IOE 691: Approximation & Online Algorithms

Lecture Notes: Max-Coverage and Set-Cover (Greedy)

Instructor: Viswanath Nagarajan   Scribe: Sentao Miao

1 Maximum Coverage

We consider a basic maximization problem, where the goal is to cover as many elements as possible in a set-system. For concreteness, one can think of covering client locations (elements) by opening facilities (picking sets).

Definition 1.1 *Given a universe $V = \{e_1, ..., e_n\}$ of elements, a list of sets $\{S_i \subseteq V\}_{i=1}^m$ (possibly overlapping) and a bound K, the goal is to find K sets $S'_1, ..., S'_K$ such that $|\bigcup_{i=1}^K S'_i|$ is maximized.*

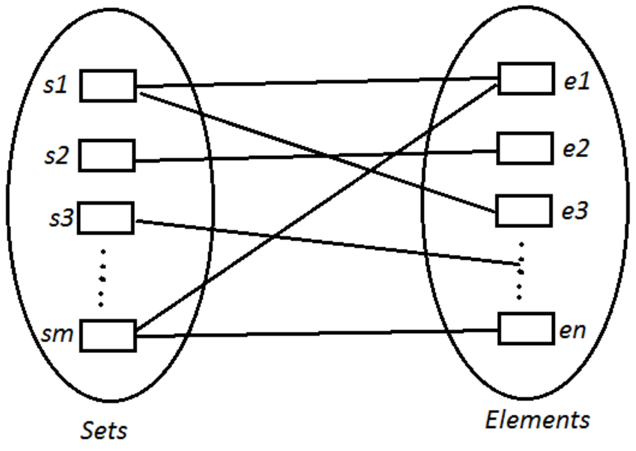

Figure 1: Maximum coverage problem

This is an **NP-hard problem**, and we are going to give a greedy algorithm which has approximation ratio $1 - 1/e$.

```
Data: $V$ : set of all elements; $S_1, ..., S_m$; $K$
Result: Approximate solution $A_1, \cdots, A_K$
$U = V$;
for $i = 1, ..., K$ do
    Let $A_i$ be one of the sets $S_1, ..., S_m$ which maximizes $|A_i \bigcap U|$;
    $U = U \backslash A_i$;
end
```

Algorithm 1: Greedy Algorithm for Maximum Coverage

The basic idea of the algorithm is to choose the set in each step which contains most of the uncovered elements. We have the following theorem.

Theorem 1.1 *Algorithm 1 is a* $(1-1/e)-$*approximation algorithm for maximum coverage.*

To prove this theorem, let us first prove the following two lemmas. Let C_i denote the set of elements covered by the end of iteration i in Algorithm 1.

Lemma 1.1 *For all* $i = 1, ..., K$, $|A_i \bigcap U| = |C_i| - |C_{i-1}| \geq \frac{OPT - |C_{i-1}|}{K}$

Proof: The number of elements covered in the optimal solution but not in the algorithm at the start of iteration i is $\geq OPT - |C_{i-1}|$. Let sets in the optimal solution be $S_1^*, ..., S_K^*$. Let $U = V \backslash C_{i-1}$.

Obviously, $\bigcup_{i=1}^K (S_i^* \bigcap U) = (\bigcup S_i^*) \backslash C_{i-1}$. This implies that

$$\sum_{i=1}^K |S_i^* \bigcap U| \geq |\bigcup_{i=1}^K (S_i^* \bigcap U)| \geq OPT - |C_{i-1}|$$

which further implies

$$\max_{i=1}^K |S_i^* \bigcap U| \geq \frac{OPT - |C_{i-1}|}{K}.$$

By definition, $|A_i \bigcap U| \geq \max_i |S_i^* \bigcap U|$ and we are done. ■

Lemma 1.2 $|C_i| \geq \frac{OPT}{K} \sum_{j=0}^{i-1} (1-1/K)^j$ *for all* $i = 1, ..., K$.

Proof: Prove by induction. The base case $i = 1$ is trivial as the first choice $A_1 = C_1$ has at least OPT/K elements by Lemma 1.1.

For inductive step, suppose i holds, and we want to prove that it holds for $i+1$.

$$\begin{aligned} |C_{i+1}| &\geq |C_i| + \frac{OPT - |C_i|}{K} \\ &= \frac{OPT}{K} + (1-1/K)|C_i| \\ &\geq \frac{OPT}{K} \sum_{j=0}^{i} (1-1/K)^j \end{aligned}$$

The first inequality is by Lemma 1.1, and the last inequality is from inductive hypothesis. ■

Now we are ready to prove Theorem 1.1.

Proof: [of Theorem 1.1]

$$\begin{aligned} |C_K| &\geq \frac{OPT}{K} \sum_{j=0}^{K-1} (1-1/K)^j \\ &= \frac{OPT}{K} \frac{1-(1-1/K)^K}{1-(1-1/K)} \\ &= OPT(1-(1-1/K)^K) \\ &\geq (1-1/e)OPT \end{aligned}$$

where the first inequality is from Lemma 1.2, and the last inequality uses $(1-1/K)^K \leq (e^{-1/K})^K$ which is obtained from $1 + x \leq e^x$ for all $x \in$ R. ■

2 Set Cover Problem

We now consider a variant of the above problem, where the goal is to cover *all* elements using the minimum number of sets.

Definition 2.1 *Given a set of elements $\{e_1, \dots e_n\}$ (called the universe) and a collection $\{S_i\}_{i=1}^m$ of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of sets whose union equals the universe.*

Notice that there is a close relationship between Set Cover and Maximum Coverage. The setting of two problems is the same, but the goals are different: in Maximum Coverage, the total number of sets is given and the goal is to cover as many elements as possible; in Set Cover the goal is to cover all elements with minimum number of sets.

A natural greedy algorithm for Set Cover problem is:

```
Data: A universe $\{e_1, \dots e_n\}$, a family $S = \{S_1, \dots S_m\}$.
/* $U$ is a set of uncovered elements.                                  */
$U = \{e_1, \dots e_n\}$;
while $U \neq \emptyset$, iteration $i = 1, 2, \dots l$ do
    pick $A_i = \arg\max_{j=1,\dots m} |S_j \cap U|$
    $U \leftarrow U \backslash A_i$
end
```

Algorithm 2: Greedy Algorithm for Set Cover Problem

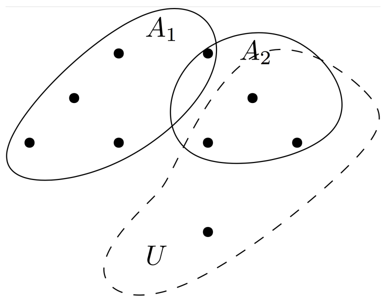

Figure 2: Diagram of first two steps of greedy algorithm for Set Cover problem.

We let l denote the number of iterations taken by the greedy algorithm.

It is clear that the first k iterations of the greedy algorithm for Set Cover are identical to that of Maximum Coverage (with bound k). In the Maximum Coverage problem, we already proved the following lemma 2.1. (Although we only had k iterations in max-coverage, the same analysis works for any number l of iterations.)

Lemma 2.1 *If C_i denotes the set of covered elements at the end of iteration i and C^* denotes the maximum coverage using k sets, then*

$$|C_i| \;\geq\; \frac{C^*}{k} \sum_{j=0}^{i-1} \left(1 - \frac{1}{k}\right)^j, \qquad \forall i = 1, \dots \ell$$

Theorem 2.1 *The greedy algorithm is $(1 + \ln(n))$-approximation for Set Cover problem.*

Proof: Suppose $k = OPT(\text{ set cover })$. Since set cover involves covering all elements, we know that the max-coverage with k sets is $C^* = n$. Our goal is to find the approximation ratio α so that $ALG(\text{ set cover }) = \ell \le \alpha k$. We apply Lemma 2.1 at the second last iteration, i.e. $i = \ell - 1$.

$$\begin{cases} |C_{\ell-1}| \le n-1 \\ |C_{\ell-1}| \ge \frac{n}{k}\sum_{j=0}^{l-2}(1-\frac{1}{k})^j = \frac{n}{k}\frac{1-(1-\frac{1}{k})^{\ell-1}}{\frac{1}{k}} = n(1-(1-\frac{1}{k})^{\ell-1}) \ge n(1-e^{-\frac{\ell-1}{k}}) \end{cases} \tag{1}$$

The first inequality is because the uncovered set must contain at least one element, otherwise the algorithm would have stopped before. The second inequality is from Lemma 2.1 and the fact that $1 + x \le e^x$ for any $x \in (-\infty, \infty)$.

From inequalities in (1), we have $ne^{-\frac{\ell-1}{k}} \ge 1$. We can take logarithm on both sides and find the approximation ratio $\alpha \le 1 + \ln n$ as claimed. ■

While $\alpha = 1 + \ln(n)$ is not a constant factor, it is still a reasonably good approximation ratio because it grows slowly with the input size n (refer to figure 3). Actually, one cannot get *any* better approximation algorithm for the Set Cover problem unless $P = NP$ [**?**].

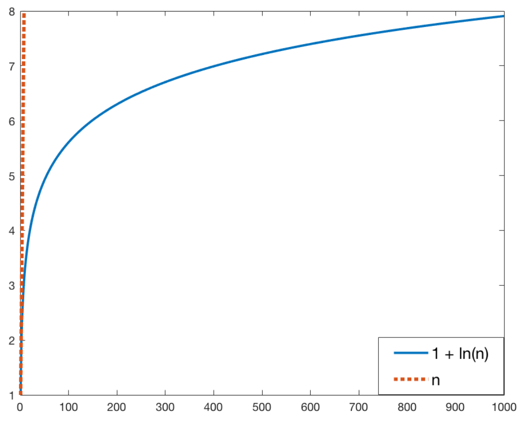

Figure 3: Plot of $y = 1 + log(n)$ solid line, $y = n$ dotted line, $n \in [1, 1000]$.

According to the form of approximation ratio α, we can categorize approximation algorithms within the region of NP-hard problems as in Figure 4. From (a) to (e) the algorithms become more and more loose in obtaining near-optimal solutions. These ratios in (c) to (e) are only examples and other functions of input size n are also possible for the approximation ratio depending on the setting of the problem.

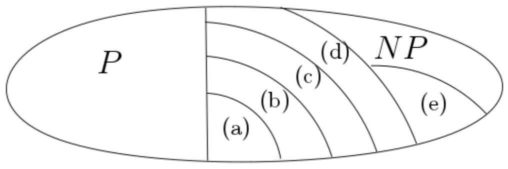

Figure 4: Taxonomy of NP problems (minimization) according to the form of α.

a. $\alpha = 1 + \epsilon$ with running time of $n^{1/\epsilon}, \epsilon > 0$. e.g. PTAS for the Knapsack problem.

b. α is constant factor, e.g. k-Center Problem, Maximum Coverage, TSP.

c. $\alpha = 1 + \log(n)$. e.g. Set Cover problem.

d. $\alpha = 1 + \log^2(n)$. e.g. group Steiner on trees.

e. α is linear in n. e.g. independent/stable set.

3 Overview of "Hardness" Results

Consider we have a problem A, it can be either a *decision problem* (if the output is yes or no) or an *optimization problem* (if the output is the maximum or minimum of the problem instance). However, we generally only consider the decision problem because decision and optimization problems are somehow "equivalent". More specifically, think about the max-coverage problem. The corresponding decision problem is given some target T, and needs to say whether/not there exists a solution whose objective (number of covered elements) is at least T. Whereas, the optimization problem of max-coverage asks to find the maximum objective value over all solutions. Obviously, if we know how to solve the optimization problem, we can immediately answer the decision problem. On the other hand, if we can answer any decision problem with any T, using a polynomial-time routine (e.g. binary search), we can find the optimal value of the optimization problem as well.

So consider a decision problem, how do we identify its hardness rigorously? There are two main classes of decision problems based on its hardness. The first class is called **P**. A problem is in **P** if given a decision instance (e.g. target T in max-coverage), we can answer whether there exists a solution satisfying this instance (e.g. some solution with objective value $\geq T$ in max-coverage) in *polynomial time* of the input size. The other class is called **NP**. Loosely speaking, a problem is in **NP** if given a decision instance *and a solution*, we can check whether this solution satisfies the instance in polynomial time.

Hence, we can obviously see that if A is in **P**, it is in **NP**. However, the other direction remains open till now and whether **P=NP** is one of the most famous problems in mathematics and computer science. Currently, it is widely assumed that **P≠NP** and this will be the underlying assumption for us as well.

Till now, we have "defined" two classes of problems with different hardness. Then the next question is, given a problem B how can we prove its hardness in a rigorous way? This is achieved from the so-called *reduction* of problems.

$$A \longrightarrow B$$

Definition 3.1 ***Reduction*** *from problem A (of input size n) to problem B is a map from instances I of A to instances J of B satisfying the following properties.*

1. *The reduction takes polynomial time in n.*

2. *I is a yes-instance if and only if J is a yes-instance.*

Then we have the important notion of **NP-hard**. That is, problem B is **NP-hard** if it can be reduced from all **NP** problems. Loosely speaking, if problem B is **NP-hard**, we believe that there is no polynomial time algorithm such that any decision instance can be checked (unless **P=NP**). Why? Because otherwise, from the definition of reduction and **NP-hard**, all **NP** problems can be solved in polynomial time, which shows **P=NP**, a statement which we assume to be wrong. Based on this argument, we immediately has the following observation.

Observation: Suppose we can reduce A to B. If A is **NP-hard**, B is also **NP-hard**.

Remark: The notion of **NP-hard** can be extended to approximate problems as well. That is, we can show for some problems that even computing an α approximation to the optimal value is **NP-hard** (for a suitable α).