Homework #11

You should try to solve these problems by yourself. I recommend that you start early and get help in office hours if needed. If you find it helpful to discuss problems with other students, go for it. **The goal is to be ready for the in class quiz that will cover the same or similar problems.**

Problem 1: Planning a Company Party

You are consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a *conviviality* rating, which is a real number. In order to make the party the most fun for all of the attendees, the president does not want both an employee and his or her immediate supervisor to attend.

You are given the tree that describes the structure of the corporation, where each node represents an employee, and a node's children represent the employees under his or her direct supervision. Each node of the tree holds, in addition to the child pointers, the name of the employee and the employee's conviviality rating.

Give a dynamic programming algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

Hints:

- Construct your solution as a *binary choice*.
- Even the president of the company may end up not invited if that turns out to be optimal.
- Argue that the problem exhibits optimal substructure, define the optimal solution via recursion, provide a description or pseudocode of the bottom-up algorithm, and analyze the running time.

Solution

First, the problem has optimal substructure. That is, an optimal solution for the entire tree contains within it optimal solutions for subtrees. Consider an optimal solution that includes some node i but none of the employees under i's direct supervision (because that wouldn't be a correct solution, let alone an optimal one). We can break this solution into components, one rooted at each of i's *grand-children* (i.e., employees under the direct supervision of employees under i's direct supervision). I claim that the sub-solution created by breaking our original optimal solution in this way is an optimal solution to the subproblem rooted at this node. The proof is simple; if it wasn't optimal, I could replace it with a sub-solution that WAS optimal, combine it with the other subproblems, and end up with a MORE optimal solution than the original one. Which is a contradiction.

The binary choice here is whether or not a particular employee is invited to the party. We will create an array of size n (the number of employees in the company), where the president is entry n in the array. The first entries in the array are employees at the bottom "level" of the tree; employees with no others under their supervision are leaves. I start by making a list of the employees to give them indices into this array; I do this with a breadth first search starting at the president, then I reverse the resulting list (so that the president is at the end and the leaves are at the beginning. This took $O(n)$ time. Then I start at the beginning of this list and use the list in conjunction with the tree that I was given. I assume c_i refers to the conviviality rating of employee i in this list (i.e., c_n is the conviviality rating of the president).

Solution

The optimal solution, defined via recursion, is:

$$C[i] = \max(c_i + \sum_{g \in \text{grandchildren}} C[g], \sum_{g \in \text{children}} C[g])$$

This gives me the *value* of the optimal solution; I can also create a second array $I[i]$ (for "invite"), which contains a 1 in location i if the first choice was the max and a 0 if the second choice was the max. This is *NOT* the guest list. See below to create the guest list. The pseudocode for the bottom up algorithm is the following (the below assumes that if there are no children (or grandchildren) then the sum of their conviviality ratings is 0):

```
PARTY
1  for i = 1 to n
2        do C[i] = max(c_i + Σ_{g∈grandchildren} C[g], Σ_{g∈children} C[g])
3           if c_i + Σ_{g∈grandchildren} C[g] > Σ_{g∈children} C[g]
4              then I[i] = 1
```

This gets the C and I tables filled in; the following tells us how to create the guest list:

```
INVITE(i)
1  if I[i] = 1
2     then output i
3          for every grandchild g of i
4               do INVITE(g)
5  if I[i] = 0
6     then for every child c of i
7               do INVITE(c)
```

The party procedure fills in n entries in the C and I tables iteratively; filling in each entry takes $O(n)$ time (it just looks up entries in the data structure or tables and does simple arithmetic, but the sum may be over $O(n)$ other elements). Therefore the overall running time to compute the value of the optimal solution is $O(n^2)$. To print out the guest list, each entry in I is visited at most once, and constant work is done on each visit. This is an overall running time of $O(n)$ as well.

Problem 2: 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 $[n/k]$ people. Give a polynomial-time algorithm that takes the

given information about the peoples locations and determines whether this is possible.

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 $floor([n/k])$. 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 from s to p_i to h_j to t, where patient i is sent to hospital j. No hospital can have more than n/k patients due to the capacity of each edge from h_j to t. This graph has $O(n+k)$ nodes and $O(nk)$ edges so the running time is $O(n^2)$ using Ford-Fulkerson.

Problem 3: 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 $s \in V$, and 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

Run the Ford-Fulkerson capacity scaled algorithm to find the max flow. Perform a graph search (BFS/DFS) from the source to identify the min cut. Find k edges that cross the cut and delete them. You have reduced the flow in the original graph by k. 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.

Problem 4: Updating a Maximum Flow

Suppose you are given a flow network $G = (V, E)$ with source s, sink t, integer capacities, and a maximum flow, f of G.

1. We increase the capacity of a single edge $(u, v) \in E$ by one. Give a $O(m+n)$ time algorithm to update the maximum flow.

Solution

The edge that we added either crossed the min cut or it didn't. If it didn't cross the min cut, then the value of the min cut is the same, and the value of the max flow is the same, so f is a max flow in G'. If the edge crossed the min cut, then the value of the old min cut went up by one. If there was another min cut of the same value, then the value of the max flow remains the same, and f is still a max flow. If the old min cut's value went up by one and it is still the min cut, then we need to find the new max flow, but we know the value of the new max flow will have increased by at most one. We can do this by executing the inner loop of the Ford-Fulkerson algorithm once on the residual graph G_f. If there is no augmenting path in G_f, then the max flow's value did not change. If there is an augmenting path, adding one unit of flow along it in f will create the new max flow. After we've found and augmented this path, we have the new max flow, and there are no remaining augmenting edges.

2. We decrease the capacity of a single edge $(u, v) \in E$ by one. Give a $O(m + n)$ time algorithm to update the maximum flow.

Solution

Let f be the maximum flow before reducing the capacity on edge (u, v). If f did not push flow equal to the capacity of (u, v) across (u, v), then f is still a max flow. Otherwise, we need to update the max flow. We define $f'(x, y) = f(x, y)$ for all $x, y \in V$ except that $f'(u, v) = f(u, v) - 1$. But now this new flow violates the conservation constraint at u (unless u is s) and v (unless v is t). We first try to reroute this flow so that it goes from u to v without going through the edge (u, v). So we search the graph $G_{f'}$ for an augmenting path from u to v (instead of from s to t). If there is such a path, augment the flow along that path (this will be possible with at most one unit of flow). If there is no such path, then we have to reduce the flow from s to u by augmenting the flow from u to s. (Such a path must exist since there is flow from s to u. Similarly we have to reduce the flow from v to t by finding an augmenting path from t to v.