IOE 691: Approximation Algorithms Date: 01/04/2017

Lecture Notes: Introduction and Minimum Spanning Tree

Instructor: Viswanath Nagarajan Scribe: Cheng Sun

1 Introduction

A discrete optimization problem involves picking the best solution out of finitely many possible solutions. For many problems, the number of possible solutions grows exponentially with the input size. So we need efficient algorithms to solve these problems. An approximation algorithm is a method that efficiently finds approximate solutions to such an optimization problem.

1.1 Basic Terminology

Let n denote the input size.

- **Running Time**. $T(n)$, the amount of time (or number of computation steps) that a "normal" physical computer would take to solve a certain computational problem using a certain algorithm. The Big-O notation is used to describe the worst case running time of an algorithm. Most natural algorithms are of two types. One has *polynomial* running time, i.e. $O(n^a)$ where a is a constant. The other has *exponential* running time, i.e. $O(e^{poly(n)})$.
- **NP-hard**. We don't expect that there is any algorithm that finds optimal solutions to an NP-hard problem in polynomial time. Many practical problems are NP-hard. For instance, the traveling salesman problem. Exact approaches to solve NP-hard problems take quite a long time and give optimal solutions. Whereas, heuristic approaches use significantly shorter time but give sub-optimal solutions.

type	running time	solution
exact approach	slow	optimal
heuristic approach	fast	sub-optimal

- **Approximation algorithm**. An approximation algorithm is a heuristic with performance guarantees. The running time is polynomial. The approximation ratio, $\alpha(n)$, describes the worst-case multiplicative gap from optimality. Note that n represents the input size, and we assume that the objective value is always positive. Formally, for an optimization problem seeking maximum value,

$$\alpha(n) = \min_{I \text{ input of size } n} \frac{ALG(I)}{OPT(I)}.$$

And for optimization problem seeking minimum value,

$$\alpha(n) = \max_{I \text{ input of size } n} \frac{ALG(I)}{OPT(I)}.$$

objective	**range of α**
minimization	$[1, \infty)$
maximization	$(0, 1]$

1.2 Course Outline

- greedy algorithm
- local-search
- dynamic programming
- linear programming
  - randomized rounding
  - primal-dual
  - iterated rounding
  - Lagrangian relaxation
- semidefinite programming

These methods will be used to solve problems such as facility location, scheduling, network design, and routing.

2 Minimum Spanning Tree

In this section, we will learn an algorithm that generates a minimum spanning tree. There will also be an analysis on why the algorithm given generates optimal solution. But first we need to know some basic concepts on minimum spanning tree.

2.1 Notations

Given undirected connected graph, $G = (V, E)$. V is the set of vertices, E is the set of edges. Connected means that there exists at least one path between arbitrary two vertices of the graph. There is a function $c : E \to \mathbb{R}$, showing the cost of each edge. Assume that there are m edges in the graph G.

Definition 2.1 *A **tree**, $T = (V_T, E_T)$, is an undirected graph in which any two vertices are connected by exactly one path.*

In other words, any acyclic connected graph is a tree.

Definition 2.2 *A **spanning tree**, $T = (V_T, E_T)$ in graph $G = (V, E)$ is a tree with $E_T \subseteq E$ that has all vertices covered, i.e. $V_T = V$.*

2.2 Greedy algorithm that generates MST

1. sort edges in non-decreasing order of cost, $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$.
2. Let $T_1 = \emptyset$
3. For $i = 1, 2, \cdots, m$, if $(T_i \cup \{e_i\})$ contains no cycle, let $T_{i+1} = T_i \cup \{e_i\}$

We can apply merge sort to assign indices to edges in the first step. Merge sort takes $O(m \log m)$ time. We can also apply quick sort that takes $O(m \log m)$ time on average. We can apply the following simple algorithm that tells whether there is a cycle in graph $T_i \cup e_i$ in step 3.

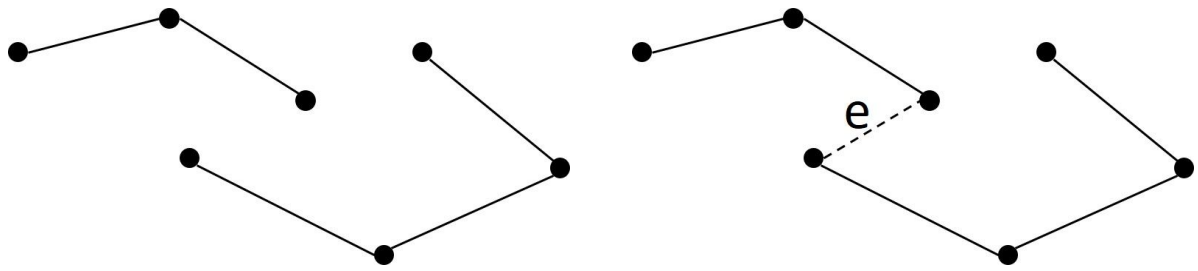
1. Get both ends of edge e_i, vertex a and b.
2. Make a list L that contains all vertices connected to a in T_i.
3. If vertex b is in L, there will be a cycle in $T_i \cup \{e_i\}$. If b is not in L, then $T_i \cup \{e_i\}$ is acyclic.

Note that the algorithm above takes $O(m)$ time. So the overall greedy algorithm runs in polynomial time (there are more efficient implementations as well).

2.3 Analysis

Lemma 2.1 *The greedy algorithm gives a tree T that covers all vertices in the graph.*

Proof: Given the graph is connected, we can prove this property by contradiction. If in the tree T generated by algorithm, not all the vertices are connected, there exist an edge $e \in E$ between two disconnected components of T (see Figure 1a). Because T is acyclic, and two components are not connected, we can declare that $T \cup \{e\}$ is also acyclic. Then according to the algorithm in 2.2, e must have been included. ■



(a) T (b) $T \cup \{e\}$

Figure 1: Lemma 2.1

To prove that the greedy algorithm gives a spanning tree with minimum cost, we will show the following lemma is true. Let $i = 1, 2, \cdots, m, m+1$ denotes the iteration in the algorithm, and $m+1$ denotes the end of the algorithm. T_i denotes the solution at the beginning of i^{th} iteration.

Lemma 2.2 *For each $i \in \{1, 2, \cdots, m, m+1\}$, there is a minimum spanning tree M with $T_i \subset M$.*

Proof: We prove this lemma by induction.

Step 1: $T_1 = \emptyset$, for any minimum spanning tree M, it is sure that $T_1 \subset M$.

Step 2: Prove that for $i \in \{1, 2, \cdots, m\}$, if there exist a minimum spanning tree M satisfying that $T_i \subset M$, there is also a minimum spanning tree M' satisfying that $T_{i+1} \subset M'$. There are two cases for T_{i+1}.

- $T_{i+1} = T_i$
  As there is a minimum spanning tree M with $T_i \subset M$, it is obvious that $T_{i+1} \subset M$.
- $T_{i+1} = T_i \cup \{e_i\}$
  - $T_{i+1} \subseteq M$. Then the induction holds.
  - $T_{i+1} \nsubseteq M$. Because every pair of vertices is connected with each other in M, we can be certain that $M \cup \{e_i\}$ contains a cycle (see Figure 2) including e_i. There must be an edge f in this cycle with index more than i: otherwise e_i would not be included in the greedy algorithm, and we can tell that $T_{i+1} = T_i$ (a contradiction). If we delete this edge f from $M \cup \{e_i\}$, what left is a new spanning tree M' (see Figure 2). Moreover, M' has lower cost than M:

$$cost(M') = cost(M \setminus \{f\} \cup \{e_i\}) = cost(M) - cost(f) + cost(e_i) \leq cost(M).$$

■

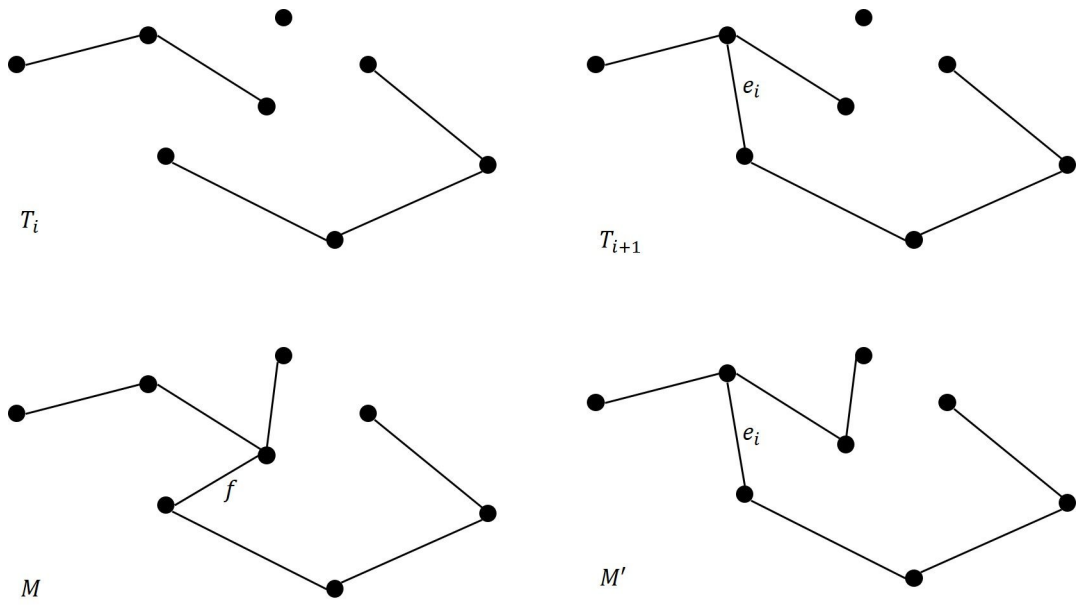


Figure 2: Lemma 2.2