CS4102 - ALGORITHMS - FALL 2019

DUE THURSDAY, NOVEMBER 14, 2019 AT 11PM VIA COLLAB

H7:ER6QT

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. **Please remember that you are not allowed to share any written notes or documents (these include but are not limited to Overleaf documents, LaTeX source code, homework PDFs, group discussion notes, etc.). Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.**

Collaborators: zh2yn

Sources: https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm, https://www.geeksforgeeks.org/bfs-disconnected-graph/, https://stackoverflow.com/questions/43012559/how-to-prove-there-always-exists-a-minimax-path-completely-on-the-mst

PROBLEM 1 *Minimax Spanning Tree*

Let $G = (V, E)$ be a connected graph with distinct positive edge weights, and let T be some spanning tree of G (not necessarily a minimum spanning tree). The dominant edge of T is the edge with the greatest weight. A spanning tree is said to be a minimax spanning tree if there is no other spanning tree with a lower-weight dominant edge. In other words, a minimax tree minimizes the weight of the heaviest edge (instead of minimizing the overall sum of edge weights).

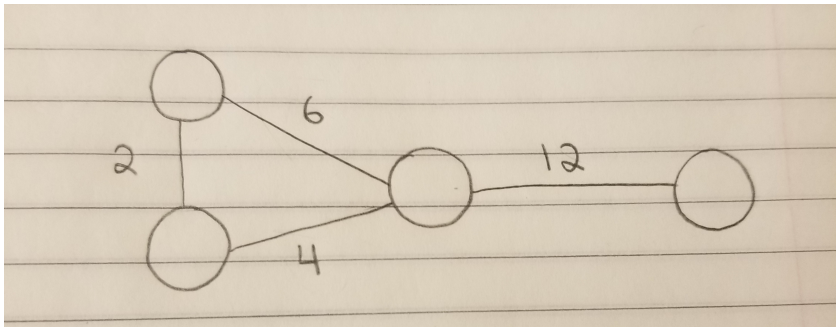

Figure 1: Minimax Spanning Tree of G

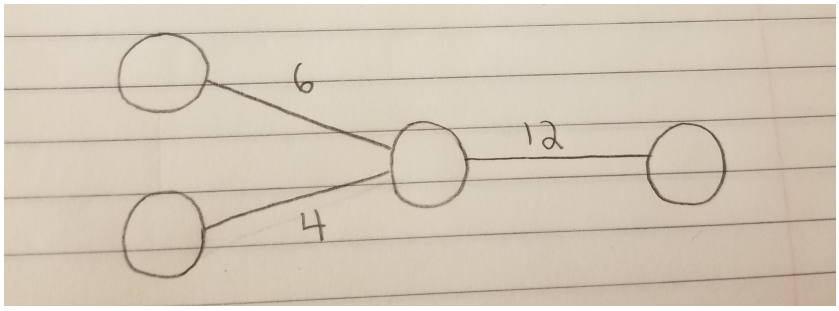

Figure 2: Minimax Spanning Tree of G that is NOT a Minimum Spanning Tree of G

1. Prove or disprove: Every minimax spanning tree of G is a minimum spanning tree of G.

   We can disprove this using the counterexample above.

   In order to disprove this argument, we need to show that every minimax spanning tree is not a minimum spanning tree. In Figure 1 above, we can see that the dominant edge with the greatest weight is 12. However, we cannot minimize the weight of the heaviest edge as it is the only edge pointing to a single node, therefore, it must be included in the spanning tree. Any minimum spanning tree in G must total to 18 as there is no other total edge length that is smaller. We get 18 by adding edges $2 + 4 + 12$. It is possible for this minimax spanning tree of G to be a minimum spanning tree of G if we add edges $2 + 4 + 12$ together, however, for counter example purposes to disprove that not **every** minimax tree is a minimum spanning tree of G, we can see in Figure 2 that there is also a possibility that this minimax tree cannot be a minimum spanning tree as edges 6, 4, 12 added together $(6 + 4 + 12)$ is equal to 22, which would not be a MST. Thus, we disprove the statement above through proof by counter example as we can see that every minimax spanning tree of G is not also a minimum spanning tree of G.

2. Prove or disprove: Every minimum spanning tree of G is a minimax spanning tree of G.

   We can prove this by contradiction.

   For contradiction purposes, let's say we have a minimum spanning tree T of graph G, as mentioned in the description, that is NOT a minimax spanning tree. By stating this, we are saying that there IS another spanning tree with a lower-weight dominant edge. If we remove the dominant edge E in tree T, we are separating/splitting the graph into 2 parts. Establishing that tree T is a minimum spanning tree and not a minimax spanning tree, we know that there exists a spanning tree that will not use the same dominant edge in T. Therefore, there must exist another lower-weight dominant edge D that we could add after removing E which would connect the point of cut(separation) between the 2 parts of the graph, but this would imply that tree T was never a minimum spanning tree of G (spanning tree, but its total weight would have decreased), which is a contradiction (T is given as being a minimum spanning tree). Thus, we can prove the above statement that every minimax spanning tree of G is a minimum spanning tree of G.

PROBLEM 2 *Coffee*

Charlottesville is known for some of its locally-roasted coffees, each with its own unique flavor combination. Suppose a group of coffee enthusiasts are given n samples $c_1, c_2, ..., c_n$ of freshly-brewed Charlottesville coffee by a coffee-skeptic. Each sample is either a *Snowing In Space* roast or a *Milli Joe* roast. The enthusiasts are given each pair (c_i, c_j) of coffees to taste, and they must collectively decide whether: (a) both are the same brand of coffee, (b) they are from different brands, or (c) they cannot agree (i.e., they are unsure whether the coffees are the same or different). Note: all n^2 pairings are tested, including (c_i, c_i), (c_i, c_j), and (c_j, c_i), but not all pairs have "same" or "different" decisions.

At the end of the tasting, suppose the coffee enthusiasts have made m judgments of "same" or "different." Give an algorithm that takes these m judgments and determines whether they are *consistent*. The m judgments are *consistent* if there is a way to label each coffee sample c_i as either *Snowing In Space* or *Milli Joe* such that for every taste-comparison (c_i, c_j) labeled "same," both c_i and c_j have the same label, and for every taste-comparison labeled "different," both c_i and c_j are labeled differently. Your algorithm should run in $O(m + n)$ time. Prove the correctness and

running time of your algorithm. Note: you do *not* need to determine the brand of each sample c_i, only whether the coffee enthusiasts are consistent in their labelings.

Algorithm
We will start off by creating two adjacency lists: one list for coffees that are classified as the "same" (list E), and one list for coffees that are classified as "different" (list D). These lists will hold all adjacent(neighboring) nodes. We can assume that the graph is undirected (undirected edges between pairs). Next, we will add all n samples of coffees to a list holding all the nodes/vertices we have not visited in the graph (list U). We will then choose one of the coffee brands, "Snowing in Space" S or "Milli Joe" M (does not matter which one), to label as our c_1 (sample 1). To be specific, we can set c_1 to equal M. Using the Breadth First Search algorithm, we will use a queue to which we can then enqueue c_1 to, and mark this source node as visited, removing it from the list U (nodes/vertices that we have not visited).

Next, we will dequeue a node held in the queue and check two things for all other nodes in the adjacency list for "different" (list D):

First: if the label of the other node in the adjacency list is not given, we will set the other node to be labeled as the opposite of our node taken out (c_1 or M), so S for Snowing in Space. We will then enqueue this other node S to the queue and remove it from the list of unvisted nodes U (mark it as visited). (Essentially, we're enqueueing adjacent nodes of the dequeued node)
Second: if the label of the other node is given and it DOES match the label of node c_1 (M), then return false as both coffee brands are the same when they are suppose to be different.

Repeat the process above, but for all other nodes in the adjacency list for the "same" (list E) decision this time:

First: if the label of the other node in the adjacency list is not given, we will set the other node to be labeled as the same as our node taken out (c_1 or M), so M for Millie Joe. We will the enqueue this other node M to the queue and remove it from the list of unvisited nodes U.
Second: if the label of the other node is given and it does NOT match the label of node c_1 (M), then return false as both coffee brands are different when they are suppose to be the same.

We repeat the processes mentioned above for each sample c_i (keep on dequeuing) until the queue is empty and we have reached all univisted nodes in the list U. Note: if the queue gets empty, but we still have nodes that have not been visited in list U, we can enqueue a vertex/node from the list into the queue and dequeue it to use as our new current node to continue the process and get all unvisited nodes.

Proof of Correctness
In order to determine if m judgements are consistent, we will label the edge between each pair of coffees to taste as the "same" or "different." These two decisions will act as edges on a graph, with the coffee brands (M or S) as our nodes. In our above algorithm, it is explained that we use c_1 to denote our first sample of freshly-brewed Charlottesville coffee, which we labeled as M Millie Joe. We will add this starting node to our queue and then dequeue it as the current node(vertex) we are looking at. We then check the nodes adjacent to our current node("other nodes") in order to label them as same or different in relation to the current node. We do this in two ways. The nodes that are connected together through a "same" edge are labeled the same thing as the current node(M), which would be M in our case for c_1, whereas the nodes that are connected together through a "different" edge are labeled the opposite thing as the current node(M), which would be S in our case for c_1. However, it is to be noted that if two nodes have the same label but are connected

together by a "different" edge or if two nodes have a different label but are connected by a "same" edge, the coffee enthusiast decided wrong about the coffee taste and, therefore, what brand it is. Hence, the algorithm would return false ("unsure"). As noted above in the algorithm, there is a chance that we will have an empty queue, but the list of unvisted nodes will still not be empty. This is due to the possibility of a disconnected graph. Therefore, we must take the unmarked nodes in list U and enqueue them when the queue gets empty. We will use these enqueued nodes in the queue to dequeue them one by one as our new current node so that we can have a new starting node to evaluate adjacent nodes with in the graph. When the queue gets emptied, the program is over, meaning that the m judgments made by the enthusiasts are consistent (return true). Thus, by labeling every node(each coffee sample c_i) with every other neighboring node accurately (classifying every taste comparison (c_i, c_j) correctly) we find consistency in the decisions/judgements made by the coffee enthusiasts, proving the correctness of our BFS algorithm.

Proof of Runtime
Our BFS algorithm does in fact run in $O(m + n)$ time because our algorithm will look at every node, or more fittingly, every coffee sample n, and every judgement m ("different" or "same" edge) once in the worst case.

PROBLEM 3 *The Rise of Skywalker*

Rey has decided to find Kylo Ren by taking the Millennium Falcon, but she doesn't quite know where he is yet. The galaxy far far away where they live contains n planets (conveniently numbered 1 through n) with $m = O(n^2)$ bidirectional interplanetary routes connecting them (i.e., there may not be a direct route connecting two planets). For each possible route (p_i, p_j), we know the amount of fuel c_{ij} the Millennium Falcon needs to traverse it; traveling within a planet consumes no fuel. For each $1 \leq i \leq n$, the price of fuel on planet p_i is fixed at f_i galactic credits per unit of fuel.

Because Han Solo was quite thrifty with his ship, he installed huge barrels of fuel in the cargo bay to store extra fuel and avoid needing to refuel on expensive planets. Therefore, the Millennium Falcon is able to carry any amount of fuel Rey wants at any moment during the trip, and she is also fine with visiting the same planet more than once on the trip if it saves money. Rey starts the trip without any fuel, so she must fuel up on the first planet.

Design an algorithm to help Rey find a route that minimizes the total number of galactic credits she needs to spend on fuel in order to travel from her current planet s to planet t. A route in this case consists of a sequence of planets (starting from s and ending at t) together with the amount of fuel Rey should purchase on each planet (if any). Your algorithm should run in time that is polynomial in n. Prove both the correctness and running time of your algorithm. Hopefully Kylo will be on planet t. *May the force be with you!*

Algorithm
A possible algorithm to help Rey find a route that minimizes the total number of galactic credits she needs to spend on fuel to travel from her current planet s to planet t to find Kylo is by first, creating one graph based on the planets and distances. Our nodes in the graph represent our n planets, and our edges in the graph represent our distances, or the amount of fuel we will be needing to travel between those planets. Next, we will apply the Floyd-Warshall algorithm to the graph so that we can get the shortest paths between any two planets/nodes (shortest path between all pairs of vertices). Taking this algorithm into account, we will add the shortest paths possible from each node to every other node based on the distance between the planets. Therefore, this will allow us to keep track of all possible paths, which is important to our solution as this is the total amount of fuel we will need to get from each planet node to the next planet node. We will then create a second graph (directed graph) that will be based on the first graph

and from the results of Floyd-Warshall. However, this time, the edges will be the total cost of fuel we need to traverse that edge from the planet node the edge came from. We will calculate these edges by multiplying the amount of fuel needed to traverse the edge from the first graph by the price of the node (price of fuel on planet p_i) that the edge comes from. Lastly, we will use Dijkstra's algorithm on the current, starting node planet s, to the final, ending node planet t. Thus, we will get the shortest optimal path with the least/minimum total number of galactic credits f_i needed to spend on fuel so that Rey can travel from her current planet s to her destination planet t.

Proof of Correctness
This algorithm will be proved using the Exchange Argument.
The algorithm implemented above is based off of two assumptions. We will explain below how these two assumptions are true, leading us to an optimal solution/algorithm.

The first assumption we make in this algorithm is that we we only spend money on fuel when our barrels(tanks) are empty:
Lets say we have three planets: E, Z, and A. If we are going from our starting planet E to our ending planet A, and we have planet Z in the middle of the two, we can have two different cases:

In the first case, we will say that the price of the fuel on planet E is cheaper than that of planet Z. Based on our algorithm, we would go directly from planet E to planet A without stopping at planet Z for fuel. We can also consider another arbitrary solution that is possibly an optimal solution that will not make the same assumption as we did, thereby stopping at planet Z to fuel up. This can be shown as such:
Our solution is: $E_i * f_i = G_1$, where E_i denotes the price of the fuel at planet E, f_i (galactic credits per unit of fuel) denotes the price of the fuel on planet E, and G_1 denotes the total cost of fuel needed to get to planet A.
The arbitrary solution is: $E_i * f_i + Z_i * f_i = G_2$, where Z_i denotes the price of the fuel at planet Z.

We know that $G_1 < G_2$ since we have already noted that the price of fuel at planet E is less than the price of fuel at planet Z. This inequality depicts that the arbitrary solution is NOT optimal. Therefore, in this case, since our solution minimizes the total fuel cost, it is the more optimal solution.

In our second case, our goal is still the same; we want to go from planet E to planet A but have an option to make a stopover at planet Z in the middle of the two. In this case, the price of fuel is cheaper at planet Z than at planet E. Taking into account what our algorithm says, we will be filling up just enough fuel needed at planet E to get to planet Z. From there, we will be filling, again, just as much fuel as we need to get us to planet A. Let's assume though, that we have an arbitrary solution which does not follow our algorithm. In that case, the Millennium Falcon will not reach planet Z with an empty tank. Therefore, any leftover fuel that was unused is a wastage of money as that same amount could have been bought at planet Z at a cheaper price. This can be shown as such:
Out solution is: $E_1 * f_i + Z_1 * f_i = G_1$
The arbitrary solution is: $E_2 * f_i + Z_2 * f_i = G_2$
In the case above, $E_1 * f_i < E_2 * f_i$. This means that our solution spends less money when the fuel is more expensive, whereas in $Z_1 * f_i > Z_2 * f_i$, we spend more money at planet Z for fuel as it is cheaper. Even though the same amount of fuel can be used to get to planet A, our solutions spends less money overall than the arbitrary solution.

Let's say we also test out a third case, in which our goal this time is to get to our ending planet A without having any fuel leftover, or in other words, an empty tank/barrels. In an arbitrary

solution, in which we don't follow this algorithm, it is possible for us to reach our destination planet A with leftover fuel. This would entail that the arbitrary solution is not optimal as leftover fuel is a wastage of money, a wasteful cost.

The second assumption we make in this algorithm is that by using the shortest possible path from planet s to planet t in our second graph, we can get the optimal solution.

If we take a path that is not the shortest in an arbitrary solution, minimizing the edge weights from planet s to planet t, we will spend more money than is necessary to find a path from that same point. Hence, our algorithm is optimal as it minimizes the amount of money needed on fuel to get from our starting planet s, to our destination planet t.
The above can be shown as such:
$s\_p * f_i < l\_p * f_i$, where s_p can be denoted as the shortest path we find using our algorithm, and l_p can be denoted as the longer path we would get if we did not follow our algorithm.
Because the total money spent on fuel is less in our implementation/algorithm than the arbitrary solution, our solution can be said to be more optimal.

To conclude, it is evident by these proofs that the implementation of our algorithm does better than an arbitrary solution. This then proves that our algorithm gives us an optimal solution as it provides the minimum amount of money we need to spend to get to our destination planet t from our starting planet s. In essence, if we have fuel left in the barrels when we get to a certain planet, we have wasted money that we could have used to buy cheaper gas at another planet, and if we do not look for the shortest possible path using Dijkstra's algorithm, we will once again be wasting money to get from the starting planet s to the ending planet t.

Proof of Runtime
The runtime of this algorithm is $O(n^3)$. We get this by:

Note: number of nodes/planets is denoted as n

Applying Floyd-Warshall on the first graph takes n^3 time.
Creating a second graph takes n^2 time as there as n^2 edges for which we are calculating the cost of each.
Implementing Dijkstra's algorithm takes $ElogV$ time, where E denotes the edges, and V denotes the vertices(nodes).

Total Runtime: $n^3 + n^2 + ElogV = O(n^3)$