**CS2040S: Data Structures and Algorithms**

# Discussion Group Problems for Week 11

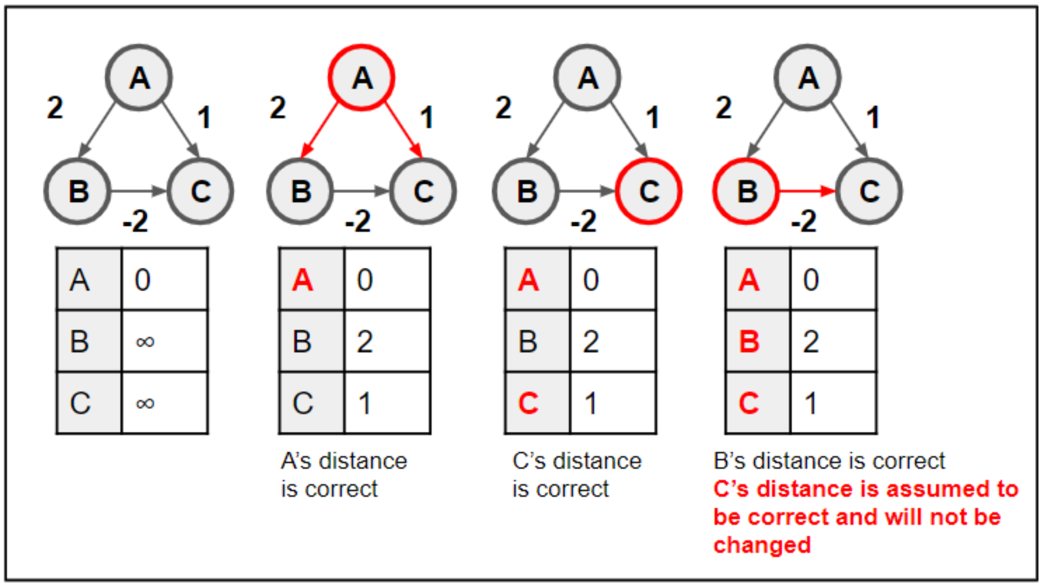
*For: March 31–April 4*

In this week's tutorial, we will be focusing on:

- Shortest Paths
- Dijkstra's Algorithm
- Directed Acyclic Graphs

**Problem 1.** (Bad Dijkstra)

**Problem 1.a.** Give an example of a graph where Dijkstra's algorithm returns the wrong answer.



**Solution:** The idea here is to create a graph with *negative* weights so that if you run Dijkstra's algorithm it fails. The graph shown above is one possible way to construct such a graph.

**Problem 1.b.** Give examples of graphs where if we used Dijkstra's algorithm, it would output the correct answer. And yet it would be unsuitable.

**Solution:** The goal of this part is to learn to evaluate and choose the right SSSP algorithm for different contexts. These are some examples:

- For a tree, BFS/DFS is more efficient than Dijkstra's algorithm. BFS/DFS takes $O(V + E)$ compared to Dijkstra's $O(E \log V)$.
- For DAG, finding a topological sort then relaxing the outgoing edges in order is more efficient than Dijkstra's. Finding a topological sort and relaxing the edges in order takes $O(V + E)$ compared to Dijkstra's $O(E \log V)$.

**Problem 2.** (Pizza Pirate Encounters)
Related Kattis Problems:

- https://open.kattis.com/problems/arbitrage
- https://open.kattis.com/problems/getshorty

Welcome to Mel's Pizzeria! We're here to deliver pizzas for them (given how tough the economy is, we're resorting to side hustles). We're given an undirected graph $G = (V, E)$ that represents the locations that we need to deliver to. We have source node $s$ that represents Mel's Pizzeria, and we have a node $t$ that represents our destination. We plan on delivering pizzas to places that we pass by on the way to home. Here's the catch! The city is rife with Pizza Pirates! With every edge $e$ that we take, the pirates will leave us with $0 < f_e \leq 1$ fraction of whatever pizza we were carrying.

For example, if we started from node $s$, and took edges $(s, a)$, $(a, b)$, $(b, t)$, where the edge $(s, a)$ has fraction 0.25 the edge $(a, b)$ has fraction 0.6 and the edge $(b, t)$ has fraction 0.1, then the remaining pizza we have is $0.25 \times 0.6 \times 0.1 = 0.015$.

We want to maximise the amount of pizza remaining.

**Problem 2.a.** Design and analyze an algorithm to determine the path from a given start vertex $s$ to a given target vertex $t$ that maximises the fraction of pizza left. We want to run Dijkstra's. For this part, think about which parts of the algorithm we should change in order to make it work.

**Solution:** Maintain a distance estimate array $d$ (you can call it fraction estimate if you want) for each node. Initially each node has fraction 0 (nothing), except for node $s$, which has fraction 1 (entire pizza). Enqueue nodes based on max-priority. In terms of the relax operation, instead of adding the estimates, we instead multiply them. For a given edge $(u, v)$, if $d[u] \times w(u, v) > d[v]$, then we increase the priority of node $v$ to $d[u] \times w(u, v)$.

**Problem 2.b.** Are there any other ways of finding the safest path without modifying Dijkstra's algorithm? Can we modify the graph instead?

**Solution:** First we modify all the weights $f_e$ such that they become $-\log(f_e)$ instead. Instead of multiplying our weights, we are now instead adding $-\log(f_e)$ (bear in mind that all edges are now positively weighted). Since $\log(a \cdot b) = \log(a) + \log(b)$, the shortest path minimises the sum of weights, which minimises $-\sum_{e \in P} \log(f_e) = -\log\left(\prod_{e \in P} f_e\right)$ where $P$ is the path we take. This in turn maximises $\log\left(\prod_{e \in P} f_e\right)$. Since log is monotonically increasing, this means we maximise $\prod_{e \in P} f_e$.

**Problem 3.** (Longest Path)
Given a directed acyclic graph $G$, compute the longest possible path that you can take in the graph.

**Problem 3.a.** For a node $u$, let $N(u)$ be the set of nodes that node $u$ has edges to (think of $N(u)$ as the set of neighbours that node $u$ can reach via a single edge). How do we find the longest possible path in graph $G$ that starts at node $u$? Write this as a recurrence. In particular $lp(u)$ should be written in terms of $lp(v)$ for $v \in N(u)$.

**Solution:** $lp(u) = \max_{v \in N(u)}\{lp(v)\} + 1$
In English: The length of the max path formed by starting at $u$, is the **length of the max path** formed by starting at any of the neighbours $v$ of $u$ (in $N(u)$), and adding 1 to it (because we can now include $u$ as the start of such a path).

**Problem 3.b.** For a given node $u$, how long does it take to compute $lp(u)$?

**Hint:** Can we say to compute this value, it is basically doing some kind of traversal on the graph? What kind?

**Solution:** Starting at node $u$, we need to recursively compute $lp(v)$ for all neighbours $v$ of $u$. After we do so, as a post-traversal operation, we take the maximum of the computed values, and add 1 to it.
This is basically again, DFS with a post-traversal operation. So $O(V + E)$.

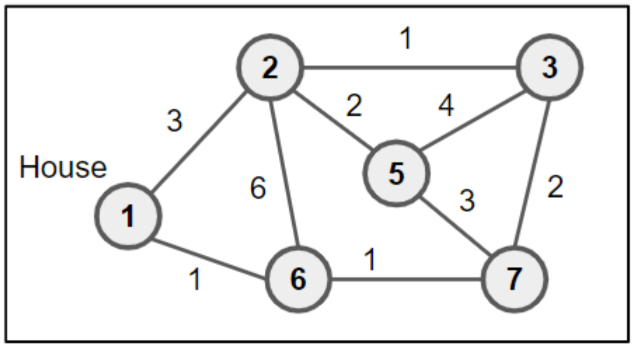
**Problem 3.c.** We want to find the maximum possible, which means we wish to compute $\max_{u \in V}\{lp(u)\}$. However, whatever your answer to the previous part might be, I think we can all agree it is not $O(1)$. Which probably means naively running the solution to the previous part but $v$ times for every node in the graph is potentially very expensive. Imagine a DAG like a linked list, and we ran the algorithm to compute $lp(u)$ for the tail node, then ran a fresh instance of the algorithm again to compute $lp(v)$ where $v$ is $u's$ parent. And then another fresh instance to compute $lp$ for its parent and so on. This likely does not only take $O(V + E)$ time.

Can we think of smart ordering in which we should compute the nodes so that we can compute the maximum possible path length in $O(V + E)$ time?

**Hint:** It's a DAG, what kind of ordering makes sense? Can we also store intermediate answers at each node to help us speed up our computation?

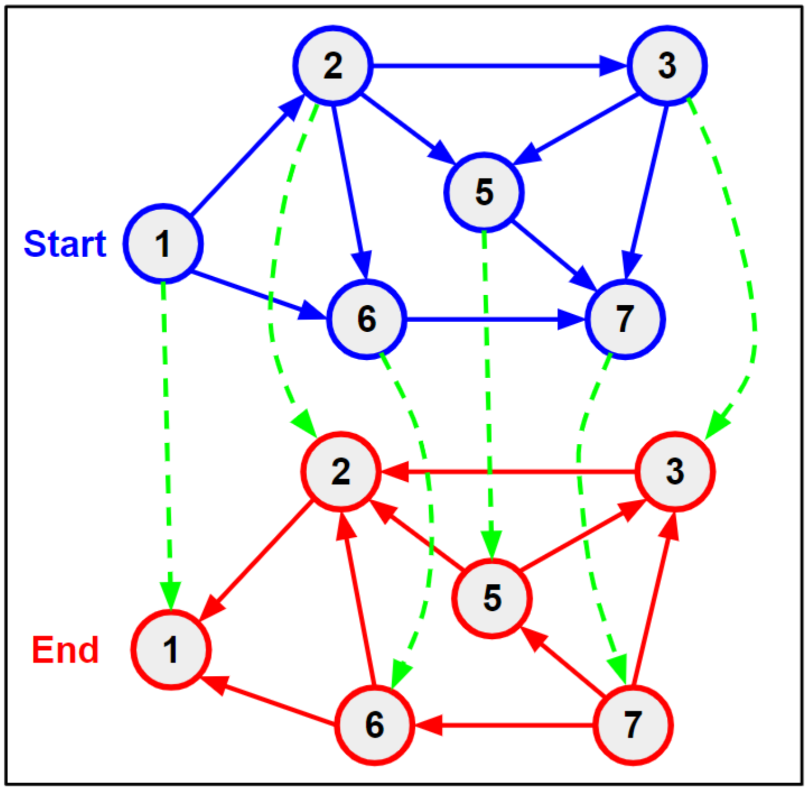
**Solution:** Toposort the graph, and compute $lp(u)$ in **reverse toposort order**. For a given node $u$, after we compute $lp(u)$, store it somewhere (like an array or a hashtable) so that we can retrieve the the value again later. This is basically **memoization**.
This way given a node $v$, when it needs to compute $\max_{v \in N(v)}\{lp(v)\}$, because we computed and stored the values in reverse toposort order, the values for $lp(v)$ have been computed previously and can be retrieved in $O(1)$ time. So the cost of computing $\max_{v \in N(v)}\{lp(v)\} + 1$ now costs $O(|N(v)|) = O(outdeg(v))$ time.

**Problem 4.** (Running Trails)



I want to go for a run. I want to go for a long run, starting from my home and ending at my home. And I want the first part of the run to be only uphill, and the second part of the run to be only downhill. I have a trail map of the nearby national park, where each location is represented as a node and each trail segment as an edge. For each node, I have the elevation (value shown in the node). Find me the longest possible run that goes first uphill and then downhill, with only one change of direction.

**Hint:** You might want to turn this into a kind of graph where the previous question can *just be called* on your graph to help compute the correct answer.



**Solution:** First, we want to model this as a more useful graph. Create two copies of the map graph, where in the first each edge is directed uphill (blue graph) and in the second each edge is directed downhill (red graph). Connect each node in the first copy with a directed edge to the corresponding node in the second copy (green edges). Edges connecting two nodes with the same height are omitted.
The problem now reduces to finding the longest route from the source (your home) in the first copy to the destination (your home) in the second copy.
In general, longest path is a hard problem (so hard that is in fact an open problem whether we can solve it efficiently). Luckily, there is a special property here: this is a directed acyclic graph. Notice there cannot be a cycle in the graph. If there were a cycle in the first copy, that would imply a cycle of only uphill edges, which is impossible. Similarly, there can't be a cycle in the second copy. And there are no edges from the second copy back to the first copy.
So we can solve this problem by using the algorithm for finding the longest path in a directed acyclic graph, i.e., negate the weights, find a topological order of the graph, and relax the outgoing edges of each node in order.

**Problem 5.** (A Random Problem with a dude called Dan)

**Problem 5.a.** Dan is on his way home from work. The city he lives in is made up of $N$ locations, labelled from 0 to $(N-1)$. His workplace is at location 0 and his home is at location $(N-1)$. These locations are connected by $M$ **directed** roads, each with an associated *non-negative* cost. To go through a road, Dan will need to pay the cost associated with that road. Usually, Dan

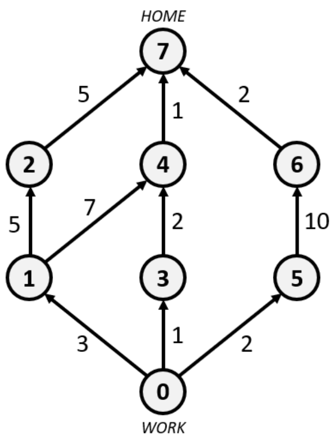


**Figure 1:** Example city 1

would try to take the cheapest path home.

The thing is, Dan has just received his salary! For reasons unknown, he wants to flaunt his wealth by going through a *really expensive road.* However, he still needs to be able to make it back home with the money he has. Given that Dan can afford to spend up to $D$ dollars on transportation, help him find **the cost of the most expensive road that he can afford to go through** on his journey back home.

Take note than Dan only cares about the most expensive road in his journey; the rest of the journey can be really cheap, or just as expensive, so long as the entire journey fits within his budget of $D$ dollars. He is also completely focused on this goal and does not mind visiting the same location multiple times, or going through the same road multiple times.

For example, suppose Dan's budget is $D = 13$ dollars. Consider the city given in Figure 1, consisting of $N = 8$ locations and $M = 10$ roads.

The path that Dan will take is $0 \to 1 \to 4 \to 7$. In this journey, the total cost is 11 dollars and the most expensive road has a cost of 7 dollars - the road from locations 1 to 4. Therefore, the expected output for this example would be "7".

Note that this path is neither the cheapest path ($0 \to 3 \to 4 \to 7$), nor is it the most expensive path that fits within his budget of 13 dollars ($0 \to 1 \to 2 \to 7$).

There is also a more expensive road within this city - the road from locations 5 to 6 with a cost of 10 dollars. However, the only path that goes through this road, $0 \to 5 \to 6 \to 7$, has a total cost of 14 dollars which exceeds Dan's budget.

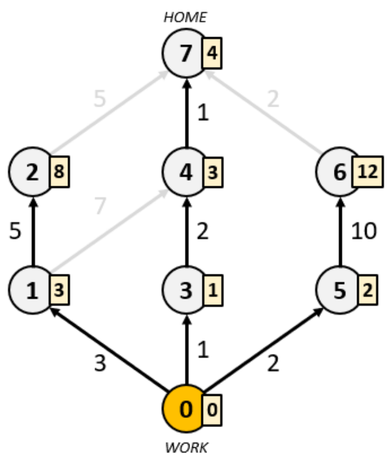
**Solution:** There are two possible solutions, both with differing approaches.
The first solution tests every edge ($u \to$ v) and checks if the sum of the following does not exceed $D$:

- The (weight of the) shortest path from vertex 0 to $u$
- The weight of the edge $u \to v$
- The (weight of the) shortest path from $v$ to vertex $N - 1$

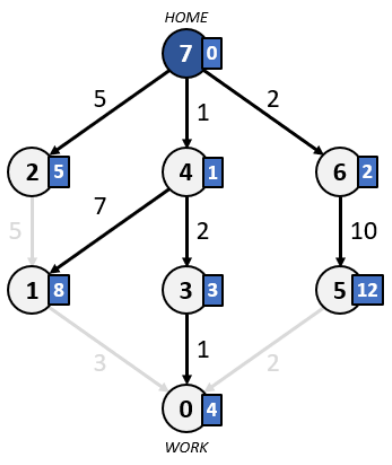
If (and only if) that sum does not exceed $D$, then Dan can afford to go through the edge ($u \to v$) on his trip home. Therefore, a simple algorithm would be to iterate through all the edges and find the edge with the maximum weight that satisfies the aforementioned property.
To perform the above check efficiently, we will need to perform some pre-processing. In particular, we need an efficient way to obtain the following:

- The shortest path from vertex 0 to any vertex.
- The shortest path from any vertex to vertex $N - 1$.

The former is achieved by performing Dijkstra's algorithm from vertex 0. This will get us the shortest path from vertex 0 to all other vertices in the graph.
The latter is achieved by performing Dijkstra's algorithm from vertex $N-1$ on the transpose graph (i.e. the same graph, but with all the edge directions reversed). For any vertex $u$, the shortest path from vertex $N - 1$ to $u$ in the transpose graph represents the shortest path from $u$ to vertex $N - 1$ in the original graph. Therefore, this will get us the shortest path from all vertices in the graph to vertex $N - 1$.



Original Graph



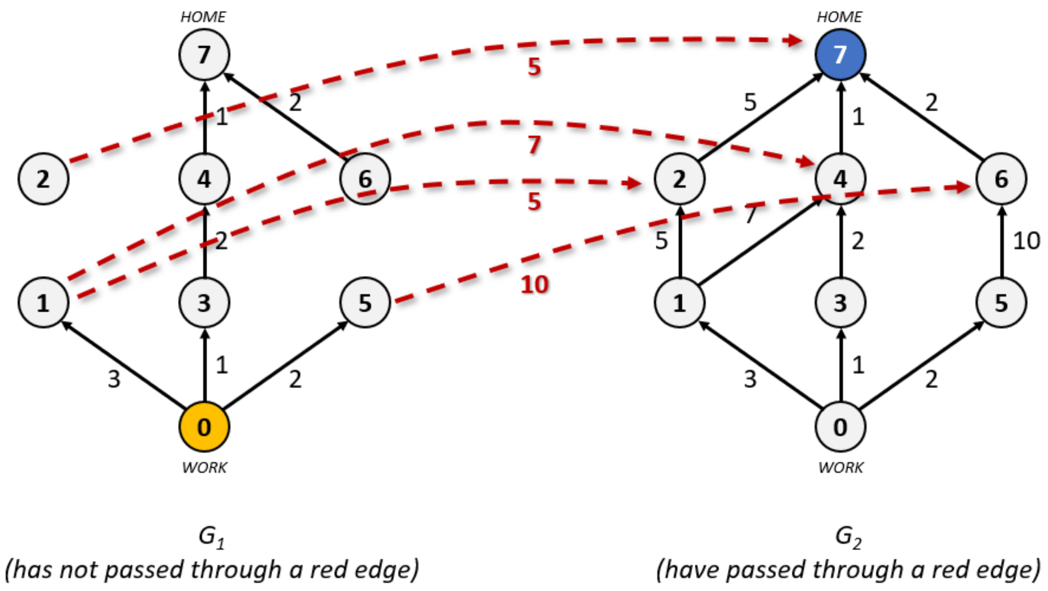
Transpose Graph

Now we can test each edge in $O(1)$ time. Including the time taken during the pre-processing, the overall time complexity is $O(M \log N + M) = O(M \log N)$.

**Solution:** For the second solution, we will define the following property:
Let $P_K$ denote the property that there exists a path *(with possibly repeating edges)* from vertices 0 to $N-1$ such that:

- The path goes through an edge of weight at least $K$.
- The total weight of the path does not exceed $D$.

For example, in the given example, $P_4$ is True because the path $(0 \to 1 \to 2 \to 7)$ satisfies the above conditions. However, $P_9$ is False because there is no such path.
It follows that the answer we are looking for is the largest value of $K$ for which $P_K$ is true. Let $K_{\max}$ represent that largest value. Notice that $P_K$ will also be true for any value $K \leq K_{\max}$. This means we can find $K_{\max}$ by binary searching on the set of all edge weights in the graph. In the example, this means binary searching on the array $[1, 1, 2, 2, 2, 3, 5, 5, 7, 10]$.
Testing if $P_K$ is true for some value of $K$ can be done in the following way:
For all the edges in the graph of weight at least $K$, colour those edges red. Now, we want to find a path from vertices 0 to $N-1$ while forcing it through a red edge. Duplicate the graph, labelling them $G_1$ and $G_2$. For any red edge $(u \to v)$, connect that edge from $u$ in $G_1$ to $v$ in $G_2$.
For example, if we're testing $P_5$ in the example graph, the duplicated graph looks like this:


HOME
7
2
4
6
1
3
5
0
WORK
3
1
2
1
2
2
5
7
5
10
HOME
7
5
1
2
2
4
6
5
7
2
10
1
3
5
3
1
2
0
WORK
$G_1$
(has not passed through a red edge)
$G_2$
(have passed through a red edge)

Now, any path from vertex 0 in $G_1$ to vertex $N-1$ in $G_2$ must pass through a red edge, since they are the only edges joining $G_1$ and $G_2$. Therefore, to check if $P_K$ is true, we simply need to check the shortest path from vertex 0 in $G_1$ to vertex $N-1$ in $G_2$. $P_K$ is true if and only if the shortest path does not exceed $D$.
Testing each value of $K$ takes $O(2M \log(2N)) = O(M \log N)$, for a total time complexity of $O(M \log N \log M) = O(M(\log N)^2)$

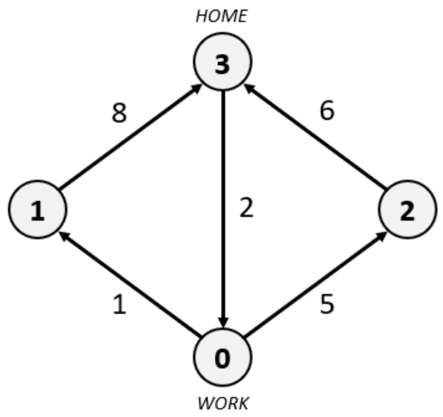


**Figure 2:** Example city 2

**Problem 5.b.** *(Optional)* Another month, another salary for Dan to flaunt. The situation is similar to that of the previous part.

This time, however, instead of maximizing the cost of the *most expensive road* in his journey, he wants to maximize the cost of *the second most expensive road* in his journey. In other words, he no longer cares about the most expensive road in his journey; that road can be 100 times more expensive than the second most expensive road in his journey for all he cares.

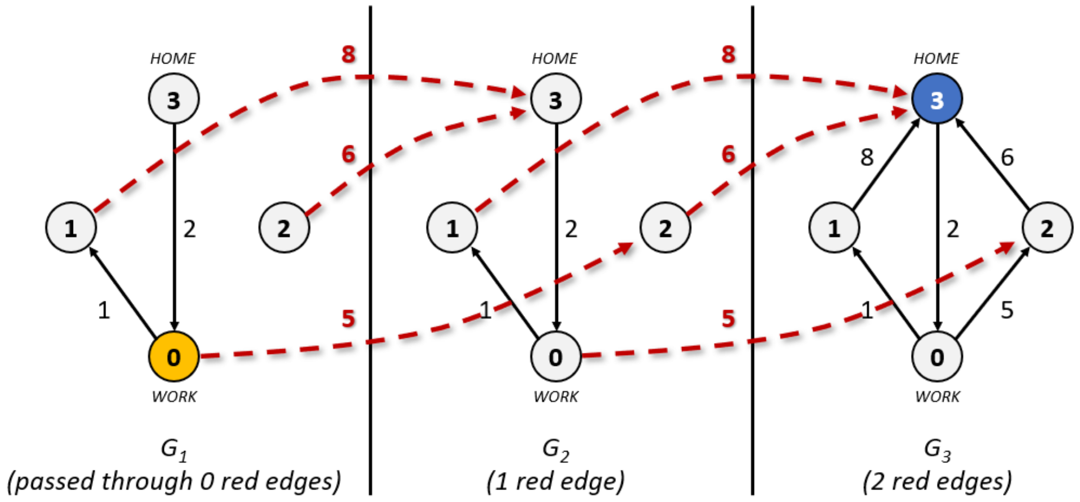
For example, consider the city in Figure 2 with $N = 4$ locations and $M = 5$ roads.

If Dan's budget is $D = 12$ dollars, the path that he will take is $0 \to 2 \to 3$. In this journey, the total cost is 11 dollars and the second most expensive road has a cost of 5 dollars, the road from locations 0 to 2. Therefore, the expected output for this example would be "5".

Notice that while he can afford to go through the path $0 \to 1 \to 3$ with an expensive 8 dollar road, the second most expensive road in that journey only costs 1 dollar.

If Dan's budget is $D = 20$ dollars, then the path he will take is $0 \to 1 \to 3 \to 0 \to 1 \to 3$. As irrational as this 20 dollar journey is, it allows him to go through the road from locations 1 to 3 twice, thus making the second most expensive road in his journey cost 8 dollars.

**Solution:** We can't really use the first solution from the previous part, but we can adapt the second solution.
Simply duplicate the graph twice to force a path to go through 2 red edges. For example, to test for $P_5$ on the example graph, the doubly-duplicated graph will look like this:



Testing each value of $K$ takes $O(3M \log(3N)) = O(M \log N)$. Binary searching over the $M$ possible edge weights, we have a total time complexity of $O(M \log N \log M) = O(M(\log N)^2)$, the same as the previous part.