Design and Analysis of Algorithms
Massachusetts Institute of Technology 6.046J/18.410J
Debayan Gupta, Aleksander Madry, and Bruce Tidor Handout 8

Recitation 13: Approximation Algorithms

1 Approximation Aglorithms Review

Approximation algorithms are useful when we want approximate solutions to *NP-Hard optimization* problems using *polynomial* time algorithms.

Suppose $\mathcal{A}$ is an algorithm for an optimization problem P. We denote the output of $\mathcal{A}$ by $C_{\mathcal{A}}$ and the optimal solution by C^*. We say that algorithm $\mathcal{A}$ is a $\rho(n)$-approximation for P if

$$\rho(n) \geq \frac{C_{\mathcal{A}}}{C^*} \qquad \text{if } P \text{ is a minimization problem}$$
$$\rho(n) \geq \frac{C^*}{C_{\mathcal{A}}} \qquad \text{if } P \text{ is a maximization problem}$$

where n is the size of the input. Note that $\rho(n) > 1$.

Given a parameter ϵ, a **polynomial-time approximation scheme (PTAS)** generates an algorithm that runs polynomially in n such that our approximation ratio $\rho(n)$ is $1 + \epsilon$. Naturally, the polynomial runtime will get larger as we decrease ϵ, since we are asking for a better approximation.

A regular PTAS allows a running time that grows exponentially in $\frac{1}{\epsilon}$, such as $n^{O(\frac{1}{\epsilon})}$. A ***fully* polynomial-time approximation scheme (FPTAS)** is one that is polynomial in both n and $\frac{1}{\epsilon}$.

1.1 Partition

Problem Definition Given a sorted list of numbers $s_1 \geq s_2 \geq \ldots \geq s_n$, partition the indices $\{1, \ldots, n\}$ into two disjoint sets A, B such that the "cost"

$$\max\{\sum_{i \in A} s_i, \sum_{j \in B} s_j\}$$

is minimized. The decision version of this problem is NP-complete.

Example. For the numbers

$$12 \;\; 10 \;\; 9 \;\; 7 \;\; 4 \;\; 4 \;\; 4$$

it is optimal to take $A = \{1, 3, 5\}$ and $B = \{2, 4, 6, 7\}$, so that $\sum_{i \in A} s_i = 25$ and $\sum_{i \in B} s_i = 25$.

At a first glance, it may seem that Partition can be solved in a greedy fashion by simply iterating through the elements and adding to the partition with smaller value. The example above shows that this is not always correct; however, this line of thinking gets us very close to a PTAS approximation scheme for partition:

Partition Approximation Algorithm

1: $m = \lfloor 1/\epsilon \rfloor$
2: By brute force, find an optimal partition $\{1, \ldots, m\} = A' \sqcup B'$ for $s_1, \ldots, s_m$
3: $A = A'$
4: $B = B'$
5: **for** $i = m + 1$ **to** n **do do**
6: **if** $\sum_{j \in A} s_j \leq \sum_{j \in B} s_j$ **then**
7: $A = A \cup \{i\}$
8: **else**
9: $B = B \cup \{i\}$
10: **end if**
11: **end for**
12: **return** $\langle A, B \rangle$

Note that this algorithm performs the simple greedy strategy mentioned before, but only after spending some extra time to find an optimal partition for *larger* elements. Finding the optimal partition for larger elements mitigates the effect of overshooting the minimum cost of the larger partition during the greedy phase.

Also note that ϵ is a parameter for our algorithm, which we can tune to balance the tradeoff between runtime and accuracy. A naive brute force in line 2 gives us $\Theta(m2^m + n)$, but a more clever implementation gives us $\Theta(2^m + n) = \Theta(2^{\frac{1}{\epsilon}} + n)$.

Claim. The above algorithm gives a $(1 + \epsilon)$-approximation.

Proof. Without loss of generality, assume

$$\sum_{i \in A'} s_i \geq \sum_{j \in B'} s_j$$

Let

$$H = \frac{1}{2} \sum_{i=1}^{n} s_i$$

Notice that solving the partition problem amounts to finding a set $A \subseteq \{1, ..., n\}$ such that $\sum_{i \in A} s_i$ is as close to H as possible. Moreover, since $\sum_{i \in A} s_i + \sum_{j \in B} s_j = 2H$, we have

$$\max\{\sum_{i \in A} s_i, \sum_{j \in B} s_j\} = H + \frac{D}{2}$$

where

$$D = \left| \sum_{i \in A} s_i - \sum_{j \in B} s_j \right|$$

- Case 1:

$$\sum_{i \in A'} s_i > H$$

In that case, the condition on line 7 is always false, as $\{\sum_{i \in A'} s_i > \sum_{i \notin A'} s_i$. So $A = A'$ and $B = \{1, ..., n\} \setminus A'$. In fact, approximation aside, we claim that this must be the *optimal partition*. To see this, first note that $\{\sum_{i \in B} s_i < H < \sum_{i \in A'} s_i$. If there were a better partition A^*, B^* then taking $A'' = A^* \cap \{1, \ldots, m\}$ and $B'' = B^* \cap \{1, \ldots, m\}$ would give a partition of $\{1, \ldots, m\}$ in which

$$\max\{\sum_{i \in A''} s_i, \sum_{j \in B''} s_j\} < \max\{\sum_{i \in A^*} s_i, \sum_{j \in B^*} s_j\} < \sum_{i \in A'} s_i$$

contradicting the brute-force solution on line 2.

- Case 2:

$$\sum_{i \in A'} s_i \leq H$$

Note that, if A ever gets enlarged (i.e., if the condition on line 7 is ever true for a certain i) then we have that on adding $i - 1$ to B, the sum of elements in B became greater than A. This means that the difference D at no point can no become greater than s_{i-1}. And if A is never enlarged, then it must be the case that $\sum_{i \in B} s_i$ never exceeded $\sum_{i \in A} s_i$ until the very last iteration of lines 6-10, in which s_n is added to B (otherwise we would be in case 1). In that case we have $D \leq s_n$. In either case, we find that

$$D \leq s_{m+1}$$

and consequently

$$\max\{\sum_{i \in A} s_i, \sum_{j \in B} s_j\} = H + \frac{D}{2} \leq H + \frac{s_{m+1}}{2}$$

Thus,

$$\begin{aligned}
\frac{\text{cost of the algorithm's output}}{\text{optimal cost}} &\leq \frac{\text{cost of the algorithm's output}}{\text{H}} \\
&\leq \frac{H + \frac{s_{m+1}}{2}}{H} \\
&\leq 1 + \frac{\frac{1}{2}s_{m+1}}{H} \\
&\leq 1 + \frac{\frac{1}{2}s_{m+1}}{\frac{1}{2} \cdot \sum_{i=1}^{m+1} s_i} \\
&= 1 + \frac{s_{m+1}}{\sum_{i=1}^{m+1} s_i} \\
&\leq 1 + \frac{s_{m+1}}{(m+1)s_{m+1}} \\
&= 1 + \frac{1}{(m+1)} \\
&< 1 + \epsilon
\end{aligned}$$

1.2 Metric TSP

Problem Definition Given an undirected, weighted, complete graph $G = (V, E, c)$ where $c(u, v) \geq 0$ and $c(u, v) \leq c(u, z) + c(z, v)\ \forall\ u, v, z \in V$ (triangle inequality), find a Hamiltonian cycle (i.e. a cycle that visits $v \in V$ exactly once) of minimum weight on G.

For convenience, we will denote the cost of a subset of edges $S \subseteq E$ as $c(S) = \sum_{(u,v)\in S} c(u, v)$. The approximation algorithm for Metric TSP is as follows:

Metric TSP Approximation Algorithm

1: $T \leftarrow \text{MST}(G)$
2: $r \leftarrow$ arbitrary $v \in V$
3: $H \leftarrow [r]$
4: Perform depth-first traversal on T starting at r, and append vertices to H only when they are first visited
5: Append r to H to complete the Hamiltonian cycle
6: **return** H

Claim. The above algorithm gives a 2-approximation to Metric TSP.

Proof. Let H^* be the optimal Hamiltonian cycle on G. Removing an edge from H^* creates a spanning tree T' of G, and by definition, $c(T) \leq c(T') \leq c(H^*)$ since all edge weights are non-negative.

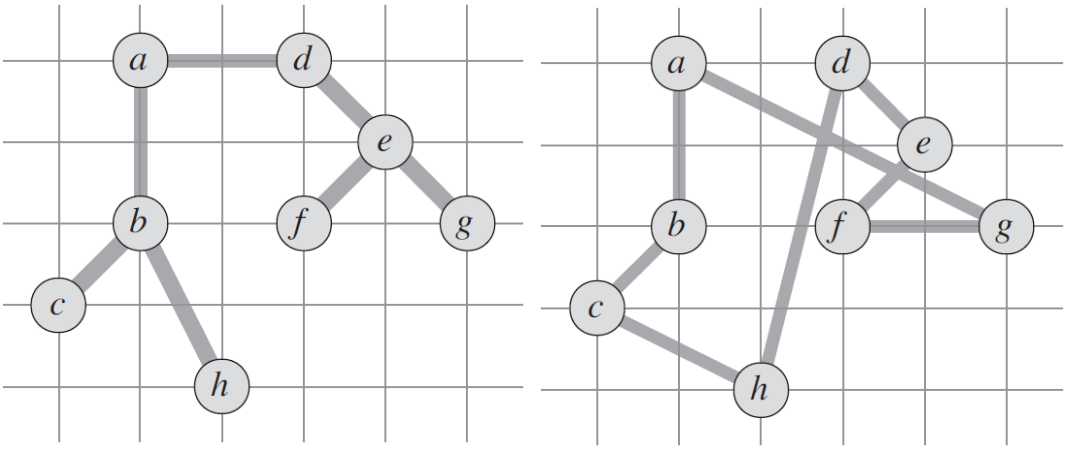

Figure 1: (adapted from CLRS Fig.35.2) The graph on the left is an an example MST, and the graph on the right is the resulting H from the traversal in line 4.

Now consider a "full walk" W of T where vertices are appended to W *every* time they are visited in a depth-first traversal of T. For example, a full walk on the graph in Fig. 1 gives us

$$W = [a, b, c, b, h, b, a, d, e, f, e, g, e, d, a]$$

Note that W visits every edge exactly twice, so $c(W) = 2c(T)$. Also note that if we replace subsequences of W that contain repeated vertices with a single edge, then we get the H returned by our approximation algorithm. To illustrate this, consider the following subsequence substitutions on our W above:

$$\begin{aligned} [c, b, h] &\to [c, h] \\ [h, b, a, d] &\to [h, d] \\ [f, e, g] &\to [f, g] \\ [g, e, d, a] &\to [g, a] \end{aligned}$$

Then our resulting list is exactly the walk depicted in Fig. 1.

By the triangle inequality, each one of these substitutions either reduces our cost or keeps it unchanged, so $c(H) \leq c(W)$. Therefore, $c(H) \leq c(W) = 2c(T) \leq 2c(H^*)$.

2 Fixed Parameter Algorithm Review

Finding the exact solution to some problems can be very computationally intensive. For instance, we have shown the existence NP-Complete problems, a set of problems for which we still know

no polynomial time solutions in the size of the input. Instead, we can take these difficult problems and isolate the exponential term to only a single parameter. Then, if the value of that parameter is small, the algorithm should run relatively quickly.

To be specific, we say that an algorithm is a **fixed-parameter algorithm** if its running time is $f(k) \cdot n^c$ where c is a constant independent of n and k. Note that these algorithms are also known as being **fixed-parameter tractable** or FPT. Notice that the running time may be exponential in k, but given a fixed k, it is only polynomial in n. Usually, the goal of these FPT algorithms is to minimize $f(k)$ and the constant c.

In this recitation, we will revise the problem of k-Vertex Cover using FPT algorithms as covered in lecture. In this problem, we are given a graph $G = (V, E)$ and would like to determine if there is a set of vertices S with fewer than k vertices that is adjacent to every edge in the graph. Note that we choose k to be the fixed parameter so we want our algorithm to run in polynomial time in V and E given a fixed value of k with the degree of the polynomial being independent of k.

2.1 Bounded Search-Tree Algorithm

The naive approach would be to just try all possible sets of k vertices, which total $\binom{V}{k} + \binom{V}{k-1} + ... + \binom{V}{0} = O(V^k)$. It also takes $O(E)$ time to check if all edges have been covered. Therefore, the running time of this approach is $O(E \cdot V^k)$. This does in polynomial time given a fixed value of k but the degree polynomial depends on k so the algorithm is not FPT.

Instead, we do the following bounded search-tree algorithm.

EXISTS-K-VERTEX-COVER$(G = (V, E), k)$

if $k == 0$ **then**
  return $E == \varnothing$
end if
Pick random edge (u, v) in G
Remove u and all its incident edges from G and call the new graph G'
Let $k' = k - 1$
Let U-VERTEX-COVER = EXISTS-K-VERTEX-COVER(G', k')
if U-VERTEX-COVER **then**
  return True
end if
Repeat the above process for v and compute V-VERTEX-COVER
return V-VERTEX-COVER

The above algorithm works because for any edge (u, v) in G, either u or v must be in S for S to be a vertex cover. The time taken to delete u or v is $O(V)$. We also only run EXISTS-K-VERTEX-COVER at most 2^k times. Hence, the total running time is $O(2^k \cdot V)$. This is FPT since the degree of the polynomial is independent of k.

2.2 Kernelization

We will improve the above runtime bound through a method called kernelization. A kernelization is a polynomial reduction from a problem with input (x, k) to the same problem with smaller input (x', k'). Specifically, by smaller we mean that that we can write $|x'| = O(f(k))$ for some function f of k. In lecture, we showed that a an algorithm is FPT if and only if there exists a kernelization for the problem.

We now try to kernelize the k-Vertex cover problem. We will do this by simplifying the graph. We first start by removing all multi-edges or duplicate edges that go between any pair of vertices. We also consider any vertex with a loop or an edge going into itself. Clearly, such a vertex must be in the vertex cover. Therefore, for all such vertices, we remove that vertex and all incident edges, add it to S, and reduce k by 1. Now, consider any vertex with degree greater than k. Such a vertex clearly must be in the vertex cover since otherwise it would take more than k vertices to cover all the edges adjacent to this vertex. Hence, we remove all such vertices and their incident edges and decrement k appropriately.

We are then left with a graph G' where all vertices have degree $\leq k$. For there to be a k-Vertex Cover, we must have $|E'| \leq k^2$ since otherwise k vertices cannot cover every edge. Finally, we remove all isolated vertices with no adjacent edges since they are not necessary in the vertex cover. Thus, we have that $|V'| \leq 2|E'| \leq 2k^2$. Hence, we have successfully kernelized the problem since the size of V' and E' after the reduction are $O(k^2)$.

The running time of the algorithm is then the time to perform the reduction added to the time taken to solve the reduced problem. We can perform the reduction in $O(V + E)$ time by maintaining a data structure that keeps track of the degree of the vertices as edges are removed. The time taken to solve the reduced is $O(V \cdot 2^k)$ using the Bounded Search-Tree Algorithm, which we know takes $O(k^2 \cdot 2^k)$. Therefore, the total running time is $O(V + E + k^2 \cdot 2^k)$.