1.

Pseudocode:

A(G,D,s,t):

S 🡨 {} /\*list of explored nodes \*/

U 🡨 {} /\*list of unexplored nodes \*/

for each node v in G:

U.append(v)

for each node u in G:

If(u == s):

shortestD = D[s,t]

if (shortestD != infinity):

while True:

w\* = argmin{D[w,t] | w in U}

S.add(w\*)

U.remove(w\*) /\*update current distances\*/

for w in U:

distance = min{D[v,t]+ D[w,t]| v is neighbor of w}

if (w == t):

break

else:

return S

return S

Proof & how it works:

We first add every node from G that is in adjacency list form to a list/array U. Since we do not know the starting point directly by looking at G itself, we need to find the starting point and then calculate the total distance moved. Therefore, by searching through every node in G, if one of the nodes equal to the starting node that we provide in the function, we begin there. If there is no such path from s to t, the question indicates that the value would be infinity. If D[s,t] is not infinity, it means that there exists a path from s to t and therefore, we can begin there. If the value is infinity, it means that such path from s to t is unavailable and we return an empty list/array of S since there cannot be path from s to t. We run an endless while loop that finds the shortest distance from the neighbor nodes of s to t. This is possible because in order to calculate the smallest distance from s to t, the distance from neighbor node of s to t should also be the smallest. We add the neighbor node of s that is closest to t to the answer list/array S until it reaches t. By following this pattern, if the neighbor node eventually reaches t, we can end the loop by adding break statement. We do not have to worry about the infinite loop of while happening. If an infinite loop occurred, it means that s did not reach t or skipped t. Since we are considering every node in G, we cannot possibly skip the node t. If s did not reach t, then shortestD = D[s,t] would have been infinity and the while loop would not have even ran. Therefore, once you go into the while loop, the break statement is guaranteed to occur since you eventually reach t from s.

Runtime:

The original code that I entered in the while loop is already proven in class to be O(n^2 \*n sigma of indegree(w) = O(n^2 + mn). However, there is another for loop above it that checks through the entire nodes, so we need to multiply it by n again, giving O(n^3 +n^2 \* m). In addition, while putting all nodes in G, we have another run time of n, giving O(n^3+n^2\*m+n). Since the runtime of last n does not matter, we take it out, giving the run time of O(n^3+n^2\*m).

When G is the upper bound on the outdegree of the points and l is an upper bound on the number of edges, the runtime becomes O(G + G(G^2 +l)), giving O(G^3 + Gl). Since G is basically the number of nodes, we can substitute that the upper bound of edges, and we run a for loop of edges that checks its edge. In the worst case, we need to check all of the nodes, G and inside it, I run a while loop that runs nodes^2 + edge, giving G^2 + l. So, the answer is O(G^3 + Gl).

\*\*Source\*\*: I got the idea of using the Dijkstra’s algorithm and based my code from the lecture slide 2\_13\_shortest\_paths.pdf that we learned on class. I learned that this question simply is the Dijkstra’s algorithm except the fact that the starting point and ending point is given by the user and the function itself. So, I added few statements to check where it begins by running the for loop, which was my idea, but based my code from the lecture slide and Dijkstra’s algorithm.

2.

Proof by counterexample

Assume that there exists an edge that has a weight that is greater than wab in a spanning tree in the undirected graph G. We can call that edge X. Spanning tree, by definition, is a group of edges that touches every node of a tree. If there exists an edge X, you can add the edge X into the maximum spanning tree and remove an edge that already exists inside the spanning tree. The removed edge that connects two nodes is replaced by the edge X that has greater weight. This means that the maximum spanning tree removed an edge and added an edge that had greater weight, meaning that the previous maximum spanning tree was not actually a maximum spanning tree since more weight could have been added to it. This is always true because the weight you subtracted by removing an edge (that is smaller than wab) is always smaller than the weight you added by adding an edge X (weight of X > weight of wab > the weight of an edge we deleted). This contradicts with the assumption in our question that a spanning tree has a maximum total edge weight since more weight could be added to that edge through the addition of edge X. Therefore, there cannot exist another edge X that has a greater weight than wab.

3.

This question simply asks to pave the minimum number of roads that safely connect the cities. If we think cities as nodes and paves as edges, the question is simply asking to construct a minimum spanning tree that goes through the roads which have puvs minimized. Since the starting point of paving the road does not matter, we can start at any random city, call it A. Then, we can add the edges that are connected to city A that are not already connected. Here, the connection fact matters the most instead of looking at the possibility of getting robbed since all cities have to be connected first to accomplish the job of paving. After paving, we go to the next nodes that are connected to city A and connect with the cities that are not connected. If we reach to the city that is already connected, we visit that city and check which of the pavement roads that are connected has the less possibility of getting robbed. For instance, if City A is connected with City B and City C and City B is connected with City C (a cycle), we do not necessarily need all the roads between those cities to be paved. Therefore, from those roads (edges in our diagram), we can only exclude one road (edge) that has the highest probability of getting robbed since the multiplication of (1-p) (1-p2) is highest when p and p2 are low. By repeating this pattern, we can visit all the cities with the least possibility of getting robbed.

\*\*Source\*\* : I got this idea of going to the edge with least weight starting from the random city from Prim’s algorithm discussed in the lecture slide 2\_20\_MST.pdf.