CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Homework #4 Solutions

November 10, 2019

Problem 1. You are given a tree T on n nodes and each node a of T is given some points, denoted by $Points[a]$. The goal in this problem is to choose a subset of the nodes that maximize the total number of points achieved while ensuring that no two chosen nodes are neighbors to each other. Design an $O(n)$ time algorithm for this problem that outputs the *value* of maximum number of points we can achieve.

You may assume that the input is specified in the following way: (1) we have a pointer r to the root of the tree (we can root the tree arbitrarily); and (2) for each node a of the tree, we have a linked-list to the child-nodes of this tree in $Child[a]$ ($Child[a] = NULL$ for every leaf-node a). **(25 points)**

Solution. We will solve this problem using Dynamic Programming.

Specification (in plain English) : For every node $a \in T$, define :

- $Val(a)$: The *maximum points* that can be achieved from the sub-tree rooted at a while ensuring that no two chosen nodes are neighbors of each other.

The answer to the problem is then $Val(r)$ where r is the root of the tree (follows from the definition).

Recursive Formula : We write the following recursive formula for every node $a \in T$:

$$Val(a) = \begin{cases} Points[a] & a \text{ is a leaf-node, i.e., } Child[a] = NULL \\ \max\left\{Points[a] + \sum_{b \in Child[a]} \sum_{c \in Child[b]} Val(c), \sum_{b \in Child[a]} Val(b)\right\} & \text{otherwise} \end{cases}.$$

The proof of correctness is as follows: the base case corresponds to a being a leaf-node. When a is a leaf, the best (only) thing we can do is to pick a in the solution (since $Points[a] \geq 0$) and hence $Val(a) = Points[a]$.

Let $T(a)$ represent the sub-tree rooted at a. If a is not a leaf-node, then the maximizing subset for $T(a)$ either includes a or it does not. If the maximizing subset contains a, then it cannot contain any child-node of a (as they are neighbors). At the same time, any combination of valid solutions in the sub-trees rooted at child-nodes of child-nodes of a ("grandchildren" of a) is a valid solution as those sub-trees cannot be neighbor to each other or a – this means that after picking a we should also pick the best solution in these sub-trees, resulting in $Points[a] + \sum_{b \in Child[a]} \sum_{c \in Child[b]} Val(c)$ points in total.

If the maximizing subset does not contain a, then we should simply pick the best solution in the combination of child-nodes of a (exactly by the same argument above), which results in $\sum_{b \in Child[a]} Val(b)$ points. Finally, by picking the maximum of these two numbers, we obtain the best of both options.

Runtime Analysis: We have n subproblems and in each subproblem rooted at a, an edge e will be traversed once when we go to the child-nodes of a, and another time while we go to the child-nodes of child-nodes of a. Since the total number of edges in a tree is $n - 1$, the total runtime would be $O(n)$.

Problem 2. You are in charge of providing access to a library for every citizen of a far far away land. A long time ago, this land had a collection of (bidirectional) roads between different pairs of cities. However,

over time these roads all got destroyed one way or another. The ruler of this land now asks for your help to determine which of these roads to repair and where to build the libraries: the objective is that after this, every city should either contain a library or the people in this city should be able to travel to another city that contains a library using these repaired roads. You are also told that repairing a road has a cost of R, and building a library has a cost L.

Design an algorithm that given n cities, the specification of m (damaged) roads that used to connect two of these cities together, and the parameters R and L, outputs the list of roads that needs to be repaired plus the name of the cities in which a library should be built, while minimizing the total cost. **(25 points)**

Solution. *Algorithm.* We create a graph with a vertex for every city and an edge between two cities if there is a (damaged) road between them. Then, if $R \geq L$, we build a library at *every* vertex of this graph. Otherwise, we run DFS on the graph to find a spanning tree of each connected component of this graph. We then repair all the roads in these spanning trees and then build one library per each connected component of the graph (in an arbitrary vertex of the component).

Proof of correctness. We first claim our algorithm is *feasible*, that is, in the output, every city is connected to a library. This s definitely true when $R \geq L$ as we build a library in every city. In the other case, each connected component has exactly one library and since we repaired the edges of a spanning tree of this component, every vertex can reach every other vertex in the component and hence can reach the library.

We now prove *optimality*, that is, the cost of our solution is minimized. Consider any solution which builds ℓ libraries. In order for this solution to be feasible, the repaired graph must have at most ℓ components, or else some component would be without a library. If the repaired graph has at most ℓ components, then at least $n - \ell$ roads were repaired, since repairing a road decreases the number of components by at most 1. Thus any solution which builds ℓ libraries must cost at least $(n - \ell) \cdot R + \ell L = n \cdot R + \ell \cdot (L - R)$.

When $R \geq L$, our algorithm gives a solution of cost $n \cdot L$, which is optimal given the previous lower bound.

We now consider the case $R < L$. If the unrepaired graph has c components, then the repaired graph has at least c components (as it is a subgraph), and thus every feasible algorithm must build at least c libraries. Thus, substituting c for ℓ in the previous lower bound, every algorithm must cost at least $n \cdot R + c \cdot (L - R)$, and this is exactly the cost of our algorithm in the case $R < L$.

Run-time analysis. The algorithm runs in time $O(n + m)$; either we build a city at every location without needing to look at the edges, or we run DFS to find the spanning trees.

Note: In the class, we went over DFS for finding spanning tree of a connected graph. To make it work on a graph which is not connected and instead finds a spanning tree for each component, we can do as follows: Run DFS from a vertex to find the spanning tree of this component; remove all these vertices from the graph, and recurse. The runtime of each connected component is proportional to number of vertices and edges *inside this component* and hence this algorithm in total requires $O(n + m)$ time.

Problem 3. Suppose you are given a directed acyclic graph $G(V, E)$ with a unique source s and a unique sink t. Any vertex $v \notin \{s, t\}$ is called an (s, t)-*cut vertex* in G if *every* path from s to t passes through v; in other words, removing v from the graph makes t unreachable from s.

(a) Prove that a vertex v is a (s, t)-cut vertex if and only if in any topological sorting of vertices of G, no edge connects a vertex with order less than v to a vertex with order higher than v. **(10 points)**

Solution.

- If v is an (s, t) cut vertex in G then in any topological sorting of vertices of G, no edge connects a vertex with order less than v to a vertex with order higher than v.

Proof of by contradiction: Suppose this is not true; it means that there is a vertex (s,t) cut vertex v where in some topological ordering of G, there is an (u,w) with $order(u) < order(v) < order(w)$ (where $order(z)$ denote the order of vertex z in the topological ordering).

Since s is the unique source for G and $order(u) < order(v)$, there is a path from s to u (say P_{su}) that does not pass through v (s being unique source guarantees at least one path to u; and since $order(u) < order(v)$, no such path can pass through v). Since t is a unique sink, the same exact argument implies existence of a path P_{wt} from w to t that does not pass through v (and moreover any vertex of P_{su} since their order in the topological order is smaller than v (and hence w)). Hence, we found a path $(P_{su}, u \to w, P_{wt})$ that goes from s to t without passing v, a contradiction with v being a (s,t)-cut vertex.

- If in any topological sorting of vertices of G, no edge connects a vertex with order less than v to a vertex with order higher than v, then v is an (s,t) cut vertex in G.

  Proof of by contradiction: Suppose this is not true; it means that there is a vertex v such that in any topological sorting of vertices of G, no edge connects a vertex with order less than v to a vertex with order higher than v, and yet v is not an (s,t)-cut vertex in G. Since v is not a (s,t)-cut, then there should exists a path P_{st} from s to t that does not pass v. Let $s, w_1, w_2, \ldots, w_k, t$ denote the vertices of this path. Since $order(s) < order(v)$ and $order(v) < order(t)$, there should exists a pair of vertices w_i, w_{i+1} where $order(w_i) < order(v) < order(w_{i+1})$ (otherwise how did we ever reach a vertex t with $order(t) > order(v)$?). But then the edge (w_i, w_{i+1}) connects a vertex with order less than v to order more than v, a contradiction.

  where in some topological ordering of G, there is an (u,w) with $order(u) < order(v) < order(w)$ (where $order(z)$ denote the order of vertex z in the topological ordering).

(b) Use part (a) to design an $O(n+m)$ time algorithm for finding *all* (s,t)-cut vertices in G. **(15 points)**

Solution. *Algorithm:* We give the following algorithm for finding all (s,t)-cut vertices in G:

- Run topological sort on G.
- For every vertex v compute the furthest vertex in the ordering from any of the vertices with order less than v. Formally, compute the maximum $order(w)$ where $(u,w) \in E$ and $order(u) < order(v)$. In order to do this:
  - Define $reach(v)$ to be the vertex $order(w)$ for w being the furthest vertex defined for v.
  - Iterate over vertices of G in the ordering, for each vertex v_i at the i-th position of the ordering (i.e., $order(v_i) = i$), let $reach(v_i) = \max\{reach(v_{i-1}, order(z)\}$ for $z \in N(v_i)$.
- For any vertex v if $reach(v) \leq order(v)$, then output v is an (s,t)-cut vertex in G. Otherwise, output v is not a (s,t)-cut vertex.

Proof of Correctness: Firstly, it can be seen that for each vertex v, $reach(v)$ is equal to $order(w)$ where $order(w)$ where $(u,w) \in E$ and $order(u) < order(v)$ (the proof is identical to the proof for maximum problem). Now, we output a vertex v as being (s,t)-cut vertex if and only if $reach(v) \leq order(v)$: by definition of $reach(v)$, this means there is no edge (u,w) in the graph G with $order(u) < order(v) < order(w)$. Part (a) of the question now proves the correctness of the algorithm.

Runtime Analysis: Topological ordering takes $O(n+m)$ time. For each vertex computing $reach(v)$ takes $O(deg(v))$ time and so this step also takes $O(n+m)$ time. The final step takes $O(n)$. Hence total runtime is $O(n+m)$.

Problem 4. Use graph reductions to design an algorithm for each of the following two problems.

(a) Given a vertex s in a directed graph $G(V, E)$, find the length of the shortest *cycle* that starts from the vertex s. **(12.5 points)**

Solution. The proof is by reduction to shortest (unweighted) path problem.

Reduction: Find the shortest path from s to all vertices in V using BFS (or any other algorithm for finding shortest paths). Let $d[v]$ denote the shortest path from s to v. Find minimum of $d[v]$ among all vertices v that have an edge back to s, namely, $(v, s) \in E$. Return the shortest path from s to v (found by BFS in the first step) plus the edge (v, s) as the cycle.

Proof of Correctness: Consider the shortest cycle C starting at vertex s. Suppose the final vertex of this cycle before s is w, i.e., the cycle is $s \rightsquigarrow w \to s$. We argue that $s \rightsquigarrow w$ is a shortest s-w path; otherwise, we could switch this part of the cycle by a shorter path and get even a shorter cycle than C, a contradiction.

This means that the shortest cycle starting at s should go to a vertex w using the shortest s-w path and then come back to s. Our algorithm takes the minimum length of all such cycles, hence returns the shortest cycle starting from s correctly.

Runtime Analysis: Running BFS takes $O(n+m)$ time and finding iterating over vertices and checking whether they have an edge to s and taking minimum can also be done in another $O(n + m)$ time, making overall runtime $O(n + m)$.

(b) Given a connected undirected graph $G(V, E)$ with distinct weights w_e on each edge $e \in E$, output a *maximum* spanning tree of G. A maximum spanning tree of G is a spanning tree of G with maximum total weight of edges. **(12.5 points)**

Solution. The proof is by reduction to the minimum spanning tree problem.

Reduction: We find the maximum weight edge in G and let W denote this edge weight *plus* one. We then create new weights $w'_e = W - w_e$ for every edge $e \in E$. Finally, we run any MST algorithm (say Kruskal's algorithm) for finding a minimum spanning tree of graph G with new weights and return this tree as a maximum spanning tree of the G under original weights.

Note that since all edge weights are *positive* in this new graph, we are allowed to run a MST algorithm on the resulting weights as well (in general, MST is only defined for positive weight graphs and so we are not allowed to run an arbitrary algorithm for MST on a negative-weight graph, even though some of the algorithm's like Kruskal's or Prim's work without any change for graphs with negative weights also. Since our goal is to do a reduction and hence we should be able to use *any* algorithm for MST, we cannot assume that algorithm works with negative weight edges.).

Proof of Correctness: Consider any spanning tree T of G. Let $w(T)$ denote the total weight of T under weight function w_e on edges $e \in E$ and $w'(T)$ denote the weight under w'_e. By our construction, $w(T) = W \cdot (n-1) - w'(T)$ since each spanning tree has exactly $n-1$ edges. As such, maximizing $w(T)$ (over the choice of T), is equivalent to minimizing $w'(T)$ (again over the choice of T). Thus, a minimum spanning tree under weights w' corresponds to a maximum spanning tree under weights w.

Runtime Analysis: Finding the maximum weight edge and defining weights w'_e takes $O(n + m)$ time to go over the graph. We can then use the best possible algorithm for MST on our graph to solve the problem; even though we do not know what necessary is runtime of that algorithm, we know that it is at most $O(m \log m)$ (since Kruskal's algorithm takes at most that time) and hence runtime of our algorithm is at most $O(m \log m)$ (but it can potentially be lower by using a better MST algorithm).
