Design and Analysis of Algorithms CS 575, Spring 2023

Theory Assignment 3.2

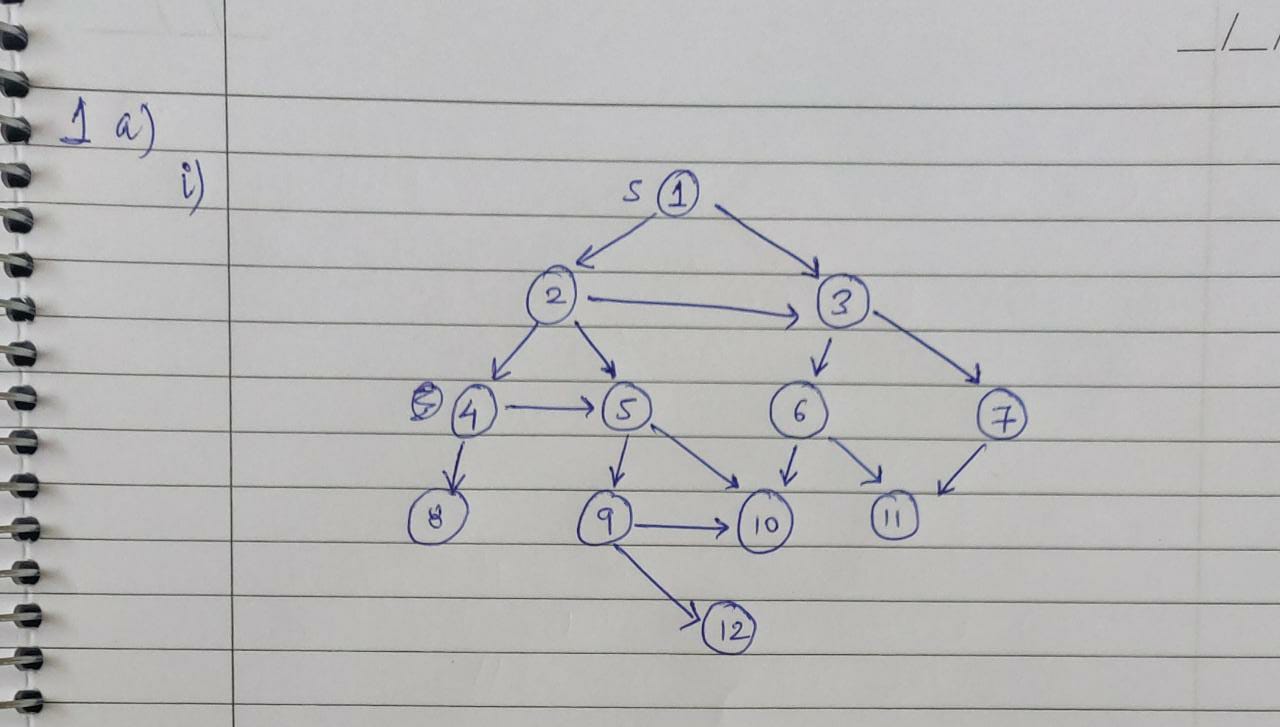
Due on 4/13/23 (Thursday) at 11:59pm

1. [20 points] In this problem, you will apply different traversal methods on a given directed graph. The start node is denoted by S. When arbitrary decisions on order must be made assume that child nodes are visited from left to right.
   1. [10 points] Perform a breadth-first search on the directed graph below (on the left). (i) *Number the nodes* according to the order in which they are visited (become gray). For example, the first node visited is *S* so *S* is numbered 1. Then, the left child of *S* is visited so it is numbered 2. Show the order numbers inside the circles. (ii) Show the distance of each node to *S* beside each circle. (iii) Show the breadth-first tree.



*S* 1

2



Diagram

Description automatically generatedLetter

Description automatically generated

* 1. [10 points] Perform a depth-first search on the directed graph below (on the left). (i) Number the nodes according to the order in which they are visited at the first time (when they become gray). Show the order numbers inside the circles. (ii) Show the depth-first tree(s).



*S*

Diagram

Description automatically generatedLetter

Description automatically generated

1. [15 points] Please modify the depth first search algorithm (slide 36 and 37 of the graphs basics lecture notes) to find all connected components in an undirected graph. Comment on where you made the modification. Your modified algorithm needs to print out each component ID (starting from 1) and the corresponding vertices. For example, your output

for the DFS example (slide 38-41 of the graphs basics lecture notes) will look like the following: Component 1: u, v, y, x. Component 2: w, z.

Solution:

DFS-Visit(u)

{

time = time + 1;

d[u] = time;

color[u] = gray;

// Add u to the current component

componentId[u] = currentComponentId;

verticesInComponent[currentComponentId].add(u);

for each v in adj[u] do

{

if color[v] == white

{

parent[v] = u;

DFS-Visit(v);

}

}

color[u] = red;

time = time + 1;

f[u] = time;

}

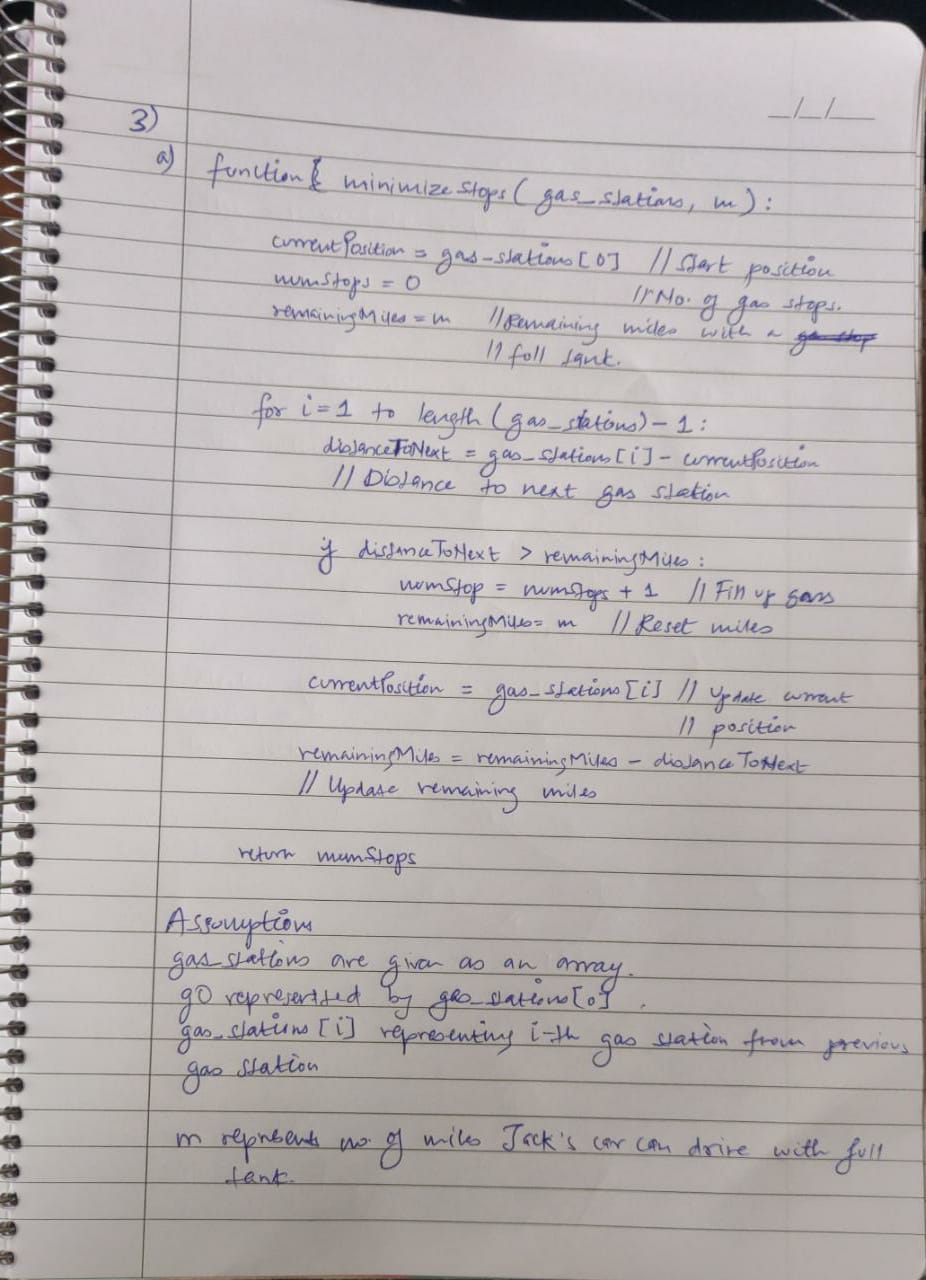
Modifications made:

* Added a componentId array to keep track of the component ID (starting from 1) for each vertex during DFS traversal.
* Added a verticesInComponent array to store the vertices in each connected component.
* During DFS traversal, when a new vertex u is visited, it is added to the current component using componentId[u] = currentComponentId and verticesInComponent[currentComponentId].add(u).
* When printing out the connected components, you can iterate over the verticesInComponent array and print the vertices for each component along with the corresponding component ID.

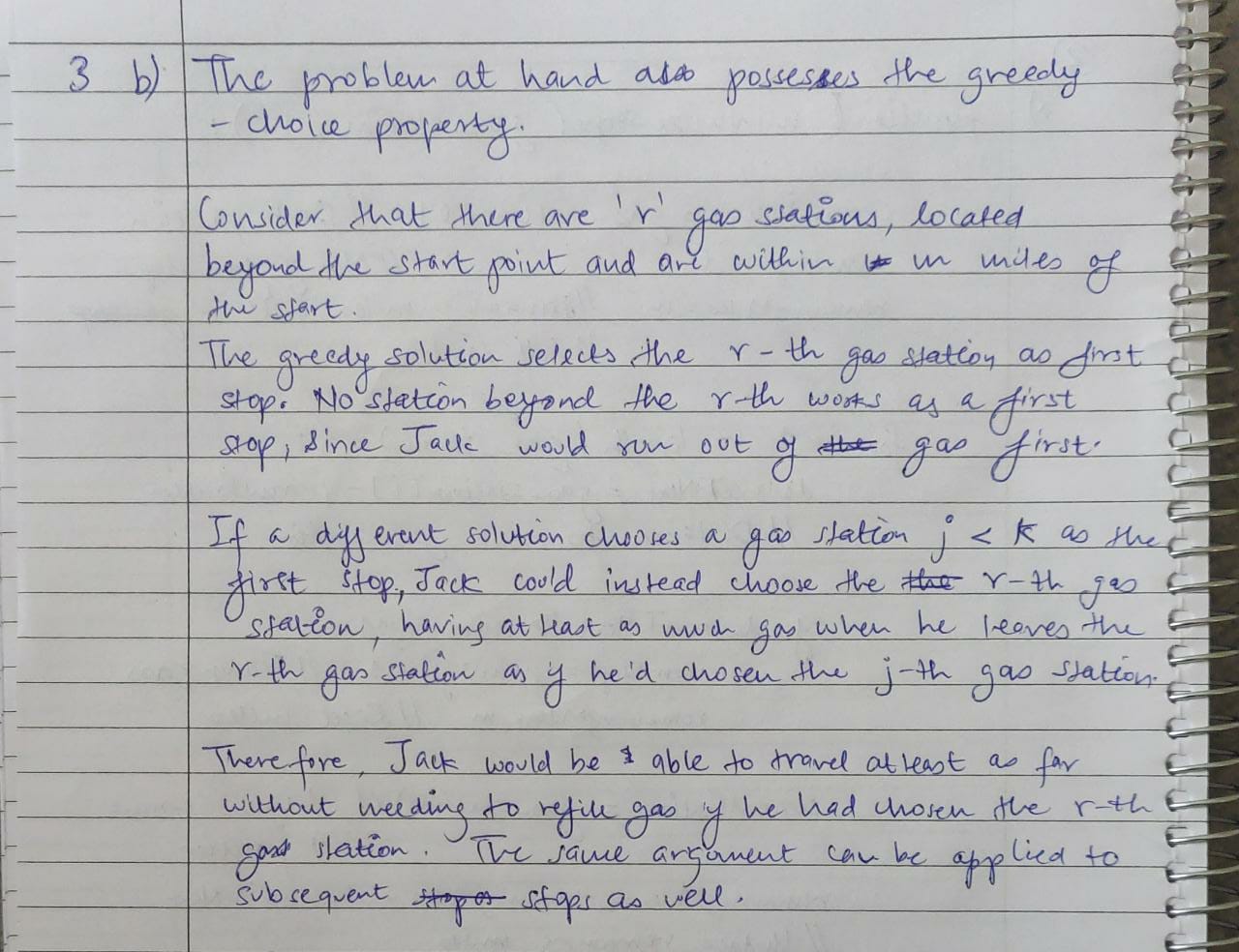
1. *[25 points]* Jack plans to drive from city *A* to city *B* along a highway. Suppose *g*0, *g*1, *g*2,

…, *gk* are the gas stations along the highway and are ordered in increasing distance (in miles) from city *A*, where *g*0 is located at the starting place (city *A*). Let *di* be the distance (in miles) between *gi* and *gi*+1 , *i* = 0, …, *k* – 1, and we assume that these distances are known to Jack. Each time Jack fills gas to his car, he gets a full tank of gas. Jack also knows the number of miles, *m*, his car can drive with a full tank of gas. Jack’s goal is to minimize the number of times he needs to stop at gas stations for his trip. You can assume that Jack starts from *g*0 with a full tank.

1. *[8 points]* Design a greedy algorithm to solve the above problem, i.e., to minimize the number of stops for gas.



1. *[12 points]* Show that your greedy algorithm has the *greedy choice property*, i.e., each local decision will lead to the optimal solution. Basically you need to argue for the correctness of your algorithm.

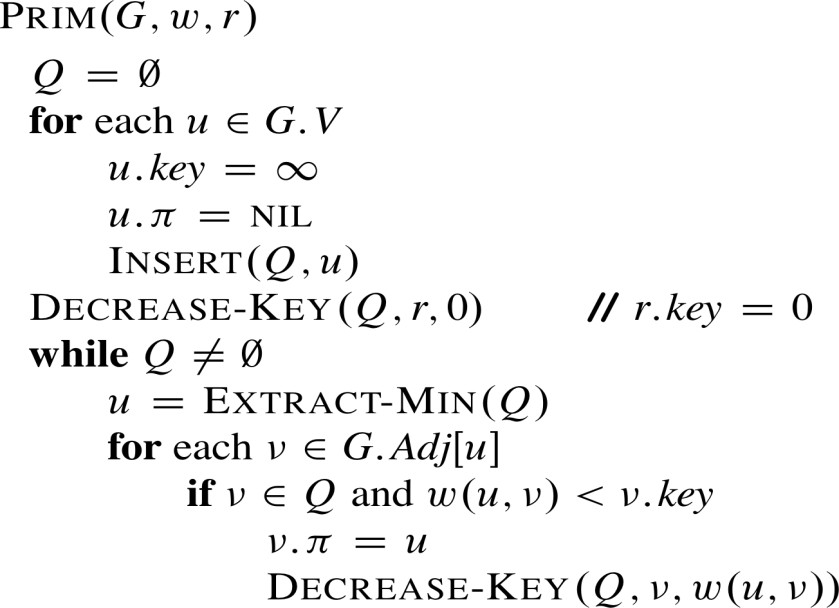


1. *[5 points]* What is the running time of your algorithm?

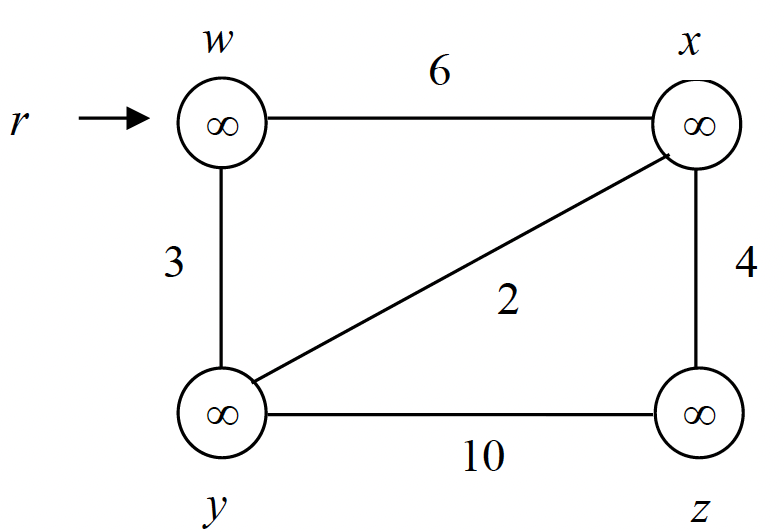
Text, letter

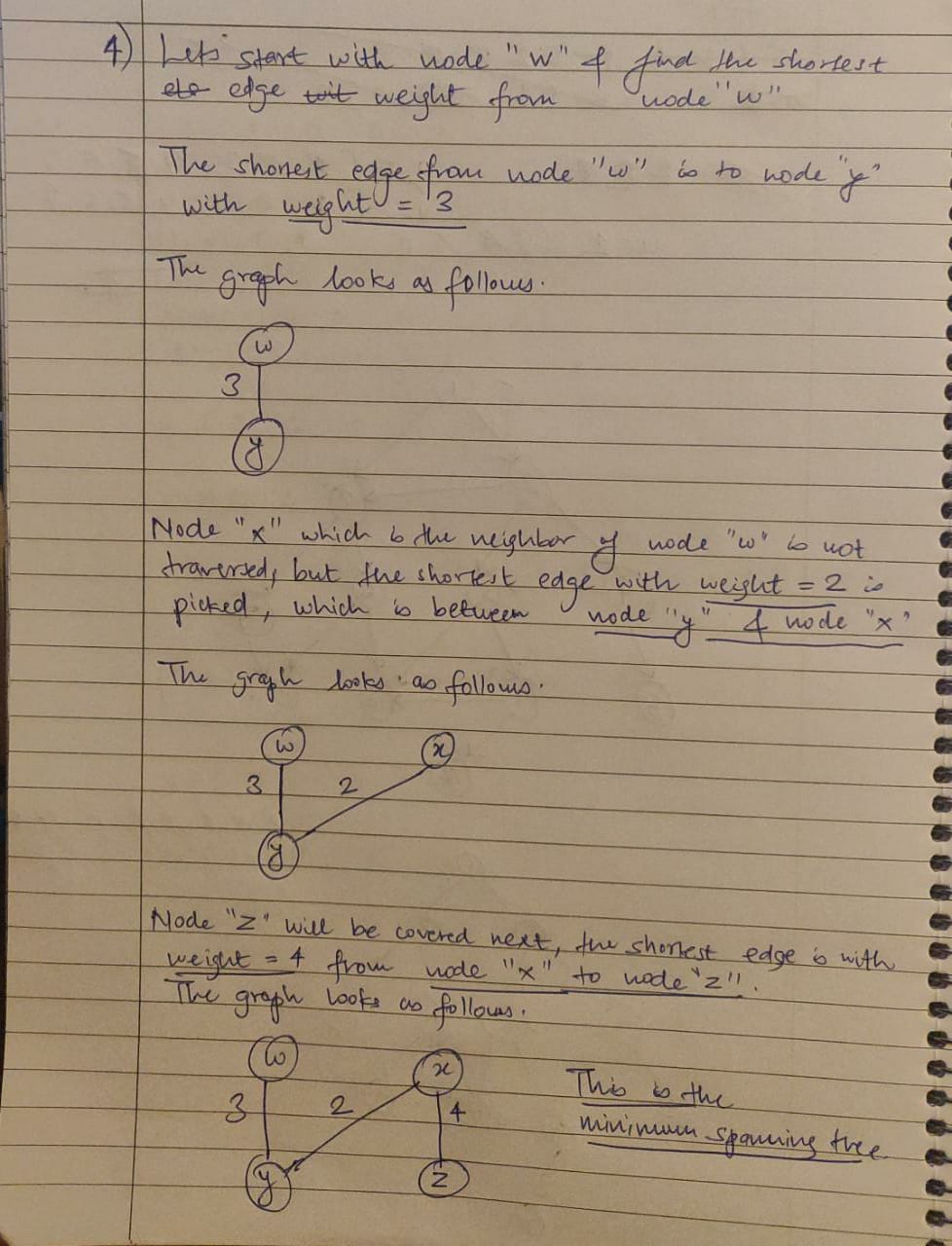
Description automatically generated

1. *[20 points]* Given the Prim’s algorithm shown below (a min-priority queue is used in the implementation):

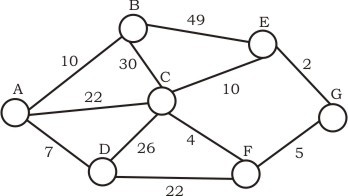


Apply the algorithm to the weighted, connected graph below (the initialization part has been done). Show a new intermediate graph after each vertex is processed in the while loop. For each intermediate graph and the final graph, you need to show the vertex being processed, the new key value for each vertex and edges in the current (partial) MST (draw a directed edge from vertex *v* to *u* if *v*. = *u*).



**

1. *[20 points]* Apply Kruskal’s algorithm to the graph below. Show new intermediate graphs with the shaded edges belong to the forest being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edges joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

  
  
  
  
  
  
  
  
A notebook with writing

Description automatically generated with low confidence

