**CSE 202: Design and Analysis of Algorithms** **(Due: 11/16/19)**

# Homework #3

*Instructor:* Ramamohan Paturi *Name:* Shihan Ran, *Netid:* A53313589

**Problem 1: Graph cohesiveness**

**Problem Description**

In sociology, one often studies a graph $G$ in which nodes represent people and edges represent those who are friends with each other. Let's assume for purposes of this question that friendship is symmetric, so we can consider an undirected graph.

Now suppose we want to study this graph $G$, looking for a "close-knit" group of people. One way to formalize this notion would be as follows. For a non-empty subset $S$ of nodes, let $e(S)$ denote the number of edges in $S$-that is, the number of edges that have both ends in $S$. We define the *cohesiveness* of $S$ as $e(S)/|S|$. A natural thing to search for would be a set S of people achieving the maximum cohesiveness.

**(Subproblem 1)**

Give a polynomial-time algorithm that takes a rational number $\alpha$ and determines whether there exists a set $S$ with cohesiveness at least $\alpha$.

***Solution***

**(High-level description)**

We represent $G$ as $(V, E)$. Then we constructed a directed graph $G' = (N, E')$. It has vertex $N_v$ for every vertex $v \in V$, vertex $N_e$ for every edge $e \in E$, and two distinguished vertex source $s$ and sink $t$.

Assume $c(N_i, N_j)$ is the capacity of edge $(N_i, N_j)$. We let $c(s, N_e) = 1$ for every edge $e \in E$, $c(N_v, t) = \alpha$ for every vertex $v \in V$, $c(N_e, N_i) = c(N_e, N_j) = \infty$ if $e = (i, j) \in E$ and $i, j \in V$.

To determine whether there exists a set $S$ with cohesiveness at least $\alpha$ is to determine if there is an $s$-$t$ cut in graph $G'$ of capacity at most $|E|$. It's a minimum cut problem. We use the maximum-flow algorithm (e.g. Ford-Fulkerson) to find an $s$-$t$ cut of minimum capacity and compare it with $|E|$.

**(Correctness)**

We define an $s$-$t$ cut as a partition $(A, B)$ of N with $s \in A, t \in B$. The above-defined network $G'$ has the following properties:

- The source node $s$ has $|E|$ outgoing edges and their capacity is all equal to 1. So the capacity of the min cut should be less than or equal to $|E|$ (we can simply get this value by letting A=$s$).
- Since $\alpha$ is a rational number. We can transform our graph by multiplying all edge weights by the same constant factor $\beta$, where $\beta$ is defined to be the smallest positive number such that $\alpha\beta$ is an integer. Then, all capacities are scaled to integers.
- If $N_e$ is in $A$ where $e = (i, j)$, then $N_i$ and $N_j$ should also be in $A$. This is because $c(N_i, N_j) = \infty$ and infinite capacity edge can not cross a min cut since our min cut is less than or equal to $|E|$. Similarly, if node $N_v$ is in $B$, then all $N_e$ where $e = (v, j)$ for $v, j \in V$ must also be in $B$.

We use $A_e$ to denote the vertices $N_e$ in $A$ and $A_v$ to denote the vertices $N_v$ in $A$. Then $|A_e|$ should be exactly the number of edges in the original graph $G$ that have both endpoints in $A_v$, and according to the problem description, there are $e(A_v)$ such edges. The edges that cross our cut are

- All edges $(N_v, t)$ for $N_v \in A_v$, each with the capacity $\alpha$. There are $|A_v|$ edges.
- All edges $s, N_e$ for which $e = (i, j)$ and $i, j \notin A_v$, each has a capacity 1. There are $|E| - e(A_v)$ edges.

Hence, the total capacity of our cut is $c(A, B) = \alpha|A_v| + |E| - e(A_v)$. We can arrange this to get

$$|E| - c(A, B) = e(A_v) - |A_v|\alpha$$

According to the problem description, we can see that $e(A_v)/|A_v| \geq \alpha$ iff $|E| - c(A, B) \geq 0$. That is $c(A, B) \leq |E|$.

So we've prove that we have a min cut with $c(A, B) \leq |E|$ then we can find a group of vertices in our original graph with cohesiveness greater than $\alpha$.

**(Subproblem 2)**

Give a polynomial-time algorithm to find a set $S$ of nodes with maximum cohesiveness.

**(High-level description)**

There are $|V|$ choices for the size of $A_v$ and $C_2^{|A_v|}$ choices for $e(A_v)$. Thus, it's a total of $O(|V|^3)$ possible values for $\alpha$. We can find the set of maximum cohesiveness by binary search on $\alpha$ by checking if there exists a subset with cohesiveness greater than a given value.

**(Time complexity)**

By using the maximum-flow algorithm to determine the minimum cut for a given $\alpha$, the final time complexity should be $O(\log |V|(|V| + |E|)^3)$.

**Problem 2: Business plan**

**Problem Description**

Consider the following problem. You are designing the business plan for a start-up company. You have identified $n$ possible projects for your company, and for, $1 \leq i \leq n$, let $c_i > 0$ be the minimum capital required to start the project $i$ and $p_i > 0$ be the profit after the project is completed. You also know your initial capital $C_0 > 0$. You want to perform at most $k$, $1 \leq i \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.

In other words, you want to pick a list of up to k distinct projects, $i_1, \ldots, i_{k'}$ with $k' \leq k$. Your accumulated capital after completing the project $i_j$ will be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$. The sequence must satisfy the constraint that you have sufficient capital to start the project $i_{j+1}$ after completing the first $j$ projects, i.e.,$C_j \geq c_{i_{j+1}}$ for each $j = 0, \ldots, k' - 1$.You want to maximize the final amount of capital, $C_{k'}$.

***Solution***

**(High-level description)**

We use the greedy algorithm to solve this problem.

According to the problem description, we have the following observations:

1. We can only perform at most $k$ projects before the IPO.

2. We want to maximize our total capital. Each project has a $p_i$ as the profit after the project is completed. Our accumulated capital after completing the project $i_j$ should be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$.

3. We should have sufficient capital to start the project $i_{j+1}$ after completing the first $j$ projects.

Intuitively, if we are to write a greedy algorithm for a problem like this, we might first consider choosing the projects based on its profit, i.e., we choose the project that has the highest profit each time. However, we should still consider another thing: we should have sufficient capital to start this profitable project. We now formally describe our algorithm:

Sort projects based on its minimum capital required to start the project. When choosing the $j$-th project, we should choose the most profitable project among all the projects $i_{j+1}$ where $c_{i_{j+1}} \leq C_j$. After chosen $k$ projects, terminate and return the final capital as well as what we've chosen.

**(Correctness)**

We use the exchange argument to prove the correctness of this algorithm.

Imagine an optimal solution OPT and our greedy solution is G. For ease of comparison, we sort projects in both solutions by increasing profits. For the purpose of comparison, we will consider projects to be equal if they have the same profit and start capital (so duplicate projects are equal to each other). Now for project $o_i$ and $g_i$, we account for the following two cases:

1. $o_i = g_i$: The solutions agree, so we can move on.

2. $o_i \neq g_i$: We argue that this is not possible. Suppose there exists an optimal solution OPT in which $o_i \neq g_i$. Then, because the greedy algorithm prioritizes projects with higher profit when the start capital is sufficient. And after the most profitable project has been done, the current capital $C_i$ should also be the highest, which will leave more choosing space for the next project. Hence, it is impossible for OPT (or any other solution) to have a higher capital than G. Therefore, this case can only lead to OPT to be worse than G. However, this would contradict our knowledge that OPT is optimal. By proof by contradiction, there is no optimal solution in which $o_i \neq g_i$.

Therefore, we have proven that any optimal solution is necessarily equal to the greedy solution.

**(Time complexity)**

The initial sort is $O(n \log n)$. Each iteration afterward can be done by $O(\log n)$ if we maintain a max-heap data structure to help us choose the most profitable project under the condition $c_{i_{j+1}} \leq C_j$. And we do that iteration for $k$ times, i.e., this takes $O(k)$ time complexity.

Overall, our algorithm can be completed in $O(n \log n + k \log n) = O(n \log n)$.

## Problem 3: Minimum Cost Sum

### Problem Description

You are given a sequence $a_1, a_2, \ldots, a_n$ of nonnegative integers, where $n \geq 1$. You are allowed to take any two numbers and add them to produce their sum. However, each such addition has a cost which is equal to the sum. The goal is to the find the sum of all the numbers in the sequence with minimum total cost. Describe an algorithm for finding the sum of the numbers in the sequence with minimum total cost. Argue the correctness of your algorithm.

***Solution***

**(High-level description)**

We use the greedy algorithm to solve this problem.

According to the problem description, we observe the fact that the larger number sum, the larger cost to do an addition. Hence, intuitively, if we are to write a greedy algorithm for a problem like this, we might first consider choosing the two numbers based on its sum. We now formally describe our algorithm:

Sort the sequence $A = a_1, a_2, \ldots, a_n$. Then we need to form a Huffman tree with all the numbers are the leaves with the same order as the sorted sequence $A'$. The key idea is each time, we choose to do the addition among the smallest two numbers. For $n$ numbers, we need to perform addition for $n-1$ times. Hence, at time step $i$, we will extract two smallest numbers from the Huffman tree, do an addition, then we add their sum to the Huffman tree. After iterating $n-1$ times, the final minimum value of the Huffman tree is the sum of the numbers in the sequence with the minimum total cost.

**(Pseudo Code)**

**Algorithm 1:** Minimum Cost Sum

**Input:** the sequence $A$
**Output:** sum of the numbers

```
sort A;
Construct a priority queue Q based on the sorted A;
for i = 1 to length(A) − 1 do
    z = new node;
    z.left = Extract − Min(Q);
    z.right = Extract − Min(Q);
    z.value = z.left.value + z.right.value;
    Insert(Q, z);
return Extract − Min(Q);
```

**(Correctness)**

With each sorted $a_i$ ordered in the leaves of the Huffman tree, we can simply cite the correctness proof of the Huffman Coding from here.

**(Time complexity)**

The sorting takes $O(n \log n)$ time. The binary Huffman tree is constructed using a priority queue and each priority queue operation takes time $O(\log n)$. We need to perform addition for $n-1$ times.

Overall, our algorithm takes $O(n \log n)$ time complexity.

**Problem 4: Cellular network**

**Problem Description**

Consider the problem of selecting nodes for a cellular network. Any number of nodes can be chosen from a finite set of potential locations. We know the benefit $b_i \geq 0$ of establishing site i. However, if sites i and j are selected as nodes, then the benefit is offset by $c_{ij}$, which is the cost of interference between the two nodes. Both the benefits and costs are non-negative integers. Find an efficient algorithm to determine the subset of sites as the nodes for the cellular network such that the sum of the node benefits less the interference costs is as large as possible.

Design an efficient polynomial-time algorithm.

Provide a high-level description of your algorithm, prove its correctness, and analyze its time complexity.

***Solution***

**(High-level description)**

**(Pseudo Code)**

**(Correctness)**

**(Time complexity)**