Question 1

Pseudocode

```

function Dijkstra(Graph, source):

create vertex set Q

for each vertex v in Graph:

dist[v] ← INFINITY

prev[v] ← UNDEFINED

add v to Q

dist[source] ← 0

while Q is not empty:

u ← vertex in Q with min dist[u]

remove u from Q

for each neighbor v of u:

alt ← dist[u] + length(u, v)

if alt < dist[v]:

dist[v] ← alt

prev[v] ← u

return dist[], prev[]

```

My code

```

function Dijkstra(Graph, source, dest):

create vertex set Q

for each vertex v in Graph:

cost[v] ← INFINITY

edges[v] ← UNDEFINED

add v to Q

cost[source] ← 0

edges[source] ← 0

while Q is not empty:

u ← vertex in Q with min cost[u]

remove u from Q

for each neighbor v of u:

alt ← cost[u] + length(u, v)

if alt < cost[v]:

cost[v] ← alt

edges[v] ← edges[u] + 1

return cost[], edges[]

```

In my implementation of the Dijkstra algorithm, I introduced two additional arrays called "cost" and "edges." The "cost" array stores tcorrespond from the source node to each individual node, while the "edges" array stores the minimum number of edges required to traverse the path from the source to each node.

During initialization, I set all elements in both arrays to infinity, except for the source node, which is initialized with a cost of 0 and an edge count of 0.

As the Dijkstra function executes, it continuously updates the values in the "cost" and "edges" arrays with the minimum cost and the minimum number of edges required to reach each node from the source.

Once the function completes its execution, the "cost" array will hold the minimum distance from the source to all other nodes, and the "edges" array will contain the minimum number of edges needed to reach each node from the source.

Question 2

Pseudocode:

U = set()

while G has edges:

    Pick arbitrary edge (v1, v2) in G

    Add v1 and v2 to U

    Remove edge (v1, v2) from G

Complexity: The algorithm has a time complexity of O(E), where E is the number of edges in the graph G. This is because the algorithm iterates through all the edges in the graph and adds the adjacent vertices to the node cover set U.

Step-by-step explanation:

The pseudocode presented above outlines a greedy approach to finding a minimum node cover for a graph G = (V, E). The algorithm initiates with an empty node cover set U and proceeds to iterate through all the edges in the graph. For each edge (v1, v2) in the graph, it adds both adjacent vertices to the node cover set U and removes the edge from the graph. This process continues until all edges in the graph have been processed.

The algorithm's time complexity is O(E) because it iterates through all the edges in the graph and adds the adjacent vertices to the node cover set U. Due to the typical scenario where the number of edges in a graph is much smaller than the number of vertices, this algorithm is relatively efficient in finding the minimum node cover.

Guaranteed Minimum Node Cover:

The algorithm is guaranteed to yield the minimum node cover of the graph since it iterates through all the edges, adding the adjacent vertices to the node cover set U until all edges have been processed. By consistently adding the least number of vertices to the node cover set U, the algorithm guarantees the minimum number of vertices in the node cover set.

Conclusion:

Overall, this algorithm provides an efficient and reliable method for finding the minimum node cover of a graph. It has a favorable time complexity and delivers the desired outcome due to its greedy approach.