Exercises in Algorithms No 3

Alexandros Kyriakakis (el12163)

December 2018

1 Longest Path in Tree

The problem can be solved very easy by applying the algorithm of the DFS tree. First of all we construct an Array $[2] * [V]$. So we start from the root of the tree and we do the algorithm of DFS. After every time the algorithm finds the longest "child" and starts returning back, for each vertex, we save in the array the price $w(u)$ of the current vertex plus all the prices of the vertices under the current vertex. At the array we store only the maximum and the second maximum prices for each vertex. At the end of the DFS algorithm we add the stored prices for each vertex and we return the vertex with the maximum price. The tree is undirected graph, so in order to find the "longest path" we have to add the paths which are on both sides of the current vertex. The complexity of this algorithm is $\mathcal{O}(V+E)$ which is the complexity of the DFS, the construction of the array costs $\mathcal{O}(2 * V)$ as for the filling of the array its part of the DFS so it does no extra cost. So in total we have $\mathcal{O}(V + E)$.

2 Cost Function in Directed Graph

2.1 DAG

We solve this problem using once again the algorithm of constructing a DFS tree. At the state that the algorithm starts returning from each "leaf" we save the current minimum $p(u)$. As the algorithm returns holding the current minimum, we save it at the passing node as the cost $c(u)$ of the current vertex and continues to the next node. At the end of the algorithm every node contains the cost $c(u)$. The time complexity of the algorithm is the same as the DFS which is $\mathcal{O}(V+E)$ because the definition of the costs is $\mathcal{O}(1)$ as also the current minimum. So the total complexity of the algorithm is $\mathcal{O}(V + E)$.

2.2 Non-DAG

First of all we use the algorithm for Strongly Connected Components to define them. Then at each (circle) component we find the minimum $p(u)$ from all the strongly connected vertices. Then we do the above algorithm for DAG, facing

every SCC's as a vertex using the minimum price as its $p(u)$. In the end we save the result of the above algorithm for each "virtual" vertex as the $c(u)$ of each vertex contained at each SCC. So all the vertices of each SCC has the same $c(u)$, which is the $c(u)$ of the "virtual" vertex we used to do the above algorithm. So, the total complexity of the algorithm is the same as before $\mathcal{O}(V+E)$. Because, the algorithm for SCC costs $\mathcal{O}(V+E)$. The replacement of each SCC costs $\mathcal{O}(V+E)$ because we construct a vertex with all the edges of the vertices that participating at each SCC except the inside edges that starts and ends at vertex that is a participant of each SCC. Then the algorithm for DAG costs the same $\mathcal{O}(V+E)$.

3 Cops and Robbers

This problem needs a dynamic solution to be solved. So we are going to use the DP method to solve it using an array N*N*2 for every possible state. A state is described by the (vertex of robber, vertex of cop, next playing). First of all we initialize (by hand) the starting states where we assign as "robber win" if robber is on vertices connected to warehouse and plays next, wherever cop is. Then we are parsing with the following way. We move from states that we know the result to states that we don't. So, if we are on a state where robber is on vertex i, cop on vertex j, cop plays next, robber wins and k is neighbour to j then at state (i,k,robber) inherits the result so state (i,k,robber) leads to "robber win". Obviously, exists the opposite where, if we are in a state (i,j,robber), leads to "cop win", k is neighbour to i then state (k,j,cop) inherits the "cop win". State (i,j,robber) with result "robber win" and k neighbour to robber leads to state (k,j,cop) that inherits the result "robber win". The same, if state (i,j,cop) with result "cop win" and k is neighbour to j then state (i,k,robber) inherits the result "cop win". Generally cop wins if in state (i,j,cop) i,j are neighbours. Also when we fill the array if a cell is already filled, if the previous result is "cop wins" and current is "robber wins" then we replace the result but we don't do the opposite. That's because robber always plays first so he has advantage. As for the draw, if all the following states leads the robber to lose but cop cannot reach him in one direct step then robber can lead to draw. The complexity of the algorithm is $\mathcal{O}(V^2+E)=\mathcal{O}(V^2)$ because for every cell of the array we use all the existing edges that lead there.

4 Spanning Trees

4.1 Exchanged Edge

We are going to prove that, using the definition of the Spanning Tree. First of all, if we remove an edge (e) from T_1 we know that now T_1 is no longer a tree but a forest which means that there is an incision. This can be proved using the definition of Spanning Tree which says that it is minimally spanning so if we remove an edge we have a forest. Now that we have an incision at that

graph, again from the definition of the Spanning Tree we know that when we constructed T_2 we used an edge (e') to bridge this same incision. So for sure there is an edge $(e' \in T_2)$ to bridge this incision. So the final tree $(T_1 \backslash e) \cup e'$ will be a Spanning Tree.

Now, as for the algorithm, the idea is simple. We remove the the edge (e) from the spanning tree T_1 and we run the algorithm for Strongly Connected Components to recognize the two components that formed on $T_1 \backslash e$. Then, linearly we try all the edges from T_2 to bridge the two components. The time complexity of this algorithm is $\mathcal{O}(n)$ because the SCC algorithm costs $\mathcal{O}(V+E)$ but $V = n$ and $E = n-1$ cause T_2 is a tree. So total time complexity is $\mathcal{O}(n)$.

4.2 Spanning tree from spanning trees

We are going to prove that for each random vertices of this graph there is a path that connects them which means generally that H is spanning. We assume that we have two random vertices of the graph H, which represents two spanning trees T_1 and T_2 on graph G with $1 \leq |T_1 \backslash T_2| \leq n-1$. From the proof we did above, we know that we can remove an edge (e) from T_1 and exchange it with an edge (e') from T_2. So there is a vertex, which represents a spanning tree $T' = (T_1 \backslash e) \cup e'$, on H that has an edge with (vertex) T_1 because $|T_1 \backslash T'| = |T' \backslash T_1| = 1$. Then we can once more exchange an edge between T' and T_2 to go to a vertex T'' which has $|T_1 \backslash T''| = |T'' \backslash T_1| = 2$ and $|T' \backslash T''| = |T'' \backslash T'| = 1$. We repeat this process until we reach T_2. This means that there is a path that connects T_1 and T_2. So, graph H is spanning. Also every time we exchange an edge between T_1 and T_2 on the process above we move a vertex closer to T_2 so the total distance between (vertex) T_1 and T_2 is $|T_1 \backslash T_2|$.

In order to find the shortest path between to vertices in H we use the algorithm above. If the given vertices are u, v that represents T and T', at first in $\mathcal{O}(n^2)$ for every edge (e) that $(e) \epsilon T \backslash T'$ we run the algorithm above with input (T, T', e). So if we exchange the edge (e) from T with the output (e') of the algorithm we have a new vertex on H that is connected with T and it's one step closer to T'. Repeating this process for $|T \backslash T'| = |T' \backslash T|$ times will lead as to T' on the graph H. This algorithm has time complexity equal to $\mathcal{O}(n^2)$ because for every edge of $T = n-1$ we run the algorithm with complexity $\mathcal{O}(n)$. So the total complexity of the algorithm will be $\mathcal{O}(n^2)$.

4.3 Spot the bridge

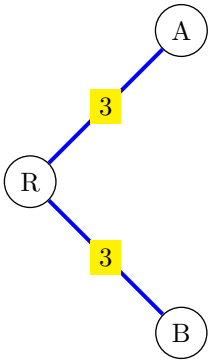
The first and more simple idea is that we transform the graph $G(V, E)$ into a weighted graph $G'(V, E, w)$. Firstly, we define every edge from E_2 with weight equal to 1 and every edge from E_1 with weight equal to 2 and then run Prim's algorithm for the weighted graph G'. From the Prim's output we save the edges with weight 2 (came from E_1) and we name a the number of these edges. Then we make a graph G'' same as G but with weight: 0 to the edges we saved, 1 to the other edges from E_1 and 2 to the edges from E_2. Then we run Prim's algorithm again for G''. In the last output from Prim's algorithm we name b

the number of edges with weight equal to 1. If $k \notin [a, a+b]$ there is no spanning tree to return. Else make a graph G''' by renaming randomly $k-a$ edges from the previous graph G'' with weight 1 to 0 and if there are still edges with weight 1 rename them to ∞. Finally return the output from Prim's algorithm in G'''. As for the complexity of this algorithm we know that Prim's algorithm has $\mathcal{O}(E + VlogV)$ and every time we rename the weights of the edges we need $\mathcal{O}(E)$. So totally we have $\mathcal{O}(E + Vlogv)$ complexity for this algorithm.

Another idea is to construct the graph H like the previous question. Go to a random vertex. Count the edges that belongs to E_1, save the number on variable x. If $x = k$ return the tree that is being represented by this vertex. If $x \leq k$ move away $k - x$ vertices from this vertex but every time gaining more edges from E_1. If this leads to a dead end return that there's no tree else return the tree that represented by the vertex we moved to. We do the opposite if $k \leq x$. We move $x - k$ steps away but every time removing edges from E_1, if this again leads to a dead end, return that there's no tree else return the tree result.

5 Uniqueness of MST

5.1 Same W unique MST



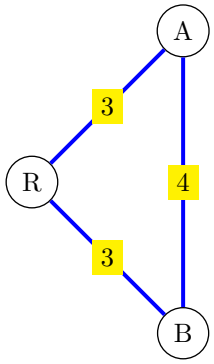
Starting from root R there is unique MST with the same weight at each two edges.

5.2 Unique minimum edge means unique MST

To prove that we are going to use proof by contradiction. Guess that there are two MSTs T and T' constructed from the same graph $G(V, E, w)$. Then from triangular inequality we know that exists at least one edge that's exchangeable between T and T'. So from the definition of MST we know that if this happens means that for the same "cut" there are two "safe" edges. Which in this example means that there are two edges that bridge the same cut and they both have minimum weight. Which leads as to contradiction. So if there is unique

minimum weighted edge that bridges every cut on graph G then there is unique MST for G.

For the opposite we use an example below. For this graph the MST is unique (uses only edges with weight 3). But the cut with R in one side and A,B on the other has two edges with minimum weight.



5.3 If and only

The mathematical condition that we are looking for can be constructed looking above. A spanning non-directed graph $G(V, E, w)$ has unique MST if and only if for every cut $(S, V \backslash S)$ on the $G(V, E, w)$ all the edges that bridge the cut $(S, V \backslash S)$ have different weight.

We have already proofed one direction, above. So we only have to prove that if we have unique MST the conditions above exists. Once more we use proof by contradiction. Guess that there's a graph $G(V, E, w)$ with unique MST and $\exists(S, V \backslash S)$ with two edges with same minimum weight. This means that these edges are exchangeable so we can form another MST exchanging these two edges. Which leads us to contradiction.

5.4 Algorithm for Uniqueness

Having the knowledge from the previous questions, the algorithm to check the uniqueness of the MST on a graph $G(V, E, w)$ leads us to Kruskal. As we know Kruskal first do sort, so if all the edges have different weight we are done, the MST is unique. So we care only for the edges with same weight. Specifically, we care only about the edges that bridge the same cut. So, only if there are two edges with the same weight that bridges the same cut the MST is not unique. To solve this we run Kruskal and in every loop we save the previous unions. While the edges have the same weight we check if they belong to the same unions that we saved at the previous step. If that happens, means that they are exchangeable and also that the MST is not unique. If Kruskal with the extra procedure ends then there is unique MST. Obviously, the extra procedure costs $\mathcal{O}(1)$ so total complexity of the algorithm is the Kruskal's complexity which is $\mathcal{O}(ElogE)$.