‘

COMPSCI 632: Approximation Algorithms August 30, 2017

Lecture 2

Lecturer: Debmalya Panigrahi *Scribe: Nat Kell*

1 Introduction

In this lecture, we examine a variety of problems for which we give greedy approximation algorithms. We first examine the classic *set cover problem* and show that the greedy algorithm achieves a $O(\log n)$ approximation. We then examine the *maximum coverage* and *submodular maximization* problems, and give $(1-1/e)$-approximate algorithms for both these problems. Finally, we examine two scheduling problems, *precedence constraint scheduling* and *list scheduling*, and give 2-approximate algorithms for both problems.

2 The Set Cover Problem

The first problem we'll examine is the *set cover problem*, which is a generalization of the vertex cover problem we saw in the previous lecture. Set cover is defined as follows.

Definition 1. SET COVER: *As input, we are given a universe of n elements U and m subsets $s_1, \ldots, s_m$ where each $s_i \subseteq U$ has cost c_i. The objective is to find a collection of subsets X of minimum cost that cover all elements in the universe, i.e., $\cup_{s_i \in X} s_i = U$.*

Since our goal is to cover all the elements with as a little cost as possible, a natural approach is to try the following greedy algorithm: On each step of the algorithm, let u_i be the number of currently uncovered elements in set i. The algorithm selects the set that minimizes the ratio c_i/u_i. We keep selecting sets until all elements in the universe are covered.

We now show that this algorithm is a $O(\log n)$ approximation, which turns out to be the best approximation one can achieve (unless P = NP).

Theorem 1. *The greedy algorithm for set cover is an $O(\log n)$ approximation.*

Proof. Consider a step of the algorithm. Let k be the number of uncovered elements before choosing a set on this step. Among the sets that have not yet been picked by the algorithm, the optimal solution uses some collection of these sets to cover the remaining elements. Let T_{opt} be the set of indices of this collection.

Let OPT be the cost of the optimal solution. Clearly, we have that $\sum_{i \in T_{\text{opt}}} c_i \leq \text{OPT}$ since these sets are a subset of the total collection used by the optimal solution. Also observe that we have $\sum_{i \in T_{\text{opt}}} u_i \geq k$ since by definition these sets cover the remaining k elements. (Note that this sum is at least k because it could be double counting elements that belong to multiple sets in T_{opt}.)

Therefore, we have:

$$\min_{i \in T_{\text{opt}}} c_i/u_i \leq \frac{\sum_{i \in T_{\text{opt}}} c_i}{\sum_{i \in T_{\text{opt}}} u_i} \leq \frac{\text{OPT}}{k}, \tag{1}$$

where the first inequality follows from an averaging argument, and the second inequality follows from our above observations. Thus, on a step where there are k elements still uncovered, there must be a subset s_i where its benefit ratio c_i/u_i is at most OPT/k.

If on given step if the algorithm selects set s_i, we will think of dividing up the cost of the set and charging it to each newly covered element, i.e., each newly covered element gets a charge c_i/u_i. We now claim that the jth element to be covered can receive a charge of at most $\mathrm{OPT}/(n-j+1)$. This follows from Eqn. (1) and the fact that when the jth element is covered, there must have been at least $n-j+1$ uncovered elements on this step (also note that an element is only ever charged once—when it is first covered by the algorithm).

Clearly, the total cost of the the algorithm is the sum of the charges over all elements. Thus the total cost of the algorithm is:

$$\sum_{j=1}^{n} \frac{\mathrm{OPT}}{n-j+1} = \mathrm{OPT} \cdot \sum_{j=1}^{n} \frac{1}{j} = \mathrm{OPT} \cdot O(\log n),$$

where the last equality follows from the fact that the nth harmonic number is $O(\log n)$. □

3 Maximum Coverage and Submodular Maximization

In this section, we first examine the *maximum coverage problem*. Maximum coverage is similar to set cover, except now instead of trying to cover all the elements with the smallest cost possible, we are given a parameter k and are asked to cover as many elements as possible with only k sets. Formally, maximum coverage is defined as follows.

Definition 2. *As input, we are given a universe of n elements U and m subsets $s_1, \ldots, s_m$ where each $s_i \subseteq U$. The objective is to find a collection of subsets X such that $|X| = k$ and the number of elements we cover, i.e., $|\cup_{s_i \in X} s_i|$, is maximized.*

The algorithm we'll use for maximum coverage will also be similar to our greedy set cover algorithm. Namely, our algorithm will be to on each step, pick the set that covers the most uncovered elements (note that there is no notion of cost for this problem). We will show that this algorithm is $(1-1/e)$-approximate.

Theorem 2. *The greedy algorithm for maximum coverage is $(1-1/e)$-approximate.*

Proof. Let OPT be the number of elements covered in the optimal solution. Define ℓ_i to be the difference between OPT and the number of elements covered by the first i steps of the algorithm. Note that ℓ_0, this difference before the start of the algorithm, is exactly OPT.

Our first claim is that on the ith step of the algorithm, there exists at least one set that covers ℓ_i/k new elements. This follows from an argument similar to one used in Theorem 1: Let T be the set of currently uncovered elements in the algorithm's solution that are covered in the optimal solution. By definition, $|T|$ is at least ℓ_i. Thus, since the optimal solution covers the elements in T with at most k sets, there must be set that covers at least $|T|/k \leq \ell_i/k$ new elements.

Hence, we have that for any i, $\ell_i - \ell_{i+1} \geq \ell_i/k$. Solving this recurrence we obtain

$$\ell_k \leq (1-1/k)^k \cdot \ell_0 = (1-1/k)^k \cdot \mathrm{OPT} \leq (1/e) \cdot \mathrm{OPT}.$$

Therefore, the algorithm covers at least $(1-1/e)\cdot\mathrm{OPT}$ elements after selecting k sets. □

Maximum coverage is actually a special case of what is known as *submodular maximization*. To define this problem, we first give the definition of a submodular function.

Definition 3. *Suppose we are given a finite universe U. A function $f : 2^U \to \mathbb{R}$ is said to be submodular if for all elements $x \in U$ and all sets $A, B \in 2^U$ such $A \subseteq B$, we have:*

$$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B). \tag{2}$$

Intuitively, a submodular function is one with *diminishing returns*, i.e., if we have a collection of elements and we add a new element to this collection, the gain in f is smaller than if we were to add the new element to a subset of this collection. For example, note that the coverage function, which was the objective we used for set cover and maximum coverage, is submodular. Here, our universe is the collection of sets $s_1, \ldots, s_m$ and our function f maps to the size of the union of the collection of sets. Clearly, this function is submodular since when we add a set to the union of a collection of sets, the number of additional elements that we cover will be smaller than if we were to add the new set to a subset of the collection.

The problem of finding a set A such that a submodular function $f(A)$ is maximized is NP-hard, and the best known algorithm is a $1/2$ approximation. However, if we also assume f is *monotone*, i.e., $f(A \cup \{x\}) \geq f(A)$ for all sets A and all elements x, then we can achieve a $(1 - 1/e)$ approximation like we did for the maximum coverage problem. In fact this algorithm is a natural generalization of our greedy maximum coverage algorithm: on each step, we add the element that maximizes the current value of our set.

Theorem 3. *The greedy algorithm for monotone submodular maximization is a $(1 - 1/e)$-approximation.*

Proof. We will use an argument similar to that used in Theorem 2. Let S_i denote the set of elements chosen by algorithm after i steps of the algorithm, and let S^* be the set that maximizes f. Let ℓ_i be the difference between the value of these two sets, i.e., $\ell_i = f(S^*) - f(S_i)$. Our goal will be again to show that $\ell_i/k \leq \ell_i - \ell_{i+1}$. Once we establish this, the proof becomes identical to that of Theorem 2.

To show this, let K_i^* be the set of elements included in S^* but not in S after i steps. Observe that since f is submodular we have the following inequality:

$$\sum_{j=1}^{|K_i^*|} f(S_i \cup \{y_1, \ldots, y_j\}) - f(S_i \cup \{y_1, \ldots, y_{j-1}\}) \leq \sum_{j=1}^{|K_i^*|} f(S_i \cup \{y_j\}) - f(S_i), \tag{3}$$

where $\{y_1, \ldots, y_0\}$ is defined as the empty set. Observe that the LHS of Eqn. (3) telescopes and therefore equals $f(S^* \cup S_i) - f(S_i)$. Since f is also monotone, we have that $f(S_i \cup S^*) \geq f(S^*)$. Thus, Eqn. (3) implies that:

$$\begin{aligned}
\ell_i = f(S^*) - f(S_i) &\leq \sum_{j=1}^{|K_i^*|} f(S_i \cup \{y_j\}) - f(S_i) \\
&\leq |K_i^*| \max_{j \in K_i^*} \left\{ f(S_i \cup \{y_j\}) - f(S_i) \right\} \\
&\leq k(\ell_i - \ell_{i+1}),
\end{aligned} \tag{4}$$

where (4) follows from the fact that the algorithm chooses the element that increases the value of f by the most and $K_i^* \leq k$ (the optimal solution can pick at most k elements). Thus, we have that $\ell_i/k \leq \ell_i - \ell_{i+1}$, as desired. As we mentioned above, the rest of the proof is identical to that of Theorem 2.

□

4 Scheduling Problems

To end the lecture, we will examine two classic scheduling problems, the first of which is known *precedence constraint scheduling*. We define this problem as follows:

Definition 4. PRECEDENCE CONSTRAINT SCHEDULING: *As input, we are given n jobs and k machines, where each job is of unit length. We are also given a DAG G on the set of jobs, which dictates the order in which jobs can be schedule, i.e., all ancestors of a job j must be processed before j can be processed. The objective is to minimize the makespan of the schedule, i.e., the completion of the job that finishes last.*

Our greedy algorithm for this problem is straightforward: On the ith step of the algorithm, let R_i be the set of jobs that are ready to be scheduled (i.e., all jobs whose ancestors have already been processed). If the $|R_i| \leq k$, the algorithm simply schedules these jobs in parallel on an arbitrary set of $|R_i|$ machines. If $|R_i| > k$, then we arbitrarily pick a subset of k jobs from R_i and schedule them in parallel. We repeat this procedure until all jobs have been scheduled.

We now show that this algorithm is a 2 approximation.

Theorem 4. *The greedy algorithm for precedence constraint scheduling is a 2 approximation.*

Proof. We will use the following two lower bounds on the optimal solution. The first lower bound is that the optimal schedule must be at least the length of the longest path p^* in G (which is immediate from the definition of the problem). The second lower bound is that the optimal solution must take time at least n/k to finish, since at any time step it can schedule at most k jobs.

Consider a run of the algorithm. We call a time step *full* if the algorithm assigns jobs to all k machines; otherwise, we call it a *non-full* time step. First observe that by definition we can have at most n/k full time steps. Next, we claim that we can have at most $|p^*|$ non-full time steps. To see this second claim, note that on ith step of the algorithm, we can think of the algorithm as having some remaining graph G_i to process (i.e., the graph after removing all jobs that have been previously scheduled), where the jobs in R_i correspond to the source nodes of this remaining graph. Since the longest path in G_i must start at one of these source nodes, the longest path in G_{i+1} must be one less than that of G_i after a non-full step (since on non-full steps we schedule all available jobs). Clearly, the value of the longest path in the remaining graph cannot increase on full time steps; thus, there can be at most $|p^*|$ non-full time steps.

Since the total steps taken by the algorithm is the sum of full and non-full steps, and the number of each respective step types lower bounds the makespan of the optimal schedule (discussed above), it follows that the algorithm's solution is 2-approximate.

□

For our last problem, we consider the *list scheduling* problem. We define this problem as follows.

Definition 5. LIST SCHEDULING: *As input, we are given n jobs and k machines, where each job j has a specified processing time p_j. The objective is to assign each job to a unique machine such that the total processing time of the machine that finishes last (i.e., the makespan of the schedule) is minimized.*

Again, our greedy algorithm for this problem is straightforward. We will simply scan the set of jobs in an arbitrary order, and then for each job we schedule it on the machine that currently has the smallest total processing time. Again, we show that this algorithm is 2-approximate.

Theorem 5. *The greedy algorithm for list scheduling is a 2 approximation.*

Proof. We again use two lower bounds on the optimal makespan, which are similar to the two lower bounds used in Theorem 4. First, it must be the case that the optimal makespan is at least $\sum_j p_j/k$ since the sum of machine completion times must add up to the total processing time. Second, we have that for any job j, the optimal makespan must be at least p_j.

Consider the machine i that finishes last in the algorithm's schedule. Let j be the last job assigned to this machine. Observe that since we schedule each job on the machine that currently has the smallest load, all machines other than i must be busy when job j starts. This implies that the start time of j is no more than $\sum_j p_j/k$. Therefore, since i is the machine that finishes last, the makespan of the schedule is at most $\sum_j p_j/k + p_j$. As we argued above, both of these terms are lower bounds on the optimal makespan. Thus, the algorithm is a 2 approximation. □