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. 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).

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. Surprisingly enough, the straightforward, simple greedy algorithm for the set cover problem is essentially the best one can hope for for this problem (see Chapter 29 for a formal proof of this statement).

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 set whose cost-effectiveness is smallest, say $S$.
     Let $\alpha = \frac{c(S)}{|S-C|}$, i.e., the cost-effectiveness of $S$.
     Pick $S$, and for each $e \in S - C$, set 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 $\text{OPT}/|\overline{C}|$, where $\overline{C} = U - C$. 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

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

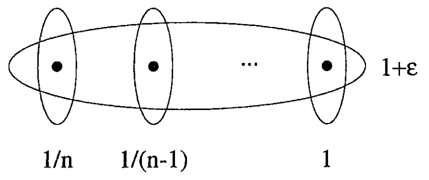
□

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 $w : V \rightarrow \mathbf{Q}^+$ 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 $w(V) \leq 2 \cdot \mathrm{OPT}$.*

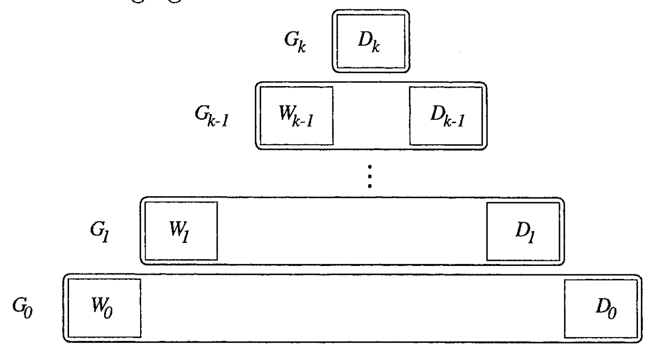
Proof: Let c be the constant such that $w(v) = c \cdot \deg(v)$, and let U be an optimal vertex cover in G. Since U covers all the edges,

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

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

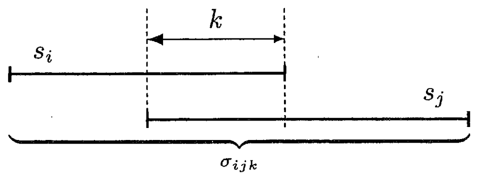
The following algorithm is given primarily to demonstrate the wide applicability of set cover. A constant factor approximation algorithm for shortest superstring will be given in Chapter 7.

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 $\pi \in S \cup M$. 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)

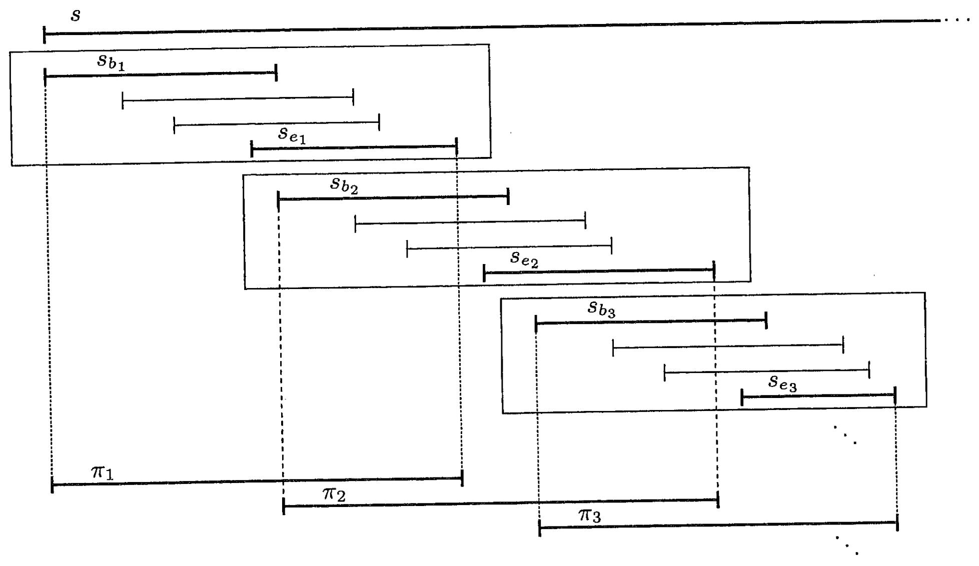
1. Use the greedy set cover algorithm to find a cover for the instance $\mathcal{S}$. Let $\text{set}(\pi_1), \ldots, \text{set}(\pi_k)$ be the sets picked by this cover.
2. Concatenate the strings $\pi_1, \ldots, \pi_k$, in any order.
3. Output the resulting string, say s.

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

Proof: Consider an optimal set cover, say $\{\text{set}(\pi_{i_j}) | 1 \leq j \leq l\}$, and obtain a string, say s, by concatenating the strings π_{i_j}, $1 \leq j \leq l$, in any order. Clearly, $|s| = \text{OPT}_{\mathcal{S}}$. Since each string of S is a substring of some π_{i_j}, $1 \leq j \leq l$, 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., $\text{cost}(\{(u \rightarrow v) \mid u \in S \text{ and } v \in \overline{S}\})$.

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, $\text{deg}_H(v) \geq (1/2)\text{deg}_G(v)$.

2.6 (Wigderson [265]) 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 [132]) 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 [157], Lovász [199], and Chvátal [50]. The hardness result for set cover, showing that this algorithm is essentially the best possible, is due to Feige [86], improving on the result of Lund and Yannakakis [206]. The application to shortest superstring is due to Li [194].