CS4234: Optimisation Algorithms Lecture 3

MIN-SET-COVER

V1.0: Seth Gilbert, V1.1: Steven Halim *August 23, 2016*

Abstract

Here we consider the problem of MIN-SET-COVER. We begin with an example, and then discuss the classical greedy solution. Finally, we analyze the Greedy Algorithm and show that it is a $O(\log n)$-approximation algorithm. In the analysis, notice how we begin by giving a lower bound on OPT, showing that OPT can only do so well. We then relate this to the performance of the Greedy Algorithm to get the approximation ratio.

1 MIN-SET-COVER

The **MIN-SET-COVER** is a Combinatorial Optimization Problem (COP) that frequently shows up in real-world scenarios, typically when you have collections of needs (e.g., tasks, responsibilities, or capabilities) and collections of resources (e.g., employees or machines) and you need to find a minimal set of resources to satisfy your needs. In fact, MIN-SET-COVER generalizes MIN-VERTEX-COVER that we have seen in the earlier lectures. In vertex cover, each vertex (i.e., set) covers the adjacent edges (i.e., elements); in set cover, each set can cover an arbitrary set of elements.

Unfortunately, MIN-SET-COVER is NP-hard (i.e., NP-complete as a decision problem). See **Example 2.** below to see that MIN-VERTEX-COVER can be reduced to MIN-SET-COVER.

In fact, it is known that MIN-SET-COVER is hard to approximate [1]. There are no polynomial-time α-approximation algorithms *for any constant*, assuming $P \neq NP$.

1.1 Problem Definition

We first define the problem, and then give some examples that show how MIN-SET-COVER problem might appear in the real world.

Definition 1 *Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n elements. Let $S_1, S_2, \ldots, S_m$ be subsets of X, i.e., each $S_j \subseteq X$. Assume that every item in X appears in some set, i.e., $\bigcup_j S_j = X$. A **set cover** of X with S is a set $I \subseteq \{1, 2, \ldots, m\}$ such that $\bigcup_{j \in I} S_j = X$. The solution for **MIN-SET-COVER** problem is a set cover I of minimum size.*

That is, a minimum set cover is the smallest set of sets $\{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\}$ that covers X.

Example 1. Assume you have a set of software developers: Alice, Bob, Collin, and Dave. Each programmer knows at least one programming language. Alice knows C and C++. Bob knows C++ and Java. Collin knows C++, Ruby, and Python. Dave knows C and Java. Your job is to hire a team of programmers. You are given two requirements: (i) there has to be at least one person on the team who knows each language (i.e., C, C++, Java, Python, and Ruby), and (ii) your team should be as small as possible (maybe your team is running on a tight budget).

This is precisely a MIN-SET-COVER problem. The base elements X are the 5 different programming languages. Each programmer represents a set. Your job is to find the minimum number of programmers (i.e., the minimum number of sets) such that every language is covered.

See Figure 1 for an example of this problem, depicted here as a (directed) bipartite graph. You will notice that any set cover problem can be represented as a (directed) bipartite graph, with the sets represented on one side, the base elements represented on the other side, and edges only go from sets to base elements.

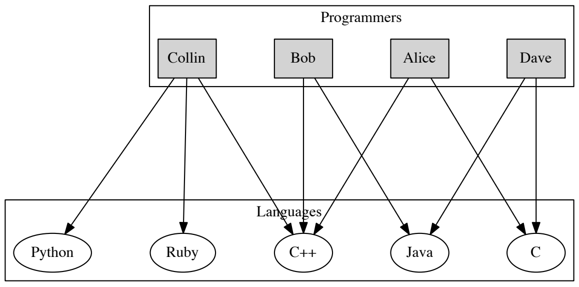
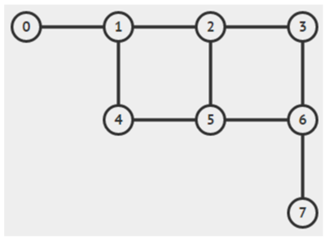


Figure 1: The problem of hiring software developers, represented as a bipartite graph. The goal is to choose a minimum sized set of programmers to ensure that every language is known by at least one of your programmers.

In this case, Alice, Bob, and Collin form a set cover of size 3. However, Collin and Dave form a set cover of size 2, which is optimal, i.e. the solution for this MIN-SET-COVER problem instance.

Example 2. Any vertex cover problem can be represented as a set cover problem, i.e. we can reduce MIN-VERTEX-COVER $\leq_p$ MIN-SET-COVER. Assume you are given a graph $G = (V, E)$ and you want to find a vertex cover. In this case, you can define $X = E$, i.e., the elements to be covered are the edges. Each vertex represents a set. We define the set $S_u = \{(u, v) : (u, v) \in E\}$, i.e., S_u is the set of edges adjacent to u. The problem of vertex cover is now to find a set of sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ such that every edge is covered. For example, see Figure 2.



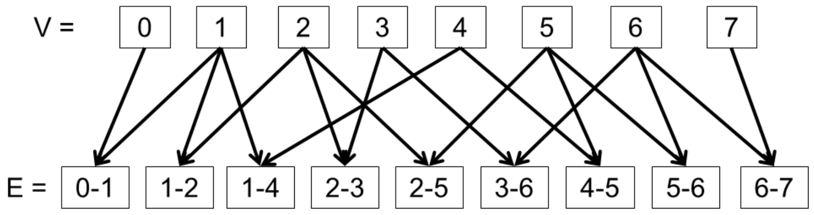


Figure 2: We can reduce an MIN-VERTEX-COVER instance into an MIN-SET-COVER instance in polynomial time.

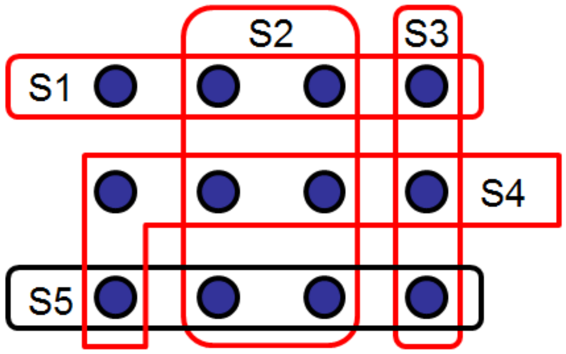
1.2 Greedy Algorithm

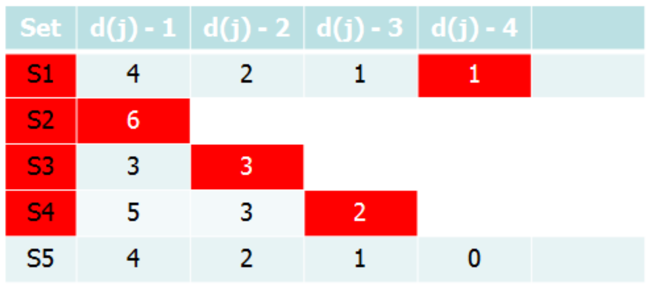
There is a simple Greedy Algorithm for tackling MIN-SET-COVER, albeit sub-optimally as we will analyze later:

```
/* This algorithm adds sets greedily, one at a time, until everything is
   covered.  At each step, the algorithm chooses the next set that will
   cover the most uncovered elements.                                   */
Algorithm: GreedySetCover($X, S_1, S_2, \ldots, S_m$)
Procedure:
    $I \leftarrow \emptyset$
    /* Repeat until every element in $X$ is covered:                    */
    while $X \neq \emptyset$ do
        Let $d(j) = |S_j \cap X|$ // This is the number of uncovered elements in $S_j$
        Let $j = \textbf{argmax}_{i \in \{1,2,\ldots,m\}} d(i)$ // Break ties by taking lower $i$
        $I \leftarrow I \cup \{j\}$ // Include set $S_j$ into the set cover
        $X \leftarrow X \setminus S_j$ // Remove elements in $S_j$ from $X$.
    return $I$
```

The algorithm proceeds greedily, adding one set at a time to the set cover until every element in X is covered by at least one set. In the first step, we add the set the covers the most elements. At each ensuing step, the algorithm chooses the set that covers the most elements that remain uncovered.

Let's look at Figure 3. Here we have 12 elements (represented by the dots) and 5 sets: $S1, S2, S3, S4, S5$. In the first iteration, we notice that set $S2$ covers the most elements, i.e., 6 elements, and hence it is added to the set cover. In the second iteration, set $S3$ and $S4$ both cover 3 new elements, and so we add set $S3$ to the set cover. In the third iteration, set $S4$ covers 2 new elements, and so we add it to the set cover. Finally, in the fourth step, set $S1$ covers one new element and so we add it to the set cover. Thus, we end up with a set cover consisting of $S1, S2, S3, S4$. Notice, though, that the optimal set cover consists of only three elements: $S1, S4, S5$.





Set	d(j) - 1	d(j) - 2	d(j) - 3	d(j) - 4	
S1	4	2	1	1	
S2	6				
S3	3	3			
S4	5	3	2		
S5	4	2	1	0	

Figure 3: On the left is an example of set cover consisting of twelve elements and five sets. On the right is a depiction of what happens when you execute the GreedySetCover algorithm on this example. Each column represents the number of new elements covered by each set at the beginning of the step.

1.3 Analysis

Our goal is to show that the Greedy Algorithm for MIN-SET-COVER shown above is a $O(\log n)$-approximation of optimal. As is typical, in order to show that it is a good approximation of optimal, we need some way to bound the optimal solution. Throughout this section, we will let OPT refer to the MIN-SET-COVER.

Intuition. To get some intuition, let's consider Figure 3 and see what we can say about the optimal solution. Notice that there are 12 elements that need to be covered, and none of the sets cover more than 6 elements. Clearly, then, any solution for this instance of MIN-SET-COVER problem requires at least $12/6 = 2$ sets. Now consider the situation after the first iteration, i.e., after adding set $S3$ to the set cover. At this point, there are 6 elements that remain to be covered, and none of the sets cover more than 3 elements. Any solution to the (smaller) MIN-SET-COVER problem requires at least $6/3 = 2$ sets.

In general, if at some point during the Greedy Algorithm, there are only k elements that remain uncovered and none of the sets covers more than t elements, then we can conclude that $OPT \geq k/t$. We will apply this intuition to show that the Greedy Algorithm is a good approximation of OPT.

Definitions. Assume we run the Greedy Algorithm for MIN-SET-COVER on elements X and sets $S_1, S_2, \dots, S_m$. When we run the algorithm, let us label the elements in the order that they are covered. For example, let's use Figure 3 again:

$$\underbrace{x_1, x_2, x_3, x_4, x_5, x_6}_{S_2}, \underbrace{x_7, x_8, x_9}_{S_3}, \underbrace{x_{10}, x_{11}}_{S_4}, \underbrace{x_{12}}_{S_1}$$

That is, x_1 is the first element covered, x_2 is the second element covered, etc. Under each element, we indicate *the first set that covered it*. In this example, notice that the first set chosen (S_2) covers 6 new elements, the second set chosen (S_3) covers 3 new elements, the third set chosen (S_4) covers 2 new elements (with element x_3, x_4, x_8 already covered by two other sets previously), etc. Each successive set covers *at most* the same number of elements as the previous one, *because the algorithm is greedy*: For example, if S_3 here had covered more new elements than S_2, then it would have been selected before S_2.

For each element x_j, let c_j be the number of elements covered at the same time. In the example, this would yield:

$$c_1 = 6, c_2 = 6, c_3 = 6, c_4 = 6, c_5 = 6, c_6 = 6, c_7 = 3, c_8 = 3, c_9 = 3, c_{10} = 2, c_{11} = 2, c_{12} = 1$$

Notice that $c_1 \geq c_2 \geq c_3 \geq \cdots$, because the algorithm is greedy.

We define $cost(x_j) = 1/c_j$. In this way, the cost of covering all the new elements for some set is exactly 1. In this example, the cost of covering $x_1, x_2, x_3, x_4, x_5, x_6$ is 1, the cost of covering x_7, x_8, x_9 is 1, etc. In general, if I is the set cover constructed by the Greedy Algorithm, then:

$$|I| = \sum_{j=1}^{n} cost(x_j) \ .$$

Key step. Let's consider the situation after elements $x_1, x_2, \dots, x_{j-1}$ have already been covered, and the elements $x_j, x_{j+1}, \dots, x_n$ remain to be covered. Let OPT be the optimal solution for covering all n elements.

What is the best that OPT can do to cover the element $x_j, x_{j+1}, \dots, x_n$? How many sets does OPT need to cover these remaining elements?

Notice that there remain $n - j + 1$ uncovered elements. However, no set covers more than $c(j)$ of the remaining elements. In particular, all the sets already selected by the Greedy Algorithm cover *zero* of the remaining elements. Of the sets not yet chosen by the Greedy Algorithm, the one that covers the most remaining elements covers $c(j)$ of those elements: Otherwise, the Greedy Algorithm would have chosen a different set.

Therefore, OPT needs at least $(n - j + 1)/c(j)$ sets to cover the remaining $(n - j + 1)$ elements. We thus conclude that:

$$OPT \geq \frac{n-j+1}{c(j)} \geq (n-j+1)cost(x_j)$$

Or to put it differently:

$$cost(x_j) \leq \frac{OPT}{(n-j+1)}$$

We can now show that the Greedy Algorithm provides a good approximation:

$$\begin{aligned}
|I| &= \sum_{j=1}^{n} cost(x_j) \\
&\leq \sum_{j=1}^{n} \frac{OPT}{(n-j+1)} \\
&\leq OPT \sum_{i=1}^{n} \frac{1}{i} \\
&\leq OPT(\ln n + O(1))
\end{aligned}$$

(Notice that the third inequality is simply a change of variable where $i = (n - j + 1)$, and the fourth inequality is because the Harmonic series $1/1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n$ can be bounded by $\ln n + O(1)$.)

We have therefore shown than the set cover constructed is at most $O(\log n)$ times optimal, i.e., the Greedy Algorithm is an $O(\log n)$-approximation algorithm:

Theorem 2 *The algorithm* GREEDYSETCOVER *is a* $O(\log n)$*-approximation*[1] *algorithm for* MIN-SET-COVER *problem.*

There are two main points to note about this proof. First, the key idea was to (repeatedly) bound OPT, so that we could relate the performance of the greedy algorithm to the performance of OPT. Second, the proof crucially depends on the fact that the algorithm is *greedy*. (Always try to understand how the structure of the algorithm, in this case the greedy nature of the algorithm, is used in the proof.) The fact that the algorithm is greedy leads directly to the bound on OPT by limiting the maximum number of new elements that any set could cover.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm.* MIT Press, 3rd edition, 2009.

[1]Approximation ratio of $O(\log n)$ means that the approximation ratio (slowly) grows as the instance size n increases. This may not be an ideal situation but we can use this Greedy Set Cover as a starting point for other advanced algorithms.

Index