polynomial-time 3-approximation algorithm for bottleneck TSP.

Hint: Show that it is possible to turn an Euler-tour traversal, E, of an MST for G into a tour visiting each vertex exactly once such that each edge of the tour skips at most two vertices of E.

C-18.7 In the ***Euclidean*** traveling salesperson problem, cities are points in the plane and the distance between two cities is the Euclidean distance between the points for these cities, that is, the length of the straight line joining these points. Show that an optimal solution to the Euclidean TSP is a simple polygon, that is, a connected sequence of line segments such that no two ever cross.

C-18.8 Consider the KNAPSACK problem, but now instead of implementing the PTAS algorithm given in the book, we use a greedy approach of always picking the next item that maximizes the ratio of value over weight (as in the optimal way to solve the fractional version of the KNAPSACK problem). Show that this approach does not produce a c-approximation algorithm, for any fixed value of c.

C-18.9 Consider a greedy algorithm for the VERTEX-COVER problem, where we repeatedly choose a vertex with maximum degree, add it to our cover, and then remove it and all its incident edges. Show that this algorithm does not, in general, produce a 2-approximation.

Hint: Use a bipartite graph where all the vertices on one side have the same degree.

C-18.10 In the HITTING-SET problem, we are given a set U of items, and a collection of subsets of U, $S_1, S_2, \ldots, S_m$. The problem is to find a smallest subset T of U such that T "hits" every subset S_i, that is, $T \cap S_i \neq \emptyset$, for $i = 1, \ldots, m$. Design a polynomial-time $O(\log n)$-approximation algorithm for HITTING-SET.

Applications

A-18.1 Suppose you are working for a cartography company, that is, a company that makes maps. Your job is to design a software package that can take as input the map of some region, R, and label as many of the cities of R as possible. Each of the n cities in such a region, R, is given by an (x, y) coordinate for the center of that city. Assume, for the sake of simplifying the problem, that the label, L_c, for each city, c, is a rectangle (which will contain the name of the city, c) whose lower-right corner is the (x, y)-location for c. The labels for two cities, c and d, ***conflict*** if L_c intersects L_d. Given your extensive algorithmic background, you realize that you can model this problem with a graph, G, where you create a vertex in G for each city and connect cities c and d with an edge if their labels conflict. Let $d = 2m/n$ be the average degree of the vertices in G, where m is the number of edges in G. Describe an $O(d)$-approximation algorithm for finding the largest number of mutually nonconflicting labels for the cities in a given region R.

A-18.2 In a synchronous optical network (SONET) ring, a collection of routers are connected with fiber-optic cables to form a single, simple cycle. A message between two routers, x and y, can then be transmitted by routing it clockwise or counter-clockwise around the ring. Given a set, M, of messages to transmit, each specified by a pair of routers (x, y), the ***ring-loading problem*** is to route all the messages in M so as to minimize the maximum load on any link in the ring (that joins two adjacent routers). Solving such an optimization problem is useful, since such a solution minimizes the bandwidth needed to transmit all the messages in M. Describe a 2-approximation for solving the ring-loading problem.

A-18.3 Suppose you work for a major package shipping company, FedUP, and it is your job to ship a set of n boxes from Rhode Island to California using a given collection of trucks. You know that these trucks will be weighed at various points along this route and FedUP will have to pay a penalty if any of these trucks are overweight. Thus, you would like to minimize the weight of the most heavily loaded truck. Assuming you know the integer weight of each of the n boxes, describe a simple greedy algorithm for assigning boxes to trucks and show that this algorithm has an approximation ratio of at most 2 for the problem of minimizing the weight of the most heavily loaded truck.

A-18.4 Suppose you work for a major package shipping company, FedUP, as in the previous exercise, but suppose there is a new law that requires every truck to carry no more than M pounds, even if it has room for more boxes. Now the optimization problem is to use the fewest number of trucks possible to carry the n boxes across the country such that each truck is carrying at most M pounds. Describe a simple greedy algorithm for assigning boxes to trucks and show that your algorithm uses a number of trucks that is within a factor of 2 of the optimal number of trucks. You may assume that no box weighs more than M pounds.

A-18.5 Suppose you are preparing an algorithm for the problem of optimally drilling the holes in an aluminum plug plate to allow it to do a spectrographic analysis of a set of galaxies. Based on your analysis of the robot drill device, you notice that the various amounts of time it takes to move between drilling holes satisfies the triangle inequality. Nevertheless, your supervisor does not want you to use the MST approximation algorithm or the Christofides approximation algorithm. Instead, your supervisor wants you to use a nearest-neighbor greedy algorithm for solving this instance of METRIC-TSP. In this greedy algorithm, one starts with city number 1 as the "current" city, and then repeatedly chooses the next city to add to the tour to be the one that is closest to the current city (then making the added city to be the "current" one). Show that your supervisor's nearest-neighbor greedy algorithm does not, in general, result in a 2-approximation algorithm for METRIC-TSP.

A-18.6 Consider the astronomy application of METRIC-TSP, as in the previous exercise, but now suppose that you have an improvement to your supervisor's nearest-neighbor idea. Your nearest-neighbor greedy algorithm works like this: you start with city number 1 and add cities one at time, always maintaining a tour, T, of the cities added so far. Given the current set of cities, C, you find the city, c, not in C that minimizes the distance to a city, d, that is in C. Then you add d to T immediately after c, add d to C, and repeat until T is a tour for all the cities. Show that your nearest-neighbor greedy approach results in a 2-approximation algorithm for METRIC-TSP.

Chapter Notes

General discussions of approximation algorithms can be found in several other books, including those by Hochbaum [101] and Papadimitriou and Steiglitz [170], as well as the chapter by Klein and Young [127]. The PTAS for KNAPSACK is modeled after a result of Ibarra and Kim [109], as presented by Klein and Young [127]. Papadimitriou and Steiglitz attribute the 2-approximation for VERTEX-COVER to Gavril and Yannakakis. The 2-approximation algorithm for the special case of TSP is due to Rosenkrantz, Stearns, and Lewis [179]. The $O(\log n)$-approximation for SET-COVER, and its proof, follow from work of Chvátal [44], Johnson [112], and Lovász [146]. The Christofides approximation algorithm is due to Nicos Christofides [43].

The discussion of backtracking and branch-and-bound is modeled after discussions by Lewis and Papadimitriou [143] and Brassard and Bratley [37], where backtracking is intended for decision problems and branch-and-bound is for optimization problems. Nevertheless, our discussion is also influenced by Neapolitan and Naimipour [164], who alternatively view backtracking as a heuristic search that uses a depth-first search of a configuration space and branch-and-bound as a heuristic search that uses breadth-first or best-first search with a lower-bound function to perform pruning. The technique of backtracking itself dates to early work of Golomb and Baumert [84].