Colorado CSCI 5454: Algorithms **Fall 2020**

Approximation Algorithms

Instructor: Bo Waggoner *Lecture 6*

Recall that in the study of algorithms, we have always asked about two things: **efficiency** and **correctness**.

We will now broaden our notion of correctness: 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 more efficient and/or simpler to implement than optimal algorithms, especially for very hard problems. We will start with an example of bipartite matching, then discuss approximation algorithms more generally. This lecture will focus on simple "greedy" algorithms, although we will see somewhat more sophisticated approximation algorithms elsewhere in the course.

Objectives:

- Know the definition of an optimization problem, in maximization and minimization forms.
- Know the definition of an approximation ratio α to a maximization problem.
- Know the definition of an approximation ratio C to a minimization problem.
- Gain practice applying greedy algorithms to achieve approximation ratios.

1 Bipartite Matching

Recall that a *bipartite* graph $G = (U_1, U_2, E)$ is a graph whose vertices can be partitioned into two disjoint sets, U_1 and U_2, such that every edge $e \in E$ has exactly one endpoint in U_1 and one in U_2.

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 $M \subseteq E$ such that no pair $e, e' \in M$ 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 M^*.

Exercise 1. Draw a bipartite graph with at least 3 edges where the maximum matching has size 2.

Exercise 2. In the same graph, give an example of a matching of size 0 and a matching of size 1 (recall the precise definition of a matching).

We saw in a previous lecture that the maximum bipartite matching 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 – producing a matching that is *approximately* maximum.

Specifically, consider the following **Greedy** algorithm:

- Start with $M = \emptyset$ (an empty set).
- Repeat: Find any edge (u, v) and add it to M. Delete u and v from the graph.

- When there are no edges remaining, halt and return M.

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 Greedy 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 M 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 M.

Approximation guarantee. As has been the case throughout class, we need some structural facts that enable our algorithms to be correct. So we first prove a couple key lemmas.

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

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

Proof. If this were not true, the remaining edge (u, v) would have both vertices not yet deleted. But then Greedy would add it to M before halting and delete it from the graph. This is a contradiction. $\square$

Lemma 2. *When Greedy adds an edge to M, at most two edges in M^* are removed from the graph.*

Proof. When Greedy adds (u, v), all edges incident to u and v are removed. M^* 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 removed edges. $\square$

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

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

Exercise 3. Give a graph where Greedy obtains exactly $\frac{1}{2}$ of the optimal matching size (you can specify in which order it processes the edges). *Hint: You only need* 4 *vertices and* 3 *edges total; take inspriation from the letter* Z*.*

Exercise 4. Give a graph where Greedy obtains the optimal matching exactly (it can even be the same graph, with a different edge ordering).

The above exercises show that Greedy can in general achieve anywhere from $\frac{1}{2}$ of optimal to fully optimal. In particular, Theorem 1 is tight: we can prove Greedy has approximation ratio at least $\frac{1}{2}$, but we cannot prove any better ratio, e.g. $\frac{3}{4}$, because Exercise 3 shows that's not true.

2 Approximation Algorithms in General

In general, we will consider *optimization* problems, where an algorithm has to make some decisions or set some variables S subject to some constraints. We can express the constraints, in general, by giving a set $\mathcal{F}$ of *feasible* solutions S that we are allowed to pick from.

The goal of the algorithm is described by some function $f(S)$ of the choices. We will consider two kinds of optimization problems: picking S to maximize f or to minimize f.

A **maximization problem** is an optimization problem of the form:

$$\max_{S \in \mathcal{F}} f(S).$$

This reads: "pick the solution S, out of the feasible solutions $\mathcal{F}$, that maximizes the function $f(S)$." We will write S^* for an optimal solution and $f(S*)$ for its value, i.e. $S^* = \arg\max_{S \in \mathcal{F}} f(S)$. Often, the optimal solution and/or value is informally referred to as OPT. Similarly, an algorithm, its solution, and/or its solution's value are informally referred to as ALG.

Definition 1. For a maximization problem, we say an algorithm outputting a solution S has **approximation ratio** α if for every input, $S \in \mathcal{F}$ and $f(S) \geq \alpha \cdot f(S^*)$.

In other words, the algorithm must always output a legal solution whose objective function is at least an α fraction of the optimal. Notice that $\alpha \leq 1$ because no algorithm can do better than OPT. Also, this definition only applies to deterministic algorithms; we will modify definitions for randomized algorithms.

Example. In the bipartite matching example, S was a set of edges. $\mathcal{F}$ was the set of all matchings of the graph, and $f(S) = |S|$, the size of the set S. So $\max_{S \in \mathcal{F}} f(S)$ meant "pick the largest set of edges that is a legal matching." Greedy obtained an approximation ratio $\alpha = \frac{1}{2}$.

Similarly, a **minimization problem** is an optimization problem of the form:

$$\min_{S \in \mathcal{F}} f(S).$$

Again, we write S^* for an optimal solution and informally refer to S^* or $f(S^*)$ as OPT.

Definition 2. For a minimization problem, we say an algorithm outputting S achieves a C**-approximation** if for every input, $f(S) \leq C \cdot f(S^*)$.

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

In order to remember these definitions, just remember that in a maximization problem, we want ALG to be as large as possible, *so we want to prove it is **bigger*** than something. Similarly, in a minimization problem, we want to prove ALG is **smaller** than something. With this memory aid, in mind:

Exercise 5. Without looking back at the notes, fill in the blank with $\geq$ or $\leq$ (hint: do we want to show ALG is big or small?): ALG guarantees an α approximation to a maximization problem if
ALG ___ $\alpha \cdot$ OPT.

Exercise 6. Without looking back at the notes, fill in the blank with $\geq$ or $\leq$ (hint: do we want to show ALG is big or small?): ALG guarantees a C approximation to a minimization problem if
ALG ___ $C \cdot$ 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 $G = (U_1, U_2, E)$ with edge weights $w_{uv} \geq 0$ for each edge $(u, v) \in E$. The output is a matching M, and the goal is to maximize the *total weight* of the matching, which we can write $f(M) = \sum_{(u,v) \in M} w_{uv}$.

This is a maximization problem. Let M^* be a max-weight matching, with objective value $f(M^*)$. (In this problem, we use $M^* =$ OPT and $M =$ ALG.) The greedy algorithm is almost unchanged:

Greedy:

- Start with $M = \emptyset$ (an empty set).
- Iterate through edges (u, v) from largest weight to smallest:
- If (u, v) is still in the graph, add it to M. 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 M, the total weight of edges in M^* that are removed from the graph is at most $2w$.*

Proof. Just as in Lemma 2, when Greedy adds an edge, at most two edges in M^* 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.*

The proof sketch is essentially the same as in the unweighted case: Each time $f(M)$ increases by some amount w, $f(M^*)$ decreases by at most $2w$, and it eventually decreases to zero. Here is a more formal proof:

Proof. Given an input, suppose Greedy runs for t rounds. Let $d(i)$ be the change in M in iteration i, i.e. the weight of the edge added, and let $d^*(i)$ be the change in M^*, i.e. the weight of M^* edges that are deleted in round i. By Lemma 1, eventually all edges in M^* are removed, so $f(M^*) = \sum_{i=1}^{t} d^*(i)$. By Lemma 3, $d^*(i) \leq 2d(i)$. And of course, $f(M) = \sum_{i=1}^{t} d(i)$. So:

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

This proves $f(M) \geq \frac{1}{2} f(M^*)$, as claimed. □

One reason this is exciting 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 as in the unweighted case. It 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 modifying Greedy slightly, we barely paid any additional runtime and still obtained the same approximation guarantee! Simple greedy algorithms also have other advantages; we will see they often work well in online algorithms settings where the entire input is not known in advance.

Exercise 7. Draw a small bipartite graph with edge weights e.g. $10, 9, 8, 7$. Simulate Greedy and compare to OPT.

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 of the jobs, $t_1, \ldots, t_n$; 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 $S[1], \ldots, S[m]$, define the **load on machine** i to be $L[i] = \sum_{j \in S[i]} t_j$. Define the **makespan** to be $M := \max_{i=1,\ldots,m} L[i]$.

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

Remark. 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.

The **Greedy** algorithm for makespan is:

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

Exercise 8. Simulate Greedy on the following instance: $m = 4$ (four machines), and $t_j = j$ for $j = 1, \ldots, 9$ (nine jobs with running times $1, 2, \ldots, 9$). What is the makespan achieved by Greedy?

Exercise 9. Argue that if $n \leq m$, then Greedy achieves the optimal solution.

The efficiency of the Greedy algorithm 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 factor: 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 i^* be the machine running the longest in the Greedy algorithm, and let j^* be the last job to run on that machine. Let T be the time at which job j^* starts to run. This gives the following key fact:

$$M = L[i^*] = T + t_{j^*}.$$

Verbally, the makespan M is the load of the longest-running machine i^*. M consists of running for time T, plus the time to run job j^*. Next, we compare the optimal makespan, M^*, to these two quantitites.

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

Proof. The optimal algorithm has to schedule job j^* on some machine, and can't split it across machines. So at least one machine runs for at least t_{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 j^* to machine 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 $M = T + t_{j^*}$.

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

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

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

Exercise 10. Can you prove that Theorem 3 is tight? That is, can you come up with an instance where Greedy's makespan is twice, or almost twice, that of the optimal algorithm? *Hint: Consider one very long job, say processing time = m; and many jobs of length 1 (how many?); have Greedy schedule the long job last.*

4.1 Improving Greedy by Sorting

We can improve the greedy algorithm by first sorting the list of jobs from longest to shortest processing time. Call this algorithm Greedy-Sort. We can show this algorithm performs better:

Theorem 4. *Greedy-Sort guarantees a $\frac{3}{2}$ approximation for the makespan problem.*

Proof. First, let us rename the jobs in order from longest to shortest, i.e. $t_1 \geq t_2 \geq \cdots \geq t_n$.

There are two cases. If $n \leq m$ (i.e. no more jobs than machines), then Greedy-Sort is optimal, as argued in Exercise 9.

On the other hand, if $n > m$, then in particular, $M^* \geq 2t_{m+1}$. This follows because of the first $m+1$ jobs, one of the m machines has at least two of them (Pigeonhole principle), and all of them take time at least t_{m+1}.

Again let i^* be the last machine to finish and j^* the last job on that machine; we must have $j^* \geq m+1$, so $t_{j^*} \leq t_{m+1} \leq \frac{1}{2}M^*$.

So now, we get $M = T + t_{j^*} \leq M^* + \frac{1}{2}M^* = \frac{3}{2}M^*$. □

In fact, Theorem 4 is *not* tight. Greedy-Sort guarantees a $\frac{4}{3}$ approximation factor (and this is tight), but we won't cover the proof here.