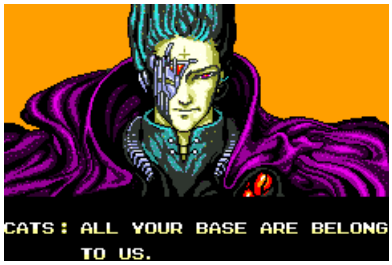
EE360C: Algorithms
University of Texas at Austin
Dr. David Soloveichik

Homework #8
Due: November 29, 2018 (*in-class quiz*)

Homework #8

You do not need to turn in these problems. The goal is to reinforce what we learned in class, as well as to cover material we didn't have time to cover in class. The in-class quiz that will cover the same or similar problems. Material on homework can also appear on exams.



Problem 1: All your base are belong to us

NOTE: Problem 1 is part of Programming Assignment 3. Consequently, there will NOT be a solution posted for Problem 1 with the rest of this homework.

You are on the planning committee of a major telecommunication company and you are given the task of setting up k base station antennas in a town with n houses along a line. Being an electrical engineer, you know that the further an antenna is from the house it covers, the more power it wastes. You'd like to minimize the maximum distance that your antennas need to cover, ensuring that you do not want to leave any house uncovered.

More precisely, you are given a set X of n houses sorted by position along a line, i.e, House 1 is at x_1, House 2 is at x_2 and so on with $x_1 < \cdots < x_n \in \mathbb{R}$. You are also given an integer k, the number of antennas. You need to find the set of base station positions, C, of k points $c_1, ..., c_k \in \mathbb{R}$ that minimizes the antenna range. The antenna range is defined as the minimum distance r such that every x_i is at most r from some c_j. Note that all base stations are identical and cover the same distance. In other words, the antenna range is:

$$r = \max_{1 \le i \le n} \left(\min_{1 \le j \le k} |x_i - c_j| \right)$$

where $\min_{1 \le j \le k} |x_i - c_j|$ represents the distance from point x_i to the closest base station, and thus a lower bound on the antenna range required by that base station, and the max condition identifies the required antenna range—the largest distance between any house and its closest base station.

Hint: The problem might make you scratch your head, but it isn't as difficult as it seems! Think of the possible subproblems that could be used. You could construct a $n \times k$ table indexed $r[t, j]$ that stores the optimal antenna range for the subproblems and another indexed $c[t, j]$ that stores a set of base stations for the subproblems. What can these sub-problems represent?

Problem 2: In it to Win it

Consider a row of n coins of values $V(1) \cdots V(n)$ where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. We consider two ways that the opponent can behave:

(a) Assume the opponent is greedy and always simply chooses the larger of the two coins.

(b) Assume the opponent "plays optimally" and chooses the coin which maximizes the amount of money they can win.

Give a dynamic programming algorithm for cases (a) and (b) to determine the maximum possible amount of money we can definitely win (for the entire game) if we move first.

(Hint: Let $W[i, j]$ be the maximum value we can definitely win if it is our turn and only coins $i \cdots j$, with values $V(i) \cdots V(j)$, remain).

Solution

We can express $W[i, j]$ by taking the maximum between: (1) the left coin's value plus the subproblem after player one takes the left coin, and (2) the right coin's value plus the subproblem after player one takes the right coin. Our base case is the following: for each $W[i, j]$ such that $j - i = 1$ (i.e., there are only two coins), $W[i, j] = \max(V(i), V(j))$. Intuitively, if there are two coins and it is our turn, the most we can definitely win is achieved by taking the larger valued coin.

Subproblem (a). Player two always chooses the larger coin. After player one takes a coin, we know which coin player two will take, so we know which subproblem's solution to use. For example, if player one takes the left coin i, player two will choose $i + 1$ or j depending on which value is greater. In general, we can write $W[i, j] = \max(L, R)$ where

$$L = V(i) + \begin{cases} W[i+2, j] & \text{if } V(i+1) > V(j) \\ W[i+1, j-1] & \text{if } V(i+1) < V(j) \end{cases}$$

$$R = V(j) + \begin{cases} W[i+1, j-1] & \text{if } V(i) > V(j-1) \\ W[i, j-2] & \text{if } V(i) < V(j-1) \end{cases}$$

Subproblem(b). Player two chooses the coin which maximizes the value they can definitely win. The key is to note this choice is the same as taking the coin which minimizes the value that player one can definitely win. Then we can write $W[i, j] = \max(L, R)$ where

$$L = V(i) + \min(W[i+1, j-1], W[i+2, j])$$

$$R = V(j) + \min(W[i, j-2], W[i+1, j-1])$$

Note that the problem assumes an even number of coins to begin with. For both subproblems, each solution for a set of coins is defined using a subproblem with two less coins. Thus, we eventually reach our base case of two coins.

Given n coins, construct an $n \times n$ matrix to store the $W[i, j]$. Assuming the appropriate entries are already set, we can compute $W[i, j]$ in constant time. Note that the number of coins in the sub-game with solution $W[i, j]$ is $j - i + 1$. We are assuming player one takes turns when there are an even number of coins, so we need only set entries $W[i, j]$ where $j - i$ is odd. For all i, j such that $j - i = 1$ (games of two coins), we can set $W[i, j] = \max(V(i), V(j))$ (player one's best choice is to take the larger of the two coins). This takes $\mathcal{O}(n)$ time. Note that $W[i, j]$ for a sub-game consisting of c coins is defined using only sub-games of $c - 2$ coins. Thus, we can fill each diagonal in the matrix where $j - i = k$ (games of $k + 1$ coins) given that we have already set the diagonal where $j - i = k - 2$ (games of $k - 1$ coins). For example, to get us started, we can fill in the n entries where $j - i = 3$ (games of four coins) using the entries where $j - i = 1$ (games of two coins), which we filled in as our base case. Each such diagonal takes $O(n)$ time. To reach our final solution $W[1, n]$, we must fill in n diagonal sets of entries, each taking $\mathcal{O}(n)$ time, thus our runtime is $\mathcal{O}(n^2)$.

Problem 3: Natural Disasters

Consider the following scenario. Due to a large-scale natural disaster in a region, a group of paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hours driving time of their current location (so different people will have different options for hospitals, depending on their locations). At the same time, we dont want to overload any one of the hospitals by sending it too many patients. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is balanced. i.e Each hospital receives at most $\lceil n/k \rceil$ people. Given the information about the people's locations, and we want to determine whether this is possible. Describe how this problem can be framed as a network flow problem and how the Ford-Fulkerson algorithm can be used to solve it.

Solution

Build a network with a node p_i for each patient i, a node h_j for each hospital j, and an edge from p_i to h_j with capacity of 1 if the patient i is within a half-hour's driving time of hospital j. Connect each of the patients to a super-source s and connect each of the hospitals to a super-sink t such that the capacity of each edge (s, p_i) is 1 and the capacity of each edge (h_j, h) is $\lceil n/k \rceil$). We claim that there is a way to send all the patients to the hospitals iff there is an $s - t$ flow of value n. If there is a way to send the patients, then we send one unit of flow from s to t along s to p_i to h_j to t, where patient i is sent to hospital j. No hospital can have more than $\lceil n/k \rceil$ patients due to the capacity of each edge from h_j to t. This graph has $O(n + k)$ nodes and $O(nk)$ edges, and the capacity out of s is n, so the running time is $O(nk^2)$ using Ford-Fulkerson.

Problem 4: Deleting Edges

Consider the following problem. You are given a flow network with unit capacity edges: It consists of a directed graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$; and $c_e = 1$ for all $e \in E$. You are also given a parameter k.

The goal is to delete k edges so as to reduce the maximum s-t flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum s-t flow in $G' = (V, E - F)$ is as small as possible subject to this.

Give a polynomial time algorithm to solve this problem. Argue (prove) that your algorithm does in fact find the graph with the smallest maximum flow.

Solution

The big idea is that all the edges across a min-cut must be used in a max-flow. Thus removing these edges is sure to reduce the flow. Note that edges NOT along the min-cut could possibly be "redundant" and that the flow through them could possibly be "re-routed" along other edges.

Run Ford-Fulkerson (FF) to find the max-flow. Identify a min-cut: perform a BFS/DFS on the residual graph given by FF, and identify all reachable nodes. Any edge from a reachable node to an unreachable node in the residual graph must be in the min-cut. Find k edges that cross the cut and delete them. You have reduced the capacity of this cut by k. This implies that you have reduced the flow in the original graph by k as well (because we started with value of flow = capacity of min-cut, and the value of the flow must be no more than the capacity of any cut). Note that removing a capacity 1 edge can at most reduce the flow by 1. Thus this algorithm is optimal. (If k such edges do not exist, just delete all of the edges that cross the cut. You have reduced the flow in the original graph to 0, which is also optimal.)