Lecture 1

In which we describe what this course is about and give two simple examples of approximation algorithms

1 Overview

In this course we study algorithms for combinatorial optimization problems. Those are the type of algorithms that arise in countless applications, from billion-dollar operations to everyday computing task; they are used by airline companies to schedule and price their flights, by large companies to decide what and where to stock in their warehouses, by delivery companies to decide the routes of their delivery trucks, by Netflix to decide which movies to recommend you, by a gps navigator to come up with driving directions and by word-processors to decide where to introduce blank spaces to justify (align on both sides) a paragraph.

In this course we will focus on general and powerful algorithmic techniques, and we will apply them, for the most part, to highly idealized model problems.

Some of the problems that we will study, along with several problems arising in practice, are NP-hard, and so it is unlikely that we can design exact efficient algorithms for them. For such problems, we will study algorithms that are worst-case efficient, but that output solutions that can be sub-optimal. We will be able, however, to prove worst-case bounds to the ratio between the cost of optimal solutions and the cost of the solutions provided by our algorithms. Sub-optimal algorithms with provable guarantees about the quality of their output solutions are called *approximation algorithms*.

The content of the course will be as follows:

- *Simple examples of approximation algorithms.* We will look at approximation algorithms for the Vertex Cover and Set Cover problems, for the Steiner Tree Problem and for the Traveling Salesman Problem. Those algorithms and their analyses will be relatively simple, but they will introduce a number of key concepts, including the importance of getting upper bounds on the cost of an optimal solution.

- *Linear Programming.* A linear program is an optimization problem over the real numbers in which we want to optimize a linear function of a set of real variables subject to a system of linear inequalities about those variables. For example, the following is a linear program:

$$\begin{aligned} &\text{maximize } x_1 + x_2 + x_3 \\ &\text{Subject to :} \\ &2x_1 + x_2 \leq 2 \\ &x_2 + 2x_3 \leq 1 \end{aligned}$$

(A linear program is not a *program* as in *computer program*; here *programming* is used to mean *planning.*) An optimum solution to the above linear program is, for example, $x_1 = 1/2$, $x_2 = 1$, $x_3 = 0$, which has cost 1.5. One way to see that it is an optimal solution is to sum the two linear constraints, which tells us that in every admissible solution we have

$$2x_1 + 2x_2 + 2x_3 \leq 3$$

that is, $x_1 + x_2 + x_3 \leq 1.5$. The fact that we were able to verify the optimality of a solution by summing inequalities is a special case of the important theory of *duality* of linear programming.

A linear program is an optimization problem over real-valued variables, while this course is about *combinatorial* problems, that is problems with a finite number of discrete solutions. The reasons why we will study linear programming are that

1. Linear programs can be solved in polynomial time, and very efficiently in practice;
2. All the combinatorial problems that we will study can be written as linear programs, provided that one adds the additional requirement that the variables only take *integer value.*

This leads to two applications:

1. If we take the integral linear programming formulation of a problem, we remove the integrality requirement, we solve it efficient as a linear program over the real numbers, and we are lucky enough that the optimal solution happens to have integer values, then we have the optimal solution for our combinatorial problem. For some problems, it can be proved that, in fact, this will happen for every input.

2. If we take the integral linear programming formulation of a problem, we remove the integrality requirement, we solve it efficient as a linear program over the real numbers, we find a solution with fractional values, but then we are able to "round" the fractional values to integer ones without changing the cost of the solution too much, then we have an efficient *approximation algorithm* for our problem.

- *Approximation Algorithms via Linear Programming.* We will give various examples in which approximation algorithms can be designed by "rounding" the fractional optima of linear programs.

- *Exact Algorithms for Flows and Matchings.* We will study some of the most elegant and useful optimization algorithms, those that find optimal solutions to "flow" and "matching" problems.

- *Linear Programming, Flows and Matchings.* We will show that flow and matching problems can be solved optimally via linear programming. Understanding why will make us give a second look at the theory of linear programming duality.

- *Online Algorithms.* An online algorithm is an algorithm that receives its input as a stream, and, at any given time, it has to make decisions only based on the partial amount of data seen so far. We will study two typical online settings: paging (and, in general, data transfer in hierarchical memories) and investing.

2 The Vertex Cover Problem

2.1 Definitions

Given an undirected graph $G = (V, E)$, a *vertex cover* is a subset of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$ at least one of u or v is an element of C.

In the *minimum vertex cover* problem, we are given in input a graph and the goal is to find a vertex containing as few vertices as possible.

The minimum vertex cover problem is very related to the *maximum independent set* problem. In a graph $G = (V, E)$ an *independent set* is a subset $I \subseteq V$ of vertices such that there is no edge $(u, v) \in E$ having both endpoints u and v contained in I. In the maximum independent set problem the goal is to find a largest possible independent set.

It is easy to see that, in a graph $G = (V, E)$, a set $C \subseteq V$ is a vertex cover if and only if its complement $V - C$ is an independent set, and so, from the point of view of exact solutions, the two problems are equivalent: if C is an optimal vertex cover for

the graph G then $V - C$ is an optimal independent set for G, and if I is an optimal independent set then $V - I$ is an optimal vertex cover.

From the point of view of approximation, however, the two problems are not equivalent. We are going to describe a linear time 2-approximate algorithm for minimum vertex cover, that is an algorithm that finds a vertex cover of size at most twice the optimal size. It is known, however, that no constant-factor, polynomial-time, approximation algorithms can exist for the independent set problem. To see why there is no contradiction (and how the notion of approximation is highly dependent on the cost function), suppose that we have a graph with n vertices in which the optimal vertex cover has size $.9 \cdot n$, and that our algorithm finds a vertex cover of size $n - 1$. Then the algorithm finds a solution that is only about 11% larger than the optimum, which is not bad. From the point of view of independent set size, however, we have a graph in which the optimum independent set has size $n/10$, and our algorithm only finds an independent set of size 1, which is terrible

2.2 The Algorithm

The algorithm is very simple, although not entirely natural:

- Input: graph $G = (V, E)$
- $C := \emptyset$
- while there is an edge $(u, v) \in E$ such that $u \notin C$ and $v \notin C$
  - $C := C \cup \{u, v\}$
- return C

We initialize our set to the empty set, and, while it fails to be a vertex cover because some edge is uncovered, we add *both* endpoints of the edge to the set. By the time we are finished with the *while* loop, C is such that for every edge $(u, v) \in E$, either $u \in C$ or $v \in C$ (or both), that is, C is a vertex cover.

To analyze the approximation, let *opt* be the number of vertices in a minimal vertex cover, then we observe that

- If $M \subseteq E$ is a *matching*, that is, a set of edges that have no endpoint in common, then we must have $opt \geq |M|$, because every edge in M must be covered using a distinct vertex.
- The set of edges that are considered inside the *while* loop form a matching, because if (u, v) and (u', v') are two edges considered in the *while* loop, and

(u, v) is the one that is considered first, then the set C contains u and v when (u', v') is being considered, and hence u, v, u', v' are all distinct.

- If we let M denote the set of edges considered in the *while* cycle of the algorithm, and we let C_{out} be the set given in output by the algorithm, then we have

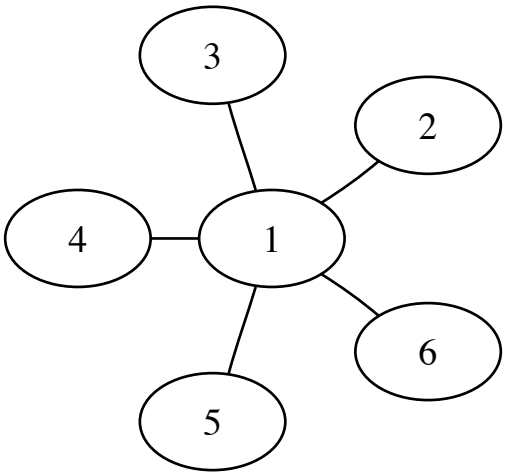
$$|C_{out}| = 2 \cdot |M| \leq 2 \cdot opt$$

As we said before, there is something a bit unnatural about the algorithm. Every time we find an edge (u, v) that violates the condition that C is a vertex cover, we add *both* vertices u and v to C, even though adding just one of them would suffice to cover the edge (u, v). Isn't it an overkill?

Consider the following alternative algorithm that adds only one vertex at a time:

- Input: graph $G = (V, E)$
- $C := \emptyset$
- while there is an edge $(u, v) \in E$ such that $u \notin C$ and $v \notin C$
  - $C := C \cup \{u\}$
- return C

This is a problem if our graph is a "star." Then the optimum is to pick the center, while the above algorithm might, in the worse case, pick all the vertices except the center.



Another alternative would be a greedy algorithm:

- Input: graph $G = (V, E)$
- $C := \emptyset$
- while C is not a vertex cover
  - let u be the vertex incident on the most uncovered edges
  - $C := C \cup \{u\}$
- return C

The above greedy algorithm also works rather poorly. For every n, we can construct an n vertex graph where the optimum is roughly $n/\ln n$, but the algorithm finds a solution of cost roughly $n - n/\ln n$, so that it does not achieve a constant-factor approximation of the optimum. We will return to this greedy approach and to these bad examples when we talk about the *minimum set cover* problem.

3 The Metric Steiner Tree Problem

In the Steiner Tree problem, we are given a set of *required* points R and a set of *optional* points S, along with a *distance function* $d : (R \cup S) \times (R \cup S) \to \mathbb{R}_{\geq 0}$. The distance function is a *metric*, that is, for every two points x, y we have $d(x, y) = d(y, x)$, and for every three points x, y, z we have the *triangle inequality*

$$d(x, y) \leq d(x, z) + d(z, y)$$

Our goal is to find a tree $T = (V, E)$ that spans all the required points, and possibly uses some of the optional points, that is, $R \subseteq V \subseteq R \cup S$, and such that the *total length* of the tree

$$\sum_{(x,y)\in E} d(x, y)$$

is minimized.

This problem is very similar to the *minimum spanning tree* problem, which we know to have an exact algorithm that runs in polynomial (in fact, nearly linear) time. In the minimum spanning tree problem, we are given a weighted graph, which we can think of as a set of points together with a distance function (which might not satisfy the triangle inequality), and we want to find the tree of minimal total length that *spans all the vertices*. The difference is that in the minimum Steiner tree problem we only require to span a subset of vertices, and other vertices are included only if they are beneficial to constructing a tree of lower total length.

We consider the following very simple approximation algorithm: *run a minimum spanning tree algorithm on the set of required vertices*, that is, find the best possible tree that uses none of the optional vertices. Next time we will prove that this algorithm achieves a factor of 2 approximation.