Fall 2021: CSOR 4231 - Analysis of Algorithms

Aayush Kumar Verma, UNI: av2955

Homework 5 (Due Date: November 30, 2021)

Problem 1:

Given that an undirected graph **G = (N, E)** is called *bipartite* if it's set N of nodes can be partitioned into two subsets A, B such that $A \cap B = \Phi$ and $A \cup B = N$ so that every edge connects a node $a \in A$ to a node $b \in B$.

(a).

Lets assume that G is bipartite and has a cycle of odd length and let's denote the set of vertices that form this cycle as:

$$C = (v_1, v_2, v_3, \ldots, v_n, v_1)$$

Without loss of generality, with the assumption that G is bipartite, we can easily state that:

$$v_1 \in A, v_2 \in B, v_3 \in A, \ldots \text{ and so on.}$$

Therefore for every vertex i in C, we have:

$$v_i \in \begin{cases} A\text{: } i \text{ is odd} \\ B\text{: } i \text{ is even} \end{cases}$$

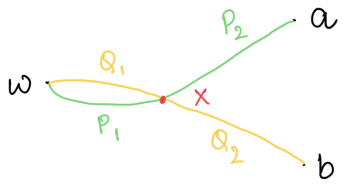
Since the cycle is of odd length, we know that n is odd, therefore $v_n \in A$ but the cycle is defined by an edge from v_n to v_1 and $v_1 \in A$. Since we know that this edge belongs to C as well as E, we can now surely say that our previous assumption that G is bipartite is incorrect by contradiction.

In addition to this, by definition of a bipartite graph, we say that a graph G is bipartite if and only if all it's connected components are bipartite. Therefore, we now prove the necessary condition for a graph to be bipartite having no odd cycles.

$X = v \in N$: d(v, w) is even
$Y = v \in N$: d(v, w) is odd

... where d is the shortest distance between v and w and we can clearly state that $X \cap Y = \Phi$ and $X \cup Y = N$.

Lets assume that G is an undirected connected graph with **no odd cycles** and lets suppose for the sake of contradiction that two vertices $(a, b) \in A$ or $(a, b) \in B$, i.e. they are adjacent ($ab \in E$). Let us select a vertex $w \in N$ which is connected to both a and b with a intersection point (another common node to a and b), say, x. Refer to the image below:



Without loss of generality, if $a = w$, then $d(a, w) = 0$. Since 0 is an even number, what follows is that $d(b, w)$ is even and therefore, $d(a, b)$ is even. However, since a and b are adjacent, we already know that $d(a, b) = 1$ which is odd, hence a contradiction of the above assumption. Therefore, we establish that:

$$a \neq w$$

$$b \neq w$$

$$a \neq b \neq w$$

Now, using the above relations, we will finally prove that if a graph is bipartite, it cannot have an odd length cycle.

Let P be the shortest path from w to a and Q be the shortest path from w to b. Since P and Q are shortest paths, it is obvious that both $|P|$ and $|Q|$ have the same parity, i.e. either both are odd or both are even. Say that the parity is not the same - let's break the shortest paths P and Q into 2 parts such that P_1 and Q_1 are the shortest paths from w to x and P_2 and Q_2 are the shortest paths from x to a and from x to b respectively. If the parity is not same, then it would no longer be the shortest path as we could take the shortest path, say, Q_1 and then P_2 to reach a which would then become the new shortest path. Hence, we can conclude that the parity will be the same if they are both the shortest paths from w to a and b respectively when going through x.

Now,

$$|P| = |P_1| + |P_2|$$

$$|Q| = |Q_1| + |Q_2|$$

Since P_1 and Q_1 are the lengths of the shortest path from w to x, they are equal in length and since the parity of $|P|$ and $|Q|$ are same, then the parity of $|P_2|$ and $|Q_2|$ are also same.

We had earlier assumed that a and b are adjacent, hence there is an edge from a to b. Now, we consider the cycle formed by a, b, and x, the length of the path between them being equal to $(|P_2| + |Q_2| + 1)$ (adding 1 for the edge between a and b). Now that the parity of $|P_2|$ and $|Q_2|$ are same, then it is obvious that the sum $|P_2| + |Q_2|$ would always be even. Hence, the length of the path of the cycle would always be odd since we add 1 for the edge between a and b, i.e., it will of the form $2k + 1$ where k is any non-negative integer. However, right at the start we had assumed that G does not have any odd length cycles. Hence, we have forced an odd length cycle when we assume 2 vertices are adjacent either in A or B. Therefore, by contradiction, we can firmly state that if there are no adjacent vertices in A and there are no adjacent vertices in B, then there must be no odd length cycle in G and hence, the partitions A and B would make G a bipartite graph.

(b).

We will follow a Breadth First Search like algorithm to determine if a graph is bipartite or not (Algorithm 1). We will approach this problem as an m way Graph Coloring Problem. Here, $m = 2$ so we maintain 2 sets U and V to keep a record of which color has been assigned to which node. For the sake of simplicity, lets divide the set of nodes into 2 colors - RED and BLUE which will be stored in sets RED and $BLUE$ respectively. Whenever a forward edge, backward edge, or a cross edge appears, the algorithm checks whether 2-coloring still holds. If, at any point, we find that 2 adjacent nodes have the same color, then it means that the graph is not bipartite and we move on to look for an odd-length cycle in the graph.

For simplicity, let's assume that the nodes are numbered from 1 to N so that it becomes easier to access a node. As it does not matter which color is assigned to the first node of a connected component, we will always assign RED to the first node of a connected component. Our algorithm will work for all connected components of a graph. For this, we will maintain an array named *color* of size equal to the number of nodes ($|V|$) which will store 3 different values depending on which color a particular node is:

$$\text{color[i]} = \begin{cases} 1 : color = RED \\ 0 : color = BLUE \\ -1 : \text{not assigned a color yet} \end{cases}$$

For finding a cycle with an odd length in a graph, we will use Graph Coloring Method to mark all nodes in the same cycle with the same color (Algorithm 2). Then we will push the nodes with the same color into an Adjacency list like structure such that all nodes of a cycle fall into 1 list. Then, if there is a list which is odd in length, we will print out that list. There may be multiple cycles with odd length but for the purpose of showing that a graph is not bipartite, any one of them will suffice. We will run a DFS on the graph to determine the cycles and store them in a list of lists named *cycles*. Then we will simply print a cycle with odd length from *cycles*. We mark the nodes as per the following color notation:

$$\text{color[i]} = \begin{cases} 0 : \text{unvisited node} \\ 1 : \text{discovered node} \\ 2 : \text{visited node} \end{cases}$$

If we run into an already visited node, then we simply return. When we find a previously discovered node, it means that we have detected a cycle so we increment the count for the number of cycles (*cycleIndex*), mark this node as a part of the new cycle and then iteratively mark all it's parents, grandparents, and so on with the same marker to signify a cycle. If we find a new node, then we update it's parent with the last visited node and change it's color to 1. Then we recursively call the function with all it's neighbor nodes. After the recursion is complete, we mark this node as visited, i.e. change it's color to 2.

Time Complexity Analysis and Proof of Correctness:

Bipartite Graphs:

Every node is explored exactly once in $O(|N|)$ time and when expanding a node's descendants, we are exploring every edge of G exactly once, hence taking another $O(|E|)$ time, hence, the overall run-time complexity of the algorithm is $O(|N| + |E|) = O(n + e)$ where n is the number of nodes in N and e is the number of edges in E.

The proof of correctness follows from the fact that the BFS algorithm is separating the nodes from one level completely from another level, therefore there are no nodes in the same set having an edge that goes from one level to the immediate next level. Had there been such nodes, then the BFS algorithm, which follows a level order traversal, would have explored the other node along with the first node. Hence, we never get a pair of nodes which lie in adjacent levels in the same set. Hence, the bi-partitioning algorithm works correctly.

Odd Length Cycle:

The cycles are explored in a depth first manner, thus every node and edge is explored twice in the recursion, hence the run-time complexity will be $O(n + e) + O(n + e) = O(n + e)$. Finally, returning the cycle with an odd length takes $O(k)$ time where k is the number of cycles. However, k can be at max equal to n. Similarly printing the odd length cycle will take $O(m)$ time, where m is the number of nodes in the current cycle and m can be at max n. Hence, the overall time complexity is $O(n + e) + O(n) + O(n) = O(n + e)$

The algorithm works correctly because in DFS, we are updating the markers and parents while looping through the nodes and a node gets explored completely only after all it's descendants have been explored. We increment the cycle number only when we find that there is an edge to a previously "discovered" nodes therefore ensuring that we always get the correct nodes in the current cycle and also the correct number of cycles. Finding out the cycle with an odd length from all the cycles is a simple linear time search and therefore, the complete algorithm of finding the cycle with an odd length is correct.

Algorithm 1 Bipartite Graphs

```
isBipartite(N: number of nodes, adj{}: Adjacency List):
    RED = set()
    BLUE = set()
    Initialize color[] with -1 up to length = N
    Create queue
    for i = 1 to N do
        if color [i] == -1 then
            color [i] = 1
            RED.add (1)
            queue.add (1)
            size = length (queue)
            for j = 1 to size do
                node = queue.remove()
                for neighbor in adj[node] do
                    if color [neighbor] == color [node] then
                        return FALSE, NULL, NULL
                    end if
                    if color [neighbor] == -1 then
                        color [neighbor] = 1 - color [node]
                        queue.add (neighbor)
                        if color [neighbor] == 1 then
                            RED.add (neighbor)
                        else
                            BLUE.add (neighbor)
                        end if
                    end if
                end for
            end for
        end if
    end for
    return TRUE, RED, BLUE


Main (N: number of nodes, adj{}: Adjacency List):
    bipartite, red, blue = isBipartite (N, adj)
    if bipartite == TRUE then
        print (red)
        print (blue)
    else
        Initialize an array color[] with 0 of size N
        Initialize an array marker[] with 0 of size N
        Initialize an array parents[] with 0 of size N
        Function Call: allCycles (N, adj, 1, -1, color, marker, parent)
        odd_cycle = oddCycle (E, marker)
        print (odd_cycle)
    end if
```

Algorithm 2 Cycle of Odd Length in a Graph

```
global cycleIndex = 0

allCycles (N: number of nodes, adj{}: Adjacency List, current, previous, color, marker, parent):
  if color[current] == 2 then
      return
  end if
  if color[current] == 1 then
      cycleIndex = cycleIndex + 1
      node = previous
      marker[node] = cycleIndex
      while node ! = current do
          node = parents[node]
          marker[node] = cycleIndex
      end while
      return
  end if
  parent[current] = previous
  color[current] = 1
  for neighbor in adj[current] do
      if neighbor ! = previous then
          allCycles (N, adj, neighbor, current, color, marker, parent)
      end if
  end for
  color[current] = 2
  return


oddCycle (E: edges, marker):
  Initialize a list of lists cycles [] as an empty list
  for i = 1 to length (E) do
      if marker[i] != 0 then
          cycles[marker[i]].append(i)
      end if
  end for
  for i = 1 to cycleIndex do
      if length(cycles[i]) % 2 == 1 then
          return cycles[i]
      end if
  end for
  return NULL
```

(c).

Given a set of non-equality constraints of the form $x_i \neq x_j$ over a set of Boolean variables $x_1, x_2, x_3, \ldots, x_n$. We notice the following analogy:

1. Boolean variables $x_1, x_2, x_3, \ldots, x_n \equiv$ Vertices of a Graph
2. Inequalities $x_i \neq x_j \equiv$ Edges of the Graph

Now, we see that the problem has been transformed into the Bipartition Problem of Graphs, i.e., we need to partition the set of vertices (V) into 2 sets such that there is no edge between two vertices belonging to the same set therefore satisfying the condition $x_i \neq x_j$.

The algorithm we will follow is:

1. Create a Graph $G = (V, E)$ using the above mentioned analogy. Here, V is the set of vertices (the Boolean variables) and E is the set of edges (the inequalities).
2. Run Algorithm 1 (Bipartite Graphs), mentioned above, on G. The returned sets (RED & BLUE) will contain the partitioned set of vertices.
3. Assign 0 to all the vertices present in the set RED.
4. Assign 1 to all the vertices present in the set BLUE.
5. Print the variables (vertices) and the Boolean values assigned to them.

Now, as mentioned above in part (1.b), the time complexity of the bi-partitioning algorithm is $O(|V| + |E|)$. Here, we have taken the set of vertices (V) as the set of Boolean variables, the size of which is n and the set of edges (E) to be the inequality constraints, the size of which is m. Therefore, the overall run-time complexity of this problem is given by $O(n + m)$. As the algorithm used here is same as the one written in part (1.b), the proof of correctness is the same as the one provided in part (1.b).

Problem 2:

(a). Let's assume that the root (R) of a graph (G_π) is an articulation point with just 1 child, say a. So, by the definition of an articulation point, removing R must disconnect the graph and thus, there must be a path from a to every other child node of R. Also, after removing R, the graph becomes disconnected, therefore, by definition of a disconnected graph, there must be another nodes b and c such that the only path between b and c would have had to go through R. A path from R to a must go through one of R's children first and this child is connected to a through a path that does not contain R. Similarly, a path from R to c must go through some child of R which is in turn connected to a through a path which does not contain R. Therefore, this implies to the fact that there actually exists a path between b and c which does not contain R thereby disproving our above assertion that the only path between b and c must go through R, hence a contradiction. Now, if R does contain at least two children, say x and y, then there must not be any path between x and y that does not contain R otherwise x would have been an ancestor of y or vice-versa. Thus, if we remove R, then it will surely disconnect the components containing x and y respectively. Hence, R becomes an articulation point.

(b). Let's take a non-root vertex (v) of G_π and we need to show that neither the direct child (s) of v nor any descendant of s has any back edge to any proper ancestor of v. Now, we remove v from the graph. Suppose that every child of v has a descendant that contains a back edge to a proper ancestor of v. Every subtree of v is a connected component and so is the set S of vertices that are not descendants of v. Hence, with a back edge from the descendants of v to vertices in S, the graph actually remains connected even after the removal of v. Therefore, v cannot be an articulation point.

Let there be an ancestor (r) of v. Since we are given an un-directed graph, the only edges possible are tree edges and back edges and every descendant of s does not have a back edge to a proper ancestor of v. Then, in this case, we cannot move up the tree without back edges thus making r unreachable from any descendant of s, i.e., the graph is now disconnected. Hence proved that now, the vertex v becomes an articulation point.

(c). As we are following a depth-first search, v is discovered before all it's descendants and the only back edges that affect $v.low$ are the ones which go from a descendant of v to an ancestor of v. If we know the values of $u.low$ for every child u of v, then the result for $v.low$ can be easily computed from the results of all $u.low$. Hence, the recursive algorithm works as follows:

$$v.low = \begin{cases} \textbf{min}\ (v.d, w.d) : v \text{ is a leaf node.} \\ \textbf{min}\ (v.d, w.d, u.low) : v \text{ is not a leaf node.} \end{cases}$$

Here, (v, w) is a back edge and u is a child of v. The computation of $v.low$ takes linear time in the degrees of the vertices the sum of which is equal to twice the number of edges, therefore, the overall run-time of this algorithm is given by $O(E)$. The proof of correctness follows from the fact that a back edge can come from either v or a descendant of v, and for every descendant of v, we are recursively calculating the value of $v.low$ which comes from the optimal values of it's descendants.

(d). For finding all the articulation points, we need to first run the recursive algorithm defined in part (2.c) to calculate $v.low$ for all the nodes in $O(E)$ time. Then we need to check for all the points that if there is at least one descendant u of v such that $u.low \geqslant v.d$, i.e., a back edge from u can go to at max v. If this condition is true, then it means that this descendant of v does not have a back edge to an ancestor of v and therefore, removing vertex v would make the graph disconnected. Hence, such a v becomes an articulation point. The comparison takes place in constant time and as we are doing this for every node which takes

another $O(V)$ time. Now, as the graph is connected, E is equal to at least $V - 1$, hence, it is deducible that the overall run-time complexity of the algorithm remains $O(E)$. The proof of correctness follows from part (2.a) if the articulation point is the root node and from part (2.b) is not a root node. Therefore, the algorithm is correct.

Problem 3:

(a). As we are given a set of consistent set of inequalities, then there must be a path from a vertex a to b such that the following inequality holds:

$$b \geqslant a$$

If the inequalities are strict, then:

$$b > a$$

The path from a to b can be written as follows:

$$a \longrightarrow x_k \longrightarrow x_l \longrightarrow \ldots \longrightarrow b$$

And, we know that there is an edge from x_i to x_j only when $x_j \geqslant x_i$, hence, we have:

$$a \leqslant x_k \leqslant x_l \leqslant \ldots \leqslant b$$

Hence, we can clearly see from the above inequalities that $b \geqslant a$.

If there are strict inequalities in between, then we have:

$$a \leqslant x_k \leqslant x_l \leqslant \ldots \leqslant x_{m-1} < x_m \leqslant \ldots \leqslant b$$

Here also, we see that all values from a to x_{m-1} are $\leqslant$ all the values from x_m to b, thereby showing that still, $b > a$.

Now, when a graph G has a cycle with a strict inequality, we can prove that C no longer remains a consistent set. We can also state the same thing by saying that if C is not consistent, then G must contain some cycle with a strict inequality.

Let's take a cycle between two vertices a and b such that there is a directed edge from a to b and another directed edge from b to a to complete the cycle which is a strict inequality. Therefore, from the above proven definitions, we can easily state that:

$$v > u$$

$$u \geqslant v$$

However, these 2 inequalities cannot hold together and therefore C is inconsistent. Now, proving the reverse that if C is inconsistent, then there are two variables a and b such that the set of inequalities in C make them greater than each other. We can also have a case when one of them is strictly greater than the other and the other one is greater than or equal to the previous one. In both cases, C is inconsistent. Let the inequalities we get are like:

$$b > a$$

$$a \geqslant b$$

Therefore, we will get a sequence in the graph like:

$$a \leqslant x_k \leqslant x_l \leqslant \ldots \leqslant x_{m-1} < x_m \leqslant \ldots < b$$

Or,

$$b \leqslant x_k \leqslant x_l \leqslant \ldots \leqslant a$$

Thus, these two sequences are for a graph G with strict inequalities and taking both paths together, we see that there is a cycle in G with some strict inequality. Hence, proved.

(b). Let's denote the strongly connected components as SCC. We create a Directed Acyclic Graph (DAG) of the SCCs using the algorithm discussed in the class lectures. This will take $O(n+m)$ time (again, from the class lectures) and we mark each SCC with some unique identifier. Every SCC will have edges that are part of some cycle but the edges connecting one SCC to another SCC are not part of any cycle. Let's denote the set of edges that are a part of any cycle by E_1. Now, the problem simply translates to the proof and lemma provided in part (3.a) that if there is any strict inequality and a cycle is formed with a strict inequality in G, then C is inconsistent. Hence, we loop through the edges in E_1 and check if there is any strict inequality or not among them. If no strict inequality is found, then C is consistent.

Time Complexity Analysis and Proof of Correctness:

As stated above, the time required to create a DAG of SCCs is $O(n+m)$ and we would require $O(m)$ time to iterate over all the edges in E_1. Hence, the total run-time complexity is $O(n+m)+O(m) = O(n+m)$.

The nodes that are a part of some SCC will be involved in a cycle whereas the rest of the nodes would not be involved in any cycle. So, if we find a strict inequality with any such node which is involved in a cycle, then from the proof in part (3.a), we can say C is inconsistent.

(c). To compute the minimum solution of a consistent set C, we can run the following algorithm:

1. Create a DAG (G) of Strongly Connected Components (SCCs) and mark every SCC with a unique identifier.
2. Do a Topological Sorting in order of shortest job first (i.e. sorting them in ascending order of finish time) of the SCCs using the DFS technique taught in the class.
3. If two SCCs are interchangeable in the topological ordering, then we keep the SCC with more number of nodes before the one with lesser number of nodes so that when we assign numbers, the lesser value is assigned to the SCC with more number of nodes thereby ensuring that we get the minimum sum at the end.
4. Assign values starting from $1, 2, \ldots, k$ to the SCCs one by one, i.e. the nodes in the first SCC get the value 1, the nodes in the next SCC get the value 2 and so on until the k^{th} SCC.
5. Now, with the assigned node values, we get the minimum solution.

Time Complexity Analysis and Proof of Correctness:

Creating a DAG of SCCs takes $O(n+m)$ time. In the worst case, we can have every node of G in a different SCC, therefore giving us a worst case run-time complexity of the Topological Sorting algorithm, using DFS, equal to $O(n+m)$ and finally, assigning the integer values takes $O(n)$ time. Hence, the total run-time complexity of the algorithm is $O(n+m)+O(n+m)+O(n) = O(n+m)$.

We know that C is consistent, therefore there are no strict inequalities in any of the SCCs so we can easily assign the same value to all the nodes in the same SCC. Hence, we choose the minimum possible integer value and assign it to all the nodes in the same SCC, thereby maintaining the minimum total sum. For an edge from one SCC to another SCC, it must be a strict inequality, hence, we must assign a higher integer to the nodes in the next SCC. Therefore, we get this assignment of SCCs using the Topological Ordering and we keep assigning the minimum possible integer in an incremental way to all the nodes in the SCCs, thus ensuring that our final sum is minimum. Hence, the algorithm is correct.

Problem 4:

(a). Increase the weight of an edge, say an edge from vertex x to y. The original weight is given by $w(x, y)$ and the new weight is given by $w'(x, y)$.

The algorithm works as follows:

1. Remove edge (x, y). Now the tree T is partitioned into two different trees, say, T_1 and T_2.
2. Run any Graph Search Algorithm such as BFS or DFS on $T - (x, y)$ to mark all the vertices that are present in T_1 and T_2 respectively. This step takes $O(E_1) + O(E_2)$ time where, E_1 and E_2 are the number of edges in T_1 and T_2 respectively which is basically the same run-time complexity as $O(|N|)$ in total because the number of vertices is 1 more than the number of edges.
3. Now, add the minimum weight edges among all the cross edges $e \in E$ such that one end of e is in T_1 and the other end is in T_2.
4. Then, find the minimum weight edge and add it to T in order to get the new MST. This can be done easily by iterating over all the edges and marking the one with the minimum weight among them which are all cross edges in $O(|E|)$ time.

Time Complexity Analysis and Proof of Correctness:

Removal of an edge takes $O(1)$ time. The search algorithm (BFS, DFS) takes $O(|N|)$ time to mark all vertices in T_1 and T_2 respectively. Finally, checking if an edge is a cross edge and choosing the minimum weight edge from these cross edges to complete the MST takes $O(|E|)$ time. Therefore, the overall time complexity of the algorithm is $O(1) + O(|N|) + O(|E|)$ time which is equal to $O(|N| + |E|) = O(n + e)$ where n and e are the total number of vertices and edges respectively.

As per the **Partition Theorem** taught in the class, for any set A of edges contained in some MST and for every partition $(R, N - R)$ of the nodes such that no edge of A crosses the partition, every minimum weight edge across partition $\in$ some MST that also contains A. Applying this property here, our partitions T_1 and T_2 correspond to R and $N - R$ and A corresponds to the edges in T_1 and T_2 and when we find the minimum weighted edge across these two partitions, we will find the edge which is in some MST containing A, thereby giving us another MST with the new minimum weighted edge that will complete the MST.

(b). In this problem, we are decreasing the weight of an edge (x, y) from $w(x, y)$ to $w'(x, y)$. So, the idea to get the new MST for this problem goes as follows:

1. Add an edge (x, y) with weight $w'(x, y)$ to T. This is the same edge whose weight had been decreased from $w(x, y)$ to $w'(x, y)$. Adding this edge will create a unique cycle in T containing (x, y).
2. Find a cycle in T and find the edge with the maximum weight in this cycle.
3. Remove the edge with the maximum weight from the cycle, leaving T to be the new MST.

Time Complexity Analysis and Proof of Correctness:

Adding an edge takes constant time and then finding a cycle in T using the algorithm taught in the class by running a DFS from vertex x will take $O(n)$ time where n is the number of vertices in T. Finally, traversing this cycle and removing the edge with the maximum weight will take another $O(n)$ time. Hence, the total time needed is $O(1) + O(n) + O(n) = O(n)$.

When we decrease the weight of an edge from $w(x, y)$ to $w'(x, y)$, then it is possible that this edge may now become a part of the MST, therefore adding it to T is correct. Now, as we have added an edge that was involved in a cycle, we traverse the cycle and remove the edge with the maximum weight. This ensures that the tree remains connected and the weight is reduced by the maximum possible value giving us a new MST, therefore the algorithm works correctly.

Problem 5:

The Dijkstra's Algorithm is a Greedy Algorithm that finds the minimum weighted vertex in every iteration and thus, will output the first shortest path with minimum weight it finds. However, this path may not be the one with the minimum number of edges. Hence, we modify the original Dijkstra's Algorithm by keeping a track of the number of edges in the minimum weight path. We will do this with the help of an array that stores the length of the path from the source vertex for every vertex in the graph and if any point of time, we find that we have a shorter path (lesser number of edges) with the same weight, then we will update the minimum number of edges. Finally, we will return the parent array to get the minimum weight path having the minimum number of edges. When we traverse backwards from the target node t parent by parent using the parent array, we will get the minimum weight path with the minimum number of edges when we reach the source vertex s.

Algorithm 3 Modified Dijkstra's Algorithm

```
Dijkstra(graph[][], s, t, weight, adj):
    Create an array of size |V| to store distance of v from s: d[]
    Create an array of size |V| to store parents of v: p[]
    Create an array of size |V| to store minimum path length of v from s: l[]
    d[s] = 0
    p[s] = NULL
    l[s] = 0
    for each v ∈ V − {s} do
        d[v] = ∞
        p[v] = NULL
        l[v] = ∞
    end for
    Initialize a Priority-Queue Q = {s}
    while Q ≠ φ do
        u = Q.remove()
        for each v ∈ adj[u] do
            if d[v] > d[u] + weight(u, v) then
                d[v] = d[u] + weight(u, v)
                p[v] = u
                l[v] = l[u] + 1
            else
                if d[v] == d[u] + weight(u, v) and l[v] > l[u] + 1 then
                    l[v] = l[u] + 1
                    p[v] = u
                end if
            end if
        end for
    end while
    return p
```

Time Complexity Analysis and Proof of Correctness:

The proposed algorithm has only a singular change in the actual Dijkstra's algorithm and that is an additional check for the minimum number of edges in the *else...if* part inside the while loop which is a constant time operation. Hence, the overall run-time complexity remains the same as that of the original Dijkstra's algorithm, i.e., $O((V + E)logV)$ when we use a Heap Data Structure for Q where V is the number of vertices and E is the number of edges in the graph G.

As the only change in the algorithm is an additional *if* condition, if we can show that this change works correctly, then the entire algorithm works correctly as the proof of correctness of the Dijkstra's algorithm has already been covered in the class lectures. If we find a path which has some path weight to a node v from the source s but includes a lesser number of edges than a previous path to v from s, then we update the parent of v to the previous node u. This does not change the minimum path weight since we are only making this change when the path weight is equal to the minimum path weight that was found earlier for the vertex v. As we know that the Dijkstra's algorithm ensures that we get the minimum weight path to a vertex v Therefore, with this small modification in the Dijkstra's algorithm, we ensure that the algorithm works correctly to give us the minimum weight path having the least number of edges between the source vertex s and the target vertex t.