Chapter 2

Set cover and its application to shortest superstring

Understanding the area of approximation algorithms involves identifying cornerstone problems: problems whose study leads to discovering techniques that become general principles in the area, and problems that are general enough that other problems can be reduced to them. Problems such as matching, maximum flow, shortest path and minimum spanning tree are cornerstone problems in the design of exact algorithms. In approximation algorithms, the picture is less clear at present. Even so, problems such as minimum set cover and minimum Steiner tree can already be said to occupy this position.

In this chapter, we will first analyse a natural greedy algorithm for the minimum set cover problem. We will then show an unexpected use of set cover to solve the minimum superstring problem. An algorithm with a much better approximation guarantee will be presented in Chapter 7 for the latter problem; the point here is to illustrate the wide applicability of the set cover problem.

Minimum set cover

Problem 2.1 (Minimum set cover) Given a universe U of n elements and a collection of subsets of U, $S_1, \ldots S_k$, with non-negative costs specified, the minimum set cover problem asks for a minimum cost collection of sets whose union is U.

Perhaps the first algorithm that comes to mind for this problem is one based on the greedy strategy of iteratively picking the most cost-effective set and removing 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., $\frac{\text{cost}(S)}{|S \cap \overline{C}|.}$ 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 $\alpha = \frac{\text{cost}(S)}{|S \cap \overline{C}|}$, i.e., the cost-effectiveness of S.

   Pick S, and for each $e \in S - C$, $\text{price}(e) \leftarrow \alpha$.
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 $k \in \{1, \ldots, n\}$, $\text{price}(e_k) \leq \frac{\text{OPT}}{n-k+1}$.*

Proof : In any iteration, the left over sets of the optimal solution can cover the remaining elements at a cost of at most OPT. Therefore, there must be a set having cost-effectiveness at most $\frac{\text{OPT}}{|\overline{C}|}$. 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

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

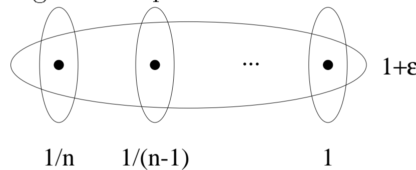
□

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 Following is a tight example:



The greedy algorithm outputs the cover consisting of the n singleton sets, since in each iteration some singleton is the most cost-effective set. So, 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 $1 + \epsilon$. □

Surprisingly enough, the obvious algorithm given above is essentially the best one can hope for for the minimum set cover problem: it is known that an approximation guarantee better than $\ln n$ is not possible, assuming $\mathbf{P} \neq \mathbf{NP}$.

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 Chapter 12 that the correct way to view the greedy set cover algorithm is in the setting of 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.

Exercise 2.6 The *maximum coverage problem* is the following: Given a universe U of n elements, with non-negative 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

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

Solving shortest superstring via set cover

Let us motivate the shortest superstring 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 sequenced. Of course, the locations of these segments on the original DNA are not known. It is hypothesised that the shortest string which contains these segments as substrings is a good approximation to the original DNA string.

Problem 2.7 (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 $2 \cdot H_n$ factor approximation algorithm by a reduction to the minimum set cover problem. 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 I contain the strings σ_{ijk} for all valid choices of i, j, k. The set cover instance $\mathcal{S}$ consists of S as the universal set. The specified subsets of S are: for each string $\pi \in S \cup I$, define $\text{set}(\pi) = \{s |\; s \in S,\, s \text{ is a substring of } \pi\}$. The cost of this set is