Problem Set 14

Data Structures and Algorithms, Fall 2019

Due: December 26, in class.

From CLRS

Exercise 34.2-1, 34.2-5, 34.2-8, 34.3-2, 34.3-7, 34.4-3, 34.4-6, 34.5-2, 34.5-7 or 34.5-8 (you do not need to solve both, pick the one you prefer among these two). Problem 34-2.

This is the last problem set. The following bonus problems may require knowledge and/or techniques not covered in class. Feel free to try but do not be frustrated if you could not solve them. If you find some of them interesting, consider taking a class dedicated to that subject. Have fun!

Bonus Problem One (Approximation Algorithms)

For an undirected graph G, a matching is a set of edges such that no two edges in the set are incident on the same vertex. We have studied how to find a maximum matching in a bipartite graph. In this problem, we will look at matchings in general undirected graphs.

(a) A *maximal matching* is a matching that is not a proper subset of any other matching. Show that a maximal matching need not be a maximum matching by exhibiting an undirected graph G and a maximal matching M in G that is not a maximum matching. *(Hint: It's easy to find such examples.)*

(b) Consider an undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in G.

In the following, we concentrate on a polynomial-time approximation algorithm for maximum matching. Whereas the fastest known algorithm for maximum matching takes super-linear (but polynomial) time[1], the approximation algorithm here will run in linear time. You will show that the linear-time greedy algorithm for maximal matching in part (b) is a 2-approximation algorithm for maximum matching.

(c) Show the size of a maximum matching in G is a lower bound on the size of any vertex cover for G.

(d) Consider a maximal matching M in G, show that $2|M|$ is the size of a vertex cover for G.

(e) Using parts (c) and (d), prove that the greedy algorithm in part (b) is a 2-approximation algorithm for the maximum matching problem.

Bonus Problem Two (Probabilistic Method)

Consider the problem of finding a large cut in an undirected graph. A cut is a partition of the vertices into two disjoint sets, and the *value* of a cut is the weight of all edges crossing from one side of the partition to the other. Here we consider the case where all edges in the graph have the same weight 1. The problem of finding a maximum cut is NP-hard.

(a) Show that the value of the maximum cut must be at least $1/2$ the number of edges in the graph.

[1] `https://en.wikipedia.org/wiki/Blossom_algorithm`

(b) Devise a deterministic polynomial time algorithm that can find a cut with a value of at least $1/2$ the number of edges in the graph. (Notice that this algorithm would also be a $1/2$ approximation algorithm for the maximum cut problem.)

Bonus Problem Three (Randomized Algorithms)

Consider a simple distributed system wherein agents contend for resources but "back-off" in the face of contention. Balls represent agents, and bins represent resources.

The system evolves over rounds. Every round, balls are thrown independently and uniformly at random into n bins. Any ball that lands in a bin by itself is *served* and removed from consideration. The remaining balls are thrown again in the next round. We begin with n balls in the first round, and we finish when every ball is served.

In the real world, a similar scheme—known as "exponential back-off"—is used in various computer networks (such as Ethernet networks) to resolve contention.

(a) If there are b balls at the start of a round, what's expected number of balls at the start of next round?

(b) Suppose that every round the number of balls served was exactly the expected number of balls to be served. Show that all the balls would be served in $O(\log \log n)$ rounds. *(Hint: If x_j is the expected number of balls left after j rounds, show and use that $x_{j+1} \leq x_j^2/n$.)*

(c) Can you show that all balls will be served in $O(\log \log n)$ rounds, with high probability? (That is, with probability at least $1 - 1/n$.)

Bonus Problem Four (Online Algorithms)

You are going skiing for an unknown number of days. Assume that renting skis costs 1 dollar per day and buying skis costs 10 dollars. Every day you have to decide whether to continue renting skis for one more day or buy a pair of skis. If you know in advance how many days you will go skiing, you can decide your minimum cost. For example, if you will be skiing for more than 10 days it will be cheaper to buy skis, while if you will be skiing for fewer than 10 days it will be cheaper to rent.

Formally, the problem can be set up as follows. There is a number of days d (unknown to you) that you will ski. We are looking for an algorithm that minimizes the ratio between what you pay using the algorithm (that does not know d) and what you would pay optimally if you knew d in advance. This ratio is often known as the *competitive ratio*. The problem is generally analyzed in the worst case, where the algorithm is fixed and then we look at the worst-case performance of the algorithm over all possible d. In particular, no assumptions are made regarding the distribution of d.

(a) Devise a deterministic algorithm, what is the (worst-case) competitive ratio of your algorithm?

(b) Suppose you can use randomization, devise a randomized algorithm, what is the (worst-case) expected competitive ratio of your algorithm? (Expectation is over the randomness of your algorithm, *not* over the distribution of d.)

Bonus Problem Five (Distributed Algorithms)

Consider a network containing n computers, some of these computers are connected by cables. Therefore, we can use an undirected graph $G = (V, E)$ to model the network, where $|V| = n$ and there is an edge between two vertices iff there is a cable connecting the two corresponding computers. Assume G is connected. For each edge $e \in E$, we associate a weight $w_e \in \mathbb{R}^+$ with it.

Initially, computers in the network do not know the topology of the network. That is, they do not know how G looks like. However, for each computer, it does know the edges incident to it (i.e., the cables connected to it), and the weights of these edges.

Two computers can communicate with each other if there is a cable connecting them. Specifically, the system evolves over rounds. At the beginning of each round, for each computer, for each of the cable connected to it, it can send a message to the computer at the other end of the cable over think link. By the end of this round, for each computer u, for each of the cable connected to it, if the computer v at the other end of the cable sent a message $m_{v\to u}$ over this link at the beginning of this round, computer u will receive the message $m_{v\to u}$.

Now, suppose the computers want to compute a minimum spanning tree of G. Try to devise an algorithm and analyze how many rounds your algorithm needs. (Notice, your algorithm would be a *distributed* algorithm. That is, each computer in the network will run this algorithm in parallel. G should *not* be the input of your algorithm, since no computer knows the entire topology initially. However, recall that each computer does know the cables connected to it and the weights of them. Moreover, computers can communicate with each other in each round to gradually gain more information regarding G.)