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

Homework #4

November 7, 2019

Name: FIRST LAST *Extension: Yes/No*

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.

Algorithm:

- **Recursive Formula:** rec[v] = max($\sum_{j=1}^{n}$rec[v], point[v] + $\sum_{j=1}^{n}$ all children of vj)

Proof of correctness:

- The basic idea here is we can't consider the sum of both nodes and its childeren at the same time
- But we can consider the sum of the node and its children's children, or we can not take the node and get the sum of its children only, and finally we chose the maximum sum among those two
- We get the max sum by using above recursive formula, but we have to run it twice, one for the parent (root) without considering its children and one for the children and take the max of the two results
- By this way we get the max number of points

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:

- There could be three possibilities,

1. L>R
2. L<R
3. L=R

- We will have to make an optimal choice for all the three cases.
- For case 1: We will make libraries in all cities
- For case 2: We will decide how many roads need to be fixed and libraries need to be made
- For case 3: It wouldn't matter, meaning we can either make all libraries or fix all the roads

Proof of correctness:

- We know from the begining that we will either have to make libraries or fix the road, and do both if needed. There is no way around that
- Now for the first case, where making a library cost less than building roads, we will make libraries in every cities rather than making some libraries and fix some roads. This will be the minimum cost, as making some libraries and fix some roads can cost more than just building libraries
- if the cost of building a libray and fixing the road is the same, then we can do either of them as it will return the same thing.
- When building roads cost less than building libraries, we can't just make roads because we need at least one libray (in the best case where every other city can reach) to be built
- In that case, we will run the DFS algorithm which will give us the path where the roads need to be fixed
- Running DFS algorithm will get rid of the duplicates or repeatitive path and help us determine the right paths that need to be fixed
- This will be the minimum cost which will be around L + R(n-1); where n is number of vertices

---

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.

Objective: To prove that no edge connects a vertex with order less than v to a vertex with order higher than v, where vertex v is a (s, t)-cut vertex

Proof of Correctness:

- We will use contradiction to prove this statement.
- First of all, to understand the objective, it basically says that for given order n of the vertex v, all vertices less than order n will have no edges to vertices with order greater than n.
- **Hypothesis:** There exists edge that connects a vertex with order less than v to a vertex with order higher than v
- The above statement basically says that there is an edge or more, from the vertices before the vertex v to the vertices after v.
- The whole point of the cut vertex is that by removing it, there is no way to reach the end of the graph
- But since there is an edge from the vertices before v to vertices after v, we can ignore v and remove the edge and still be able to reach the end of the graph (according to hypothesis)
- Hence, vertex v is not a cut vertex because even after cutting/removing it we can reach the end of the graph
- Since v is a cut vertex (given in question), and according to our hypothesis it is not. Hence, our hypothesis is incorrect, and we proved that there is no edge that connects a vertex with order less than v to a vertex with order higher than v whoch makes v a cut vertex
- Hence, proved by contradiction

(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 will implement the DFS algorithm here and will go over the vertices in a tree form, and since we traverse the graph in a tree form we will have a parent p and vertices v
- the vertex/parent p will be the cut vertex if,

1. p is the root of the tree menaing it has at least two children
2. p is not the root in that case it will have one child v and since there are no elements before p (because p is the root), the vertx v will have no back edges

- If both of the above conditions are satisfies, then we can say that the vertex u is the cut vertex

Proof of Correctness:

- As mentioned above, we will use the DFS traversal in a tree form to find the cut vetex
- We will have to keep the parents or p vertices and we can do so by creating an array that stores the p vertices
- If there are more than two children to the parent then we found the cut vertex
- If there are not more than two children, then it gets more comprehensive
- That can be done by finding lowest numbered vertex reachable from the parent p using 0 which has at most one back edge
- We basically check if there is a child v of p that cannot reach a vertex visited before p. If so, then removing v will disconnect the graph and we will get our cut vertex
- Hence by just using DFS tree traversal we can determine the cut vertices

Time Complexity:

- The above allgorithm is simple DFS with additional arrays that stores the parent and cut vertices. So time complexity is same as DFS which 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.

Algorithm:

- Vertices: For all nodes in the given graph G(V, E), we initiate all vertices' Vs' distances to ∞ and source s distance to 0 and create a set of those distances

- Edges: For any two adjacent nodes, we keep adding the edges until reached end and update the set created above

- We will first initialize distances to all vertices to ∞ anddistance to source by 0

- After that, we will use topological searching to sort the graph, and once we have a topological order, we will go through all vertices and update the distance of its adjacent by comparing it to the distance of the current vertex

- Here are the three steps of the algorithm:

1. Create a set, initially null, to keep track of vertices that creates the shortest path

2. Initializing the distances to ∞ and source distance to 0.

3. We pick a vertex at random which is not in out set and include it in the set. We will then look at it adjacent vertices and compare the distance

Proof of Correctness:

- As mentioned in the algorithm, we will have a set containg distances to all vertices. We initialize each distance to ∞ and source distance to 0

- Since we topologicaly sort it, and we set the source distance to 0, source will be the first one to be selected

- By doing that we can assure that the source will be in our shortest path

- We start from source and update the distance set, so the distance set will look like {0, 1} in the beginning

- Now we look at the adjacent vertices and add/update the distance set

- We do this for all other vertices and in the end we will have multiple paths, not all of those will be the shortest, but they will be the paths from source to the end, of course containing source

- We will check which one is the shortest among them and hence we found the shortest path

- There could be the case where the output is ∞, in that case there will be no short path to reach the end (the graph will be acyclic in that case)

Time Complexity:

- We go through each node and add the distances and in the end select the shortest result. This entire process takes $O(n^2)$ time

- The process of going through each nodes will take O(n) time and adding/updating the distances will also take O(n) time. Since they happen at the same time (meaning they are nested, in a coding point of view), it will take $O(n^2)$ time in total

---

(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.

Algorithm:

- Vertices: For all nodes in the given graph, we randomly choose a node i and look at the other nodes adjacent to it only if the weight is maximum. We make sure that we mark the nodes that we choose
- Edges: For any two adjacent nodes, we look at the randomly chosen node's edge e and check that weight with other edges in order to find the max edge. We don't consider the edge between two marked nodes

Proof of Correctness:

- The first step is to choose a node at random and then look at the adjacent nodes given that the edge weight is max
- We have to make the selection when deciding which way to go when there are multiple edges coming out of one node
- We will choose the edge with max weight and mark both the nodes so we do not repeat them
- We will make sure that we don't consider the edges between the two marked nodes
- We repeat the same process for all the other nodes and we will get the maximum spannig tree

---

Challenge Yourself. A *cut-vertex* in an undirected connected graph $G(V, E)$ is any vertex v such that removing v (and all its edges) from G makes the graph disconnected. Design an $O(n + m)$ time algorithm for finding all cut-vertices of a given graph with n vertices and m edges.

Solution.

Algorithm:

- We can run the same algorithm that we have on question 3b. The only difference will be that we will be running DFS algorithm in an undirected tree. Everything else will stay the same

---