Green University

Lab report of CSE 206

Course Title: Algorithms Lab
Course Code: CSE 206
Section: PC DA

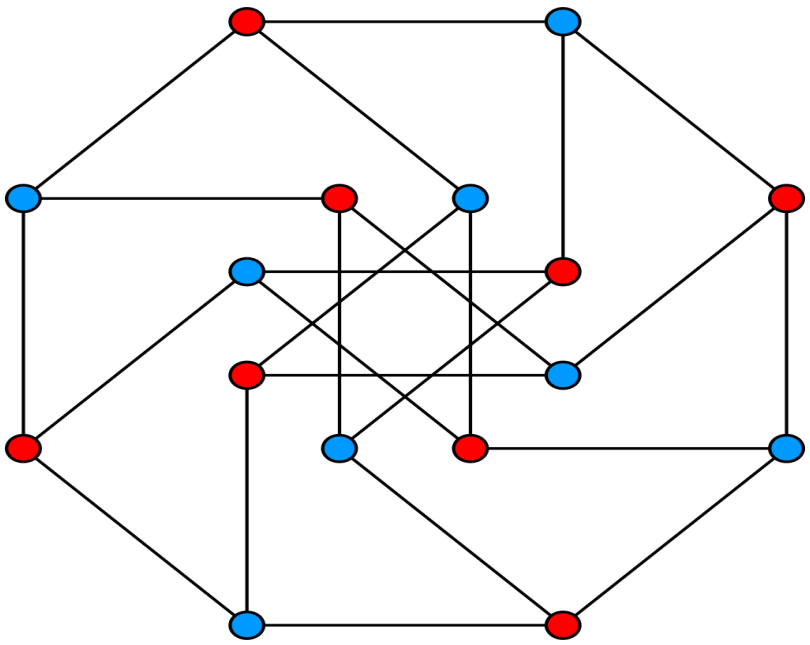
Lab report 05

Submitted to
Dr. Shah Murtaza Rashid Al Masud
Assistant Professor
Dept. of CSE
Green University of Bangladesh

Submitted by:
Mohammad Nazmul Hossain
ID:193902031
Dept. of CSE

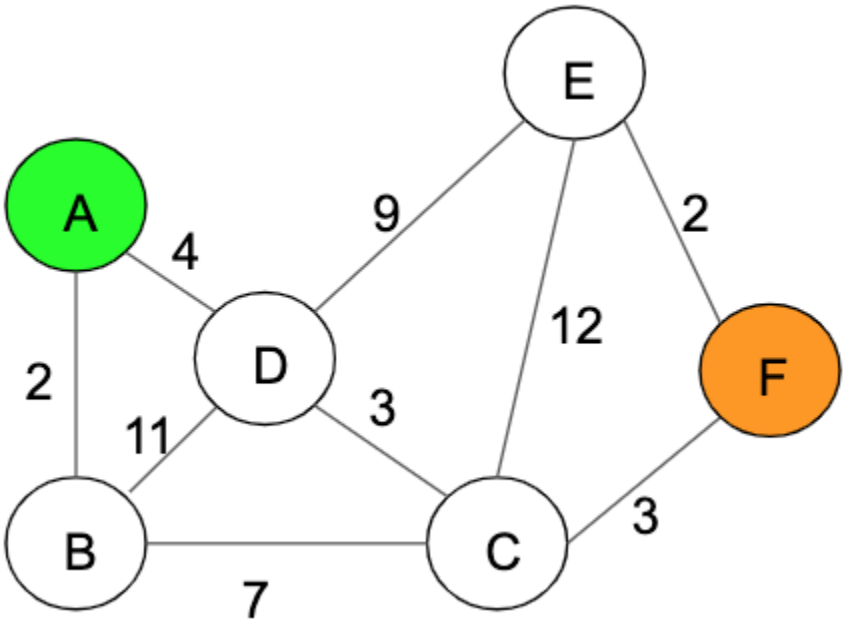
What is Dijkstra's Algorithm?

What if you were shown a graph of nodes in which each node is connected to a number of other nodes at different distances? What is the shortest path to every other node in the graph if you start from one of the nodes in the graph?
Well, Dijkstra's Algorithm is a simple algorithm that is used to find the shortest distance, or path, from a starting node to a target node in a weighted graph.
The shortest path from the starting node, the source, to all other nodes (points) in the graph is created by this algorithm.
Dijkstra's algorithm makes use of weights of the edges for finding the path that minimizes the total distance (weight) among the source node and all other nodes. This algorithm is also known as the single-source shortest path algorithm.
Dijkstra's algorithm is an iterative algorithm that finds the shortest path from a single starting node to all other nodes in a graph. It differs from the minimum spanning tree in that the shortest distance between two vertices may not include all of the graph's vertices.
It's important to note that Dijkstra's algorithm works only when all of the weights are positive since the weights of the edges are added to find the shortest path during execution.
Dijkstra's algorithm, in general, works on the principle of relaxation, in which an approximation of the exact distance is gradually displaced by more appropriate values until the shortest distance is reached.
Furthermore, the estimated distance to each node is always an overestimate of the true distance, and it is usually replaced with the distance of a recently determined path.



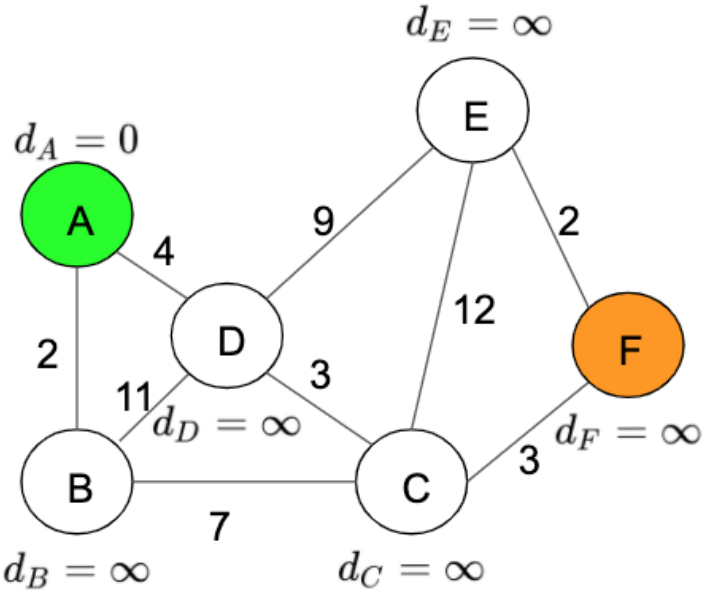
HOW IT WORKS

To explain how this algorithm works first I need to show you the graph that we are going to use in the rest of this section:



In the previous graph we can see 6 nodes from A-F, node A (highlighted green) is our start point and node F (highlighted orange) is our destination. Let's now analyse this algorithm step by step :

1. Initialise all the distances with the value infinite, except the initial node which is going to be initialized with the value 0.



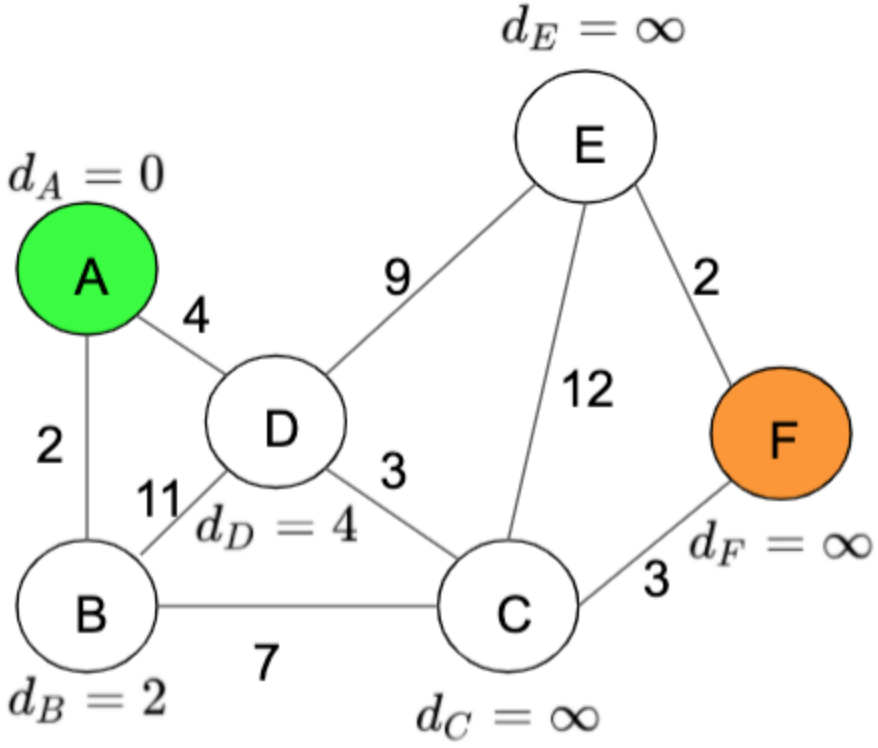
Distance table:

Current Node	Distance	Previous closed Node
A	0	None
B	Infinity	None
C	Infinity	None
D	Infinity	None
E	Infinity	None
F	Infinity	None

2. Add the current node, in this case 'Node A', into the visited array. We have to maintain this information to prevent analysing the same node more than once.

Visited = | A | | | | | |

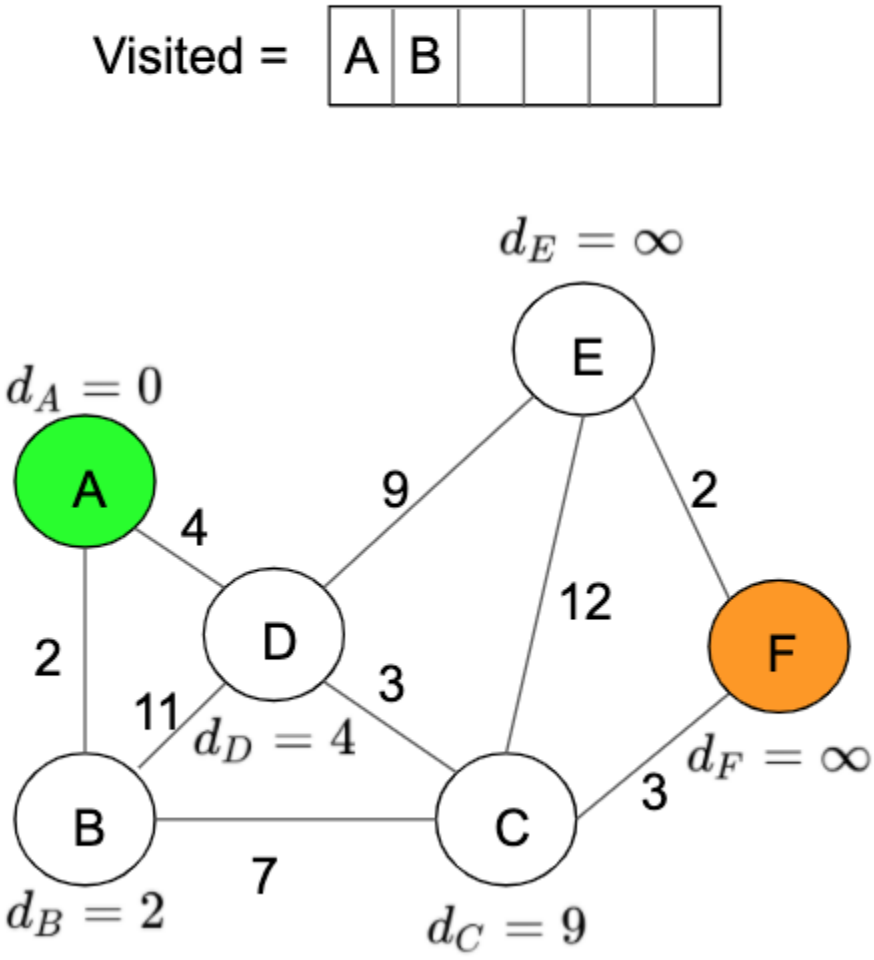
3. Calculate the distance between the current node and its children, the formula to calculate this distance is: total_distance = current_node_distance + path_distance, for example, between A and B the total_distant = 0 + 2 = 2. After this iteration the graph looks like this:



Distance table:

Current Node	Distance	Previous closed Node
A	0	None
B	2	A
C	Infinity	None
D	4	A
E	Infinity	None
F	Infinity	None

4. Now, analyse the next node. This node will be the no visited node with the minimum distance, in our case this node will be 'Node B'. In this step we are going to repeat steps 2 and 3:

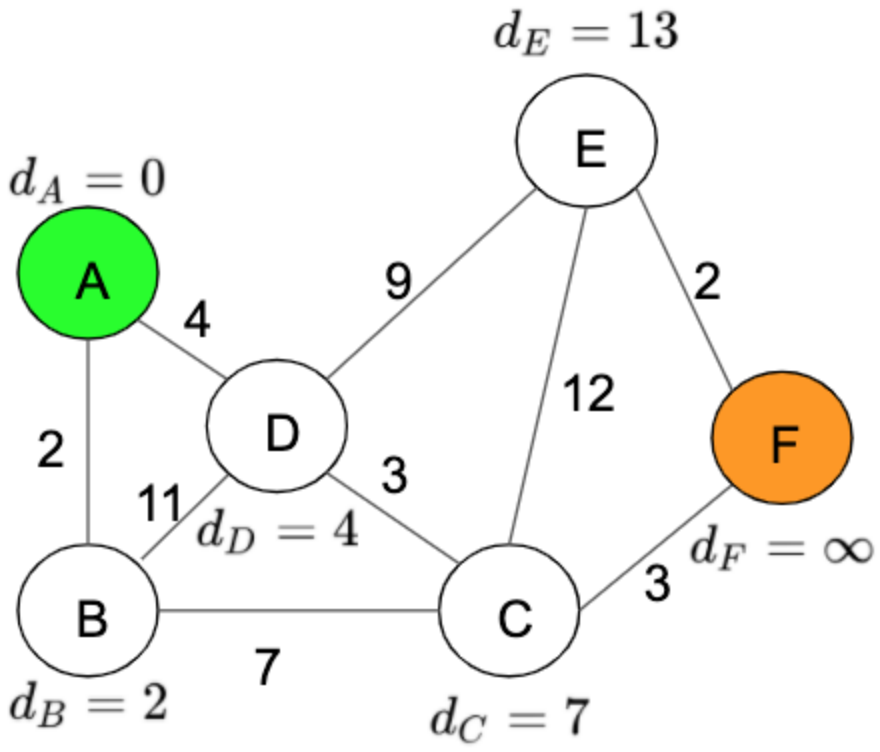


Distance table:

Current Node	Distance	Previous closed Node
A	0	None
B	2	A
C	9	B
D	4	A
E	Infinity	None
F	Infinity	None

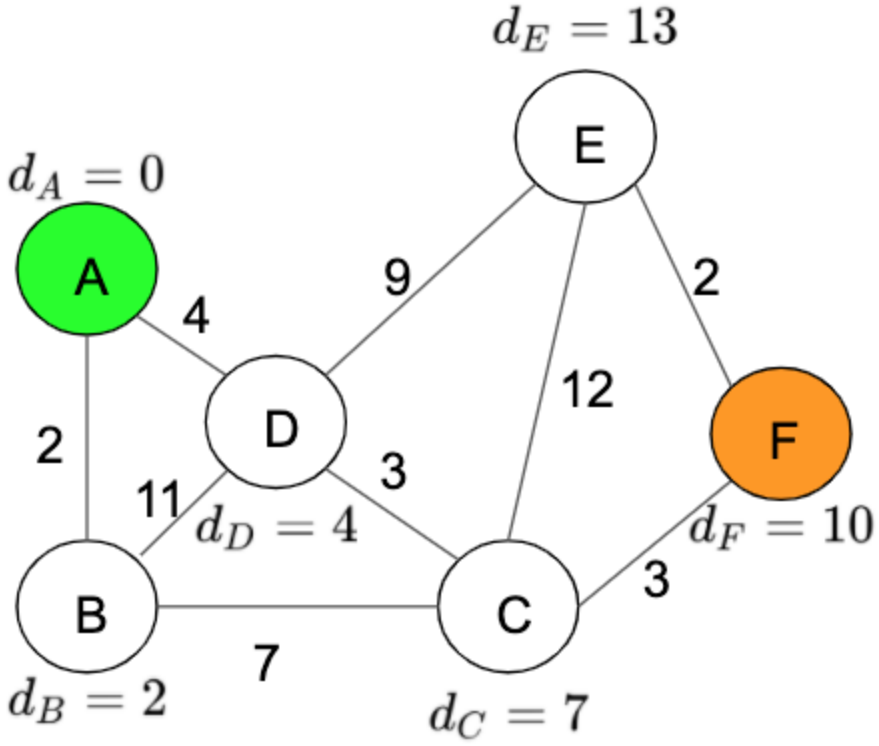
5. After that, analyse the next node, the next node will be the no visited node with the minimum distance, in our case this node will be 'Node D'. In this step we are going to repeat steps 2 and 3 again.

Visited = | A | B | D | | | |



Current Node	Distance	Previous closed Node
A	0	None
B	2	A
C	7	D
D	4	A
E	13	D
F	Infinity	None

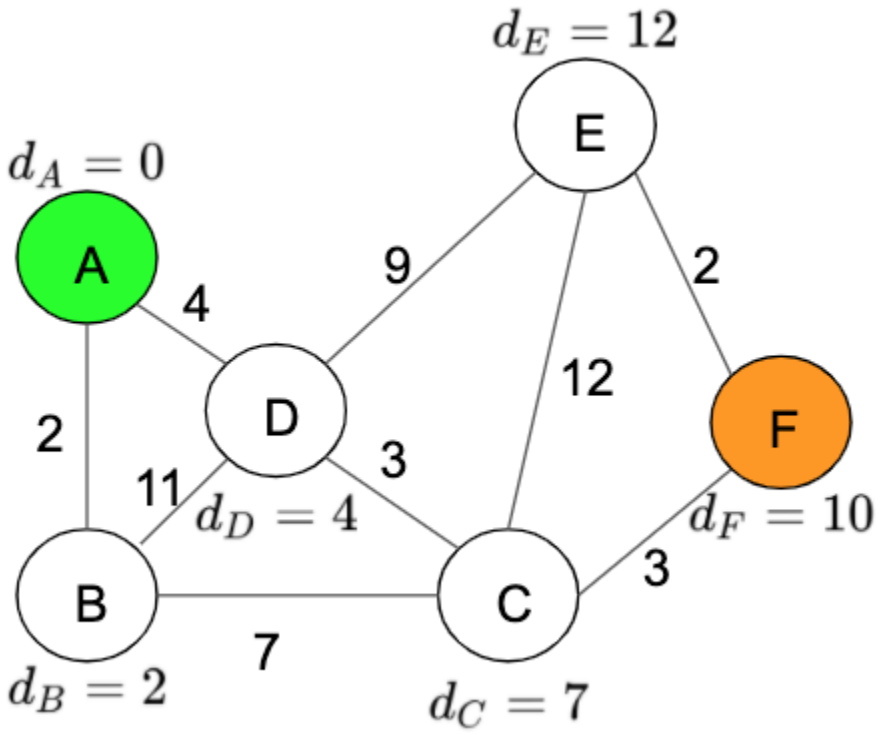
6. We repeat the process this time with ‘Node C’.



Current Node	Distance	Previous closed Node
A	0	None
B	2	A
C	7	D
D	4	A
E	13	D
F	10	C

7. We repeat the process this time with ‘Node F’.

Visited = | A | B | D | C | F | |

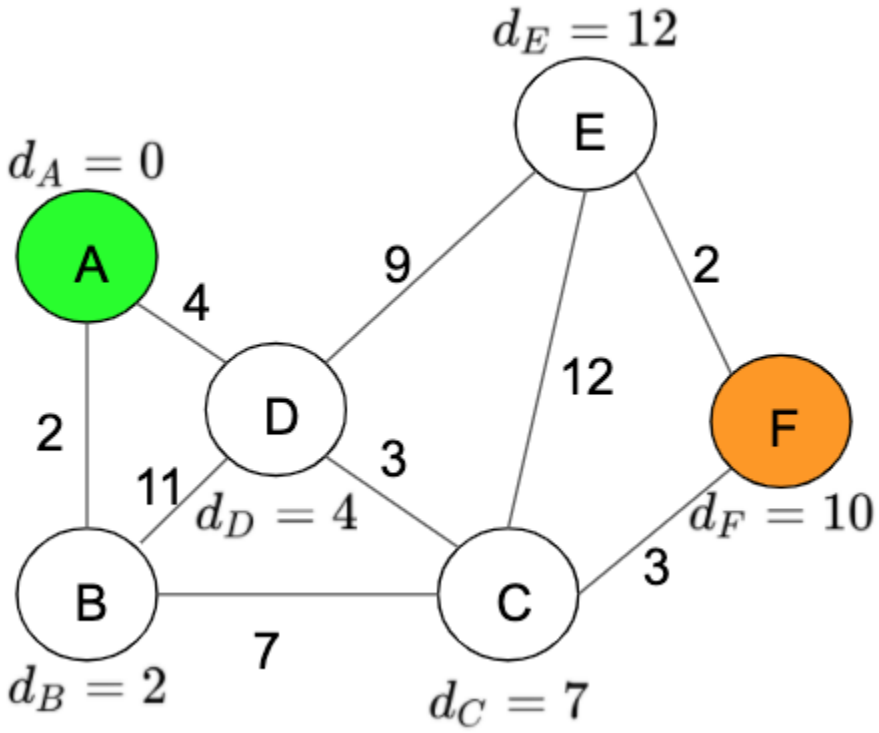


Distance table:

Current Node	Distance	Previous closed Node
A	0	None
B	2	A
C	7	D
D	4	A
E	12	F
F	10	C

8. Finally, we analyse the last node in this case, ‘Node E’.

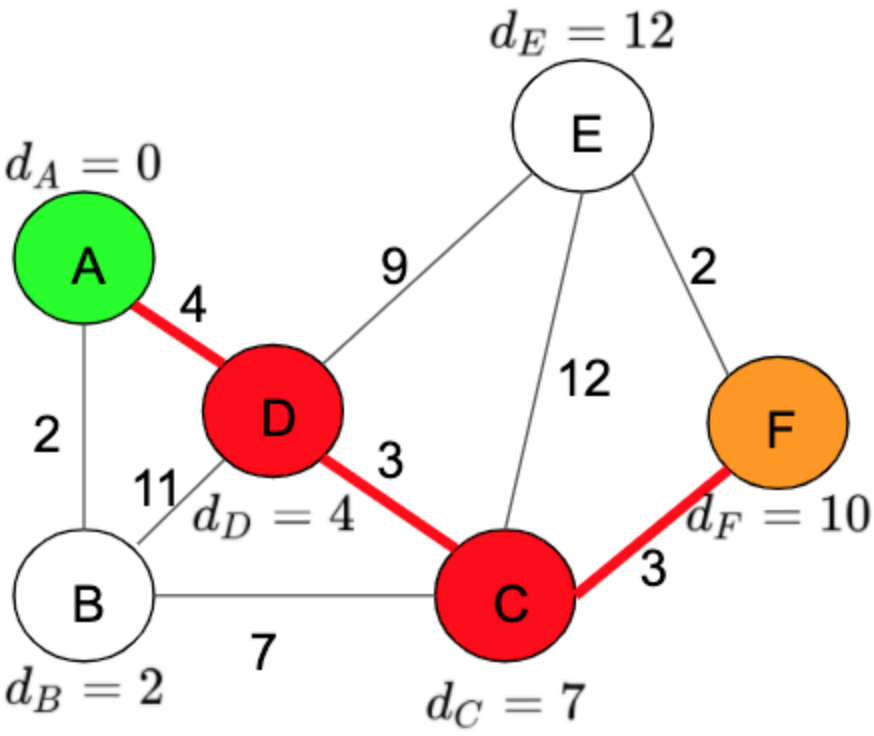
Visited = | A | B | D | C | F | E |



Current Node	Distance	Previous closed Node
A	0	None
B	2	A
C	7	D
D	4	A
E	12	F
F	10	C

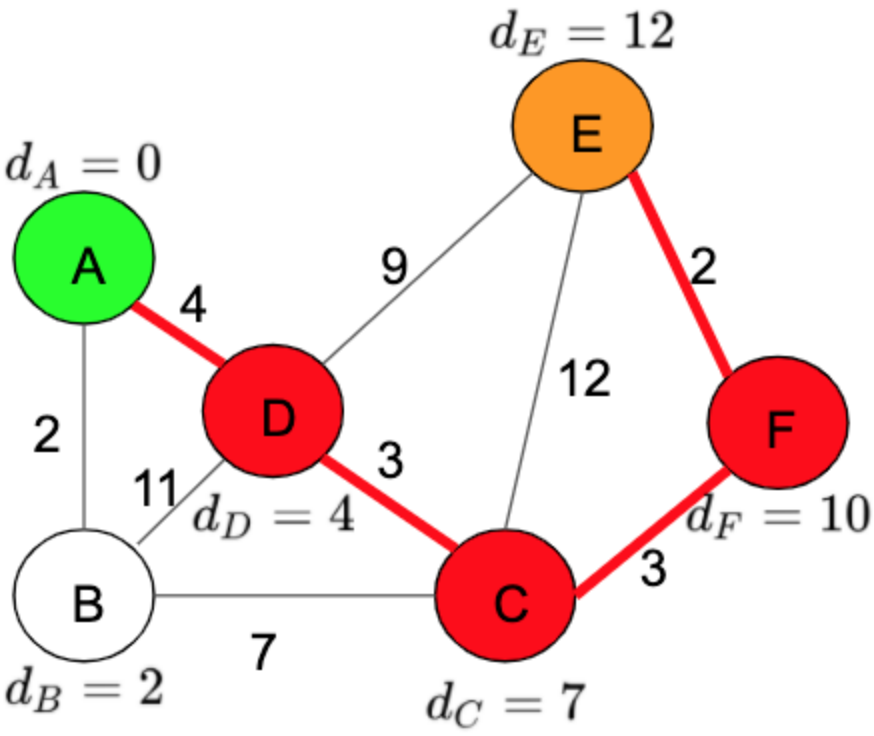
Now that we have our table with all the distances from 'Node A', we can find the shortest path from F. To do this, we have to find the distance from the current node F to Node A which is 10, and the path is calculated using the previous closed nodes results which are:

A -> D -> C -> F



As I said before, we can use the information in our table to find the shortest paths between A and any of the nodes in our graph. For example, between Node E and Node A the shortest path will be:

A -> D -> C -> F -> E



Algorithm

The implementation in our code is really similar to the steps I have shown you before. The only difference between our code and the steps explained is that in our code we stop as soon as we find the final node. The reason behind this decision is because we are not saving the information for future searches in order to simplify our code because the goal of this application is to show the algorithm, not the performance.

```
calculateShorterPath = (treeNodeArray: Array<TreeNode>) => {

    let currentNode: TreeNode | null = treeNodeArray[0];

    do {
        this.visited.add(currentNode.id);
        const distant: number = currentNode.distant;
        const currentDijkstra: DijkstraModel = new ...;
        this.dijkstraModelArray.push(currentDijkstra);
        this.iterateChild(currentDijkstra);
        currentNode = this.getNextNotVisitedNode();
```

```
    } while (!isNil(currentNode) && !currentNode.isEnd);

    if (isNil(currentNode)) return new DijkstraResModel([], []);

    const path: Array<Point> = this.createPath(currentNode);

    return new DijkstraResModel(this.dijkstraModelArray, path);
  }
```

We can see in our code the following steps:

1. Save visited node
2. Iterate child to calculate the distance

3. Find next minimum node not visited yet
4. Repeat until the end Node is found
5. Create a path with the previously calculated information.