1.Solution:

If Graph G = (V, E) is given by an adjacency matrix, for a vertex u, to find its adjacent vertices, we can search the row of u in the adjacency matrix instead of searching the adjacency list. We assume that the adjacency matrix stores the edge weights, and those unconnected edges have weights 0.

Pseudocode:

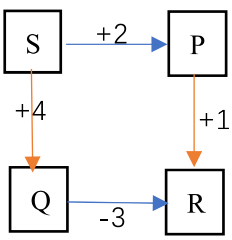
Prim’s algorithm():

```
for each u ∈ V[G] do
    key[u] = ∞;
    π[u] = NIL;
end
key[r] = 0;
Q=V[G];
while Q ≠ ∅ do
    u=EXTRACT-MIN(Q);
    for each v ∈ V[G] do
        if A[u,v]6= 0 and v ∈ Q and A[u,v] < key[v] then
            π[v] = u;
            key[v] = A[u, v];
        end
    end
end
```

Time Complexity:

The outer loop (while) has |V| variables and the inner loop (for) has |V| variables. Hence the algorithm runs in $O(|V|^2)$.

2. Solution:



In the graph shown, Dijkstra's algorithm incorrectly produces the path S->P->R, with a cost of +3, whereas the path S->Q->R has a cost of only +1.

3. Solution:

Pseudocode:

FindMaxMin():

Input:G:the stored graph

s: a fixed source

prev:array to store the prev of each node

Output: Returns the maximum and minimum capacity starting from node s

1:capacity=-∞

2:prev = undefined # an array which will store(for each node u) the node that gets us to reach node u with the maximum path capacity

3: capacity [s]← ∞

4:Q←all nodes in graph # Q: to find the node with the maximum capacity so far quickly

5:**while** Q.empty() **do**

6: u←getNodeWithMaxCapacity()

get the node with the maximum path capacity from Q using the function getNodeWithMaxCapacity.

7: **if** capacity [u]= -∞ **then** # If the extracted node has a capacity equal to -∞, then all the remaining nodes inside Q can't be reached from node s

8: **break**

9: **for** v∈neighbors(u) **do** # for each neighbor calculate the new capacity

10: w←min(capacity[u], capacityBetween(u,v))

new capacity :the minimum between the capacity of node u and the capacity of the edge between u and v.

11: **if** w> capacity[v] **then** # compare the new capacity to the old one for node v

12: capacity[v]←w

13: prev[v]←u

14: Q.update(v)

15: **end if**

16: **end**

17:**return** capacity # return the calculated capacity

Prove of correctness:

We will always process the node with the largest capacity. So that when the goal node is reached, all the remaining nodes must have a smaller capacity. Thus the found path would be the one with the maximum-minimum capacity.

Time Complexity:

The algorithm's time complexity is O(E log(V)), similar to Dijkstra's algorithm, where E is the number of edges and V is the number of vertices.

4. Solution:

Pseudocode:

Input:

C: the maximum cost of any shortest path that leaves s (the source). It's at most K|V|.

L[i]:Each L[i] is a doubly-linked list which keeps track of all the vertices that are currently at distance i from the source.

d[v]:the shortest distance (so far) from the source to vertex v

s:source

Upgrated Dijkstra's Algorithm ():

1:L[i] <- {}, 0 <= i <= C

2:L[inf] <- {}

3:L[0] <- {s}

```
for v in V do
    push v into L[inf]
    d[v] <- inf
end
d[s] <- 0
p <- 0
while p < C do
    while L[p] != {} do
        pop v from L[p]
        for (v, u) with d[u] > d[v] + w(v, u) do
            remove u from L[d[u]]
            d[u] <- d[v] + w(v, u)
            push u in L[d[u]]
        end
    end
    p <- p + 1
end
```

Time Complexity:

Each edge and vertex of the graph is visited only once, and the outer loop runs in time O(C), therefore the complexity of the algorithm is $O(|E|+|V|+C)$. Since $C \leq K|V|$, this reduces to $O(|E|+K|V|)$.