Introduction to Approximation Algorithms[1]

- Welcome to a course on *approximation algorithms*. These are "efficient" algorithms which return a solution "close" to the desired solution, where close is deliberately left vague at this point. Why should one care? For many reasons. Most importantly, there are many problems for which finding the desired solution may be too hard. For instance, if we take an NP-hard problem[2] such as finding what is the longest path from a to b in a graph, then no one knows (and many don't expect) an efficient algorithm which runs in time at most some polynomial of the number of vertices and returns the correct solution on every graph. An approximate solution could be a path which may not be the longest but is at least half as long.

  Approximation also makes sense even when there are efficient algorithms known for a problem, but they are not efficient enough. For instance, a classic undergraduate algorithms problem is the *edit distance* problem : given two strings find the smallest number of inserts/deletes/swaps which takes the first string to the second. There is a dynamic programming algorithm whose running time is proportional to the product of the length of the two strings. But if the strings are, say, genome sequences, then each could have billions of characters, and it is infeasible to run such an algorithm. Perhaps one can approximate this number much faster? Perhaps even without looking at whole of these strings.

  In this course, most of the effort will be spent on designing approximation algorithms for NP-hard problems. A formal definition follows in the next bullet point. The main objective, however, is to explain an array of techniques that have been developed in the past four decades, and these techniques also find use in other notions of approximation.

- ***A bit of formality.*** Formally, an optimization problem Π consists of instances $\mathcal{I}$ and *feasible solutions*, $\mathcal{S}$, to these instances. Each solution $S \in \mathcal{S}$ is associated a cost $c(S)$. To work with an example, the optimization problem *minimum spanning tree (MST)* has instances described by undirected graphs and costs on the edges. Solutions are the spanning trees of the graph, that is acyclic subgraphs containing all the vertices, and the cost of each solution is the sum of costs of the edges in the tree. An optimization problem Π is called a *minimization* problem if one wishes to find minimum cost solutions for every instance; it is called a *maximization* problem, if one wishes to find maximum cost (in which case cost is often called profit) solution.

  An algorithm $\mathcal{A}$ for an optimization problem Π, maps every instance $\mathcal{I}$ to a feasible solution $S \in \mathcal{S}$. So an MST algorithm takes an undirected graphs with edge costs and returns a spanning tree. Algorithm $\mathcal{A}$ is called the **optimal** algorithm for a minimization (respectively, maximization) problem Π if for each instance $\mathcal{I}$, the algorithm returns the solution $S \in \mathcal{S}$ of the *minimum* (respectively, *maximum*) cost. As mentioned above, for many optimization problems which are NP-hard, we do not expect to find optimal algorithms. This motivates the following definition.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 4th Jan, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2] We assume the reader has familiarity with a basic undergraduate algorithms course, and also knows a bit of computational complexity. In particular, the reader understands what an NP-hard problem is.

Definition 1. An algorithm $\mathcal{A}$ for a minimization problem Π is an α-factor approximation algorithm for Π, for some $\alpha \geq 1$, if for every instance $\mathcal{I}$, the solution $S_{\mathcal{A}}$ returned by $\mathcal{A}$ satisfies

$$c(S_{\mathcal{A}}) \leq \alpha \cdot \min_{S \in \mathcal{S}} c(S)$$

An algorithm $\mathcal{A}$ for a maximization problem Π is an α-factor approximation algorithm for Π, for some $\alpha \leq 1$, if for every instance $\mathcal{I}$, the solution $S_{\mathcal{A}}$ returned by $\mathcal{A}$ satisfies

$$c(S_{\mathcal{A}}) \geq \alpha \cdot \max_{S \in \mathcal{S}} c(S)$$

It should be clear that for any problem Π, we would like to design *polynomial time* algorithms with α as close to 1 as possible. For NP-hard problems we don't expect to reach $\alpha = 1$. How close can we get?

- ***The Steiner Tree Problem.*** Let us begin with a simple cousin of the minimum spanning tree algorithm which is already NP-hard. As in the minimum spanning tree problem, the input is an undirected graph $G = (V, E)$ with costs on edges. Additionally, one is given a subset $R \subseteq V$ of *required vertices*. The objective is to find the minimum cost subtree of G which contains all the vertices of R. It may or may not contain all the vertices in $V \setminus R$, the so-called *Steiner vertices*. Such trees are called Steiner trees. For example, in Figure 1 the right-most tree is a spanning tree which is naturally a Steiner tree as well, while the middle tree is a Steiner tree since it doesn't contain the bottom-right Steiner vertex.

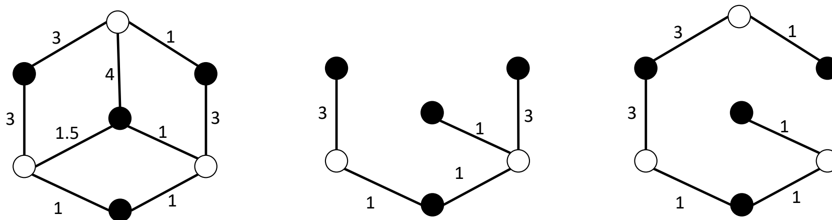

Figure 1: A graph and two Steiner trees. The dark, filled circles are required vertices.

It is perhaps surprising that although the minimum spanning tree has a simple exact algorithm, the Steiner tree problem is NP-hard. One therefore wishes to find approximation algorithms. An α-approximation algorithm would take any graph G and return a Steiner tree T with the guarantee that $\texttt{cost}(T) \leq \alpha \cdot \texttt{opt}(G)$ where $\texttt{opt}(G)$ is the cost of the minimum cost Steiner tree. Note that the algorithm would have no idea what $\texttt{opt}(G)$ is, indeed even finding that value is NP-hard, and yet one should be able to argue that $\texttt{cost}(T) \leq \alpha \cdot \texttt{opt}(G)$.

- ***A simple 2-approximation algorithm.*** Is there an algorithm that comes to mind? Here is one : since finding MST is easy, let's start by finding T' which is an MST of G. Starting with this tree T', while there is a leaf which is a Steiner vertex, keep deleting it till we end up with a Steiner tree T all of whose leaves are vertices in R. In the example of Figure 1, the algorithm would return the tree to the right. And in that example, we see that $\texttt{cost}(T) \leq \frac{10}{9} \cdot \texttt{opt}(G)$, where the middle tree is the cheapest

Steiner tree (figured by brute-force check). Does this mean the algorithm is a $\frac{10}{9}$-approximation? No! Because the approximation algorithm needs to return a tree within $\alpha \cdot \texttt{opt}(G)$ for *every* graph G. Indeed, the example shows that the factor can be no better than $\frac{10}{9}$, and in fact the factor is pretty bad.

Exercise: *For any constant α, describe a graph $G = (R \subseteq V, E)$ such that if T is the tree returned by the above "mst-and-prune" algorithm, then $\texttt{cost}(T) > \alpha\texttt{opt}(G)$.*

However, there is a simple 2-approximation algorithm using minimum spanning trees, except that it is run on a different graph. Given $G = (V, E)$, $R \subseteq V$, and $c(e)$ on edges, define a *complete* graph $H = (R, F)$ with costs $w(u, v)$ for *all* pairs $u, v \in R \times R$, where

$$w(u, v) = \text{shortest cost path from } u \text{ to } v \text{ in } G \text{ with costs } c$$

See Figure 2's first arrow to get the metric completion for the graph in Figure 1. Note that the metric completion can be obtained efficiently using, say, an all pairs shortest path algorithm.

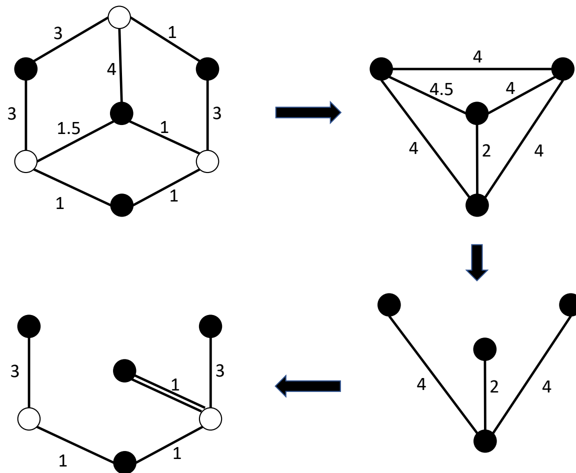

Figure 2: Illustration of metric completion and the algorithm.

The algorithm is as follows : compute the MST in H with weights w. Let this tree be $T' \subseteq H$. Start with an initial graph $T = (V, \emptyset)$. For every edge $e = (u, v) \in T'$ of weight $w(e)$, add the edges of the minimum cost path from u to v of cost $w(u, v)$ to T. Note that we may be taking multiple copies of the same edge. After processing all edges in T', we end up with a multi-graph T such that (a) T contains all the vertices in R, and (b) $\texttt{cost}(T) = w(T') = \mathsf{mst}(H)$. Now delete edges arbitrarily from T to make it a simple tree. This includes parallel copies and edges that form cycles, if any. In the example in Figure 2, the algorithm, in fac, recovers the optimum solution.

Exercise: *As the illustration shows, T can contain parallel edges. Can it contain cycles of length 3 or more? Why or why not?*

1: **procedure** MST-STEINER($G = (V, E)$, $R \subseteq V$, $c(e)$ on edges):
2: Obtain $(H, w) \leftarrow (G, c)$ via metric completion.
3: $T' \leftarrow \mathsf{mst}(H, w)$.
4: For every edge $(u, v) \in T'$, add path to T.
5: Prune T till it becomes a tree.
6: **return** T.

- ***Analysis.*** The above algorithm in fact is "not-too-bad" on any graph.

Theorem 1. MST-STEINER is a 2-approximation algorithm. That is, for any (G, c) with non-negative costs on edges, the algorithm returns a Steiner tree T with $\texttt{cost}(T) \leq 2\texttt{opt}(G)$.

Proof. We already know that $\texttt{cost}(T) \leq \mathsf{mst}(H, w)$. To prove the theorem, it suffices to argue that $\mathsf{mst}(H, w) \leq 2\texttt{opt}(G)$. To do this, consider the optimal Steiner tree T^*. The idea is to describe a tree W in H whose *weight* is at most $2\texttt{cost}(T^*)$.

Duplicate every edge in T^* to get a multi-subgraph T^{**} of G. Note that $\texttt{cost}(T^{**})$, the cost of all the edges, is precisely $2\texttt{opt}(G)$. Upon duplicating, we get that the degree of any vertex $v \in T^{**}$ is *even*. We are now going to use another graph theoretic fact. In any multi-graph, and in particular in T^{**}, where all vertices are even, there exists a *walk* starting at any vertex u and ending at the same vertex u which traverses every edge *exactly* once. Such a walk is called the Eulerian walk or Eulerian tour. Let τ be this tour.

Now we start τ at a required vertex u. Initialize $W \leftarrow \emptyset$. Any time we visit another required vertex $v \in R$ in τ, we add an edge (u, v) to W with cost $w(u, v)$. Note that this weight $w(u, v)$ is, by design, at most the *cost* of the edges in traversed in τ. Upon continuing thus throughout τ, we end with a subgraph W of H which is (a) spanning since all vertices of R are visited by τ since $R \subseteq T^{**}$, and (b) the total weight of W is at most $\texttt{cost}(\tau) = \texttt{cost}(T^{**}) = 2\texttt{opt}(G)$. Since the MST of H wrt w is only less than the weight of W, we get $\mathsf{mst}(H, w) \leq 2\texttt{opt}(G)$. QED. □

Exercise: ☕ *For any constant $\varepsilon > 0$, find a graph $G = (V, E)$ with costs c, such that* MST-STEINER *returns a tree T with $\texttt{cost}(T) > (2 - \varepsilon)\texttt{opt}(G)$. So, the approximation factor of the algorithm is no better than* 2.

Notes

The Steiner tree problem has its roots in the following plane geometry problem : given three points A, B, C on the place, which point O minimizes $|OA| + |OB| + |OC|$. This point is called the Fermat-Torricelli point. The 2-approximation algorithm described above is one of the earliest approximation algorithms known, and can be found in the paper [2] by Choukhmane. The first improvement of $\frac{11}{6}$ is in the paper [3] by Zelikovsky. The current best approximation faction is $\ln 4 \approx 1.39$ which was obtained in the paper [1] by Byrka, Grandoni, Rothvoss, and Sanita.

References

[1] J. Byrka, F. Grandoni, T. Rothvoss, and L. Sanità. Steiner tree approximation via iterative randomized rounding. *Journal of the ACM*, 60(1):1–33, 2013.

[2] C. El-Arbi. Une heuristique pour le problème de l'arbre de Steiner. *RAIRO-Operations Research*, 12(2):207–212, 1978.

[3] A. Z. Zelikovsky. An $11/6$-approximation algorithm for the network Steiner problem. *Algorithmica*, 9:463–470, 1993.