Optimization and Approximation ENS Lyon

Lecture 1

Lecturer: Alantha Newman *September 14, 2016*

1 Optimization

What is an Optimization Problem? Optimize (minimize or maximize) an objective function over a set of feasible solutions (defined by a set of constraints).

1.1 Example: Polly's Diet

Suppose we are given the following nutrition table:

	Chicken	Eggs	Bread	Milk
Protein	1	1	2	1
Fat	2	3	0	1
Sugar	0	1	1	1
Calories	4	5	2	3

A daily diet should obey the following guidelines.

It should contain the following recommended amounts:

- at least 10 units of protein,
- at least 15 units of fat,
- and at least 6 units of sugar.

It should **not** contain the following:

- more than 4 units of chicken,
- more than 5 units of milk.

Problem: Find a diet with the minimum calories.

Solution: Model this problem mathematically.

Define variables: let x_1, x_2, x_3, x_4 denote units of chicken, egg, bread and milk, respectively. (The number of units for each foood may be fractional.)

Translate the diet requirements into constraints.

Define the objective function (minimize calories).

$$\begin{aligned} & \min & 4x_1 + 5x_2 + 2x_3 + 3x_4 \\ x_1 + x_2 + 2x_3 + x_4 & \geq & 10 \\ 2x_1 + 3x_2 + x_4 & \geq & 15 \\ x_2 + x_3 + x_4 & \geq & 6 \\ x_1 & \leq & 4 \\ x_4 & \leq & 5 \\ x_1, x_2, x_3, x_4 & \geq & 0. \end{aligned}$$

1.2 A Generalized Diet Problem

Suppose there are n different foods and m different nutrients. The following table shows the nutritional content in a unit of each food.

	food 1	...	food n
nutrient 1	a_{11}	...	a_{1n}
nutrient 2	a_{21}	...	a_{2n}
.	.		.
.	.		.
nutrient m	a_{m1}	...	a_{mn}

Let $\mathbf{A}$ be the $m \times n$ matrix with entries $\{a_{ij}\}$. The j^{th} column $\mathbf{A_j}$ represents the nutritional content of the j^{th} food. Let $\mathbf{b}$ be a vector with the requirements of an *ideal* diet. (For this problem, it makes sense to assume that the entries in both $\mathbf{A}$ and $\mathbf{b}$ are non-negative.)

1. Does there always exist an ideal diet? In other words, does there always exist a solution $\mathbf{x} \geq \mathbf{0}$ such that $\mathbf{Ax} = \mathbf{b}$?

   No. Why? Give an example. Suppose food i is the only food with non-zero nutritional content for nutrients j and j'. In other words, $a_{hj} = a_{hj'} = 0$ for all $h \neq i$. Then $x_i = \frac{b_j}{a_{ij}} = \frac{b_{j'}}{a_{ij'}}$, which may not be the case. Such an instance would be *infeasible*.

2. If $\mathbf{Ax} \geq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$, say that $\mathbf{x}$ represents a *good* diet. Does there always exist a good diet?

   No. Why? Suppose the entries in row j are all zero (i.e. there is some nutrient not contained in any food), but the corresponding entry b_j is positive, then we can not meet nutrient requirement b_j.

3. Suppose that all the entries in $\mathbf{A}$ and $\mathbf{b}$ are strictly *positive*. Then is there always a good diet?

Suppose that we are also given the following caloric information for the n foods:

	food 1	...	food n
calories	c_1	...	c_n

From the set of good diets, find the diet with the minimum calories. This is an example of a *linear program*.

2 Linear Programming

A *linear program* is a problem where the objective is to minimize or maximize a linear function of a set of variables subject to linear inequalities and/or equalities, called *constraints*.

Given an $m \times n$ matrix $\mathbf{A}$, an m-dimensional vector $\mathbf{b}$ and an n-dimensional vector $\mathbf{c}$, all with entries in $\mathbb{R}$, the *linear programming* problem is to find a solution $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$ for the following:

$$\begin{aligned} & & \min & \quad \sum_{i=1}^{n} c_i x_i & \\ \text{subject to:} & \quad \sum_{i=1}^{n} a_{ji} x_i & \geq & \quad b_j, & j : 1 \leq j \leq m, \\ & \quad x_i & \geq & \quad 0, & i : 1 \leq i \leq n. \end{aligned}$$

Abbreviate this as follows:

$$\begin{aligned} \min \ & \mathbf{c^T x} \\ \mathbf{Ax} \ & \geq \mathbf{b} \\ \mathbf{x} \ & \geq 0. \end{aligned} \tag{P}$$

Equivalent to allowing $=$ and $\leq$ constraints. (Prove as exercise.)

Linear programming was studied in the 1930's by Kantorovich. In the 1940's, George Dantzig invented the simplex algorithm to solve linear programs. Shor, Nemirovski and Judin invented the ellipsoid algorithm in the 1960's and 70's. And in 1979, Khachiyan showed that the ellipsoid algorithm has polynomial complexity. In other words, he proved that linear programming is in the complexity class **P**.

Thus, we can use linear programming to find Polly's optimal diet. Can we use linear programming to solve more challenging problems, i.e. classical problems in discrete optimization?

2.1 Minimum Cut

Given a graph $G = (V, E)$, a *cut* is defined as a subset of vertices, $S \subset V$, $S \neq \emptyset$. The value of a cut S is denoted by $\delta(S, \bar{S})$. The *minimum s-t-cut* problem is to find the cut S such that $s \in S$ and $t \notin S$ that minimizes the value of $\delta(S, \bar{S})$.

In the following linear program, we have the following variables corresponding to edges and vertices:

- $\ell : E \to \mathbb{R}$, where $\ell(i, j)$ is the *length* of an edge ij,
- $p : V \to \mathbb{R}$, where $p(i)$ is the *potential* of a vertex i.

$$\begin{aligned} & \min \sum_{ij \in E} \ell(ij) \\ \text{subject to:} \quad & \ell(ij) \geq p(i) - p(j), \quad ij \in E, \\ & p(s) = 1, \\ & p(t) = 0 \\ & \ell(ij) \geq 0, \quad ij \in E. \end{aligned} \tag{$P_{min-cut}$}$$

Lemma 1. *The optimal value of* $(P_{min-cut})$ *equals the value of the minimum s-t-cut.*

Proof. Suppose G has a minimum s-t-cut with value k. This means that G is k-edge connected.

$\Rightarrow$ Assign all vertices in S value 1 and all those in $\bar{S}$ value 0. This shows that the solution to $(P_{min-cut})$ is at most k.

$\Leftarrow$ By Menger's Theorem, there exist k edge-disjoint paths between s and t. Each path has value at least 1. Thus, the solution to $(P_{min-cut})$ is at least k. □

2.2 Maximum Cut

Given a graph $G = (V, E)$, the *maximum cut* problem is to find a cut S such that $\delta(S, \bar{S})$ is maximized. How can we model this problem?

$$\begin{aligned} & \max \sum_{ij \in E} (z_{ij} + z_{ji}) \\ & z_{ij} \leq x_i, \\ & z_{ij} \leq 1 - x_j, \\ & x_i \in \{0, 1\}. \end{aligned} \tag{$P^1_{max-cut}$}$$

$(P^1_{max-cut})$ is an exact formulation for the maximum cut problem, but it is not a linear program, and thus not necessarily solvable in polynomial time. Suppose we relax the constraint $x_i \in \{0,1\}$ to $0 \leq x_i \leq 1$. Then we will obtain a linear program whose value is at least that of the maximum cut. Relaxing an integer constraint to obtain a linear program is called a *linear programming relaxation*.

Does this linear program model that maximum cut problem, as in the case of the minimum s-t-cut problem? No, because the maximum cut problem is NP-hard.

Why restrict ourselves to linear programs? Maybe we can use quadratic programs? Let us consider the following relaxation, i.e. non-integer program:

$$\begin{aligned} \max \sum_{(i,j)\in E} (x_i(1-x_j) + x_j(1-x_i)) \\ \text{subject to: } \quad 0 \leq x_i \leq 1, \quad i \in V. \end{aligned} \tag{$P^2_{max-cut}$}$$

Lemma 2. *The optimal value of* $(P^2_{max-cut})$ *equals the value of the maximum cut.*

Proof. Suppose the max-cut has size k.

$\Rightarrow$ Assign all vertices in S value 1, and the rest 0. Then the value of $(P^2_{max-cut})$ is at least k.

$\Leftarrow$ For any vertex x_i, try setting x_i to both 0 and 1. One of these choices does not decrease the objective function. Thus, the final result is a cut that is at least as large as $(P^2_{max-cut})$, which by definition, can not be greater than k. □

Since this is an exact relaxation, quadratic programming can not have a polynomial time algorithm! This is why we stick to linear programming for now. In future lectures, we will see other (vector) relaxations of quadratic programming called *semidefinite programming*.

3 Approximation

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.

3.1 What is an Approximation Algorithm?

An α-approximation algorithm A for a problem P is an algorithm that:

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

3.2 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.

3.3 Lower Bounds for Minimization Problems

How can we design an α-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 may be hard to 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 α-approximation algorithm is actually within factor α 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 α of this lower bound, then we can guarantee that our solution is within factor α 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.

3.3.1 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 3. *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 4. *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 4 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.

3.3.2 Example 2: Vertex Cover

Given a graph $G = (V, E)$, a *vertex cover* is a subset S of vertices such that for each edge $ij \in E$, either i or j belongs to S. The *vertex cover* problem is to find a vertex cover of minimize size. Vertex cover is NP-hard. How well can it be approximated? What is a good lower bound? One natural method for obtaining efficiently computable lower bounds for NP-hard problems is to use the linear programming relaxation of an integer program.

Consider the following linear programming relaxation for vertex cover:

$$\begin{aligned} \min \sum_{i \in V} x_i & \\ x_i + x_j \geq 1, \quad & ij \in E, \\ x_i \geq 0, \quad & i \in V. \end{aligned} \tag{P_{vc}}$$

Note that since this is a relaxation of the original (discrete) problem, an integral solution (valid vertex cover) is a feasible solution for (P_{vc}). Thus, the value of (P_{vc}) is a lower bound on the value of an optimal vertex cover.

Now we would like to use (P_{vc}) to obtain a feasible vertex cover (i.e. integer solution to (P_{vc})). Our goal is to *round* the fractional values $x_1, x_2, \ldots, x_n$ to obtain a vertex cover whose size is not too much larger than the value of (P_{vc}). To do this,

How can we round this LP to obtain a feasible vertex cover with a solution that is not too much larger than this lower bound? For this problem, we can take all the vertices whose corresponding $x_i \geq \frac{1}{2}$ and put them into the vertex cover. Since for each edge $ij \in E$, it is the case that at least one of x_i or x_j has value at least half, this is solution is a valid vertex cover. Finally, note that this solution has value at most twice the value of (P_{vc}), since each vertex contributes at most $2x_i$ to the vertex cover.

References

[Chv83] Vasek Chvatal. *Linear programming.* Macmillan, 1983.

[Sti45] George J. Stigler. The cost of subsistence. *Journal of Farm Economics*, 27(2):303–314, 1945.

These lecture notes are partly based on the following sources: lectures notes by Stéphan Thomassé from previous versions of the same course, and lectures notes from a course on Approximation Algorithms at EPFL (`http://theory.epfl.ch/osven/courses/Approx13`). The example in Section 1.1 is from [Chv83], and the example in Section 1.2 is from [Sti45].