CS 598CSC: Approximation Algorithms	Lecture date: January 21, 2009
Instructor: Chandra Chekuri	Scribe: Nitish Korula

1 Introduction/Administrivia

- Course website: `http://www.cs.uiuc.edu/class/sp09/cs598csc/`. Join the newsgroup!
- Text book (recommended): *Approximation Algorithms* by Vijay Vazirani, Springer-Verlag, 2004.
- 6 homework sets, the first 3 required. The last 3 sets can be replaced by a project.

Course Objectives

1. To appreciate the rich landscape of optimization problems. **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 $\mathbf{P} \neq \mathbf{NP}$. 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.

Commonly used heuristic approaches include mathematical programming, local seach, 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; this measures the quality of solutions.

For each instance I of a problem, let $\mathrm{OPT}(I)$ denote the value of an optimal solution to instance I. We say that an algorithm $\mathcal{A}$ is an α-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 α of $\mathrm{OPT}(I)$. Equivalently, we say that $\mathcal{A}$ is an approximation algorithm with *approximation ratio* α.

Notes:

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

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

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 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 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 too long. In contrast, the greedy algorithm takes linear time and outputs a matching at least 1/2 the size of the maximum matching.

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 δ; it is currently known that it is hard to approximate the STEINER TREE problem to within a ratio of 95/94 [3].[1]

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?

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: $\forall u, v, w, \quad cost(uv) \leq cost(uw) + cost(wv)$.

[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.

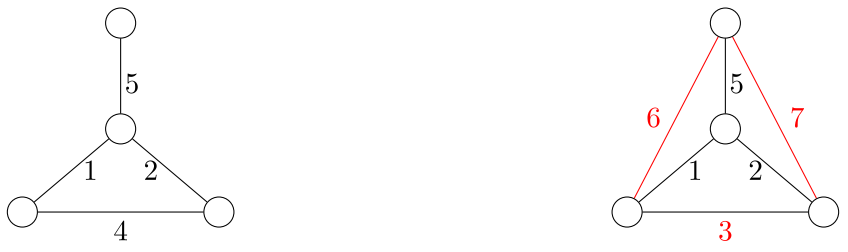

Figure 1: On the left, a graph. On the right, its metric completion, with new edges and modified edge costs in red.

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

The MST Algorithm

The following algorithm, with an approximation ratio of 2 is due to [10]:

```
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.)

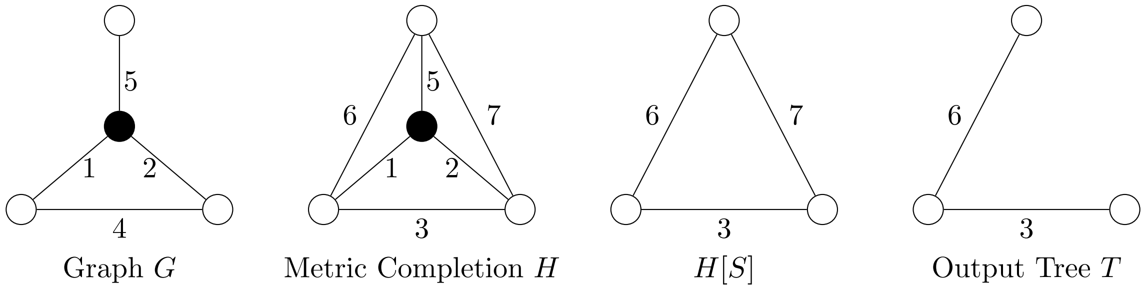

Figure 2: Illustrating the MST Heuristic for Steiner Tree

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 $2(1 - \frac{1}{|S|})\mathrm{OPT}$, where 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 $2(1 - \frac{1}{|S|})\mathrm{OPT}$, 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

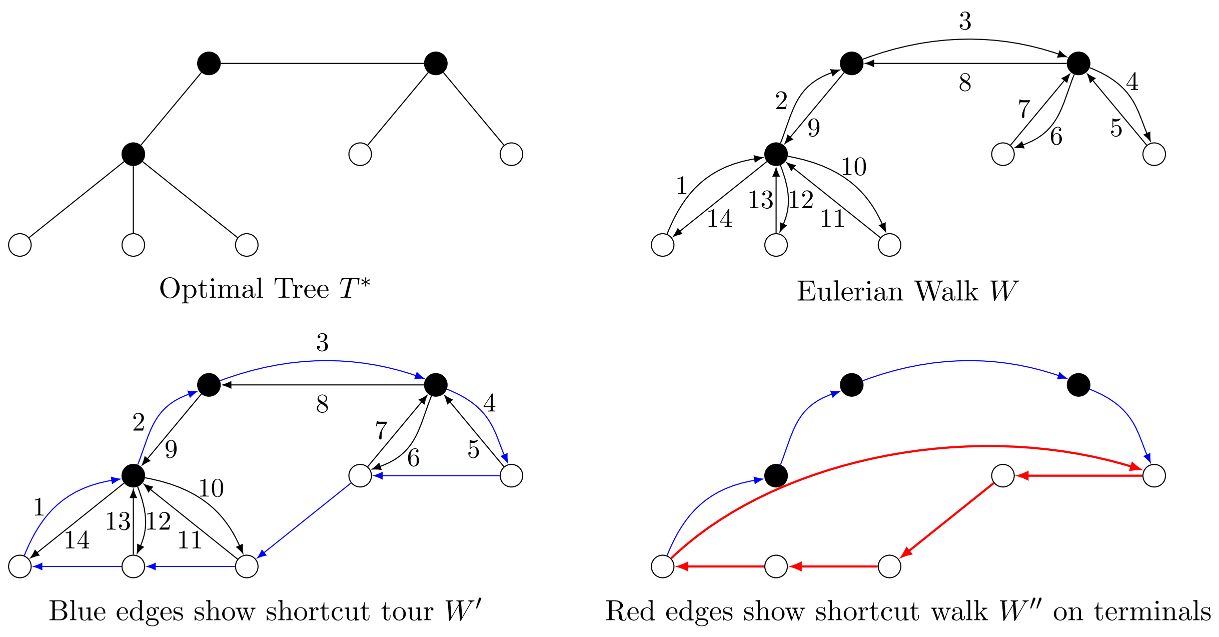
Optimal Tree T^*
Eulerian Walk W
Blue edges show shortcut tour W'
Red edges show shortcut walk W'' on terminals
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Figure 3: Doubling edges of T^* and shortcutting gives a low-cost spanning tree on terminals.

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 $2(1-\frac{1}{S})$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 $2(|S| - 1)$.

The Greedy/Online Algorithm

We now describe another simple algorithm for the Steiner Tree problem, due to [7]:

```
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 $T$.
```

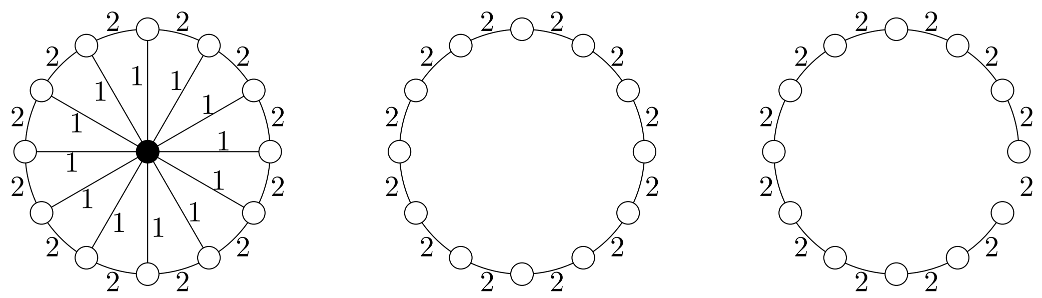
Graph G; not all edges shown $H[S]$; not all edges shown.   An MST of $H[S]$.

Figure 4: A tight example for the SteinerMST algorithm

GreedySteiner is a $\lceil \log_2 n \rceil$-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 *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 ith 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* $2\mathrm{OPT}/j$*, where* OPT *is the cost of an optimal solution to the given instance.*

Proof: Suppose by way of contradiction this were not true; since s_{i_j} is the terminal with jth highest cost of connection, there must be j terminals that each pay more than $2\mathrm{OPT}/j$ to connect to the tree that exists when they are considered. Let $S' = \{s_{i_1}, s_{i_2}, \ldots s_{i_j}\}$ denote this set of terminals.

We argue that no two terminals in $S' \cup \{s_1\}$ are within distance $2\mathrm{OPT}/j$ 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 $2\mathrm{OPT}/j$, and we have a contradiction.

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

Given this claim, it is easy to prove Theorem 2.

$$\sum_{i=1}^{|S|} c(i) = \sum_{j=1}^{|S|} c(i_j) \leq \sum_{j=1}^{|S|} \frac{2\mathrm{OPT}}{j} = 2\mathrm{OPT} \sum_{j=1}^{|S|} \frac{1}{j} = 2H_{|S|} \cdot \mathrm{OPT}.$$

Note: We emphasize again that this algorithm considers vertices in *any* order. 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?

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 [11]; the approximation ratio of this algorithm was $11/6 \approx 1.83$. Currently, the best known approximation ratio in general graphs is $1 + \frac{\ln 3}{2} \approx 1.55$; this is achieved by an algorithm of [9] that combines the MST heuristic with Local Search.

2. The *bidirected cut* LP relaxation for the Steiner Tree was proposed by [5]; it has an integrality gap of at most $2(1 - \frac{1}{|S|})$, 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 [6, 8] 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 $2/\sqrt{3} \approx 1.15$ [4]. 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].

References

[1] S. Arora. Polynomial-time Approximation Schemes for Euclidean TSP and other Geometric Problems. *J. of the ACM* 45(5): 753–782, 1998.

[2] M. Bern and P. Plassmann. The Steiner problems with edge lengths 1 and 2. *Information Processing Letters* 32, 171–176, 1989.

[3] M. Chlebik and J. Chlebikova. Approximation Hardness of the Steiner Tree Problem on Graphs. *Proc. 8th Scandinavian Workshop on Algorithm Theory*, Springer-Verlag, 170–179, 2002.

[4] D. Z. Du and F. K. Hwang. A proof of Gilbert-Pollak's conjecture on the Steiner ratio. *Algorithmica*, 7: 121–135, 1992.

[5] J. Edmonds. Optimum branchings. *J. of Research of the National Bureau of Standards, Section B*, 71:233–240, 1967.

[6] M. Goemans. Unpublished Manuscript, 1996.

[7] M. Imaze and B.M. Waxman. Dynamic Steiner tree problem. *SIAM J. on Discrete Math.*, 4(3): 369–184, 1991.

[8] J. Konemann, D. Pritchard, and K. Tan. A partition based relaxation for Steiner trees. Manuscript, 2007. `arXiv:0712.3568`, `http://arxiv.org/abs/0712.3568v1`

[9] G. Robins and A. Zelikovsky. Tighter Bounds for Graph Steiner Tree Approximation. *SIAM J. on Discrete Math.*, 19(1): 122–134, 2005.

[10] J. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Jap.*, 24: 573–577, 1980.

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