Homework 6

CMPSC 465

Kinner Parikh

October 20, 2022

Problem 1:

I worked with Sahil Kuwadia and Ethan Yeung
I did not consult without anyone my group member
I did not consult any non-class materials

Problem 2: *DFS Variations*

a) To find if there is a feasible path from s to t, we can simply run DFS on the graph starting at s, ignoring all edges whose weight is greater than L. Thus, the only edges we can travel are those for which $\ell_e \leq L$.

Algorithm 1: feasibleRoute

```
Input  : $G = (V, (E, \ell_e))$ (in adjacency list format), $s$, $t$, $L$
Output: A feasible path exists or not
Initialize an empty stack, $st$, that will store nodes
Create visited list for all nodes and set all to false
Push $s$ into $st$
while $st$ is not empty do
    $x \leftarrow$ pop first element in $st$
    if $x$ not in visited then
        add $x$ to visited
        forall neighbors $a$ of $x$ do
            if $\ell_a \leq L$ then
                if $a$ is $t$ then
                    return True
                end if
                push $a$ onto stack
            end if
        end forall
    end if
end while
return False
```

<u>Proof of correctness</u>

Initialization:

The stack is initialized with only s at the beginning. This represents the first city you leave from, thus the base case holds.

Maintenance:

Assuming we are at an arbitrary city k, which means that there is a path to k that is valid and we know k is the top of the stack, we can pop k off. If k has already been visited, then we don't do anything. If it hasn't been visited, then we add k to the visited list and look at the cities that are adjacent to k. We can iterate through all the cities that are adjacent to k, and check if the weights on those edges are less than L. If they are, then we can push them onto the stack. When we push them onto the stack, we know that they are reachable from k, thus we can check if the adjacent node is t, in which case we know that t is reachable, and we can return true. If it isn't, we simply continue.

Termination:

When the stack is empty, which means that all the reachable nodes have been visited, we can confidently say that there is no feasible route from s to t and return False. $\square$

<u>Runtime:</u> Since this algorithm is simply DFS, and all the modifications we make to the lgorithm complete in constant time, this will run in $O(|V| + |E|)$.

b) To find the minimum fuel tank capacity we need to find a path that has the least maximum distance. We start by finding all the unique edge distances in the graph. We can go through the graph's adjacency list and add all the distances to list l, making sure not to add duplicate distances into l. Once we have all distances, we sort l using merge sort such that all the unique distances are increasing in order. We can then call leastMaxDistance using this sorted list.

Algorithm 2: leastMaxDistance

```
Input  : $l$, $G = (V, (E, \ell_e))$ (in adjacency list format), $s$, $t$
Output: Least maximum distance if path exists, -1 if a path does not exist
1  low ← 1
2  high ← length of $l$
3  lowMax ← -1
4  while $low \neq high$ do
5      mid ← (low + high) / 2
6      if feasibleRoute($G, s, t, l[mid]$) is True then
7          high := mid - 1
8          lowMax := $l$[mid]
9      end if
10     else
11         low := mid + 1
12     end if
13 end while
14 return lowMax
```

Proof of correctness:

Initialization:

We can assume that we have a sorted array l with all the distances. We initialize the current lowest maximum distance (lowMax) to -1. We find the lower value (low) to be 1 and the upper value (high) to be the number of entries in l. This sets up the bounds for which we will search through the array, thus the statement holds.

Maintenance:

Assuming we are at arbitrary iteration k, we know that $\text{low}_k < \text{high}_k$. Running through the loop, we find that mid_k is $(\text{low}_k + \text{high}_k)/2$. We now check if there is a feasible route on the length $l[\text{mid}_k]$. If a feasible route does exist, then we can set high (which will be high_{k+1}) to mid_k - 1 and lowMax to $l[\text{mid}_k]$. In this case, we know that $\text{low}_{k+1} = \text{low}_k$. If a path does not exist, then we set low (which will be low_{k+1}) to mid + 1.

Termination:

When low is equal to high, then we know we must terminate our loop and return whatever is in lowMax. If there is never a feasible route, then lowMax will be -1. □

Runtime:

Gathering all the edges will take $\Theta(2E)$. Then sorting will be $O(E \log E)$. The binary traversal takes $\Theta(\log E)$. And with each iteration of the binary traversal, feasibleRoute takes $O(|V|+|E|)$. So, the leastMaxDistance will take $\Theta(2E) + O(E \log E) + \Theta(\log E) \cdot O(|V| + |E|)$ which is equal to $O((|V| + |E|) \log E)$.

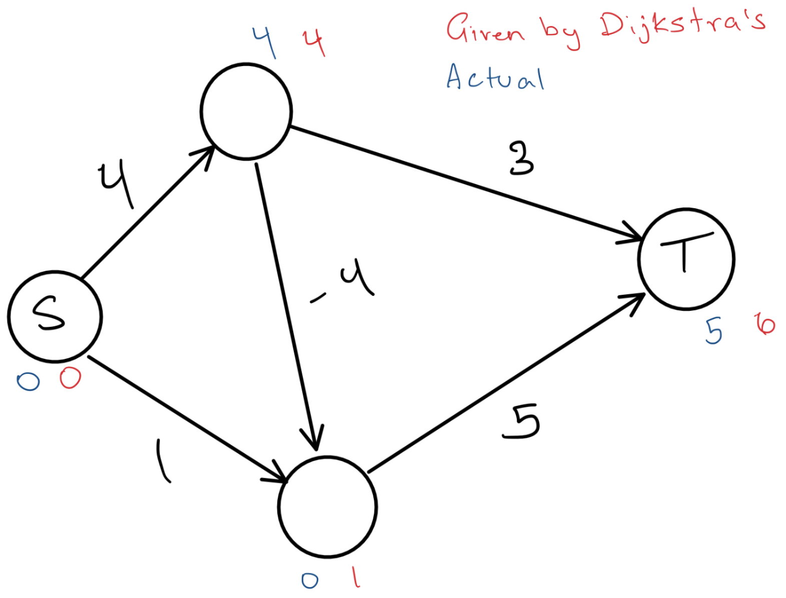
Problem 3: *Shortest bitonic paths*

a) Because the property states that all paths from s to v are strictly increasing and unique, we can determine that there is exactly one order of increasing weights for the path of s to v. Because of this fact, we can get the path from s to v and its corresponding edge weights, and sort these weights ($O(E \log E)$). Then we can relax the edges in that order, which will take $O(E)$ time. This is an adaptation of the Bellman-Ford Algorithm, we can say that the preprocessing will take $O(V)$, thus the overall runtime will be $O(E \log E) + O(E) + O(V) = O(V + E \log E)$.

b) Similar to the algorithm above, because the paths are strictly bitonic, this means that there are exactly two orders of increasing and two orders of decreasing weights for the path from s to v. Applying the same principle above, we can relax the edges of the graph first in the order of the two increasing weight paths then the two decreasing weight paths. Thus, same to the algorithm above, sorting will take $O(E \log E)$ and relaxing the edges will take $O(E)$, so the final runtime will be $O(V + E \log E)$.

Problem 4: *Dijkstra's on negative*

a)



b)

The termination of the while loop is that the heap is empty. Every single time the loop runs, the vertex that has the minimum distance is removed from H. In the event of a negative edge going to a node that has already been removed, it will update the distance of that already removed node, but it will not update the rest of the nodes that come after the removed node because it will never be visited in the loop again.