Colorado CSCI 5454: Algorithms September 24, 2019

Lecture 9

Lecturer: Bo Waggoner *Scribe: Bo Waggoner*

Approximation Algorithms

In this lecture, instead of trying to solve the given problem optimally, we will output a solution that is, not exactly correct, but still provably good: approximately as good as the optimal possible output, as measured by some objective function. The idea is that such approximation algorithms can be much more efficient than optimal algorithms, especially for very hard problems. Let's see an example, then discuss approximation algorithms more generally.

1 Bipartite Matching

Recall that a *bipartite* graph $G = (U, V, E)$ is a graph whose vertices can be partitioned into two disjoint sets, U and V, such that every edge $e \in E$ has exactly one endpoint in U and exactly one endpoint in V.

Given edges $e = (u, v)$ and $e' = (w, x)$, let us say they *overlap* if they share at least one vertex. Formally, they overlap if $|\{u, v\} \cap \{w, x\}| \geq 1$.

A *matching* in a graph is a set of edges $S \subseteq E$ such that no pair $e, e' \in S$ overlap. In other words, any vertex can be an endpoint of at most one edge in the matching.

The *maximum bipartite matching problem* has as input an undirected, unweighted bipartite graph. It asks us to output a matching of largest size (most number of edges), let us call it S^*. This is the optimal solution.

We saw previously that this problem can be solved in polynomial time, for example using a reduction to max flow. However, the goal here will be to show that a very simple and fast algorithm can still do pretty well.

Greedy:

- Start with $S = \emptyset$ (an empty set).
- Repeat: Find any edge (u, v) and add it to S. Delete u and v from the graph.
- When there are no edges remaining, halt and return S.

By "delete from the graph", we mean remove u and v and also all edges where u is an endpoint or where v is an endpoint.

Efficiency. Note we can implement this in linear time as follows: Initially create an array `deleted` of length n (number of vertices), all set to False. Iterate through the edges of the graph using the adjacency list. For each edge (u, v), check if `deleted[u] == deleted[v] == False`. If so, add the edge to S and set `deleted[u] = deleted[v] = True`.

After iterating through all edges, every edge has either been added or deleted from the graph, so the algorithm halts and returns S.

Approximation guarantee. We prove a couple key lemmas (phrased slightly differently from in-class, but essentially the same).

Recall that S^* is a maximum matching. It is a set of edges, so its size is written $|S^*|$. We want to show that Greedy's output, S, has a large size, ideally some fraction of $|S^*|$.

Lemma 1. *By the end of the algorithm, every edge is removed from the graph.*

Proof. This follows immediately from the definition of the algorithm and the running time analysis. (Or, as phrased in class: If this were not true, the remaining edge (u, v) would have both vertices not yet deleted, so Greedy would be able to add it to S and continue. This is a contradiction.) $\square$

Lemma 2. *Each iteration, when Greedy adds an edge to S, at most two edges in S^* are removed from the graph.*

Proof. When Greedy adds (u, v), all edges with an endpoint equalling either u or v are removed. S^* is a matching, so it has at most one edge with endpoint u and at most one edge with endpoint v, which adds up to at most two. $\square$

Theorem 1. *The greedy algorithm satisfies $|S| \geq \frac{1}{2}|S^*|$, in other words, its matching always has at least half the optimal number of edges.*

Proof. By Lemma 2, each time $|S|$ increases by one, at most two edges of S^* are deleted. By Lemma 1, eventually all edges of S^* are deleted. So $|S^*| \leq 2|S|$, or in other words, $|S| \geq \frac{1}{2}|S^*|$. $\square$

2 Approximation Algorithms in General

In general, we will consider *optimization* problems, where an algorithm has to make some decisions or set some variables $\vec{x}$ subject to some constraints. In the above example, the algorithm had to choose a subset of edges S subject to the constraint that they form a matching.

The goal of the algorithm is some function $f(\vec{x})$ of the choices. In the above example, the function was the size of the matching.

This was an example of a *maximization* problem:

$OPT = \max_{\vec{x}} f(\vec{x})$ subject to the constraints.

Above, OPT was $|S^*|$, the size of the maximum possible matching.

Given an algorithm such as Greedy, we often write ALG for the objective function obtained by the algorithm. Above, ALG was $|S|$ where S was the matching output by greedy.

Definition 1. For a maximization problem, we say an algorithm has *approximation ratio α* if for every input, $ALG \geq \alpha \cdot OPT$.

Notice that $\alpha \leq 1$ since no algorithm can do better than OPT. In the above example, Greedy obtained an approximation ratio $\frac{1}{2}$. Note: this only applies to deterministic algorithms; we will modify definitions for randomized algorithms.

We also have *minimization* problems (we will see an example soon):

$OPT = \min_{\vec{x}} f(\vec{x})$ subject to the constraints.

Definition 2. For a minimization problem, we say an algorithm achieves a *C-approximation* if for every input, $ALG \leq C \cdot OPT$.

Notice that $C \geq 1$ since no algorithm can do better than OPT.

3 Max-Weighted Bipartite Matching

To see some of the power of approximation algorithms, let us put a twist on the bipartite matching problem.

In *max-weighted bipartite matching*, the input is a bipartite, undirected graph $G = (U, V, E)$ with edge weights w_{uv} for each edge $(u, v) \in E$. (Let us assume the weights are all at least zero, to make life simpler.) The output is a matching, and the goal is to maximize the *total weight* of the matching S, which we can write $f(S) = \sum_{(u,v)\in S} w_{uv}$.

This is a maximization problem. Let S^* be a max-weight matching, with objective value $f(S^*)$. The greedy algorithm is almost unchanged.

Greedy:

- Start with $S = \emptyset$ (an empty set).
- Iterate through edges (u, v) from largest weight to smallest:
- If (u, v) is still in the graph, add it to S. Delete u and v from the graph.
- Otherwise, skip this edge and continue.

Efficiency: The sorting step requires $|E| \log |E|$ time to sort the edges by weight, but after that, the rest of the algorithm can be implemented in linear time just as before. So the running time is bounded by $O(|E| \log |E| + |V|)$.

Approximation: The proof will look very similar. First, notice that Lemma 1 is still true in this setting: By the end of the algorithm, every edge is removed from the graph. Next:

Lemma 3. *Each iteration, when Greedy adds an edge of weight w to S, the total weight of edges in S^* that are removed from the graph is at most $2w$.*

Proof. Just as in Lemma 3, when Greedy adds an edge, at most two edges in S^* are deleted from the graph. But Greedy adds the edge remaining in the graph of largest weight, w. So the two deleted edges each have weight at most equal to w, so the total weight deleted is at most $2w$. □

Theorem 2. *The greedy algorithm for max-weighted bipartite matching guarantees a $\frac{1}{2}$ approximation ratio.*

Proof. By Lemma 3, each time $f(S)$ increases by some amount w, $f(S^*)$ decreases by at most $2w$. By Lemma 1, eventually $f(S^*)$ decreases to zero and $f(S)$ increases to its final value. So $f(S^*) \leq 2f(S)$.

One can be a bit more careful or formal as follows. Given any input, suppose Greedy runs for t rounds. Let $d(i)$ be the change in ALG in iteration i, i.e. the weight of the edge added, and let $d^*(i)$ be the change in OPT, i.e. the weight of OPT edges that are deleted in round i. Then

$$\begin{aligned} f(S^*) &= \sum_{i=1}^{t} d^*(i) \\ &\leq \sum_{i=1}^{t} 2d(i) \\ &= 2f(S). \end{aligned}$$

□

One reason this is cool is that the max-weighted bipartite matching problem is pretty difficult. It can be solved in polynomial time, but we can't use the same simple reduction to max flow. This is an example of what is called "the assignment problem" (because a matching is an assignment), and it requires fancy algorithms like the Hungarian algorithm.

Yet, by simply modifying Greedy slightly, we barely paid any additional runtime and still obtained the same approximation guarantee!

(We will also see that simple greedy algorithms work well in online algorithms settings.)

4 Load balancing

In this problem, we are given n computing tasks to execute on m identical machines in parallel. We must assign the tasks to the machines such that the final completion time is as quick as possible. This is a minimization problem whose objective is the total time to completion. The constraints are that each task must be scheduled on exactly one machine.

Formally, the input consists of processing times $t_1, \dots, t_n$; and an integer m, the number of machines. The output is an assignment of jobs to machines. Let $S[i]$ = the set of jobs on machine i.

Given the assignments $S[1], \dots, S[m]$, define the *load* to be $L[i] = \sum_{j \in S[i]} t_j$. Define the makespan to be $M := \max_{i=1,\dots,m} L[i]$.

The objective is to choose $S[1], \dots, S[m]$ to minimize the makespan.

Aside: exactly solving this problem is NP-hard even with $m = 2$. With two machines, notice that if the tasks can be divided such that both machines have equal processing time, then this is optimal. If we could solve this problem efficiently, then we could efficiently solve the Subset-Sum or Partition problems, which are NP-complete.

Now consider **Greedy:**

- Iterate through the tasks in any order.
- Assign each task to the machine whose load is currently the smallest.

(Quiz: argue that if $n \leq m$, Greedy achieves the optimal solution.)

Efficiency: This depends on the priority queue we use to keep track of the machine loads, and we will skip it (but you can tell it's quite efficient).

Approximation: Let M be the makespan of Greedy, and M^* the optimal makespan. Recall that we want to prove that Greedy is a C-approximation, meaning that $M \leq C \cdot M^*$, for some constant C, the smaller the better.

The key idea is to look at the last job that finishes, and the time when it started running. Let $\bar{i}$ be the machine running the longest in the Greedy algorithm, and let $\bar{j}$ be the last job to run on that machine. Let T be the time at which job $\bar{j}$ starts to run. This gives the following key fact:

$$M = L[\bar{i}] = T + t_{\bar{j}}.$$

Verbally, the makespan is the load of the longest-running machine, $\bar{i}$, which consists of T plus the time to run job $\bar{j}$.

Lemma 4. *$M^* \geq t_{\bar{j}}$.*

Proof. The optimal algorithm has to schedule job $\bar{j}$ on some machine, and can't split it across machines. So at least one machine runs for at least $t_{\bar{j}}$ time. □

Lemma 5. *$M^* \geq T$.*

Proof. We will first prove that the total time of all the jobs is at least $m \cdot T$.

The Greedy algorithm assigned task $\bar{j}$ to machine $\bar{i}$ when its load was T. So every other machine had load at least T, i.e. $L[i] \geq T$ for all i. Note that the total load always equals the total time of all the jobs. This gives:

$$\begin{aligned}\sum_{j=1}^{n} t_j &= \sum_{i=1}^{m} L[i] \\ &\geq \sum_{i=1}^{m} T \\ &= m \cdot T.\end{aligned}$$

Now, we will argue that the average load of OPT is at least T, therefore its makespan is at least T.

Let $L^*[i]$ be the load on machine i under the optimal solution OPT. Again, the total load equals the total time of all jobs, so we have

$$\begin{aligned}\sum_{i=1}^{m} L^*[i] &= \sum_{j=1}^{n} t_j \\ &\geq T \cdot m\end{aligned}$$

which implies the average load is at least T:

$$\frac{1}{m}\sum_{i=1}^{m} L^*[i] \geq T.$$

Therefore, there exists some i such that $L^*[i] \geq T$ (otherwise we would get a contradiction). So the maximum load satisfies $M^* = \max_i L^*[i] \geq T$. □

Theorem 3. *The Greedy algorithm guarantees a 2-approximation for the makespan problem.*

Proof. As observed, Greedy's makespan is $M = T + t_{\bar{j}}$.

By Lemma 5, $T \leq M^*$.

By Lemma 4, $t_{\bar{j}} \leq M^*$.

This proves $M \leq M^* + M^* = 2M^*$. □