**Approximation Algorithms and Hardness of Approximation** February 19, 2013

Lecture 1

*Lecturer: Ola Svensson* *Scribes: Alantha Newman*

# 1 Class Information

- 4 credits
- Lecturers:
  - Ola Svensson (**ola.svensson@epfl.ch**)
  - Alantha Newman (**alantha.newman@epfl.ch**)
- Mailing list: **approx13-subscribe@listes.epfl.ch**
- Books:
  - *Approximation Algorithms*, Vazirani, 2001,
  - *The Design of Approximation Algorithms*, David Williamson and David Shmoys, 2010. (Available at: www.designofapproxalgs.com),
  - *Computational Complexity: A Modern Approach*, Sanjeev Arora and Boaz Barak, 2007. (Available at: www.cs.princeton.edu/theory/complexity).

# 2 Motivation: "Fast. Reliable. Cheap. Choose two."

Some desirable properties for an algorithm are:

1. runs in polynomial time,
2. finds exact optimal solution,
3. robust: works for any input instance of a problem.

Most natural optimization problems are NP-hard (e.g. set cover, chromatic number). We can find exact solutions using techniques from mathematical programming, for example. But we do not know how to do so with a guarantee of efficiency or polynomial running time. In this class, we study algorithms that satisfy the first and third properties—fast algorithms for every input—and so we must relax the second property.

## 2.1 What is an Approximation Algorithm?

An $\alpha$-approximation algorithm $A$ for a problem $P$ is an algorithm that:

1. runs in polynomial time,
2. for any instance $I$ of problem $P$, algorithm $A$ produces a solution with value $val_A(I)$ such that:
   (a) $\frac{val_A(I)}{OPT(I)} \leq \alpha$ (if $P$ is a minimization problem),
   (b) $\frac{val_A(I)}{OPT(I)} \geq \alpha$ (if $P$ is a maximization problem).

### 2.2 Examples of Approximation Guarantees

Let us consider the problem of visiting all cities in Switzerland in the minimum amount of time. Suppose that a fixed amount of time $c_{ij}$ is required to travel between cities $i$ and $j$. Suppose further that there is a way to visit all the cities in 1000 hours and that this is the minimum amount of time required. However, we do not know this optimal ordering of the cities. Suppose that using Google maps, we find a way to visit all the cities in 1500 hours. Using another, superior method, we obtain an ordering of the cities that uses 1333 hours. We say the Google solution is a 1.5-approximation to the optimal solution. The superior solution is a 1.333-approximation to the optimal solution. Note that even if we obtain a solution requiring 1333 hours, we only know that this is a 1.333-approximation if we know that an optimal solution requires at least 1000 hours.

### 2.3 Why Approximation Algorithms?

Approximation algorithms give us a metric to compare techniques and problems. Someone who designs, say, a 1.333-approximation algorithm for a problem probably has a better understanding of the problem than someone who can only design a 1.5-approximation algorithm for that problem. Sometimes, approximation algorithms are easier to implement than exact algorithms, since we have relaxed the objective of providing an exact solution. Additionally, approximation algorithms can provide ideas for heuristics, which we could implement to obtain exact solutions and sacrifice the guarantee of polynomial running time.

## 3 Lower Bounds for Minimization Problems

How can we design an $\alpha$-approximation algorithm for a problem $P$ if we can not efficiently compute the value of an optimal solution? In other words, if problem $P$ is NP-hard, it is most likely impossible to efficiently determine the optimal value of $P$ on an input instance $I$. If we can not compute this value, how do we know that the output of our supposed $\alpha$-approximation algorithm is actually within factor $\alpha$ of the value of an optimal solution?

The main idea is that, although it is often the case that computing the value of an optimal solution is infeasible, we can nevertheless often efficiently compute a *lower bound* (for a minimization problem) on the value of an optimal solution. Then, if we prove that our algorithm outputs a solution within factor $\alpha$ of this lower bound, then we can guarantee that our solution is within factor $\alpha$ of the value of an optimal solution. Analogously, we need to find efficiently computable upper bounds in order to design efficient approximation algorithms for maximization problems.

### 3.1 Example 1: Traveling Salesman Problem

Given $n$ cities with costs $c_{ij}$ to go between city $i$ and $j$, the problem is to find a minimum cost tour that visits each city at least once. We can think of this problem on a graph, $G = (V, E)$, where the $n$ cities are represented by vertices in $V$ and each edge has weight $c_{ij}$. We can assume the triangle inequality: for all $i, j, j \in V$, $c_{ij} \leq c_{ik} + c_{kj}$, and with this assumption, our goal is to find the minimum cost tour that visits each vertex exactly once. We will refer to this problem as the *metric traveling salesman problem* or *metric TSP*.

What is a polynomial-time computable lower bound on the cost of an optimal solution for the metric TSP problem?

**Lemma 1** *The cost of a minimum spanning tree is at most that of an optimal tour.*

**Proof** Consider an optimal tour. Removing one edge results in a spanning tree with cost at most that of the tour. Since the minimum spanning tree has cost at most the cost of any tree, it follows that the cost of the minimum spanning tree is at most the cost of an optimal tour. ■

ALGORITHM "DOUBLE"
*Input:* A $G = (V, E)$ with edge costs $\{c_{ij}\}$.

1. Find a minimum spanning tree of $G$ with cost $c(MST)$.
2. Double each edge in the spanning tree.
3. Take an Eulerian tour $T$ of the doubled spanning tree.
4. Delete each previously visited vertex in the Eulerian tour $T$.

*Output:* Tour of the vertices, $T$.

Note that in Step 2. of Algorithm "Double", we obtain a graph in which each vertex has even degree. It is well known that such a graph has an Eulerian tour, i.e. a tour that visits each edge exactly once, and there are many efficient ways to compute such a tour. The resulting tour can be viewed as an ordering of the vertices in $V$ in which each vertex appears at least once but may appear multiple times. The last step of Algorithm "Double" (Step 4) is to go through the vertices in the order determined by the Eulerian tour and delete an occurence of a vertex if it appeared previously in the order. Note that this step does not increase the cost of the tour due to the assumed triangle inquality on the edge costs.

**Lemma 2** *Algorithm "Double" is a 2-approximation algorithm for metric TSP.*

**Proof** Call the cost of the final output tour of the vertices $c(T)$. Since Step 3 produces a tour that covers each vertex at least once, and Step 4 removes only repeated occurences of a vertex, the output tour $T$ is a feasible solution. Moreover, the Eulerian tour computed at Step 3 has cost at most $2 \cdot c(MST)$. As mentioned earlier, due to the triangle inequality, we do not increase the cost of the tour in Step 4. Thus, for the cost of the output tour $T$, applying Lemma 2 we have:

$$c(T) \leq 2 \cdot c(MST) \leq 2 \cdot OPT.$$

■

The best known approximation guarantee for the metric TSP problem is 1.5. This is due to Christofides and dates to 1976. There is a famous conjecture that there exists an efficient algorithm that produces a tour of cost at most 4/3-optimal.

## 3.2 Example 2: Max-3SAT

Given $m$ clauses $C_1, C_2, \ldots C_m$, where each clause is a disjunction of three literals, e.g. $(x \vee y \vee \bar{z})$, find a truth assignment that maximizes the number of satisfied clauses.

For example, if we are given two clauses: $(x \vee y \vee z)$ and $(\bar{x} \vee \bar{y} \vee \bar{z})$, out of eight possible truth assignment to the three variables, only two assignments leave a clause unsatisfied.

Since this is a maximization problem, we need to find an efficiently computable upper bound. A simple upper bound is that the number of clauses that can be satisfied is at most $m$ or the total number of input clauses.

ALGORITHM "RANDOM"
*Input:* Clauses $C_1, C_2, \ldots C_m$ on $n$ literals, $L$.

1. For each variable $x \in L$,
   - set $x$ to "true" with probability $\frac{1}{2}$.
   - set $x$ to 0 "false" with probability $\frac{1}{2}$.

*Output:* Truth assignment for each literal in $L$.

**Lemma 3** *Algorithm "Random" is a $\frac{7}{8}$-approximation algorithm.*

**Proof**

$$\mathbb{E}[\text{number of satisfied clauses}] = \sum_{i=1}^{m} \Pr[C_i \text{ is satisfied}] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7m}{8}.$$

■

For Max-2SAT, Algorithm "Random" gives an approximation guarantee of $\frac{3}{4}$. For $Max-3SAT$, the approximation guarantee of $\frac{7}{8}$ is the best possible and we will see the proof of this theorem later in the course.

## 3.3 Example 3: Scheduling on Identical Machines

Given $n$ jobs of processing times $p_1, p_2, \ldots p_n$, respectively, and $m$ identical machines, find a schedule that minimizes the completion time, i.e. the completion time of the job that finishes last.

For example, suppose we are given three machines ($m = 3$), and four jobs ($n = 4$) of processing times $1, 1, 1, 2$. If we schedule the unit processing time jobs on one machine each, and the job with processing time two on an arbitrary machine, then the total completion time of this schedule is three.

What is the lower bound on the completion time of an optimal schedule? For this problem, there are two simple lower bounds. The first is the average time each machine is used or $\sum_{i=1}^{m} p_i/m$. The second lower bound is the largest processing time or $\max_i\{p_i\}$. We consider the following algorithm:

ALGORITHM "LIST SCHEDULING"
*Input:* $n$ jobs with processing time $\{p_1, p_2, \ldots p_n\}$ and $m$ machines.

1. Order the jobs arbitrarily.
2. In this order, assign each job to the machine that currently has the least work assigned to it.

*Output:* The assignment of jobs to machines.

The following theorem is due to Graham from 1969.

**Theorem 4** *Algorithm "List Scheduling" is a 2-approximation algorithm.*

**Proof** Consider job $j$ that completes last. Then the completion time of the schedule is the completion time of job $j$. Let $start_j$ denote the starting time of job $j$. Since job $j$ was assigned to the machine with the least amount of work, it follows that:

$$start_j \leq \frac{\sum_{i=1}^{m} p_i}{m}.$$

Thus, $start_j$ is also a lower bound (on a lower bound) for the minimum completion time of a schedule. The completion time of job $j$ is $start_j + p_j$. Since both of these quantities are lower bounds on the completion time of an optimal schedule, our algorithm outputs a schedule requiring time no more than twice that of an optimal schedule. ■