2 Set Cover

The set cover problem plays the same role in approximation algorithms that the maximum matching problem played in exact algorithms – as a problem whose study led to the development of fundamental techniques for the entire field. For our purpose this problem is particularly useful, since it offers a very simple setting in which many of the basic algorithm design techniques can be explained with great ease. In this chapter, we will cover two combinatorial techniques: the fundamental greedy technique and the technique of layering. In Part II we will explain both the basic LP-based techniques of rounding and the primal–dual schema using this problem.

Among the first strategies one tries when designing an algorithm for an optimization problem is some form of the greedy strategy. Even if this strategy does not work for a specific problem, proving this via a counterexample can provide crucial insights into the structure of the problem.

Perhaps the most natural use of this strategy in approximation algorithms is to the set cover problem. Besides the greedy set cover algorithm, we will also present the technique of layering in this chapter. Because of its generality, the set cover problem has wide applicability, sometimes even in unexpected ways. In this chapter we will illustrate such an application – to the shortest superstring problem (see Chapter 7 for an improved algorithm for the latter problem).

Problem 2.1 (Set cover) Given a universe U of n elements, a collection of subsets of U, $\mathcal{S} = \{S_1, \ldots, S_k\}$, and a cost function $c : \mathcal{S} \to \mathbf{Q}^+$, find a minimum cost subcollection of $\mathcal{S}$ that covers all elements of U.

Define the *frequency* of an element to be the number of sets it is in. A useful parameter is the frequency of the most frequent element. Let us denote this by f. The various approximation algorithms for set cover achieve one of two factors: $O(\log n)$ or f. Clearly, neither dominates the other in all instances. The special case of set cover with $f = 2$ is essentially the vertex cover problem (see Exercise 2.7), for which we gave a factor 2 approximation algorithm in Chapter 1.

2.1 The greedy algorithm

The greedy strategy applies naturally to the set cover problem: iteratively pick the most cost-effective set and remove the covered elements, until all elements are covered. Let C be the set of elements already covered at the beginning of an iteration. During this iteration, define the *cost-effectiveness* of a set S to be the average cost at which it covers new elements, i.e., $c(S)/|S-C|$. Define the *price* of an element to be the average cost at which it is covered. Equivalently, when a set S is picked, we can think of its cost being distributed equally among the new elements covered, to set their prices.

Algorithm 2.2 (Greedy set cover algorithm)

1. $C \leftarrow \emptyset$
2. While $C \neq U$ do
   Find the most cost-effective set in the current iteration, say S.
   Let $\alpha = \frac{\text{cost}(S)}{|S-C|}$, i.e., the cost-effectiveness of S.
   Pick S, and for each $e \in S - C$, set $\text{price}(e) = \alpha$.
   $C \leftarrow C \cup S$.
3. Output the picked sets.

Number the elements of U in the order in which they were covered by the algorithm, resolving ties arbitrarily. Let $e_1, \ldots, e_n$ be this numbering.

Lemma 2.3 *For each $k \in \{1, \ldots, n\}$, $\text{price}(e_k) \leq \text{OPT}/(n-k+1)$.*

Proof: In any iteration, the leftover sets of the optimal solution can cover the remaining elements at a cost of at most OPT. Therefore, among these sets, there must be one having cost-effectiveness of at most $\text{OPT}/|\overline{C}|$. In the iteration in which element e_k was covered, $\overline{C}$ contained at least $n-k+1$ elements. Since e_k was covered by the most cost-effective set in this iteration, it follows that

$$\text{price}(e_k) \leq \frac{\text{OPT}}{|\overline{C}|} \leq \frac{\text{OPT}}{n-k+1}.$$

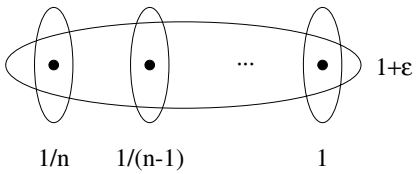
□

From Lemma 2.3 we immediately obtain:

Theorem 2.4 *The greedy algorithm is an H_n factor approximation algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$.*

Proof: Since the cost of each set picked is distributed among the new elements covered, the total cost of the set cover picked is equal to $\sum_{k=1}^{n} \text{price}(e_k)$. By Lemma 2.3, this is at most $\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) \cdot \text{OPT}$. □

Example 2.5 The following is a tight example for Algorithm 2.2:



When run on this instance the greedy algorithm outputs the cover consisting of the n singleton sets, since in each iteration some singleton is the most cost-effective set. Thus, the algorithm outputs a cover of cost

$$\frac{1}{n} + \frac{1}{n-1} + \cdots + 1 = H_n.$$

On the other hand, the optimal cover has a cost of $1 + \varepsilon$. □

Surprisingly enough, for the minimum set cover problem the obvious algorithm given above is essentially the best one can hope for; see Sections 29.7 and 29.9.

In Chapter 1 we pointed out that finding a good lower bound on OPT is a basic starting point in the design of an approximation algorithm for a minimization problem. At this point the reader may be wondering whether there is any truth to this claim. We will show in Section 13.1 that the correct way to view the greedy set cover algorithm is in the setting of the LP-duality theory – this will not only provide the lower bound on which this algorithm is based, but will also help obtain algorithms for several generalizations of this problem.

2.2 Layering

The algorithm design technique of layering is also best introduced via set cover. We note, however, that this is not a very widely applicable technique. We will give a factor 2 approximation algorithm for vertex cover, assuming arbitrary weights, and leave the problem of generalizing this to a factor f approximation algorithm for set cover, where f is the frequency of the most frequent element (see Exercise 2.13).

The idea in layering is to decompose the given weight function on vertices into convenient functions, called degree-weighted, on a nested sequence of subgraphs of G. For degree-weighted functions, we will show that we will be within twice the optimal even if we pick all vertices in the cover.

Let us introduce some notation. Let $w : V \to \mathbf{Q}^+$ be the function assigning weights to the vertices of the given graph $G = (V, E)$. We will say that a function assigning vertex weights is *degree-weighted* if there is a constant

$c > 0$ such that the weight of each vertex $v \in V$ is $c \cdot \deg(v)$. The significance of such a weight function is captured in:

Lemma 2.6 *Let $w : V \to \mathbf{Q}^+$ be a degree-weighted function. Then $w(V) \leq 2 \cdot \mathrm{OPT}$.*

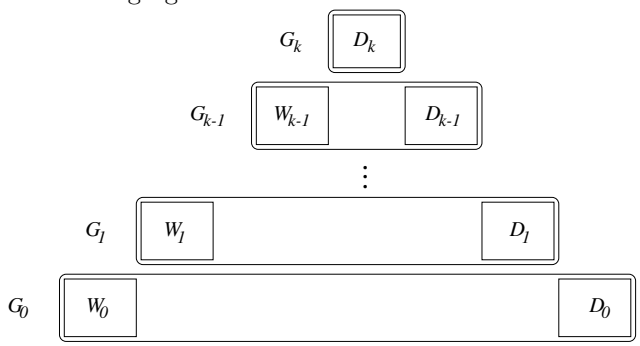
Proof: Let c be the constant such that $w(v) = c \cdot \deg(v)$, and let U be an optimal vertex cover in G. Since U covers all the edges,

$$\sum_{v \in U} \deg(v) \geq |E|.$$

Therefore, $w(U) \geq c|E|$. Now, since $\sum_{v \in V} \deg(v) = 2|E|$, $w(V) = 2c|E|$. The lemma follows. □

Let us define the *largest degree-weighted function in w* as follows: remove all degree zero vertices from the graph, and over the remaining vertices, compute $c = \min\{w(v)/\deg(v)\}$. Then, $t(v) = c \cdot \deg(v)$ is the desired function. Define $w'(v) = w(v) - t(v)$ to be the *residual weight function*.

The algorithm for decomposing w into degree-weighted functions is as follows. Let $G_0 = G$. Remove degree zero vertices from G_0, say this set is D_0, and compute the largest degree-weighted function in w. Let W_0 be vertices of zero residual weight; these vertices are included in the vertex cover. Let G_1 be the graph induced on $V - (D_0 \cup W_0)$. Now, the entire process is repeated on G_1 w.r.t. the residual weight function. The process terminates when all vertices are of degree zero; let G_k denote this graph. The process is schematically shown in the following figure.



Let $t_0, ..., t_{k-1}$ be the degree-weighted functions defined on graphs $G_0, ..., G_{k-1}$. The vertex cover chosen is $C = W_0 \cup \ldots \cup W_{k-1}$. Clearly, $V - C = D_0 \cup \ldots \cup D_k$.

Theorem 2.7 *The layer algorithm achieves an approximation guarantee of factor 2 for the vertex cover problem, assuming arbitrary vertex weights.*

Proof: We need to show that set C is a vertex cover for G and $w(C) \leq 2 \cdot \mathrm{OPT}$. Assume, for contradiction, that C is not a vertex cover for G. Then

there must be an edge (u, v) with $u \in D_i$ and $v \in D_j$, for some i, j. Assume $i \leq j$. Therefore, (u, v) is present in G_i, contradicting the fact that u is a degree zero vertex.

Let C^* be an optimal vertex cover. For proving the second part, consider a vertex $v \in C$. If $v \in W_j$, its weight can be decomposed as

$$w(v) = \sum_{i \leq j} t_i(v).$$

Next, consider a vertex $v \in V - C$. If $v \in D_j$, a lower bound on its weight is given by

$$w(v) \geq \sum_{i < j} t_i(v).$$

The important observation is that in each layer i, $C^* \cap G_i$ is a vertex cover for G_i, since G_i is a vertex-induced graph. Therefore, by Lemma 2.6, $t_i(C \cap G_i) \leq 2 \cdot t_i(C^* \cap G_i)$. By the decomposition of weights given above, we get

$$w(C) = \sum_{i=0}^{k-1} t_i(C \cap G_i) \leq 2 \sum_{i=0}^{k-1} t_i(C^* \cap G_i) \leq 2 \cdot w(C^*).$$

□

Example 2.8 A tight example is provided by the family of complete bipartite graphs, $K_{n,n}$, with all vertices of unit weight. The layering algorithm will pick all $2n$ vertices of $K_{n,n}$ in the cover, whereas the optimal cover picks only one side of the bipartition. □

2.3 Application to shortest superstring

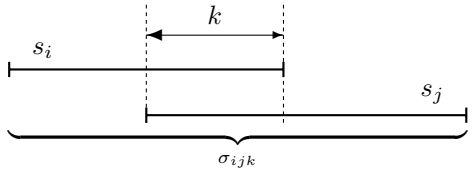
The following algorithm is given primarily to demonstrate the wide applicability of set cover. A constant factor approximation algorithm for shortest superstring will be given in Chapter 7.

Let us first provide motivation for this problem. The human DNA can be viewed as a very long string over a four-letter alphabet. Scientists are attempting to decipher this string. Since it is very long, several overlapping short segments of this string are first deciphered. Of course, the locations of these segments on the original DNA are not known. It is hypothesized that the shortest string which contains these segments as substrings is a good approximation to the original DNA string.

Problem 2.9 (Shortest superstring) Given a finite alphabet Σ, and a set of n strings, $S = \{s_1, \ldots, s_n\} \subseteq \Sigma^+$, find a shortest string s that contains each s_i as a substring. Without loss of generality, we may assume that no string s_i is a substring of another string s_j, $j \neq i$.

This problem is **NP**-hard. Perhaps the first algorithm that comes to mind for finding a short superstring is the following greedy algorithm. Define the *overlap* of two strings $s, t \in \Sigma^*$ as the maximum length of a suffix of s that is also a prefix of t. The algorithm maintains a set of strings T; initially $T = S$. At each step, the algorithm selects from T two strings that have maximum overlap and replaces them with the string obtained by overlapping them as much as possible. After $n - 1$ steps, T will contain a single string. Clearly, this string contains each s_i as a substring. This algorithm is conjectured to have an approximation factor of 2. To see that the approximation factor of this algorithm is no better than 2, consider an input consisting of 3 strings: ab^k, b^kc, and b^{k+1}. If the first two strings are selected in the first iteration, the greedy algorithm produces the string ab^kcb^{k+1}. This is almost twice as long as the shortest superstring, $ab^{k+1}c$.

We will obtain a $2H_n$ factor approximation algorithm, using the greedy set cover algorithm. The set cover instance, denoted by $\mathcal{S}$, is constructed as follows. For $s_i, s_j \in S$ and $k > 0$, if the last k symbols of s_i are the same as the first k symbols of s_j, let σ_{ijk} be the string obtained by overlapping these k positions of s_i and s_j:



Let M be the set that consists of the strings σ_{ijk}, for all valid choices of i, j, k. For a string $\pi \in \Sigma^+$, define $\text{set}(\pi) = \{s \in S \mid s \text{ is a substring of } \pi\}$. The universal set of the set cover instance $\mathcal{S}$ is S, and the specified subsets of S are $\text{set}(\pi)$, for each string $\pi \in S \cup I$. The cost of $\text{set}(\pi)$ is $|\pi|$, i.e., the length of string π.

Let $\text{OPT}_{\mathcal{S}}$ and OPT denote the cost of an optimal solution to $\mathcal{S}$ and the length of the shortest superstring of S, respectively. As shown in Lemma 2.11, $\text{OPT}_{\mathcal{S}}$ and OPT are within a factor of 2 of each other, and so an approximation algorithm for set cover can be used to obtain an approximation algorithm for shortest superstring. The complete algorithm is:

Algorithm 2.10 (Shortest superstring via set cover)

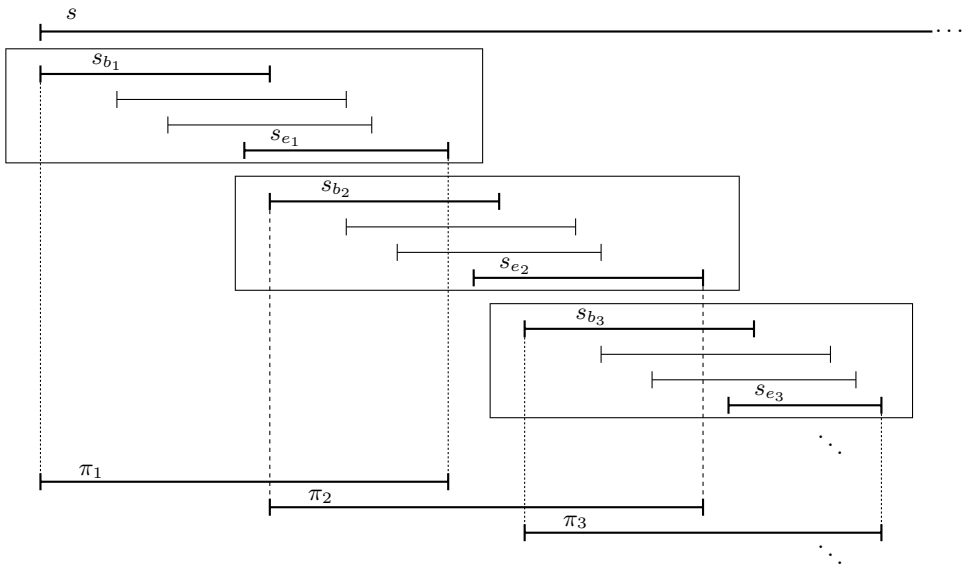
1. Use the greedy set cover algorithm to find a cover for the instance $\mathcal{S}$. Let $\text{set}(\pi_1), \ldots, \text{set}(\pi_k)$ be the sets picked by this cover.
2. Concatenate the strings $\pi_1, \ldots, \pi_k$, in any order.
3. Output the resulting string, say s.

Lemma 2.11 $\text{OPT} \leq \text{OPT}_{\mathcal{S}} \leq 2 \cdot \text{OPT}$.

Proof: Consider an optimal set cover, say $\{\text{set}(\pi_i) | 1 \leq i \leq l\}$, and obtain a string, say s, by concatenating the strings π_i, $1 \leq i \leq l$, in any order. Clearly, $|s| = \text{OPT}_{\mathcal{S}}$. Since each string of S is a substring of some π_i, $1 \leq i \leq l$, it is also a substring of s. Hence $\text{OPT}_{\mathcal{S}} = |s| \geq \text{OPT}$.

To prove the second inequality, let s be a shortest superstring of $s_1, \ldots, s_n$, $|s| = \text{OPT}$. It suffices to produce *some* set cover of cost at most $2 \cdot \text{OPT}$.

Consider the leftmost occurrence of the strings $s_1, \ldots, s_n$ in string s. Since no string among $s_1, \ldots, s_n$ is a substring of another, these n leftmost occurrences start at distinct places in s. For the same reason, they also end at distinct places. Renumber the n strings in the order in which their leftmost occurrences start. Again, since no string is a substring of another, this is also the order in which they end.



We will partition the ordered list of strings $s_1, \ldots, s_n$ in groups as described below. Each group will consist of a contiguous set of strings from this

list. Let b_i and e_i denote the index of the first and last string in the ith group ($b_i = e_i$ is allowed). Thus, $b_1 = 1$. Let e_1 be the largest index of a string that overlaps with s_1 (there exists at least one such string, namely s_1 itself). In general, if $e_i < n$ we set $b_{i+1} = e_i + 1$ and denote by e_{i+1} the largest index of a string that overlaps with $s_{b_{i+1}}$. Eventually, we will get $e_t = n$ for some $t \leq n$.

For each pair of strings (s_{b_i}, s_{e_i}), let $k_i > 0$ be the length of the overlap between their leftmost occurrences in s (this may be different from their maximum overlap). Let $\pi_i = \sigma_{b_i e_i k_i}$. Clearly, $\{\text{set}(\pi_i) | 1 \leq i \leq t\}$ is a solution for $\mathcal{S}$, of cost $\sum_i |\pi_i|$.

The critical observation is that π_i does not overlap π_{i+2}. We will prove this claim for $i = 1$; the same argument applies to an arbitrary i. Assume, for contradiction, that π_1 overlaps π_3. Then the occurrence of s_{b_3} in s overlaps the occurrence of s_{e_1}. However, s_{b_3} does not overlap s_{b_2} (otherwise, s_{b_3} would have been put in the second group). This implies that s_{e_1} ends later than s_{b_2}, contradicting the property of endings of strings established earlier.

Because of this observation, each symbol of s is covered by at most two of the π_i's. Hence $\text{OPT}_{\mathcal{S}} \leq \sum_i |\pi_i| \leq 2 \cdot \text{OPT}$. □

The size of the universal set in the set cover instance $\mathcal{S}$ is n, the number of strings in the given shortest superstring instance. This fact, Lemma 2.11, and Theorem 2.4 immediately give the following theorem.

Theorem 2.12 *Algorithm 2.10 is a $2H_n$ factor algorithm for the shortest superstring problem, where n is the number of strings in the given instance.*

2.4 Exercises

2.1 Given an undirected graph $G = (V, E)$, the *cardinality maximum cut problem* asks for a partition of V into sets S and $\overline{S}$ so that the number of edges running between these sets is maximized. Consider the following greedy algorithm for this problem. Here v_1 and v_2 are arbitrary vertices in G, and for $A \subset V$, $d(v, A)$ denotes the number of edges running between vertex v and set A.

Algorithm 2.13

1. Initialization:
   $A \leftarrow \{v_1\}$
   $B \leftarrow \{v_2\}$
2. For $v \in V - \{v_1, v_2\}$ do:
   if $d(v, A) \geq d(v, B)$ then $B \leftarrow B \cup \{v\}$,
   else $A \leftarrow A \cup \{v\}$.
3. Output A and B.

Show that this is a factor 1/2 approximation algorithm and give a tight example. What is the upper bound on OPT that you are using? Give examples of graphs for which this upper bound is as bad as twice OPT. Generalize the problem and the algorithm to weighted graphs.

2.2 Consider the following algorithm for the maximum cut problem, based on the technique of *local search*. Given a partition of V into sets, the basic step of the algorithm, called *flip*, is that of moving a vertex from one side of the partition to the other. The following algorithm finds a *locally optimal solution* under the flip operation, i.e., a solution which cannot be improved by a single flip.

The algorithm starts with an arbitrary partition of V. While there is a vertex such that flipping it increases the size of the cut, the algorithm flips such a vertex. (Observe that a vertex qualifies for a flip if it has more neighbors in its own partition than in the other side.) The algorithm terminates when no vertex qualifies for a flip. Show that this algorithm terminates in polynomial time, and achieves an approximation guarantee of 1/2.

2.3 Consider the following generalization of the maximum cut problem.

Problem 2.14 (MAX k-CUT) Given an undirected graph $G = (V, E)$ with nonnegative edge costs, and an integer k, find a partition of V into sets $S_1, \ldots, S_k$ so that the total cost of edges running between these sets is maximized.

Give a greedy algorithm for this problem that achieves a factor of $(1 - \frac{1}{k})$. Is the analysis of your algorithm tight?

2.4 Give a greedy algorithm for the following problem achieving an approximation guarantee of factor 1/4.

Problem 2.15 (Maximum directed cut) Given a directed graph $G = (V, E)$ with nonnegative edge costs, find a subset $S \subset V$ so as to maximize the total cost of edges out of S, i.e., $\text{cost}(\{(u \to v) \mid u \in S \text{ and } v \in \overline{S}\})$.

2.5 (N. Vishnoi) Use the algorithm in Exercise 2.2 and the fact that the vertex cover problem is polynomial time solvable for bipartite graphs to give a factor $\lceil \log_2 \Delta \rceil$ algorithm for vertex cover, where Δ is the degree of the vertex having highest degree.
Hint: Let H denote the subgraph consisting of edges in the maximum cut found by Algorithm 2.13. Clearly, H is bipartite, and for any vertex v, $\deg_H(v) \geq (1/2)\deg_G(v)$.

2.6 (Wigderson [257]) Consider the following problem.

Problem 2.16 (Vertex coloring) Given an undirected graph $G = (V, E)$, color its vertices with the minimum number of colors so that the two endpoints of each edge receive distinct colors.

1. Give a greedy algorithm for coloring G with $\Delta + 1$ colors, where Δ is the maximum degree of a vertex in G.
2. Give an algorithm for coloring a 3-colorable graph with $O(\sqrt{n})$ colors.
   Hint: For any vertex v, the induced subgraph on its neighbors, $N(v)$, is bipartite, and hence optimally colorable. If v has degree $> \sqrt{n}$, color $v \cup N(v)$ using 3 distinct colors. Continue until every vertex has degree $\leq \sqrt{n}$. Then use the algorithm in the first part.

2.7 Let 2SC denote the restriction of set cover to instances having $f = 2$. Show that 2SC is equivalent to the vertex cover problem, with arbitrary costs, under approximation factor preserving reductions.

2.8 Prove that Algorithm 2.2 achieves an approximation factor of H_k, where k is the cardinality of the largest specified subset of U.

2.9 Give a greedy algorithm that achieves an approximation guarantee of H_n for set multicover, which is a generalization of set cover in which an integral coverage requirement is also specified for each element and sets can be picked multiple numbers of times to satisfy all coverage requirements. Assume that the cost of picking α copies of set S_i is $\alpha \cdot \text{cost}(S_i)$.

2.10 By giving an appropriate tight example, show that the analysis of Algorithm 2.2 cannot be improved even for the cardinality set cover problem, i.e., if all specified sets have unit cost.
Hint: Consider running the greedy algorithm on a vertex cover instance.

2.11 Consider the following algorithm for the weighted vertex cover problem. For each vertex v, $t(v)$ is initialized to its weight, and when $t(v)$ drops to 0, v is picked in the cover. $c(e)$ is the amount charged to edge e.

Algorithm 2.17

1. Initialization:
   $C \leftarrow \emptyset$
   $\forall v \in V$, $t(v) \leftarrow w(v)$
   $\forall e \in E$, $c(e) \leftarrow 0$
2. While C is not a vertex cover do:
   Pick an uncovered edge, say (u, v). Let $m = \min(t(u), t(v))$.
   $t(u) \leftarrow t(u) - m$
   $t(v) \leftarrow t(v) - m$
   $c(u, v) \leftarrow m$
   Include in C all vertices having $t(v) = 0$.
3. Output C.

Show that this is a factor 2 approximation algorithm.
Hint: Show that the total amount charged to edges is a lower bound on OPT and that the weight of cover C is at most twice the total amount charged to edges.

2.12 Consider the layering algorithm for vertex cover. Another weight function for which we have a factor 2 approximation algorithm is the constant function – by simply using the factor 2 algorithm for the cardinality vertex cover problem. Can layering be made to work by using this function instead of the degree-weighted function?

2.13 Use layering to get a factor f approximation algorithm for set cover, where f is the frequency of the most frequent element. Provide a tight example for this algorithm.

2.14 A *tournament* is a directed graph $G = (V, E)$, such that for each pair of vertices, $u, v \in V$, exactly one of (u, v) and (v, u) is in E. A *feedback vertex set* for G is a subset of the vertices of G whose removal leaves an acyclic graph. Give a factor 3 algorithm for the problem of finding a minimum feedback vertex set in a directed graph.
Hint: Show that it is sufficient to "kill" all length 3 cycles. Use the factor f set cover algorithm.

2.15 (Hochbaum [125]) Consider the following problem.

Problem 2.18 (Maximum coverage) Given a universal set U of n elements, with nonnegative weights specified, a collection of subsets of U, $S_1, \ldots, S_l$, and an integer k, pick k sets so as to maximize the weight of elements covered.

Show that the obvious algorithm, of greedily picking the best set in each iteration until k sets are picked, achieves an approximation factor of

$$1 - \left(1 - \frac{1}{k}\right)^k > 1 - \frac{1}{e}.$$

2.16 Using set cover, obtain approximation algorithms for the following variants of the shortest superstring problem (here s^R is the reverse of string s):

1. Find the shortest string that contains, for each string $s_i \in S$, both s_i and s_i^R as substrings.
   Hint: The universal set for the set cover instance will contain $2n$ elements, s_i and s_i^R, for $1 \leq i \leq n$.
2. Find the shortest string that contains, for each string $s_i \in S$, either s_i or s_i^R as a substring.
   Hint: Define $\text{set}(\pi) = \{s \in S \mid s \text{ or } s^R \text{ is a substring of } \pi\}$. Choose the strings π appropriately.

2.5 Notes

Algorithm 2.2 is due to Johnson [150], Lovász [192], and Chvátal [48]. The hardness result for set cover, showing that this algorithm is essentially the best possible, is due to Feige [80], improving on the result of Lund and Yannakakis [199]. The application to shortest superstring is due to Li [187].

3 Steiner Tree and TSP

In this chapter, we will present constant factor algorithms for two fundamental problems, metric Steiner tree and metric TSP. The reasons for considering the metric case of these problems are quite different. For Steiner tree, this is the core of the problem – the rest of the problem reduces to this case. For TSP, without this restriction, the problem admits no approximation factor, assuming $\mathbf{P} \neq \mathbf{NP}$. The algorithms, and their analyses, are similar in spirit, which is the reason for presenting these problems together.

3.1 Metric Steiner tree

The Steiner tree problem was defined by Gauss in a letter he wrote to Schumacher (reproduced on the cover of this book). Today, this problem occupies a central place in the field of approximation algorithms. The problem has a wide range of applications, all the way from finding minimum length interconnection of terminals in VLSI design to constructing phylogeny trees in computational biology. This problem and its generalizations will be studied extensively in this book, see Chapters 22 and 23.

Problem 3.1 (Steiner tree) Given an undirected graph $G = (V, E)$ with nonnegative edge costs and whose vertices are partitioned into two sets, *required* and *Steiner*, find a minimum cost tree in G that contains all the required vertices and any subset of the Steiner vertices.

We will first show that the core of this problem lies in its restriction to instances in which the edge costs satisfy *the triangle inequality*, i.e., G is a complete undirected graph, and for any three vertices u, v, and w, $\text{cost}(u, v) \leq \text{cost}(u, w) + \text{cost}(v, w)$. Let us call this restriction the *metric Steiner tree problem*.

Theorem 3.2 *There is an approximation factor preserving reduction from the Steiner tree problem to the metric Steiner tree problem.*

Proof: We will transform, in polynomial time, an instance I of the Steiner tree problem, consisting of graph $G = (V, E)$, to an instance I' of the metric Steiner tree problem as follows. Let G' be the complete undirected graph on

vertex set V. Define the cost of edge (u, v) in G' to be the cost of a shortest u–v path in G. G' is called the *metric closure* of G. The partition of V into required and Steiner vertices in I' is the same as in I.

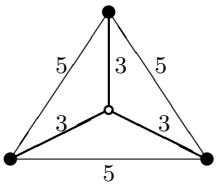
For any edge $(u, v) \in E$, its cost in G' is no more than its cost in G. Therefore, the cost of an optimal solution in I' does not exceed the cost of an optimal solution in I.

Next, given a Steiner tree T' in I', we will show how to obtain, in polynomial time, a Steiner tree T in I of at most the same cost. The cost of an edge (u, v) in G' corresponds to the cost of a path in G. Replace each edge of T' by the corresponding path to obtain a subgraph of G. Clearly, in this subgraph, all the required vertices are connected. However, this subgraph may, in general, contain cycles. If so, remove edges to obtain tree T. This completes the approximation factor preserving reduction. □

As a consequence of Theorem 3.2, any approximation factor established for the metric Steiner tree problem carries over to the entire Steiner tree problem.

3.1.1 MST-based algorithm

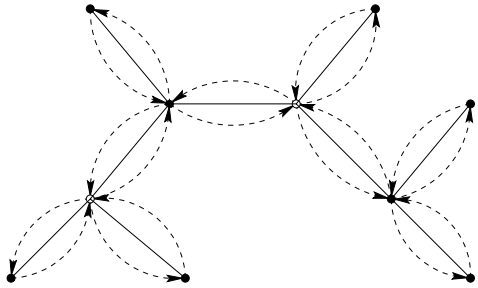
Let R denote the set of required vertices. Clearly, a minimum spanning tree (MST) on R is a feasible solution for this problem. Since the problem of finding an MST is in **P** and the metric Steiner tree problem is **NP**-hard, we cannot expect the MST on R to always give an optimal Steiner tree; below is an example in which the MST is strictly costlier.



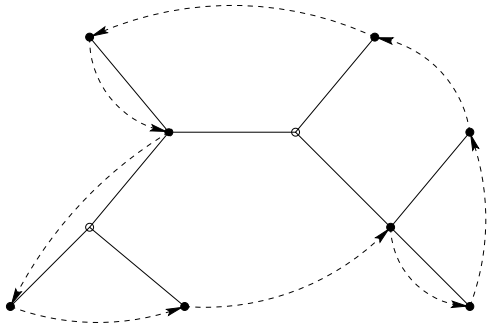
Even so, an MST on R is not much more costly than an optimal Steiner tree:

Theorem 3.3 *The cost of an MST on R is within $2 \cdot \mathrm{OPT}$.*

Proof: Consider a Steiner tree of cost OPT. By doubling its edges we obtain an Eulerian graph connecting all vertices of R and, possibly, some Steiner vertices. Find an Euler tour of this graph, for example by traversing the edges in DFS (depth first search) order:



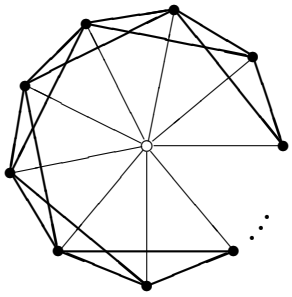
The cost of this Euler tour is $2 \cdot \text{OPT}$. Next obtain a Hamiltonian cycle on the vertices of R by traversing the Euler tour and "short-cutting" Steiner vertices and previously visited vertices of R:



Because of triangle inequality, the shortcuts do not increase the cost of the tour. If we delete one edge of this Hamiltonian cycle, we obtain a path that spans R and has cost at most $2 \cdot \text{OPT}$. This path is also a spanning tree on R. Hence, the MST on R has cost at most $2 \cdot \text{OPT}$. □

Theorem 3.3 gives a straightforward factor 2 algorithm for the metric Steiner tree problem: simply find an MST on the set of required vertices. As in the case of set cover, the "correct" way of viewing this algorithm is in the setting of LP-duality theory. In Chapters 22 and 23 we will see that LP-duality provides the lower bound on which this algorithm is based and also helps solve generalizations of this problem.

Example 3.4 For a tight example, consider a graph with n required vertices and one Steiner vertex. An edge between the Steiner vertex and a required vertex has cost 1, and an edge between two required vertices has cost 2 (not all edges of cost 2 are shown below). In this graph, any MST on R has cost $2(n-1)$, while $\text{OPT} = n$.



□

3.2 Metric TSP

The following is a well-studied problem in combinatorial optimization.

Problem 3.5 (Traveling salesman problem (TSP)) Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.

In its full generality, TSP cannot be approximated, assuming $\mathbf{P} \neq \mathbf{NP}$.

Theorem 3.6 *For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $\mathbf{P} = \mathbf{NP}$.*

Proof: Assume, for a contradiction, that there is a factor $\alpha(n)$ polynomial time approximation algorithm, $\mathcal{A}$, for the general TSP problem. We will show that $\mathcal{A}$ can be used for deciding the Hamiltonian cycle problem (which is **NP**-hard) in polynomial time, thus implying $\mathbf{P} = \mathbf{NP}$.

The central idea is a reduction from the Hamiltonian cycle problem to TSP, that transforms a graph G on n vertices to an edge-weighted complete graph G' on n vertices such that

- if G has a Hamiltonian cycle, then the cost of an optimal TSP tour in G' is n, and
- if G does not have a Hamiltonian cycle, then an optimal TSP tour in G' is of cost $> \alpha(n) \cdot n$.

Observe that when run on graph G', algorithm $\mathcal{A}$ must return a solution of cost $\leq \alpha(n) \cdot n$ in the first case, and a solution of cost $> \alpha(n) \cdot n$ in the second case. Thus, it can be used for deciding whether G contains a Hamiltonian cycle.

The reduction is simple. Assign a weight of 1 to edges of G, and a weight of $\alpha(n) \cdot n$ to nonedges, to obtain G'. Now, if G has a Hamiltonian cycle, then the corresponding tour in G' has cost n. On the other hand, if G has

no Hamiltonian cycle, any tour in G' must use an edge of cost $\alpha(n) \cdot n$, and therefore has cost $> \alpha(n) \cdot n$. □

Notice that in order to obtain such a strong nonapproximability result, we had to assign edge costs that violate triangle inequality. If we restrict ourselves to graphs in which edge costs satisfy triangle inequality, i.e., consider *metric TSP*, the problem remains **NP**-complete, but it is no longer hard to approximate.

3.2.1 A simple factor 2 algorithm

We will first present a simple factor 2 algorithm. The lower bound we will use for obtaining this factor is the cost of an MST in G. This is a lower bound because deleting any edge from an optimal solution to TSP gives us a spanning tree of G.

Algorithm 3.7 (Metric TSP – factor 2)

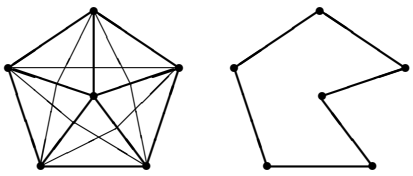
1. Find an MST, T, of G.
2. Double every edge of the MST to obtain an Eulerian graph.
3. Find an Eulerian tour, $\mathcal{T}$, on this graph.
4. Output the tour that visits vertices of G in the order of their first appearance in $\mathcal{T}$. Let $\mathcal{C}$ be this tour.

Notice that Step 4 is similar to the "short-cutting" step in Theorem 3.3.

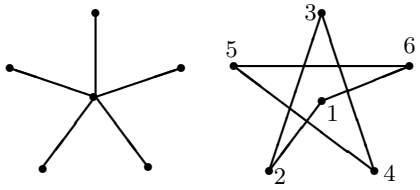
Theorem 3.8 *Algorithm 3.7 is a factor* 2 *approximation algorithm for metric TSP.*

Proof: As noted above, $\text{cost}(T) \leq \text{OPT}$. Since $\mathcal{T}$ contains each edge of T twice, $\text{cost}(\mathcal{T}) = 2 \cdot \text{cost}(T)$. Because of triangle inequality, after the "short-cutting" step, $\text{cost}(\mathcal{C}) \leq \text{cost}(\mathcal{T})$. Combining these inequalities we get that $\text{cost}(\mathcal{C}) \leq 2 \cdot \text{OPT}$. □

Example 3.9 A tight example for this algorithm is given by a complete graph on n vertices with edges of cost 1 and 2. We present the graph for $n = 6$ below, where thick edges have cost 1 and remaining edges have cost 2. For arbitrary n the graph has $2n-2$ edges of cost 1, with these edges forming the union of a star and an $n-1$ cycle; all remaining edges have cost 2. The optimal TSP tour has cost n, as shown below for $n = 6$:



Suppose that the MST found by the algorithm is the spanning star created by edges of cost 1. Moreover, suppose that the Euler tour constructed in Step 3 visits vertices in order shown below for $n = 6$:



Then the tour obtained after short-cutting contains $n-2$ edges of cost 2 and has a total cost of $2n-2$. Asymptotically, this is twice the cost of the optimal TSP tour. □

3.2.2 Improving the factor to 3/2

Algorithm 3.7 first finds a low cost Euler tour spanning the vertices of G, and then short-cuts this tour to find a traveling salesman tour. Is there a cheaper Euler tour than that found by doubling an MST? Recall that a graph has an Euler tour iff all its vertices have even degrees. Thus, we only need to be concerned about the vertices of odd degree in the MST. Let V' denote this set of vertices. $|V'|$ must be even since the sum of degrees of all vertices in the MST is even. Now, if we add to the MST a minimum cost perfect matching on V', every vertex will have an even degree, and we get an Eulerian graph. With this modification, the algorithm achieves an approximation guarantee of 3/2.

Algorithm 3.10 (Metric TSP – factor 3/2)

1. Find an MST of G, say T.
2. Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T and obtain an Eulerian graph.
3. Find an Euler tour, $\mathcal{T}$, of this graph.
4. Output the tour that visits vertices of G in order of their first appearance in $\mathcal{T}$. Let $\mathcal{C}$ be this tour.

Interestingly, the proof of this algorithm is based on a second lower bound on OPT.

Lemma 3.11 *Let $V' \subseteq V$, such that $|V'|$ is even, and let M be a minimum cost perfect matching on V'. Then,* $\text{cost}(M) \leq \text{OPT}/2$.

Proof: Consider an optimal TSP tour of G, say τ. Let τ' be the tour on V' obtained by short-cutting τ. By the triangle inequality, $\text{cost}(\tau') \leq$

cost(τ). Now, τ' is the union of two perfect matchings on V', each consisting of alternate edges of τ. Thus, the cheaper of these matchings has cost $\leq$ cost(τ')/2 $\leq$ OPT/2. Hence the optimal matching also has cost at most OPT/2. □

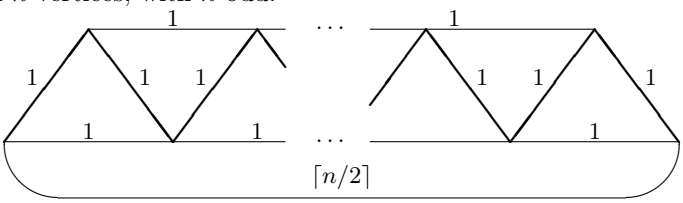
Theorem 3.12 *Algorithm 3.10 achieves an approximation guarantee of* 3/2 *for metric TSP.*

Proof: The cost of the Euler tour,

$$\text{cost}(\mathcal{T}) \leq \text{cost}(T) + \text{cost}(M) \leq \text{OPT} + \frac{1}{2}\text{OPT} = \frac{3}{2}\text{OPT},$$

where the first inequality follows by using the two lower bounds on OPT. Using the triangle inequality, cost($\mathcal{C}$) $\leq$ cost($\mathcal{T}$), and the theorem follows. □

Example 3.13 A tight example for this algorithm is given by the following graph on n vertices, with n odd:



Thick edges represent the MST found in step 1. This MST has only two odd vertices, and by adding the edge joining them we obtain a traveling salesman tour of cost $(n-1) + \lceil n/2 \rceil$. In contrast, the optimal tour has cost n. □

Finding a better approximation algorithm for metric TSP is currently one of the outstanding open problems in this area. Many researchers have conjectured that an approximation factor of 4/3 may be achievable.

3.3 Exercises

3.1 The hardness of the Steiner tree problem lies in determining the optimal subset of Steiner vertices that need to be included in the tree. Show this by proving that if this set is provided, then the optimal Steiner tree can be computed in polynomial time.
Hint: Find an MST on the union of this set and the set of required vertices.

3.2 Let $G = (V, E)$ be a graph with nonnegative edge costs. S, the *senders* and R, the *receivers*, are disjoint subsets of V. The problem is to find a minimum cost subgraph of G that has a path connecting each receiver to a

sender (any sender suffices). Partition the instances into two cases: $S \cup R = V$ and $S \cup R \neq V$. Show that these two cases are in **P** and **NP**-hard, respectively. For the second case, give a factor 2 approximation algorithm.
Hint: Add a new vertex which is connected to each sender by a zero cost edge. Consider the new vertex and all receivers as required and the remaining vertices as Steiner, and find a minimum cost Steiner tree.

3.3 Give an approximation factor preserving reduction from the set cover problem to the following problem, thereby showing that it is unlikely to have a better approximation guarantee than $O(\log n)$.

Problem 3.14 (Directed Steiner tree) $G = (V, E)$ is a directed graph with nonnegative edge costs. The vertex set V is partitioned into two sets, *required* and *Steiner*. One of the required vertices, r, is special. The problem is to find a minimum cost tree in G rooted into r that contains all the required vertices and any subset of the Steiner vertices.
Hint: Construct a three layer graph: layer 1 contains a required vertex corresponding to each element, layer 2 contains a Steiner vertex corresponding to each set, and layer 3 contains r.

3.4 (Hoogeveen [130]) Consider variants on the metric TSP problem in which the object is to find a simple path containing all the vertices of the graph. Three different problems arise, depending on the number (0, 1, or 2) of endpoints of the path that are specified. Obtain the following approximation algorithms.

- If zero or one endpoints are specified, obtain a 3/2 factor algorithm.
- If both endpoints are specified, obtain a 5/3 factor algorithm.

Hint: Use the idea behind Algorithm 3.10.

3.5 (Papadimitriou and Yannakakis [219]) Let G be a complete undirected graph in which all edge lengths are either 1 or 2 (clearly, G satisfies the triangle inequality). Give a 4/3 factor algorithm for TSP in this special class of graphs.
Hint: Start by finding a minimum 2-matching in G. A 2-matching is a subset S of edges so that every vertex has exactly 2 edges of S incident at it.

3.6 (Frieze, Galbiati, and Maffioli [89]) Give an $O(\log n)$ factor approximation algorithm for the following problem.

Problem 3.15 (Asymmetric TSP) We are given a directed graph G on vertex set V, with a nonnegative cost specified for edge $(u \to v)$, for each pair $u, v \in V$. The edge costs satisfy *the directed triangle inequality*, i.e., for any three vertices u, v, and w, $\text{cost}(u \to v) \leq \text{cost}(u \to w) + \text{cost}(w \to v)$. The problem is to find a minimum cost cycle visiting every vertex exactly once.

Hint: Use the fact that a minimum cost cycle cover (i.e., disjoint cycles covering all the vertices) can be found in polynomial time. Shrink the cycles and recurse.

3.7 Let $G = (V, E)$ be a graph with edge costs satisfying the triangle inequality, and $V' \subseteq V$ be a set of even cardinality. Prove or disprove: The cost of a minimum cost perfect matching on V' is bounded above by the cost of a minimum cost perfect matching on V.

3.8 Given n points in $\mathbf{R}^2$, define the optimal Euclidean Steiner tree to be a minimum length tree containing all n points and any other subset of points from $\mathbf{R}^2$. Prove that each of the additional points must have degree three, with all three angles being 120°.

3.9 (Rao, Sadayappan, Hwang, and Shor [230]) This exercise develops a factor 2 approximation algorithm for the following problem.

Problem 3.16 (Rectilinear Steiner arborescence) Let $p_1, \ldots, p_n$ be points given in $\mathbf{R}^2$ in the positive quadrant. A path from the origin to point p_i is said to be *monotone* if it consists of segments traversing in the positive x direction or the positive y direction (informally, going right or up). The problem is to find a minimum length tree containing monotone paths from the origin to each of the n points; such a tree is called *rectilinear Steiner arborescence*.

For point p, define x_p and y_p to be its x and y coordinates, and $|p|_1 = |x_p| + |y_p|$. Say that point p *dominates* point q if $x_p \leq x_q$ and $y_p \leq y_q$. For sets of points A and B, we will say that A *dominates* B if for each point $b \in B$, there is a point $a \in A$ such that a dominates b. For points p and q, define $\mathrm{dom}(p, q) = (x, y)$, where $x = \min(x_p, x_q)$ and $y = \min(y_p, y_q)$. If p dominates q, define segments(p, q) to be a monotone path from p to q. Consider the following algorithm.

Algorithm 3.17 (Rectilinear Steiner arborescence)

1. $T \leftarrow \emptyset$.
2. $P \leftarrow \{p_1, \ldots, p_n\} \cup \{(0, 0)\}$.
3. while $|P| > 1$ do:
   Pick $p, q = \arg\max_{p,q \in P}(|\mathrm{dom}(p, q)|_1)$.
   $P \leftarrow (P - \{p, q\}) \cup \{\mathrm{dom}(p, q)\}$.
   $T \leftarrow T \cup \mathrm{segments}(\mathrm{dom}(p, q), p) \cup \mathrm{segments}(\mathrm{dom}(p, q), q)$.
4. Output T.

For $z \geq 0$, define ℓ_z to be the line $x + y = z$. For a rectilinear Steiner arborescence T, let $T(z) = |T \cap \ell_z|$. Prove that the length of T is

$$\int_{z=0}^{\infty} T(z)\, \mathrm{d}z.$$

Also, for every $x \geq 0$ define $P_z = \{p \in P \; s.t. \; |p|_1 > z\}$, and

$$N(z) = \min\{|C| \; : \; C \subset \ell_z \text{ and } C \text{ dominates } P_z\}.$$

Prove that

$$\int_{z=0}^{\infty} N(z)\, \mathrm{d}z$$

is a lower bound on OPT.

Use these facts to show that Algorithm 3.17 achieves an approximation guarantee of 2.

3.10 (I. Măndoiu) This exercise develops a factor 9 approximation algorithm for the following problem, which finds applications in VLSI clock routing.

Problem 3.18 (Rectilinear zero-skew tree) Given a set S of points in the rectilinear plane, find a minimum length *zero-skew tree* (ZST) for S, i.e., a rooted tree T embedded in the rectilinear plane such that points in S are leaves of T and all root-to-leaf paths in T have equal length. By *length* of a path we mean the sum of the lengths of edges on it.

1. Let T be an arbitrary zero-skew tree, and let R' denote the common length of all root-to-leaf paths. For $r \geq 0$, let $T(r)$ denote the number of points of T that are at a length of $R' - r$ from the root. Prove that the length of T is

$$\int_0^{R'} T(r)\mathrm{d}r$$

2. A closed ℓ_1 ball of radius r centered at point p is the set of all points whose ℓ_1-distance from p is $\leq r$. Let R denote the radius of the smallest ℓ_1-ball that contains all points of S. For $r \geq 0$, let $N(r)$ denote the minimum number of closed ℓ_1-balls of radius r needed to cover all points of S. Prove that

$$\int_0^{R} N(r)\mathrm{d}r$$

is a lower bound on the length of the optimum ZST.

3. Consider the following algorithm. First, compute R and find a radius R ℓ_1-ball enclosing all points of S. The center of this ball is chosen as the root of the resulting ZST. This ball can be partitioned into 4 balls, called its *quadrants*, of radius $R/2$ each. The root can be connected to the center of any of these balls by an edge of length $R/2$. These balls can be further partitioned into 4 balls each of radius $R/4$, and so on.
   The ZST is constructed recursively, starting with the ball of radius R. The center of the current ball is connected to the centers of each of its quadrants that has a point of S. The algorithm then recurses on each of these quadrants. If the current ball contains exactly one point of S, then this ball is not partitioned into quadrants. Let r' be the radius of this ball, c its center, and $p \in S$ the point in it. Clearly, the ℓ_1 distance between c and p is $\leq r'$. Connect c to p by a rectilinear path of length exactly r'.
   Show that for $0 \leq r \leq R$, $T(r) \leq 9N(r)$. Hence, show that this is a factor 9 approximation algorithm.

3.4 Notes

The Steiner tree problem has its origins in a problem posed by Fermat, and was defined by Gauss in a letter he wrote to his student Schumacher on March 21, 1836. Parts of the letter are reproduced on the cover of this book. Courant and Robbins [55] popularized this problem under the name of Steiner, a well known 19th century geometer. See Hwang, Richards, and Winter [133] and Schreiber [236] for the fascinating history of this problem.

The factor 2 Steiner tree algorithm was discovered independently by Choukhmane [44], Iwainsky, Canuto, Taraszow, and Villa [136], Kou, Markowsky, and Berman [177], and Plesník [221]. The factor 3/2 metric TSP algorithm is due to Christofides [45], and Theorem 3.6 is due to Sahni and Gonzalez [232]. The lower bound in Exercise 3.10 is from Charikar, Kleinberg, Kumar, Rajagopalan, Sahai, and Tomkins [41]. The best factor known for the rectilinear zero-skew tree problem, due to Zelikovsky and Măndoiu [263], is 3.

Given n points on the Euclidean plane, the minimum spanning tree on these points is within a factor of $2/\sqrt{3}$ of the minimum Steiner tree (which is allowed to use any set of points on the plane as Steiner points). This was shown by Du and Hwang [63], thereby settling the conjecture of Gilbert and Pollak [100].

4 Multiway Cut and k-Cut

The theory of cuts occupies a central place in the study of exact algorithms. In this chapter, we will present approximation algorithms for natural generalizations of the minimum cut problem. These generalizations are **NP**-hard.

Given a connected, undirected graph $G = (V, E)$ with an assignment of weights to edges, $w : E \to \mathbf{R}^+$, a *cut* is defined by a partition of V into two sets, say V' and $V - V'$, and consists of all edges that have one endpoint in each partition. Clearly, the removal of the cut from G disconnects G. Given *terminals* $s, t \in V$, consider a partition of V that separates s and t. The cut defined by such a partition will be called an *s–t cut*. The problems of finding a minimum weight cut and a minimum weight s–t cut can be efficiently solved using a maximum flow algorithm. Let us generalize these two notions:

Problem 4.1 (Multiway cut) Given a set of terminals $S = \{s_1, s_2, \ldots, s_k\} \subseteq V$, a *multiway cut* is a set of edges whose removal disconnects the terminals from each other. The multiway cut problem asks for the minimum weight such set.

Problem 4.2 (Minimum k-cut) A set of edges whose removal leaves k connected components is called a *k-cut*. The k-cut problem asks for a minimum weight k-cut.

The problem of finding a minimum weight multiway cut is **NP**-hard for any fixed $k \geq 3$. Observe that the case $k = 2$ is precisely the minimum s–t cut problem. The minimum k-cut problem is polynomial time solvable for fixed k; however, it is **NP**-hard if k is specified as part of the input. In this chapter, we will obtain factor $2 - 2/k$ approximation algorithms for both problems. In Chapter 19 we will improve the guarantee for the multiway cut problem to 3/2.

4.1 The multiway cut problem

Define an *isolating cut* for s_i to be a set of edges whose removal disconnects s_i from the rest of the terminals.

Algorithm 4.3 (Multiway cut)

1. For each $i = 1, \ldots, k$, compute a minimum weight isolating cut for s_i, say C_i.
2. Discard the heaviest of these cuts, and output the union of the rest, say C.

Each computation in Step 1 can be accomplished by identifying the terminals in $S - \{s_i\}$ into a single node, and finding a minimum cut separating this node from s_i; this takes one max-flow computation. Clearly, removing C from the graph disconnects every pair of terminals, and so is a multiway cut.

Theorem 4.4 *Algorithm 4.3 achieves an approximation guarantee of* $2-2/k$.

Proof: Let A be an optimal multiway cut in G. We can view A as the union of k cuts as follows: The removal of A from G will create k connected components, each having one terminal (since A is a minimum weight multiway cut, no more than k components will be created). Let A_i be the cut separating the component containing s_i from the rest of the graph. Then $A = \bigcup_{i=1}^{k} A_i$.

Since each edge of A is incident at two of these components, each edge will be in two of the cuts A_i. Hence,

$$\sum_{i=1}^{k} w(A_i) = 2w(A).$$

Clearly, A_i is an isolating cut for s_i. Since C_i is a minimum weight isolating cut for s_i, $w(C_i) \leq w(A_i)$. Notice that this already gives a factor 2 algorithm, by taking the union of all k cuts C_i. Finally, since C is obtained by discarding the heaviest of the cuts C_i,

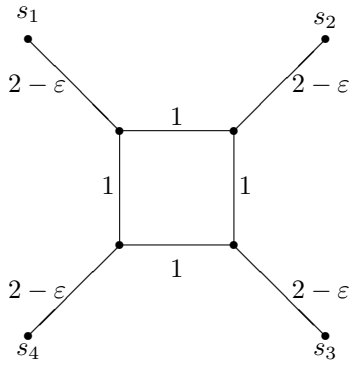
$$w(C) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} w(C_i) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} w(A_i) = 2\left(1 - \frac{1}{k}\right) w(A).$$

□

Once again, Algorithm 4.3 is not based on a lower bounding scheme. Exercise 19.2 gives an algorithm with the same guarantee using an LP-relaxation as the lower bound. The use of LP-relaxations is fruitful for this problem as well. Section 19.1 gives an algorithm with an improved guarantee, using another LP-relaxation.

Example 4.5 A tight example for this algorithm is given by a graph on $2k$ vertices consisting of a k-cycle and a distinct terminal attached to each vertex of the cycle. The edges of the cycle have weight 1 and edges attaching terminals to the cycle have weight $2 - \varepsilon$ for a small fraction $\varepsilon > 0$.

For example, the graph corresponding to $k = 4$ is:



For each terminal s_i, the minimum weight isolating cuts for s_i is given by the edge incident to s_i. So, the cut C returned by the algorithm has weight $(k-1)(2-\varepsilon)$. On the other hand, the optimal multiway cut is given by the cycle edges, and has weight k. □

4.2 The minimum k-cut problem

A natural algorithm for finding a k-cut is as follows. Starting with G, compute a minimum cut in each connected component and remove the lightest one; repeat until there are k connected components. This algorithm does achieve a guarantee of $2 - 2/k$, however, the proof is quite involved. Instead we will use the *Gomory–Hu tree representation of minimum cuts* to give a simpler algorithm achieving the same guarantee.

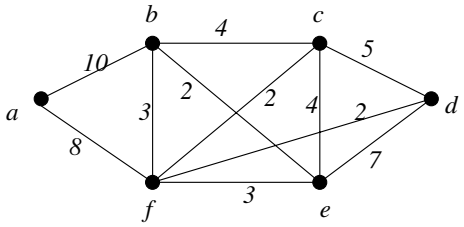
Minimum cuts, as well as sub-optimal cuts, in undirected graphs have several interesting structural properties, as opposed to cuts in directed graphs (the algorithm of Section 28.2 is based on exploiting some of these properties). The existence of Gomory–Hu trees is one of the remarkable consequences of these properties.

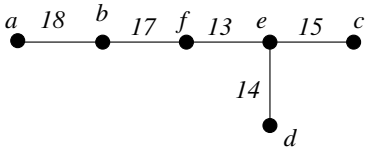
Let T be a tree on vertex set V; the edges of T need not be in E. Let e be an edge in T. Its removal from T creates two connected components. Let S and $\overline{S}$ be the vertex sets of these components. The cut defined in graph G by the partition $(S, \overline{S})$ is the *cut associated with e in G*. Define a weight function w' on the edges of T. Tree T will be said to be a Gomory–Hu tree for G if

1. for each pair of vertices $u, v \in V$, the weight of a minimum u–v cut in G is the same as that in T.
2. for each edge $e \in T$, $w'(e)$ is the weight of the cut associated with e in G, and

A Gomory–Hu tree encodes, in a succinct manner, a minimum u–v cut in G, for each pair of vertices $u, v \in V$ as follows. A minimum u–v cut in T is given by a minimum weight edge on the unique path from u to v in T, say e. By the properties stated above, the cut associated with e in G is a minimum u–v cut, and has weight $w'(e)$. So, for the $\binom{n}{2}$ pairs of vertices $u, v \in V$, we need only $n-1$ cuts, those encoded by the edges of a Gomory–Hu tree, to give minimum u–v cuts in G.

The following figure shows a weighted graph and its associated Gomory–Hu tree. Exercise 4.6 shows how to construct a Gomory–Hu tree for an undirected graph, using only $n-1$ max-flow computations.





We will need the following lemma.

Lemma 4.6 *Let S be the union of cuts in G associated with l edges of T. Then, the removal of S from G leaves a graph with at least $l+1$ components.*

Proof: Removing the corresponding l edges from T leaves exactly $l+1$ connected components, say with vertex sets $V_1, V_2, \ldots, V_{l+1}$. Clearly, removing S from G will disconnect each pair V_i and V_j. Hence we must get at least $l+1$ connected components. □

As a consequence of Lemma 4.6, the union of $k-1$ cuts picked from T will form a k-cut in G. The complete algorithm is given below.

Algorithm 4.7 (Minimum k-cut)

1. Compute a Gomory–Hu tree T for G.
2. Output the union of the lightest $k-1$ cuts of the $n-1$ cuts associated with edges of T in G; let C be this union.

By Lemma 4.6, the removal of C from G will leave at least k components. If more than k components are created, throw back some of the removed edges until there are exactly k components.

Theorem 4.8 *Algorithm 4.7 achieves an approximation factor of* $2 - 2/k$.

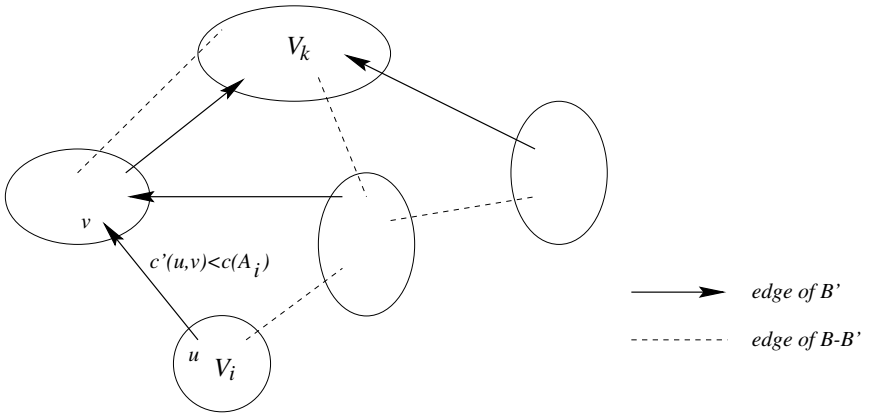
Proof: Let A be an optimal k-cut in G. As in Theorem 4.4, we can view A as the union of k cuts: Let $V_1, V_2, \ldots, V_k$ be the k components formed by removing A from G, and let A_i denote the cut separating V_i from the rest of the graph. Then $A = A_1 \cup \cdots \cup A_k$, and, since each edge of A lies in two of these cuts,

$$\sum_{i=1}^{k} w(A_i) = 2w(A).$$

Without loss of generality assume that A_k is the heaviest of these cuts. The idea behind the rest of the proof is to show that there are $k-1$ cuts defined by the edges of T whose weights are dominated by the weight of the cuts $A_1, A_2, \ldots, A_{k-1}$. Since the algorithm picks the lightest $k-1$ cuts defined by T, the theorem follows.

The $k-1$ cuts are identified as follows. Let B be the set of edges of T that connect across two of the sets $V_1, V_2, \ldots, V_k$. Consider the graph on vertex set V and edge set B, and shrink each of the sets $V_1, V_2, \ldots, V_k$ to a single vertex. This shrunk graph must be connected (since T was connected). Throw edges away until a tree remains. Let $B' \subseteq B$ be the left over edges, $|B'| = k-1$. The edges of B' define the required $k-1$ cuts.

Next, root this tree at V_k (recall that A_k was assumed to be the heaviest cut among the cuts A_i). This helps in defining a correspondence between the edges in B' and the sets $V_1, V_2, \ldots, V_{k-1}$: each edge corresponds to the set it comes out of in the rooted tree.



Suppose edge $(u, v) \in B'$ corresponds to set V_i in this manner. The weight of a minimum u–v cut in G is $w'(u, v)$. Since A_i is a u–v cut in G,

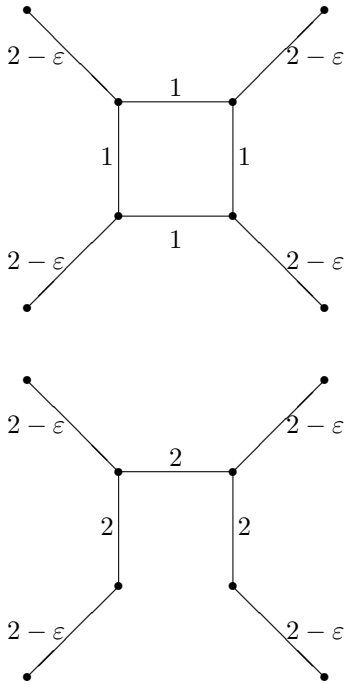
$$w(A_i) \geq w'(u, v).$$

Thus each cut among $A_1, A_2, \ldots, A_{k-1}$ is at least as heavy as the cut defined in G by the corresponding edge of B'. This, together with the fact that C is the union of the lightest $k-1$ cuts defined by T, gives:

$$w(C) \leq \sum_{e \in B'} w'(e) \leq \sum_{i=1}^{k-1} w(A_i) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} w(A_i) = 2\left(1 - \frac{1}{k}\right) w(A).$$

□

Example 4.9 The tight example given above for multiway cuts on $2k$ vertices also serves as a tight example for the k-cut algorithm (of course, there is no need to mark vertices as terminals). Below we give the example for $k = 4$, together with its Gomory–Hu tree.



The lightest $k-1$ cuts in the Gomory–Hu tree have weight $2 - \varepsilon$ each, corresponding to picking edges of weight $2 - \varepsilon$ of G. So, the k-cut returned

by the algorithm has weight $(k-1)(2-\varepsilon)$. On the other hand, the optimal k-cut picks all edges of weight 1, and has weight k. □

4.3 Exercises

4.1 Show that Algorithm 4.3 can be used as a subroutine for finding a k-cut within a factor of $2-2/k$ of the minimum k-cut. How many subroutine calls are needed?

4.2 A natural greedy algorithm for computing a multiway cut is the following. Starting with G, compute minimum s_i–s_j cuts for all pairs s_i, s_j that are still connected and remove the lightest of these cuts; repeat this until all pairs s_i, s_j are disconnected. Prove that this algorithm also achieves a guarantee of $2-2/k$.

The next 4 exercises provide background and an algorithm for finding Gomory–Hu trees.

4.3 Let $G=(V,E)$ be a graph and $w : E \to \mathbf{R}^+$ be an assignment of nonnegative weights to its edges. For $u, v \in V$ let $f(u,v)$ denote the weight of a minimum u–v cut in G.

1. Let $u, v, w \in V$, and suppose $f(u,v) \leq f(u,w) \leq f(v,w)$. Show that $f(u,v) = f(u,w)$, i.e., the two smaller numbers are equal.
2. Show that among the $\binom{n}{2}$ values $f(u,v)$, for all pairs $u, v \in V$, there are at most $n-1$ distinct values.
3. Show that for $u, v, w \in V$,

$$f(u,v) \geq \min\{f(u,w), f(w,v)\}.$$

4. Show that for $u, v, w_1, \ldots, w_r \in V$

$$f(u,w) \geq \min\{f(u,w_1), f(w_1,w_2), \ldots, f(w_r,v)\} \tag{4.1}$$

4.4 Let T be a tree on vertex set V with weight function w' on its edges. We will say that T is a *flow equivalent tree* if it satisfies the first of the two Gomory–Hu conditions. i.e., for each pair of vertices $u, v \in V$, the weight of a minimum u–v cut in G is the same as that in T. Let K be the complete graph on V. Define the weight of each edge (u,v) in K to be $f(u,v)$. Show that any maximum weight spanning tree in K is a flow equivalent tree for G. **Hint:** For $u, v \in V$, let $u, w_1, \ldots, w_r, v$ be the unique path from u to v in T. Use (4.1) and the fact that since T is a maximum weight spanning tree, $f(u,v) \leq \min\{f(u,w_1), \ldots, f(w_r,v)\}$.

4.5 Let $(A, \bar{A})$ be a minimum s–t cut such that $s \in A$. Let x and y be any two vertices in A. Consider the graph G' obtained by collapsing all vertices of $\bar{A}$ to a single vertex $v_{\bar{A}}$. The weight of any edge $(a, v_{\bar{A}})$ in G' is defined to be the sum of the weights of edges (a, b) where $b \in \bar{A}$. Clearly, any cut in G' defines a cut in G. Show that a minimum x–y cut in G' defines a minimum x–y cut in G.

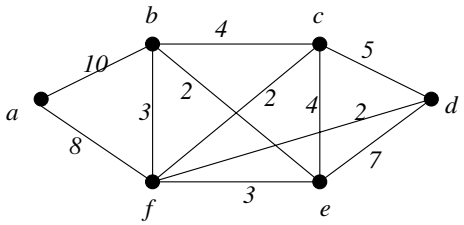
4.6 Now we are ready to state the Gomory–Hu algorithm. The algorithm maintains a partition of V, $(S_1, S_2, \ldots S_t)$, and a spanning tree T on the vertex set $\{S_1, \ldots, S_t\}$. Let w' be the function assigning weights to the edges of T. Tree T satisfies the following invariant.
Invariant: For any edge (S_i, S_j) in T there are vertices a and b in S_i and S_j respectively, such that $w'(S_i, S_j) = f(a, b)$, and the cut defined by edge (S_i, S_j) is a minimum a–b cut in G.

The algorithm starts with the trivial partition V, and proceeds in $n - 1$ iterations. In each iteration, it selects a set S_i in the partition such that $|S_i| \geq 2$ and refines the partition by splitting S_i, and finding a tree on the refined partition satisfying the invariant. This is accomplished as follows. Let x and y be two distinct vertices in S_i. Root the current tree T at S_i, and consider the subtrees rooted at the children of S_i. Each of these subtrees is collapsed into a single vertex, to obtain graph G' (besides these collapsed vertices, G' contains all vertices of S_i). A minimum x–y cut is found in G'. Let (A, B) be the partition of the vertices of G' defining this cut, with $x \in A$ and $y \in B$, and let w_{xy} be the weight of this cut. Compute $S_i^x = S \cap A$ and $S_i^y = S \cap B$, the two sets into which S_i splits.

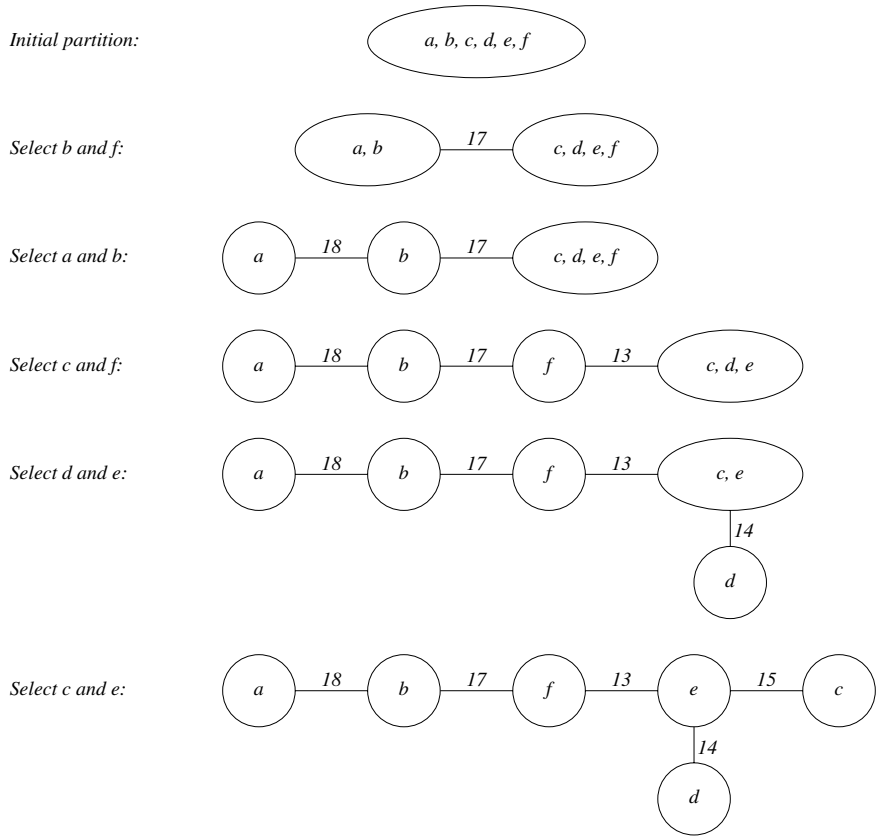
The algorithm updates the partition and the tree as follows. It refines the partition by replacing S_i with two sets S_i^x and S_i^y. The new tree has the edge (S_i^x, S_i^y), with weight w_{xy}. Consider a subtree T' that was incident at S_i in T. Assume w.l.o.g. that the node corresponding to T' lies in A. Then, T' is connected by an edge to S_i^x. The weight of this connecting edge is the same as the weight of the edge connecting T' to S_i. All edges in T' retain their weights.

Show that the new tree satisfies the invariant. Hence show that the algorithm terminates (when the partition consists of singleton vertices) with a Gomory–Hu tree for G.

Consider the graph:



The execution of the Gomory–Hu algorithm is demonstrated below:



4.7 Prove that if the Gomory–Hu tree for an edge-weighted undirected graph G contains all $n-1$ distinct weights, then G can have only one minimum weight cut.

4.4 Notes

Algorithm 4.3 is due to Dahlhaus, Johnson, Seymour, Papadimitriou and Yannakakis [57]. Algorithm 4.7 is due to Saran and Vazirani [233]; the proof given here is due to R. Ravi. For Gomory–Hu trees see Gomory and Hu [110].

5 *k*-Center

Consider the following application. Given a set of cities, with intercity distances specified, pick k cities for locating warehouses in so as to minimize the maximum distance of a city from its closest warehouse. We will study this problem, called the k-center problem, and its weighted version, under the restriction that the edge costs satisfy the triangle inequality. Without this restriction, the k-center problem cannot be approximated within factor $\alpha(n)$, for any computable function $\alpha(n)$, assuming $\mathbf{P} \neq \mathbf{NP}$ (see Exercise 5.1).

We will introduce the algorithmic technique of *parametric pruning* for solving this problem. In Chapter 17 we will use this technique in a linear programming setting.

Problem 5.1 (Metric k-center) Let $G = (V, E)$ be a complete undirected graph with edge costs satisfying the triangle inequality, and k be a positive integer. For any set $S \subseteq V$ and vertex $v \in V$, define connect(v, S) to be the cost of the cheapest edge from v to a vertex in S. The problem is to find a set $S \subseteq V$, with $|S| = k$, so as to minimize $\max_v\{\text{connect}(v, S)\}$.

5.1 Parametric pruning applied to metric *k*-center

If we know the cost of an optimal solution, we may be able to prune away irrelevant parts of the input and thereby simplify the search for a good solution. However, as stated in Chapter 1, computing the cost of an optimal solution is precisely the difficult core of **NP**-hard **NP**-optimization problems. The technique of parametric pruning gets around this difficulty as follows. A parameter t is chosen, which can be viewed as a "guess" on the cost of an optimal solution. For each value of t, the given instance I is pruned by removing parts that will not be used in any solution of cost $> t$. Denote the pruned instance by $I(t)$. The algorithm consists of two steps. In the first step, the family of instances $I(t)$ is used for computing a lower bound on OPT, say t^*. In the second step, a solution is found in instance $I(\alpha \cdot t^*)$, for a suitable choice of α.

A restatement of the k-center problem shows how parametric pruning applies naturally to it. Sort the edges of G in nondecreasing order of cost, i.e., $\text{cost}(e_1) \leq \text{cost}(e_2) \leq \ldots \leq \text{cost}(e_m)$, and let $G_i = (V, E_i)$, where $E_i =$

$\{e_1, e_2, \ldots, e_i\}$. A *dominating set* in an undirected graph $H = (U, F)$ is a subset $S \subseteq U$ such that every vertex in $U - S$ is adjacent to a vertex in S. Let $\mathrm{dom}(H)$ denote the size of a minimum cardinality dominating set in H. Computing $\mathrm{dom}(H)$ is **NP**-hard. The k-center problem is equivalent to finding the smallest index i such that G_i has a dominating set of size at most k, i.e., G_i contains k stars spanning all vertices, where a *star* is the graph $K_{1,p}$, with $p \geq 1$. If i^* is the smallest such index, then $\mathrm{cost}(e_{i^*})$ is the cost of an optimal k-center. We will denoted this by OPT. We will work with the family of graphs $G_1, \ldots, G_m$.

Define the *square of graph* H to be the graph containing an edge (u, v) whenever H has a path of length at most two between u and v, $u \neq v$. We will denote it by H^2. The following structural result gives a method for lower bounding OPT.

Lemma 5.2 *Given a graph H, let I be an independent set in H^2. Then, $|I| \leq \mathrm{dom}(H)$.*

Proof: Let D be a minimum dominating set in H. Then, H contains $|D|$ stars spanning all vertices. Since each of these stars will be a clique in H^2, H^2 contains $|D|$ cliques spanning all vertices. Clearly, I can pick at most one vertex from each clique, and the lemma follows. □

The k-center algorithm is:

Algorithm 5.3 (Metric k-center)

1. Construct $G_1^2, G_2^2, \ldots, G_m^2$.
2. Compute a maximal independent set, M_i, in each graph G_i^2.
3. Find the smallest index i such that $|M_i| \leq k$, say j.
4. Return M_j.

The lower bound on which this algorithm is based is:

Lemma 5.4 *For j as defined in the algorithm,* $\mathrm{cost}(e_j) \leq \mathrm{OPT}$.

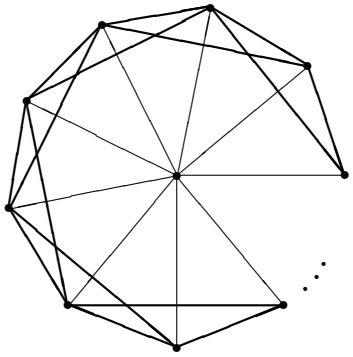
Proof: For every $i < j$ we have that $|M_i| > k$. Now, by Lemma 5.2, $\mathrm{dom}(G_i) > k$, and so $i^* > i$. Hence, $j \leq i^*$. □

Theorem 5.5 *Algorithm 5.3 achieves an approximation factor of 2 for the metric k-center problem.*

Proof: The key observation is that a maximal independent set, I, in a graph is also a dominating set (for, if some vertex v is not dominated by I, then $I \cup \{v\}$ must also be an independent set, contradicting I's maximality). Thus,

there exist stars in G_j^2, centered on the vertices of M_j, covering all vertices. By the triangle inequality, each edge used in constructing these stars has cost at most $2 \cdot \text{cost}(e_j)$. The theorem follows from Lemma 5.4. □

Example 5.6 A tight example for the previous algorithm is given by a wheel graph on $n+1$ vertices, where all edges incident to the center vertex have cost 1, and the rest of the edges have cost 2:



(Here, thin edges have cost 1 and thick edges have cost 2; not all edges of cost 2 are shown.)

For $k = 1$, the optimal solution is the center of the wheel, and $\text{OPT} = 1$. The algorithm will compute index $j = n$. Now, G_n^2 is a clique and, if a peripheral vertex is chosen as the maximal independent set, then the cost of the solution found is 2. □

Next, we will show that 2 is essentially the best approximation factor achievable for the metric k-center problem.

Theorem 5.7 *Assuming* $\mathbf{P} \neq \mathbf{NP}$*, there is no polynomial time algorithm achieving a factor of* $2 - \varepsilon$*,* $\varepsilon > 0$*, for the metric* k*-center problem.*

Proof: We will show that such an algorithm can solve the dominating set problem in polynomial time. The idea is similar to that of Theorem 3.6 and involves giving a reduction from the dominating set problem to metric k-center. Let $G = (V, E)$, k be an instance of the dominating set problem. Construct a complete graph $G' = (V, E')$ with edge costs given by

$$\text{cost}(u, v) = \begin{cases} 1, \text{ if } (u, v) \in E, \\ 2, \text{ if } (u, v) \notin E. \end{cases}$$

Clearly, G' satisfies the triangle inequality. This reduction satisfies the conditions:

- if $\text{dom}(G) \leq k$, then G' has a k-center of cost 1, and

- if $\text{dom}(G) > k$, then the optimum cost of a k-center in G' is 2.

In the first case, when run on G', the $(2-\varepsilon)$-approximation algorithm must give a solution of cost 1, since it cannot use an edge of cost 2. Hence, using this algorithm, we can distinguish between the two possibilities, thus solving the dominating set problem. □

5.2 The weighted version

We will use the technique of parametric pruning to obtain a factor 3 approximation algorithm for the following generalization of the metric k-center problem.

Problem 5.8 (Metric weighted k-center) In addition to a cost function on edges, we are given a weight function on vertices, $w : V \to R^+$, and a bound $W \in R^+$. The problem is to pick $S \subseteq V$ of total weight at most W, minimizing the same objective function as before, i.e.,

$$\max_{v \in V}\{\min_{u \in S}\{\text{cost}(u,v)\}\}.$$

Let $\text{wdom}(G)$ denote the weight of a minimum weight dominating set in G. Then, with respect to the graphs G_i defined above, we need to find the smallest index i such that $\text{wdom}(G_i) \leq W$. If i^* is this index, then the cost of the optimal solution is $\text{OPT} = \text{cost}(e_{i^*})$.

Given a vertex weighted graph H, let I be an independent set in H^2. For each $u \in I$, let $s(u)$ denote a lightest neighbor of u in H, where u is also considered a neighbor of itself. (Notice that the neighbor is picked in H and not in H^2.) Let $S = \{s(u) |\ u \in I\}$. The following fact, analogous to Lemma 5.2, will be used to derive a lower bound on OPT:

Lemma 5.9 $w(S) \leq \text{wdom}(H)$.

Proof: Let D be a minimum weight dominating set of H. Then there exists a set of disjoint stars in H, centered on the vertices of D and covering all the vertices. Since each of these stars becomes a clique in H^2, I can pick at most one vertex from each of them. Thus, each vertex in I has the center of the corresponding star available as a neighbor in H. Hence, $w(S) \leq w(D)$. □

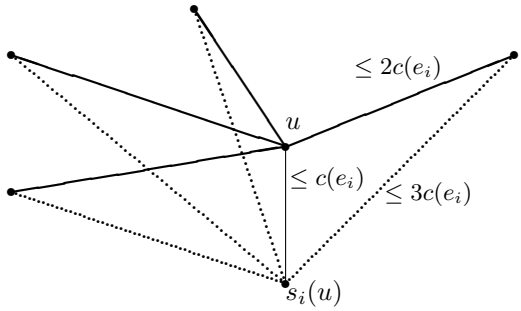
The algorithm is given below. In it, $s_i(u)$ will denote a lightest neighbor of u in G_i; for this definition, u will also be considered a neighbor of itself.

Algorithm 5.10 (Metric weighted k-center)

1. Construct $G_1^2, G_2^2, \ldots, G_m^2$.
2. Compute a maximal independent set, M_i, in each graph G_i^2.
3. Compute $S_i = \{s_i(u) |\ u \in M_i\}$.
4. Find the minimum index i such that $w(S_i) \leq W$, say j.
5. Return S_j.

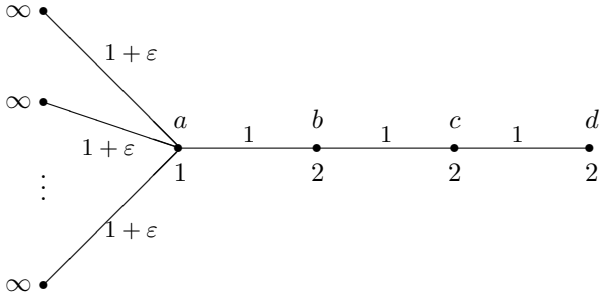
Theorem 5.11 *Algorithm 5.10 achieves an approximation factor of 3 for the weighted k-center problem.*

Proof: By Lemma 5.9, $\text{cost}(e_j)$ is a lower bound on OPT; the argument is identical to that in Lemma 5.4 and is omitted here. Since M_j is a dominating set in G_j^2, we can cover V with stars of G_j^2 centered in vertices of M_j. By the triangle inequality these stars use edges of cost at most $2 \cdot \text{cost}(e_j)$.



Each star center is adjacent to a vertex in S_j, using an edge of cost at most $\text{cost}(e_j)$. Move each of the centers to the adjacent vertex in S_j and redefine the stars. Again, by the triangle inequality, the largest edge cost used in constructing the final stars is at most $3 \cdot \text{cost}(e_j)$. □

Example 5.12 A tight example is provided by the following graph on $n+4$ vertices. Vertex weights and edge costs are as marked; all missing edges have a cost given by the shortest path.



It is not difficult to see that for $W = 3$ the optimum cost of a k-center is $1 + \varepsilon$: a k-center achieving this cost is $\{a, c\}$. For any $i < n + 3$, the set S_i computed by the algorithm will contain a vertex of infinite weight. Suppose that, for $i = n+3$, the algorithm chooses $M_{n+3} = \{b\}$ as a maximal independent set. Then $S_{n+3} = \{a\}$, and this is the output of the algorithm. The cost of this solution is 3. □

5.3 Exercises

5.1 Show that if the edge costs do not satisfy the triangle inequality, then the k-center problem cannot be approximated within factor $\alpha(n)$ for any computable function $\alpha(n)$.
Hint: Put together ideas from Theorems 3.6 and 5.7.

5.2 Consider Step 2 of Algorithm 5.3, in which a maximal independent set is found in G_i^2. Perhaps a more natural choice would have been to find a minimal dominating set. Modify Algorithm 5.3 so that M_i is picked to be a minimal dominating set in G_i^2. Show that this modified algorithm does not achieve an approximation guarantee of 2 for the k-center problem. What approximation factor can you establish for this algorithm?
Hint: With this modification, the lower bounding method does not work, since Lemma 5.2 does not hold if I is picked to be a minimal dominating set in H^2.

5.3 (Gonzalez [111]) Consider the following problem.

Problem 5.13 (Metric k-cluster) Let $G = (V, E)$ be a complete undirected graph with edge costs satisfying the triangle inequality, and let k be a positive integer. The problem is to partition V into sets $V_1, \ldots, V_k$ so as to minimize the costliest edge between two vertices in the same set, i.e., minimize

$$\max_{1 \le i \le k,\ u,v \in V_i} \text{cost}(u, v).$$

1. Give a factor 2 approximation algorithm for this problem, together with a tight example.
2. Show that this problem cannot be approximated within a factor of $2 - \varepsilon$, for any $\varepsilon > 0$, unless $\mathbf{P} = \mathbf{NP}$.

5.4 (Khuller, Pless, and Sussmann [169]) The *fault-tolerant* version of the metric k-center problem has an additional input, $\alpha \le k$, which specifies the

number of centers that each city should be connected to. The problem again is to pick k centers so that the length of the longest edge used is minimized.

A set $S \subseteq V$ in an undirected graph $H = (V, E)$ is an *α-dominating set* if each vertex $v \in V$ is adjacent to at least α vertices in S (assuming that a vertex is adjacent to itself). Let $\mathrm{dom}_\alpha(H)$ denote the size of a minimum cardinality α-dominating set in H.

1. Let I be an independent set in H^2. Show that $\alpha|I| \leq \mathrm{dom}_\alpha(H)$.
2. Give a factor 3 approximation algorithm for the fault-tolerant k-center problem.
   Hint: Compute a maximal independent set M_i in G_i^2, for $1 \leq i \leq m$. Find the smallest index i such that $|M_i| \leq \lfloor \frac{k}{\alpha} \rfloor$, and moreover, the degree of each vertex of M_i in G_i is $\geq \alpha - 1$.

5.5 (Khuller, Pless, and Sussmann [169]) Consider a modification of the problem of Exercise 5.4 in which vertices of S have no connectivity requirements and only vertices of $V - S$ have connectivity requirements. Each vertex of $V - S$ needs to be connected to α vertices in S. The object again is to pick S, $|S| = k$, so that the length of the longest edge used is minimized.

The algorithm for this problem works on each graph G_i. It starts with $S_i = \emptyset$. Vertex $v \in V - S_i$ is said to be *j-connected* if it is adjacent to j vertices in S_i, using edges of G_i^2. While there is a vertex $v \in V - S_i$ that is not k-connected, pick the vertex with minimum connectivity, and include it in S_i. Finally, find the minimum index i such that $|S_i| \leq k$, say l. Output S_l. Prove that this is a factor 2 approximation algorithm.

5.4 Notes

Both k-center algorithms presented in this chapter are due to Hochbaum and Shmoys [127], and Theorem 5.7 is due to Hsu and Nemhauser [132].