1. [5] Sometimes it is desirable not only to find the shortest path in a graph (assuming only positive weights here), but also among the shortest paths, the one that has the minimum number of edges.

   Modify Dijkstra's algorithm so that it makes sure that for each vertex t, the shortest path we get from s (source) to t, is the one with the minimum cost but also, among all the minimum cost paths, the one with the minimum number of edges.

   Take the pseudo-code from DPV and add your additional code to it. Also explain your logic in a few lines of English.

   Answer From §4.4.1 in *Algorithms* by Dasgupta-Papadimitriou-Vazirani (DPV)

   Algorithm 1 Dijkstra (from text)

   Dijkstra's shortest-path algorithm
   Input: Graph $G = (V, E)$, directed or undirected
   positive edge lengths $\{l_e : e \in E\}$ vertex $s \in V$
   Output: For all vertices u reachable from s,
   dist(u) is set to the distance from s to u.
   Time Complexity: $O((|V| + |E|) \log |V|)$
   Space Complexity: $O(V)$

```
procedure DIJKSTRA($G, l, s$)
    for all $u \in V$ do
        dist($u$) $\leftarrow \infty$
        prev($u$) $\leftarrow$ nil
    end for
    dist($s$) $\leftarrow 0$
    $H \leftarrow$ makequeue($V$)                          ▷ using dist-values as keys
    while $H$ is not empty do
        $u \leftarrow$ deletemin($H$)
        for all edges $(u, v) \in E$ do
            if dist($v$) > dist($u$) + $l(u, v)$ then
                dist($v$) $\leftarrow$dist($u$) + $l(u, v)$
                prev($v$) $\leftarrow u$
                decreasekey($H, v$)
            end if
        end for
    end while
end procedure
```

   Modified solution Below is the modified pseudo-code which will output not only the minimum distance based on edge weights but also the minimum number of edges (with the lowest weight) that need be traversed to get from the source node s to the desired destination node t

Algorithm 2 Modified Dijkstra

Dijkstra's shortest-path algorithm modified to
return the path that has both the shortest distance and least number of edges.
Input: Graph $G = (V, E)$, directed or undirected positive edge
lengths $\{l_e : e \in E\}$ vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set
to the distance from s to u and edgeDist(u) is set to the minimum
number of edges between s and u.
Time Complexity: $O((|V| + |E|) \log |V|)$
Space Complexity: $O(V)$

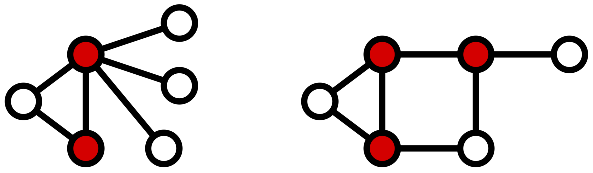
```
 1: procedure DIJKSTRA($G, l, s$)
 2:     for all $u \in V$ do
 3:         dist($u$) $\leftarrow \infty$
 4:         prev($u$) $\leftarrow$ nil
 5:         edgeDist($u$) $\leftarrow 0$
 6:         visited($u$) $\leftarrow$ false
 7:     end for
 8:     $H \leftarrow$ makequeue($V$)                    ▷ using dist-values as keys
 9:     dist($s$) $\leftarrow 0$
10:     $H$.enqueue($s$)
11:     while $H$ is not empty do
12:         $u \leftarrow$ deletemin($H$)            ▷ dequeues the minimum distance vertex
13:         visited($u$) $\leftarrow$ true
14:         $curDist \leftarrow$ dist($u$) $+ l(u, v)$
15:         $curEdgeDist \leftarrow$ edgeDist($u$) $+ 1$
16:         for all edges $(u, v) \in E$ do
17:             if dist($v$) $= curDist$ and edgeDist($v$) $> curEdgeDist$ then
18:                 edgeDist($v$) $\leftarrow curEdgeDist$
19:                 prev($v$) $\leftarrow u$
20:             end if
21:             if dist($v$) $> curDist$ then
22:                 dist($v$) $\leftarrow curDist$
23:                 prev($v$) $\leftarrow u$
24:                 decreasekey($H, v$)
25:                 if visited($v$) = false then
26:                     $H$.enqueue($v$)
27:                 end if
28:             end if
29:         end for
30:     end while
31: end procedure
```

Plain English explanation To modify the original algorithm presented in §4.4.1 of the DPV, we add edgeDist(v) which will hold the number of edges in the path from the source vertex to the node v. Like dist(v), this value will be stored for each node v in the graph. At each step in the **for** loop we check to see if the current minimum distance (based on edge weights) is equal to the node we are checking. If it is then we modify the edge distance for that node to reflect the current minimum edges traversed plus one. This has the effect of storing the minimum number of edges which need be traversed to get to each node when starting from the source vertex (node). Furthermore, if a node has two or more traversals which have the same edge weights then by storing the number of edges needed to traverse to that node we are able to differentiate paths which appear to equidistant based on weights alone.

Like the original algorithm from the text, we will store the minimum distance for each node in the graph structure as well as the minimum number of edges which need be traversed to reach said node. Then, upon the completion of the algorithm and given a graph, G, the minimum weighted distance from source node s to destination node t can be obtained with G.dist(t) and the minimum number of edges can be obtained with G.edgeDist(t)

Complexity analysis We have not added any extra loops to the provided function and the atomic operations we have added have no impact on the worst case complexity. Therefore the worst case complexity remains $O((|V| + |E|) \log |V|)$, like the original solution. Furthermore, as we have only added single array with length $|V|$ the space complexity has only risen by the additional $c|V|$ for some constant $c \geq 2$. Therefore the total space complexity is $O(|V|)$

2. [5] Let $G = (V, E)$ be an undirected graph. A node cover of G is a subset U of the vertex set V such that every edge in E is incident to at least one vertex in U. A minimum node cover is one with the fewest number of vertices.



Shaded vertices shows minimum node cover. Find a greedy approach to find the minimum node cover. Write the pseudocode and explain its complexity. (Think in terms of degrees of vertices, as discussed)

Pseudo-code solution From §35.1 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms* (CLRS)

Algorithm 3 Approx-Vertex-Cover

An approximate algorithm to calculate the minimum vertex cover.

Require: Graph $G = (V, E)$, where V, E are sets.

Ensure: Minimum vertex cover C, where C is a set.

```
$C \leftarrow \emptyset$
$E' \leftarrow G.E$
while $E' \neq \emptyset$ do
    $(u, v) \leftarrow E'[0]$
    $C \leftarrow C \cup \{u, v\}$
    for $e \in E'$ do
        if $u \in e$ or $v \in e$ then
            $E'$.remove($e$)
        end if
    end for
end while
return $C$
```

Modified solution We modify the above pseudo-code to use the vertex degrees in a greedy approach to find the minimum node cover using vertex degrees

Algorithm 4 Greedy-Vertex-Cover

A Greedy algorithm to calculate the minimum node cover.
Require: Graph $G = (V, E)$, where V, E are sets.
Ensure: Minimum vertex cover C, where C is a set.

```
1: $C \leftarrow \emptyset$
2: $G' \leftarrow G.copy()$                                  ▷ copy graph to preserve original
3: while $G'.E \neq \emptyset$ do
4:     $v \leftarrow G.V'[0]$
5:     for each $u \in G'.V$ do                              ▷ Find vertex with max degree
6:         if $u$.degree() $>$ $v$.degree() then
7:             $v \leftarrow u$
8:         end if
9:     end for
10:    $C \leftarrow C \cup \{v\}$                            ▷ add max degree vertex to cover-set
11:    for each $e \in G'.E$ do
12:        if $v \in e$ then
13:            $G'.E$.remove($e$)
14:        end if
15:    end for
16:    $G'$.calculate-degrees()                               ▷ recalculate degrees
17:    for each $v \in G'.V$ do                              ▷ remove nodes with degree of 0
18:        if $v$.degree() $= 0$ then
19:            $G'.V$.remove($v$)
20:        end if
21:    end for
22: end while
23: return $C$
```

Plain English explanation and analysis The above algorithm first copies the graph we want to generate a minimum cover-set for, G, and stores it as G'. We do this so we can remove nodes from the graph during our algorithm while still preserving the original structure should we need it later. As maintaining the original graph is not a program requirement we do not consider the additional complexity this copy adds to our algorithm. We begin each iteration of our algorithm by finding the vertex with the highest degree, v, in our graph which is accomplished in $O(|V|)$ time. Once found, we add this vertex to our cover-set C. We then search through every edge in G' and remove any edges incident to our vertex v. This is accomplished in $O(|E|)$ time. Now we must recalculate all of the degrees in graph G' so the next iteration correctly finds the node with the highest degree. Finally, we remove any vertex with a degree of 0 from our graph as they will have already been covered by the nodes in our cover-set. This will be accomplished in $O(|V|)$ time as we will have to search every node in the graph. This continues until there are no edges left in G'. The resulting set C will contain the nodes which, at each iteration of our algorithm, have the highest degree and thus provide the most edge coverage with the fewest nodes. While this algorithm will find a correct answer this answer may not always be guaranteed to be the most optimal. As a greedy algorithm we try and make choices as quickly as possible while still being correct. In situations where multiple vertices have the same degree we arbitrarily choose one without considering the other choices which can lead to a less than optimal final solution.

Total complexity of solution is $O((|V| + |E|)^2)$ since for every edge we loop through every vertex and every other edge while we consider which edges should be removed from the resulting graph. While the number of edges we must search through seems to decrease quickly, there can be edge cases where at each iteration the total number of edges only decreases by one.