Design and Analysis of Algorithms
Massachusetts Institute of Technology 6.046J/18.410J
Debayan Gupta, Aleksander Madry, and Bruce Tidor Handout 13

Recitation 13: Approximate and Distributed Algorithms

1 Approximation Algorithms 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

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

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 $\{1, \dots, m\} = A' \sqcup B'$ for $s_1, \dots, s_m$
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,

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

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: $T \leftarrow$ 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.

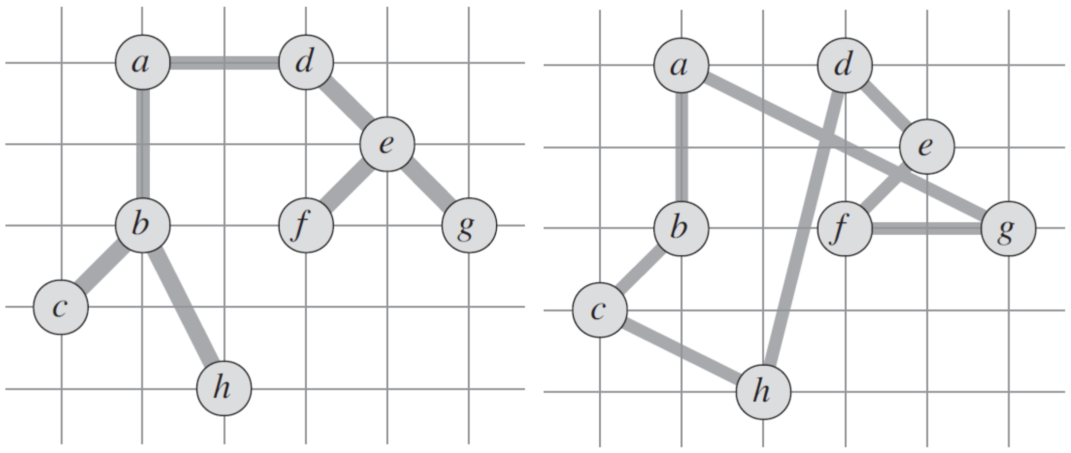

Figure 1: (adapted from CLRS Fig.35.2) The graph on the left is an an example MST, and the graph on the right is the resulting H from the traversal in line 4.

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 $c(H) \le c(W)$. Therefore, $c(H) \le c(W) = 2c(T) \le 2c(H^*)$.

2 Distributed Algorithms Review

Recall that in our distributed model, we have a network (graph) of n nodes where each node has its own processor and is connected to a subset of other nodes via two-way links (undirected edges) with which they may communicate. Nodes may only send/receive messages to/from neighboring nodes, but we assume there exists a global synchronization mechanism across the network such that communication happens in synchronized "rounds." Importantly, the nodes do not have full information about the graph; we assume node v knows only the number of vertices in the graph, n, and its set of neighbors $N(v)$.

Our goal is to develop a protocol (either randomized or deterministic) which will run on each node (without individual modification) and compute some quantity over the entire network. In lecture, we saw how to determine a single "leader" node and compute a maxim*al* independent set. Today, we'll demonstrate how to compute an MST in a distributed setting.

2.1 Distributed MST

In addition to the undirected edges, suppose we associate a "weight" with each edge, designating some quantity we'd like to minimize globally. For example, if the weights represents the average latency of the link, communicating over edges in the MST reduces the overall broadcast time from a given node to the rest of the graph. For simplicity, let's assume the edge weights are distinct.

The algorithm proceeds similarly to Melon Usk's MST algorithm from Quiz 1. At a high level, we build the MST in $\log n$ phases maintaining a set of connected components (constituting a spanning forest) which grow in size and shrink in number as the algorithm proceeds. Each component at phase k ($0 \leq k \leq \log n$) contains at least 2^k nodes, and in transitioning to phase $k+1$, we merge components via their minimum weight outgoing edge (hereon abbreviated "MWOE").

The key modification for the distributed setting is to designate a representative element ("leader") for each component and efficiently update the leader as we merge. We'll assume each node has a unique ID for simplicity.

Formally, the protocol is as follows:

1. Initialize each node as the leader of its own component.
2. For each component, find the MWOE to some other component. Specifically, the leader of the component broadcasts a request to all other nodes in its component to report the set of edges leaving the component. Since edge weights are unique, there will be a unique MWOE (u, v) for each component. Let v be the node in an adjacent component. After determining the MWOE, the leader of u's component sends a "request-to-merge" message to v's leader with u's leader's ID.

   If there are no outgoing edges (i.e., all nodes constitute a single component), the leader broadcasts a termination message to all other nodes.
3. Let C denote a subset of the components requesting to merge with each other. Merge all components in C along their MWOE's, determine a new leader, and broadcast the leader's ID to all nodes in C (see below for details). Do this for all C.

4. Repeat step 2-3 until termination (on step 2).

One way to select a new leader for a merged component in step 3 is to run the Leader Election algorithm from lecture. However, a more efficient approach is as follows: let us consider the components in $C = \{c_1, c_2, ...\}$ as vertices in a graph G_C connected via *directed* edges representing the MWOE's. That is, if c_i's MWOE connects to c_j, then $v_i \rightarrow v_j$ exists in G_C.

Lemma 1. A weakly connected[1] directed graph with exactly one outgoing edge per vertex has a unique cycle.

Proof. Suppose for the sake of contradiction that there exist no cycles in the graph. Then, traversing the graph from a given node (e.g., via DFS without pruning), we must terminate at some node (or set of nodes) after $\leq |V| - 1$ steps. The node(s) at which we terminate must have no outgoing edges, which contradicts that each vertex has exactly one outgoing edge. Therefore, we know the graph has at least 1 cycle.

Now, suppose for the sake of contradiction that there exists more than one cycle. Since each vertex has exactly one outgoing edge, the outgoing edges from vertices in a cycle must constitute exactly one cycle. This implies two things: (1) that all cycles are disjoint, and (2) if we collapsed the vertices of each disjoint cycle into a single "mega-vertex," there exist no outgoing edges from this mega-vertex. In order for the graph to be weakly connected, there must exist a path between each pair of these mega-vertices in the digraph's undirected counterpart. Let m be the number of vertices in this path. There must be $m - 1$ edges on this undirected path, and if neither mega-vertices at the endpoints contribute an outgoing edge to the set of $m-1$ edges, then one of the $m-2$ remaining vertices must have 2 outgoing edges by the pidgeonhole principle. This contradicts our construction that each vertex has *exactly* one outgoing edge.

□

Since G_C is weakly connected by construction, Lemma 1 implies a unique cycle in G_C. By definition, the MWOE's constituting this unique cycle must have nonincreasing weights. Since we assume weights are distinct, the unique cycle in G_C must be length 2 (where the two directed edges constituting the cycle represent the same underlying MWOE, shared by the adjacent components). In other words, there exists a unique pair of components in C which send "request-to-merge" messages to one another. We can simply designate the leader with larger ID in this pair as the leader of the entire merged component. All other nodes may simply wait for a broadcast from the endpoints of the MWOE crossing this pair.

[1] A "weakly connected" digraph is one where converting directed edges to undirected ones produces a connected graph.