CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Homework #4

April 4, 2021

Name: Yinfeng Cong *Extension: No*

Homework Policy

- If you leave a question completely blank, you will receive 25% of the grade for that question. This however does not apply to the extra credit questions.
- You are allowed to discuss the homework problems with other students in the class. **But you must write your solutions independently.** You may also consult all the materials used in this course (video recordings, notes, textbook, etc.) while writing your solution, but no other resources are allowed.
- Do not forget to write down your name and whether or not you are using one of your two extensions. Submit your homework on Canvas.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "use Prim's or Kruskal's algorithm to find an MST of the input graph in $O(m \log m)$ time". You are **strongly encouraged to use graph reductions** instead of designing an algorithm from scratch whenever possible (even when the question does not ask you to do so explicitly).
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question).
- The "Challenge yourself" and "Fun with algorithms" are both extra credit. These problems are significantly more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.

Problem 1. You are given the map of n cities with m bidirectional roads between different cities. You are asked to construct airports in some of the cities such that each city either has an airport itself or there is a way to go from this city to a city with an airport using the given roads—moreover, you can also construct a road between any two cities if there is no road between them already. Finally, you are told that the cost of constructing an airport is a and the cost of connecting any two cities by a new road is r.

Design and analyze an $O(n+m)$ time algorithm that given the number n of the cities, the m roads between them, and the costs a and r, outputs the locations of airports and roads to be constructed to satisfy the conditions above, while having the minimum possible cost. **(25 points)**

Examples:

1. *Input:* $n = 4$ cities with $m = 2$ roads $(1, 2), (2, 3)$, cost of constructing an airport $a = 7$, and constructing a road is $r = 5$.

   Output: The minimum cost needed is 12 – we construct an airport in city 4 and connect it this city via a road to any of the cities 1, 2, or 3 (chosen arbitrarily).

2. *Input:* $n = 4$ cities with $m = 2$ roads $(1, 2), (2, 3)$, cost of constructing an airport $a = 7$, and constructing a road is $r = 8$.

   Output: The minimum cost needed is 14 – we construct an airport in city 4 and another one in any one of the cities 1, 2, or 3 (chosen arbitrarily).

Solution. Solution to Problem 1 goes here.
Algorithm:
Run BFS (what we have learned in the last week's kecture) on the cities [treat cities as the vertices, and treat the roads as edges]
After applying the BFS, we mark the cities that have been visited as connected, and for those cities we have not visited, we put them into an array unconnected[ ]
Assume there are x unconnected cities
Set cost = 0
—— if a > r
for x = 1 to i, we build a road between connected cities and unconnected [x], cost = cost + r.
Then, we build an airport in any one of connected cities. cost = cost + a.
Output cost
——If a < r, we build an airport in any one of the connected cities. cost = cost + a.
For x =1 to i, we build an airport in the unconnected city unconnected[x]. cost = cost + a.
Proof of correctness:
We have known the correctness of BFS, and we need to prove for the other part of the algorithm.
We know that we need to calculate the minimum of the cost, and the magnitude of the cost depends on the cost of airports and the cities.
Assume the cost of airports is more expensive than the cost of roads, then we would just build one airport and try to connect all the cities to make sure that we have the minimum cost.
because a>r, and $a + x \cdot r$ must be smaller than $ma + (x - m + 1) \cdot r$
so it is proved for the situation of $a > r$
For the other situation, we assume the cost of the airports is cheaper than the cost of the roads, then we would all build airports and no roads to make sure that we have the minimum cost.
because $a < r$, so we can get to know that $x \cdot a < (x - m) \cdot a + m \cdot r$
so it is proved also for this situation.
As a result, the correctness is proved.
Runtime analysis:
The runtime of BFS is (m+n), and the runtime of the for loop is i
So the total runtime is O(m + n + 2i) = O(m+n)

Problem 2. We say that an undirected graph $G = (V, E)$ is **2-edge-connected** if we need to remove *at least two* edges from G to make it disconnected. Prove that a graph $G = (V, E)$ is 2-edge-connected if and only if for every cut $(S, V - S)$ in G, there are *at least two cut edges*, i.e., $|\delta(S)| \geq 2$. **(25 points)**

Solution. Solution to Problem 2 goes here.
Proof:
We prove each part separately:
If G is 2-edge-connected then every cut in G has at least two cut edges. We prove this by contradiction.
Suppose G is 2-edge-connected but there exists a cut (S,V - S) with $|\delta(S)| < 2$
Pick any two vertex u, m in S and any two vertex v,n in V - S. Since G is 2-edge-connected,there should be a path from u to v, and a path from m to n in G. Let us say the path is P(1) = u, w1, w2, ...v. And the path P(2) = m, x1, x2, ...n.
For the first index i such that w_i belongs to S and w_{i+1} belongs to V - S, such an index should exists because u belongs to S and v belongs to V - S and so along this path we should eventually move from S-part to (V - S) part.
For another index j such that x_j belongs to S amd x_{j+1} belongs to V - S, such an index should exists because m belongs to S and n belongs to V - S and so along this path we should eventually move from S-part to (V - S) part.
But now the edges (w_i,w_{i+1}), (x_j, x_{j+1}) are two cut edges of S, a contradiction.
If G is not 2-edge-connected then there exists a cut in G with less than two cut edges. Since G is not

2-edge-connected, it has more than one connected component.
Let S be any connected component of G nad consider the cut (S, V - S). By definition, there is less than 2 edges going out of S and hence $|\delta(S)| < 2$.

Problem 3. The Muddy City consists of n houses but no proper streets; instead, the different houses are connected to each other via m bidirectional muddy roads. The newly elected mayor of the city aims to pave some of these roads to ease the travel inside the city but also does not want to spend too much money on this project, as paving each road e between houses u and v has a certain cost c_e (different across the muddy roads). The mayor thus specifies two conditions:

- Enough streets must be paved so that everyone can travel from any house to another one using only the paved roads (you may assume that this is always possible);
- The paving should cost as little as possible.

You are chosen to help the mayor in this endeavor.

(a) Design and analyze an $O(m \log m)$ time algorithm for this problem. **(10 points)**

Solution. Solution to Problem 3 part (a) goes here.
To find the pave that costs as little as possible, I think we can use the kruskal's algorithm. For the reason that it sorts the edges in increasing order.
Algorithm:
1. Sort the roads in increasing order of their costs
2. Let F = empty
3.For i = 1 to m (in the sorted ordering of edges):
If adding e_i to F does not create a cycle, let F = F + e_i
4. return F
Proof of correctness:
It is the same proof of kruskal's algorithm, because we just used kruskal's algorithm to solve this problem.
Consider the forest F maintained by the algorithm. We prove that if F is MST-good at some iteration i and in that iteration we inserted an edge ei to F, then ei was safe with respect to F. This then implies that we only added safe edges to F. Moreover, the output of this algorithm isalways a tree since whenever we see an edge that does not create a cycle we add it to F(and since G was connected, we will find a tree eventually this way). That means that we added n-1 safe edges. This would imply the correctness of the algorithm as we now can say that this algorithm is an implementation of themeta-algorithm discussed in the previous lecture which we already proved its correctness.
Runtime analysis:
As what we did in kruskal's algorithm, we use union-find in only O(logn) time with a preprocessing of O(n) time .
So the total runtime would be O(n+mlogm+mlogn) =O(mlogm)

(b) The mayor of a neighboring city is feeling particularly generous and has made the following offer to Muddy City: they have identified a list of $O(\log m)$ different muddy roads in the city and are willing to entirely pay the cost of paving *exactly one* of them (in exchange for calling the new street after the neighboring city).

Design and analyze an $O(m \log m)$ time algorithm that identifies the paving of which of these roads, if any, the mayor should delegate to the neighboring city to further minimize the total cost—note that

if you decide to pave one of the roads payed by the neighboring city, you only need to pay a cost of 1 (for making a plaque of the name of the street). **(15 points)**

Hint: Design an algorithm that given a MST T of a graph G, and a single edge e, in only $O(m)$ time finds an MST T' for the graph G' obtained by changing the weight of the edge e to 1.

Solution. Solution to Problem 3 part (b) goes here.
Algorithm:
For this part of the problem, there is not a huge change in the algorithm, but it still has some difference
First, we have had the tree we build in the part (a) without any cycle in it. And we set a number count = 0.
Then, We know that the mayor of the neighbor ciry provides us with a list of O(logm) roads (these are the roads they can help us build)
We set the roads mayor provides us as z.
For z from 0 to logm:
if the road in the list is in the tree (the tree we had in (a)), then we set count = (max of $count_{old}$ and $count_{new}$) [if we replace the old count with the new one, we mark the road i, and delete the old mark we had]
If the road in the list is not in the tree, and it creates a cycle, then we find the road that has the max cost in the cycle, and we compare this cost and the old count, the bigger one of them would be the new count. [if the cost $> count_{old}$, we mark the road i, and delete the old mark we had]
At last, we get the max cost we can replace, and it is from road i.
So we just replace the cost of i with 1 (which means ask the mayor of neighbor to build this road)
proof of correctness:
What we get in (a) we used kruskal algorithm, so it does not need to prove
To get the road with maximum cost we want, we need to make sure that the road is in the list they provide.
So we need to go through the list of z first.
Then we have two situations: z is in the tree, or z is not in the tree but in the graph.
For the situation z is in the tree, it is easy because we can just replace that road with 1, but we need to make sure that we replace the max road, so we can just record its cost and its location, and compare it with other z later.
For the situation z is not in the tree, we would build a cycle in the tree (because we need to connect all the cities in the tree, if we add one more edge, it must create a cycle), if we have a cycle, we just need to find the road which has the max cost, and delete it from the tree, and add z into the tree. Again, we are not sure if the one we want to delete is the best choice for us, so we record its cost and its location, and compare it later.
After comparing what we had above, we got our best choice, and the algorithm is proved.
Runtime analysis:
Time for go through the list is O(logm), and for going through the tree to find it z is in the tree we need at most m time, and the runtime for the kruskal's algorithm is mlogm
So the total runtime is O(mlogm) + O(mlogm) = O(mlogm)

Problem 4. You are given a weighted undirected graph $G = (V, E)$ with integer weights $w_e \in \{1, 2, \ldots, W\}$ on each edge e, where $W = O(1)$. Given two vertices $s, t \in V$, the goal is to find the minimum weight path (or shortest path) from s to t. Recall that Dijkstra's algorithm solves this problem in $O(n + m \log m)$ time even if we do not have the condition that $W = O(1)$. However, we now want to use this extra condition to design an even faster algorithm.

Design and analyze an algorithm to find the minimum weight (shortest) s-t path in $O(n + m)$ time.

Solution. Solution to Problem 4 goes here.
We still use Dijkstra's Algorithm here, but we have W = O(1) instead
Algorithm:
1. Let mark[1:n] = FALSE and s be the designated source vertex
2. Let d[1:n] = infinity and set d[s] = 0
3. Set mark[s] = TRUE and let S be the set of edges incident on s and assign a value value(e) = d[s] + w_e to each of these edges
4. While S is non-empty:
(a). Let e = (u,v) be the minimum value edge in S and remove e from S.
(b). If mark[v] = TRUE ignore this edge and go to the next iteration of the while-loop.
(c). Otherwise, set mark[v] = TRUE, d[v] = value(e), and insert all edges e' incident on v to S with value(e') = d[v] + w_e'.
Retunr d.
Proof of correctness:
Because what we use here is the same as mentioned in the lecture of the Dijkstra's Algorithm part, and it is proved in the lecture.
The only difference is that w_e must belong to O(1) now
We prove by induction that in every iteration of the while-loop in the algorithm
basecase:
iteration 0 of the while-loop is true since s is the closest vertex to s and d[s] = dist(s,s) = 0 satisfying part two.
Now suppose this is true for some iteration i of the while-loop and we prove it for iteration i+ 1. Let e= (u, v) be the edge removed fromSin this iteration. If mark[v] =TRUE we simply ignore this edge and hence the set C remains the same after this step and by induction hypothesis, we have the above two properties. Now suppose mark[v] = FALSE. In this case
Every edge among the cut edges of (C, V-C) belong to the set S at this point. We are setting d[v] =value(e) =d[u] + w_e=dist(s, u) + w_e
Since dist(s, w)> dist(s, u) for all w belongs to V - C by induction hypothesis, we know that the shortest path from s to v does not visit any of the vertices in V-C, hence, by setting d[v] =dist(s, u) + w_e where edge e minimizes the right hand side of this equation, we will have that d[v] =dist(s, v). This proves the second part of the induction hypothesis. For the first part also, notice that value(e) is minimized and hence v is the "closest" vertex to s in V-C and hence after adding v to C,dist(s, w)> dist(s, v) for all w belongs to V-C. This proves the second part of the induction hypothesis.
(the above part is the same as it is mentioned in the lecture)
Runtime analysis:
This is the biggest diffrerent part from the normal Dijkstra's Algorithm
In the lecture notes, we know that we used min-heap to help us run the Dijkstra's Algorithm much faster
But here, we do not need to apply the min-heap method for the reason that we have known that W = O(1), and it means that we do not have logm runtime for the min-heap part here, but O(1) time
So by iteration of while-loop to extract the minimum and O(deg(v)*

Challenge Yourself. A **bottleneck spanning tree (BST)** of a undirected connected graph $G = (V, E)$ with positive weights w_e over each edge e, is a spanning tree T of G such that the weight of maximum-weight edge in T is minimized across all spanning trees of G. In other words, if we define the cost of T as $\max_{e \in T} w_e$, a BST has minimum cost across all spanning trees of G. Design and analyze an $O(n + m)$ time algorithm for finding a BST of a given graph. **(+10 points)**

Fun with Algorithms. Let us go back to the bottleneck spanning tree (BST) problem defined above.

(a) Prove that any MST of any graph G is also a BST. Use this to obtain an $O(m \log m)$ time algorithm for the BST problem (notice that this is slower than the algorithm from the previous question).

(+8 points)

(b) Give an example of a BST of some graph G which is *not* an MST in G. **(+2 points)**