Notes edited by instructor in 2011.

1 Introduction/Administrivia


- Recommended books:

- 6 homework sets, the first 3 required. The last 3 sets can be replaced by a project.

Course Objectives

1. To appreciate that not all intractable problems are the same. NP optimization problems, identical in terms of exact solvability, can appear very different from the approximation point of view. This sheds light on why, in practice, some optimization problems (such as Knapsack) are easy, while others (like Clique) are extremely difficult.

2. To learn techniques for design and analysis of approximation algorithms, via some fundamental problems.

3. To build a toolkit of broadly applicable algorithms/heuristics that can be used to solve a variety of problems.

4. To understand reductions between optimization problems, and to develop the ability to relate new problems to known ones.

The complexity class P contains the set of problems that can be solved in polynomial time. From a theoretical viewpoint, this describes the class of tractable problems, that is, problems that can be solved efficiently. The class NP is the set of problems that can be solved in non-deterministic polynomial time, or equivalently, problems for which a solution can be verified in polynomial time. NP contains many interesting problems that often arise in practice, but there is good reason to believe P ≠ NP. That is, it is unlikely that there exist algorithms to solve NP optimization problems efficiently, and so we often resort to heuristic methods to solve these problems.

Heuristic approaches include backtrack search and its variants, mathematical programming methods, local search, genetic algorithms, tabu search, simulated annealing etc. Some methods are guaranteed to find an optimal solution, though they may take exponential time; others are guaranteed to run in polynomial time, though they may not return an optimal solution. Approximation algorithms fall in the latter category; however, though they do not find an optimal solution, we can give guarantees on the quality of the solution found.
Approximation Ratio

To give a guarantee on solution quality, one must first define what we mean by the quality of a solution. We discuss this more carefully in the next lecture; for now, note that each instance of an optimization problem has a set of feasible solutions. The optimization problems we consider have an objective function which assigns a (real/rational) number/value to each feasible solution of each instance \( I \). The goal is to find a feasible solution with minimum objective function value or maximum objective function value. The former problems are minimization problems and the latter are maximization problems.

For each instance \( I \) of a problem, let \( \text{OPT}(I) \) denote the value of an optimal solution to instance \( I \). We say that an algorithm \( \mathcal{A} \) is an \( \alpha \)-approximation algorithm for a problem if, for every instance \( I \), the value of the feasible solution returned by \( \mathcal{A} \) is within a (multiplicative) factor of \( \alpha \) of \( \text{OPT}(I) \). Equivalently, we say that \( \mathcal{A} \) is an approximation algorithm with approximation ratio \( \alpha \). For a minimization problem we would have \( \alpha \geq 1 \) and for a maximization problem we would have \( \alpha \leq 1 \). However, it is not uncommon to find in the literature a different convention for maximization problems where one says that \( \mathcal{A} \) is an \( \alpha \)-approximation algorithm if the value of the feasible solution returned by \( \mathcal{A} \) is at least \( \frac{1}{\alpha} \cdot \text{OPT}(I) \); the reason for using convention is so that approximation ratios for both minimization and maximization problems will be \( \geq 1 \). In this course we will for the most part use the convention that \( \alpha \geq 1 \) for minimization problems and \( \alpha \leq 1 \) for maximization problems.

Remarks:

1. The approximation ratio of an algorithm for a minimization problem is the maximum (or supremum), over all instances of the problem, of the ratio between the values of solution returned by the algorithm and the optimal solution. Thus, it is a bound on the worst-case performance of the algorithm.

2. The approximation ratio \( \alpha \) can depend on the size of the instance \( I \), so one should technically write \( \alpha(|I|) \).

3. A natural question is whether the approximation ratio should be defined in an additive sense. For example, an algorithm has an \( \alpha \)-approximation for a minimization problem if it outputs a feasible solution of value at most \( \text{OPT}(I) + \alpha \) for all \( I \). This is a valid definition and is the more relevant one in some settings. However, for many \( \mathbf{NP} \) problems it is easy to show that one cannot obtain any interesting additive approximation (unless of course \( P = \mathbf{NP} \)) due to scaling issues. We will illustrate this via an example later.

Pros and cons of the approximation approach:

Some advantages to the approximation approach include:

1. It explains why problems can vary considerably in difficulty.

2. The analysis of problems and problem instances distinguishes easy cases from difficult ones.

3. The worst-case ratio is robust in many ways. It allows reductions between problems.

4. Algorithmic ideas/tools are valuable in developing heuristics, including many that are practical and effective.
As a bonus, many of the ideas are beautiful and sophisticated, and involve connections to other areas of mathematics and computer science.

Disadvantages include:
1. The focus on worst-case measures risks ignoring algorithms or heuristics that are practical or perform well on average.
2. Unlike, for example, integer programming, there is often no incremental/continuous tradeoff between the running time and quality of solution.
3. Approximation algorithms are often limited to cleanly stated problems.
4. The framework does not (at least directly) apply to decision problems or those that are inapproximable.

Approximation as a broad lens

The use of approximation algorithms is not restricted solely to NP-Hard optimization problems. In general, ideas from approximation can be used to solve many problems where finding an exact solution would require too much of any resource.

A resource we are often concerned with is time. Solving NP-Hard problems exactly would (to the best of our knowledge) require exponential time, and so we may want to use approximation algorithms. However, for large data sets, even polynomial running time is sometimes unacceptable. As an example, the best exact algorithm known for the Matching problem in general graphs requires $O(n^3)$ time; on large graphs, this may be not be practical. In contrast, a simple greedy algorithm takes near-linear time and outputs a matching of cardinality at least $1/2$ that of the maximum matching; moreover there have been randomized sub-linear time algorithms as well.

Another often limited resource is space. In the area of data streams/streaming algorithms, we are often only allowed to read the input in a single pass, and given a small amount of additional storage space. Consider a network switch that wishes to compute statistics about the packets that pass through it. It is easy to exactly compute the average packet length, but one cannot compute the median length exactly. Surprisingly, though, many statistics can be approximately computed.

Other resources include programmer time (as for the Matching problem, the exact algorithm may be significantly more complex than one that returns an approximate solution), or communication requirements (for instance, if the computation is occurring across multiple locations).

2 The Steiner Tree Problem

In the Steiner Tree problem, the input is a graph $G(V,E)$, together with a set of terminals $S \subseteq V$, and a cost $c(e)$ for each edge $e \in E$. The goal is to find a minimum-cost tree that connects all terminals, where the cost of a subgraph is the sum of the costs of its edges.

The Steiner Tree problem is NP-Hard, and also APX-Hard [2]. The latter means that there is a constant $\delta > 1$ such that it is NP-Hard to approximate the solution to within a ratio of less than $\delta$; it is currently known that it is hard to approximate the Steiner Tree problem to within a ratio of $95/94$ [5].

1 Variants of the Steiner Tree problem, named after Jakob Steiner, have been studied by Fermat, Weber, and others for centuries. The front cover of the course textbook contains a reproduction of a letter from Gauss to Schumacher on a Steiner tree question.
**Note:** If $|S| = 2$ (that is, there are only 2 terminals), an optimal Steiner Tree is simply a shortest path between these 2 terminals. If $S = V$ (that is, all vertices are terminals), an optimal solution is simply a minimum spanning tree of the input graph. In both these cases, the problem can be solved exactly in polynomial time.

**Question:** Can you find an efficient algorithm to exactly solve the Steiner Tree problem with 3 terminals? How about the case when $|S|$ is a fixed constant?

Observe that to solve the Steiner Tree problem on a graph $G$, it suffices to solve it on the metric completion of $G$, defined below. (Why is this true?)

**Definition:** Given a connected graph $G(V, E)$ with edge costs, the metric completion of $G$ is a complete graph $H(V, E')$ such that for each $u, v \in V$, the cost of edge $uv$ in $H$ is the cost of the shortest path in $G$ from $u$ to $v$.

The graph $H$ with edge costs is metric, because the edge costs satisfy the triangle inequality: $\forall u, v, w, \text{cost}(uv) \leq \text{cost}(uw) + \text{cost}(wv)$.

![Figure 1: On the left, a graph. On the right, its metric completion, with new edges and modified edge costs in red.](image)

We now look at two approximation algorithms for the Steiner Tree problem.

### 2.1 The MST Algorithm

The following algorithm, with an approximation ratio of $(2 - 2/|S|)$ is due to [12]:

```plaintext
STEINERMST($G(V, E), S \subseteq V$):
Let $H(V, E') \leftarrow$ metric completion of $G$.
Let $T \leftarrow$ MST of $H[S]$.
Output $T$.
```

(Here, we use the notation $H[S]$ to denote the subgraph of $H$ induced by the set of terminals $S$.)

The following lemma is central to the analysis of the algorithm STEINERMST.

**Lemma 1** For any instance $I$ of Steiner Tree, let $H$ denote the metric completion of the graph, and $S$ the set of terminals. There exists a spanning tree in $H[S]$ (the graph induced by terminals) of cost at most $2(1 - 1/|S|)\text{OPT}$, where $\text{OPT}$ is the cost of an optimal solution to instance $I$.  

Before we prove the lemma, we note that if there exists some spanning tree in $H[S]$ of cost at most $2(1 - \frac{1}{|S|})OPT$, the minimum spanning tree has at most this cost. Therefore, Lemma 1 implies that the algorithm SteinerMST is a $2(1 - \frac{1}{|S|})$-approximation for the Steiner Tree problem.

**Proof of Lemma 1** Let $T^*$ denote an optimal solution in $H$ to the given instance, with cost $c(T^*)$. Double all the edges of $T^*$ to obtain an Eulerian graph, and fix an Eulerian Tour $W$ of this graph. (See Fig. 3 above.) Now, shortcut edges of $W$ to obtain a tour $W'$ of the vertices in $T^*$ in which each vertex is visited exactly once. Again, shortcut edges of $W'$ to eliminate all non-terminals; this gives a walk $W''$ that visits each terminal exactly once.

It is easy to see that $c(W'') \leq c(W') \leq c(W) = 2c(T^*)$, where the inequalities follow from the fact that by shortcutting, we can only decrease the length of the walk. (Recall that we are working in the metric completion $H$.) Now, delete the heaviest edge of $W''$ to obtain a path through all the terminals in $S$, of cost at most $(1 - \frac{1}{|S|})c(W'')$. This path is a spanning tree of the terminals, and contains only terminals; therefore, there exists a spanning tree in $H[S]$ of cost at most $2(1 - \frac{1}{|S|})c(T^*)$. □
A tight example: The following example (Fig. 4 below) shows that this analysis is tight; there are instances of Steiner Tree where the SteinerMST algorithm finds a tree of cost $2(1 - \frac{1}{S})\text{OPT}$. Here, each pair of terminals is connected by an edge of cost 2, and each terminal is connected to the central non-terminal by an edge of cost 1. The optimal tree is a star containing the central non-terminal, with edges to all the terminals; it has cost $|S|$. However, the only trees in $H[S]$ are formed by taking $|S| - 1$ edges of cost 2; they have cost $2(|S| - 1)$.

![Graph G; not all edges shown](image1)

![H[S]; not all edges shown](image2)

![An MST of H[S]](image3)

Figure 4: A tight example for the SteinerMST algorithm

2.2 The Greedy/Online Algorithm

We now describe another simple algorithm for the Steiner Tree problem, due to [9]:

**GreedySteiner**($G(V,E), S \subseteq V)$:
Let $\{s_1, s_2, \ldots, s_{|S|}\}$ be an arbitrary ordering of the terminals.
Let $T \leftarrow \{s_1\}$
For ($i$ from 2 to $|S|$):
Let $P_i$ be the shortest path in $G$ from $s_i$ to $T$.
Add $P_i$ to $T$.

GreedySteiner is a $\lceil \log_2 |S| \rceil$-approximation algorithm; here, we prove a slightly weaker result.

**Theorem 2** The algorithm GreedySteiner has an approximation ratio of $2H_{|S|} \approx 2 \ln |S|$, where $H_i = \sum_{j=1}^{i} 1/j$ denotes the $i$'th harmonic number.

Note that this is an online algorithm; terminals are considered in an arbitrary order, and when a terminal is considered, it is immediately connected to the existing tree. Thus, even if the algorithm could not see the entire input at once, but instead terminals were revealed one at a time and the algorithm had to produce a Steiner tree at each stage, the algorithm GreedySteiner outputs a tree of cost no more than $O(\log |S|)$ times the cost of the optimal tree.

To prove Theorem 2 we introduce some notation. Let $c(i)$ denote the cost of the path $P_i$ used in the $i$th iteration to connect the terminal $s_i$ to the already existing tree. Clearly, the total cost of the tree is $\sum_{i=1}^{|S|} c(i)$. Now, let $\{i_1, i_2, \ldots, i_{|S|}\}$ be a permutation of $\{1, 2, \ldots, |S|\}$ such that $c(i_1) \geq c(i_2) \geq \ldots \geq c(i_{|S|})$. (That is, relabel the terminals in decreasing order of the cost paid to connect them to the tree that exists when they are considered by the algorithm.)
Claim 3 For all \( j \), the cost \( c(i_j) \) is at most \( 2\text{OPT}/j \), where \( \text{OPT} \) is the cost of an optimal solution to the given instance.

Proof: Suppose by way of contradiction this were not true; since \( s_{ij} \) is the terminal with \( j \)th highest cost of connection, there must be \( j \) terminals that each pay more than \( 2\text{OPT}/j \) to connect to the tree that exists when they are considered. Let \( S' = \{s_{i1}, s_{i2}, \ldots, s_{ij}\} \) denote this set of terminals.

We argue that no two terminals in \( S' \cup \{s_1\} \) are within distance \( 2\text{OPT}/j \) of each other. If some pair \( x, y \) were within this distance, one of these terminals (say \( y \)) must be considered later by the algorithm than the other. But then the cost of connecting \( y \) to the already existing tree (which includes \( x \)) must be at most \( 2\text{OPT}/j \), and we have a contradiction.

Therefore, the minimum distance between any two terminals in \( S' \cup \{s_1\} \) must be greater than \( 2\text{OPT}/j \). Since there must be \( j \) edges in any MST of these terminals, an MST must have cost greater than \( 2\text{OPT} \). But the MST of a subset of terminals cannot have cost more than \( 2\text{OPT} \), exactly as argued in the proof of Lemma 1. Therefore, we obtain a contradiction.

Given this claim, it is easy to prove Theorem 2

\[
\sum_{i=1}^{\vert S \vert} c(i) = \sum_{j=1}^{\vert S \vert} c(i_j) \leq \sum_{j=1}^{\vert S \vert} \frac{2\text{OPT}}{j} = 2\text{OPT} \sum_{j=1}^{\vert S \vert} \frac{1}{j} = 2H_{\vert S \vert} \cdot \text{OPT}.
\]

Question: Give an example of a graph and an ordering of terminals such that the output of the Greedy algorithm is \( \Omega(\log \vert S \vert) \text{OPT} \).

Remark: We emphasize again that the analysis above holds for every ordering of the terminals. A natural variant might be to adaptively order the terminals so that in each iteration \( i \), the algorithm picks the terminal \( s_i \) to be the one closest to the already existing tree \( T \) built in the first \( i \) iterations. Do you see that this is equivalent to using the MST Heuristic with Prim’s algorithm for MST? This illustrates the need to be careful in the design and analysis of heuristics.

Other Results on Steiner Trees

The 2-approximation algorithm using the MST Heuristic is not the best approximation algorithm for the Steiner Tree problem currently known. Some other results on this problem are listed below.

1. The first algorithm to obtain a ratio of better than 2 was due to due to Alexander Zelikovsky [13]; the approximation ratio of this algorithm was \( 11/6 \approx 1.83 \). Until very recently the best known approximation ratio in general graphs is \( 1 + \frac{\ln 3}{2} \approx 1.55 \); this is achieved by an algorithm of [11] that combines the MST heuristic with Local Search. Very recently Byrka et al. gave an algorithm with an approximation ratio of 1.39 [4] which is currently the best known for this problem.

2. The bidirected cut LP relaxation for the Steiner Tree was proposed by [7]; it has an integrality gap of at most \( 2(1 - \frac{1}{\vert S \vert}) \), but it is conjectured that the gap is smaller. No algorithm is currently known that exploits this LP relaxation to obtain an approximation ratio better than that of the SteinerMST algorithm. Though the true integrality gap is not known, there are examples due to [8, 10] that show it is at least \( 8/7 \approx 1.14 \).
3. For many applications, the vertices can be modeled as points on the plane, where the distance between them is simply the Euclidean distance. The MST-based algorithm performs fairly well on such instances; it has an approximation ratio of $2/\sqrt{3} \approx 1.15$ [6]. An example which achieves this bound is three points at the corners of an equilateral triangle, say of side-length 1; the MST heuristic outputs a tree of cost 2 (two sides of the triangle) while the optimum solution is to connect the three points to a Steiner vertex which is the circumcenter of the triangle. One can do better still for instances in the plane (or in any Euclidean space of small-dimensions); for any $\epsilon > 0$, there is a $1 + \epsilon$-approximation algorithm that runs in polynomial time [1]. Such an approximation scheme is also known for planar graphs [3] and more generally bounded-genus graphs.

References


