

Lecture 13: Approximation Algorithms

Suppose that you need to solve an NP-hard problem. What can you do? Ideally, we want an algorithm that:

- runs in polynomial time.
- can be used on any instance of the problem.
- finds the optimal solution.

For NP-hard problems, we must in practice relax at least one of these conditions. A commonly chosen tradeoff is to find an *approximation algorithm*. A ρ -approximation algorithm:

- runs in polynomial time.
- can be used on any instance of the problem.
- finds a solution within a ratio ρ of the optimum.

Job Scheduling Problem

Consider that we have m machines and n jobs each of which takes time t_i to process. Let A_j be the set of jobs assigned to machine j . Let $L_j = \sum_{i \in A_j} t_i$ be the *load* of machine j .

The *makespan* of an assignment is the maximum load on a machine (i.e. $\max_i L_i$). The goal of *load balancing* is to find an assignment of jobs to machines that minimizes the makespan.

A “greedy” approach is to iteratively assign each job to the machine with the smallest load.

Algorithm 1 Greedy

```
 $\forall j, A_j \leftarrow \emptyset$   
 $\forall j, L_j \leftarrow 0$   
for  $i = 1$  to  $n$  do  
   $j \leftarrow \operatorname{argmin}_k L_k$   
   $A_j = A_j \cup i$   
   $L_j = L_j + t_i$   
end for
```

The greedy algorithm can be implemented in $O(n \log n)$ time with a priority queue.

Theorem 1 (Graham, 1966) *Greedy scheduling is a 2-approximation to minimum makespan.*

This was the first worst-case analysis of an approximation algorithm. To prove this result, we need to be able to make comparisons to the *optimal solution* L^* .

Lemma 1 *The optimal makespan $L^* \geq \max_i t_i$.*

Proof: The most time consuming job must be assigned to some machine. ■

Lemma 2 *The optimal makespan $L^* \geq \frac{1}{m} \sum_i^n t_i$.*

Proof: The maximum load must be larger than the average load. ■

Lemma 3 *The solution of the greedy makespan algorithm is at most*

$$\frac{1}{m} \sum_i^n t_i + \max_i t_i$$

Proof: Consider machine j with maximum load L_j . Let i be the last job scheduled on machine j . When i was scheduled, j had the smallest load, so j must have had smaller than the average load. We have,

$$L_j = (L_j - t_j) + t_j \leq \frac{1}{m} \sum_i^n t_i + \max_i t_i .$$

Therefore the total load on machine j is less than the average load plus the largest job. ■

Combining these three lemmas implies Theorem 1. Is this analysis tight? The following example shows that it essentially is.

Example: Consider m machines, with $m(m-1)$ jobs of length 1 and one job of length m . The optimal solution is to assign the largest jobs to one machine, and m of the small jobs to each of the remaining $m-1$ machines, resulting in an optimal makespan of m . The greedy algorithm may assign the largest job last, at which point each machine has load $m-1$, making a makespan of $2m-1$.

The key to this counterexample seems to be the order in which we assign the jobs to processors. If we simply assigned the largest job first, the greedy algorithm would actually achieve the optimum value. Does this mean that we can get a better approximation guarantee if assign jobs in decreasing order of longest processing time (LPT)? It turns out we can.

Algorithm 2 Longest Processing Time (LPT)

 Sort jobs so that $t_1 \geq t_2 \geq \dots \geq t_n$.
 $\forall j, A_j \leftarrow \emptyset$ $\forall j, L_j \leftarrow 0$ **for** $i = 1$ to n **do** $j \leftarrow \operatorname{argmin}_k L_k$ $A_j = A_j \cup i$ $L_j = L_j + t_i$ **end for**

Lemma 4 *If there are at most m jobs, LPT scheduling is optimal.*

Proof: Put each job on its own machine. ■

Lemma 5 *If there are more than m jobs then $L^* \geq 2t_{m+1}$.*

Proof: By the pigeonhole principle, at least one processor must get 2 of the first $m + 1$ jobs. Each of these jobs is at least as big as t_{m+1} . ■

Theorem 2 *LPT scheduling is a $3/2$ approximation.*

Proof: Consider machine j assigned maximum load L_j in LPT scheduling (where $j > m$ since otherwise we are done). Let i be the last job assigned to j . By lemma 5 we have that $t_j \leq L^*/2$ for $j \geq m + 1$. Then by the same reasoning as the 2-approximation proof

$$L_j = (L_j - t_i) + t_i \leq L^* + \frac{1}{2}L^* = \frac{3}{2}L^*$$

■

By a more careful analysis, Graham was able to show that the $3/2$ analysis of LPT scheduling was not tight.

Theorem 3 *LPT scheduling is a $4/3$ -approximation.*

The k -center Problem

In the k -center problem, we are given as input a set of n sites s_1, \dots, s_n and a distance function d . We wish to place k centers such that the maximum distance from each site to its nearest center is minimized.

Let C be a set of centers. We make the following definitions.

- $d(x, y)$: distance from x to y
- $d(s_i, C) = \min_{x \in C} d(s_i, x)$: distance from s_i to the nearest center
- $r(C) = \max_i d(s_i, C)$: a covering radius r

The solution to the k -center problem is then

$$r^* = \min_{C:|C|=k} r(C)$$

Recall that a distance function satisfies the following properties:

$$\begin{aligned} d(x, x) &= 0 & \forall x & \quad (\text{identity}) \\ d(x, y) &= d(y, x) & \forall x, y & \quad (\text{symmetry}) \\ d(x, z) &\leq d(x, y) + d(y, z) & \forall x, y, z & \quad (\text{triangle inequality}) \end{aligned}$$

A naive greedy algorithm for this problem would be to place the first center in the best possible location, and iteratively add centers so as to minimize the radius at each iteration. Unfortunately, naive greedy is not a good algorithm.

This can be seen when there are two clusters of sites each with radius ϵ at a distance α from each other. The 2-center optimum is then ϵ , whereas naive greedy will have value $\alpha/2$, which is arbitrarily bad.

A different approach is based on the following observation: if we knew what the optimum radius $r^* = r(C^*)$ were, then every site must be within r^* of the nearest center. Furthermore, sites that share a common nearest center must be within distance $2r^*$ of each other. This suggests the following approximation algorithm.

Algorithm 3 k -center approximation given r^*

```

 $C \leftarrow \emptyset$ 
while  $S \neq \emptyset$  do
  pick an element  $s \in S$ 
   $S \leftarrow S - \{x \in S : d(x, s) \leq 2r^*\}$ 
   $C \leftarrow C \cup \{s\}$ 
end while
if  $|C| \leq k$  then
  return  $C$ 
else
  Report that  $k$ -center with radius  $r^*$  does not exist
end if

```

Claim 1 *At each step in the algorithm we are removing at least the sites in an r^* ball around some center.*

Proof: For any site v , it must be within r^* of some center j , so every site within r^* of j must be within $2r^*$ of v by the triangle inequality. Therefore, each $2r^*$ ball we remove must cover the r^* ball of at least one center. ■

This algorithm is very nice, but it has a problem: it uses the optimum solution r^* as an input! We can remedy this by using binary search. However, there is a more elegant solution. We first observe that in the previous algorithm we pick each new center guaranteeing that it is further than $2r^*$ from the centers picked thus far. But notice that if we pick the furthest such point we automatically satisfy this condition if that is possible. This leads to the following 2-approximation algorithm for the k -center problem.

Algorithm 4 k -center approximation (*)

```

 $C \leftarrow \{s\}$  (arbitrary point in  $S$ )
while  $|C| < k$  do
     $C \leftarrow C \cup \{\operatorname{argmax}_{x \in S} d(x, C)\}$ 
end while
return  $C$ 

```

Theorem 4 *Algorithm (*) is a 2-approximation algorithm for k center.*

Proof: Let C be the set of k -centers found by the algorithm and let C^* be the optimal set. Lets assume that $r(C) > 2r(C^*) = 2r^*$ and arrive at a contradiction.

Since $r(C) > 2r^*$ then there exists a point x (not a center) with $d(x, C) > 2r^*$. Consider a step of the algorithm at which we have a center set C' and have found the next center s . Then,

$$d(s, C') \geq d(x, C') \geq d(x, C) > 2r^* .$$

Thus at each of the k iterations it is possible to select a point further than $2r^*$ from the intermediate set of centers C' . And by our assumption, after these k iterations, we still have a point s further than $2r^*$ from C . This contradicts the fact that a set of k centers C^* of covering radius r^* exists. Hence $r(C) \leq 2r^*$. ■

Maximum Cut Problem

Given a graph $G(V, E)$, the cardinality maximum cut problem asks for a partition of V into S_1, S_2 such that the number of edges between S_1 and S_2 is maximized.

It is possible to show that the above problem is NP-hard and we will give a factor $1/2$ approximation algorithm for this problem using the technique of *local search*. Given a partition of V into two sets, the basic step of the algorithm, called a *flip*, is to move a vertex from one

Algorithm 5 Approximation algorithm for Max-cut

```
while there exists a vertex  $v$  such that flipping  $v$  increases the size of the cut do  
    flip  $v$   
end while
```

side of the partition to the other. The following algorithm finds a *locally optimum* solution under the flip operation i.e. a solution that can not be improved by a single flip.

Claim 2 *Algorithm 5 terminates in polynomial number of steps and produces a cut with at least $|E|/2$ edges.*

Proof: Each step of the while loop improves the cut by at least one edge. This means that there can be at most $|E|$ steps.

When the algorithm terminates, more than half of the neighbors of each vertex $v \in V$ lie in the subset other than the subset in which v lies. Therefore, at the end of the algorithm, at least half of the edges incident to every vertex are running across the cut (S_1, S_2) . Summing over all vertices gives us the desired claim. ■

Corollary 1 *Algorithm 5 is a factor $1/2$ approximation algorithm for maximum cut problem.*