**⭐Dijkstra's Algorithm⭐**

One of the most famous and widely used algorithms around!

Finds the shortest path between two vertices on a graph

"What's the fastest way to get from point A to point B?"

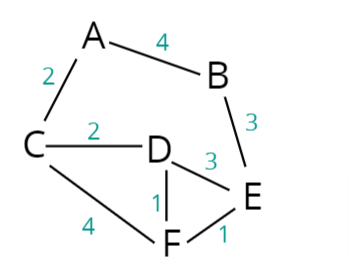
**Edsger Dijkstra** was a Dutch programmer, who has discovered this Algorithm.

**WHY IS IT USEFUL?**

* GPS - finding fastest route
* Network Routing - finds open shortest path for data
* Biology - used to model the spread of viruses among humans
* Airline tickets - finding cheapest route to your destination
* Many other uses!

**Weighted Graph:**

**Visualization:**

****

It would be look like this,

{

"A": [{node: "B", weight: 10}]

}

**Code Implementation:**

class WeightedGraph {

constructor() {

this.adjacencyList = {};

}

addVertex(vertex){

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2, weight){

if(this.adjacencyList[vertex1])

this.adjacencyList[vertex1].push({node: vertex2, weight: weight});

if(this.adjacencyList[vertex2])

this.adjacencyList[vertex2].push({node: vertex1, weight: weight});

}

}

let weightedGraph = new WeightedGraph();

**//Adding vertex;**

weightedGraph.addVertex('A');  
weightedGraph.addVertex('B');  
weightedGraph.addVertex('C');  
weightedGraph.addVertex('D');  
weightedGraph.addVertex('E');  
weightedGraph.addVertex('F');

**//Adding Edge between Vertices**

weightedGraph.addEdge('A', 'B', 4);  
weightedGraph.addEdge('B', 'E', 3);  
weightedGraph.addEdge('A', 'C', 2);  
weightedGraph.addEdge('C', 'D', 2);  
weightedGraph.addEdge('C', 'F', 4);  
weightedGraph.addEdge('D', 'F', 1);  
weightedGraph.addEdge('D', 'E', 3);  
weightedGraph.addEdge('E', 'F', 1);

**Output:**

weightedGraph.adjacencyList; //***To get the Output i.e. AdjacencyList***

**A Simple Priority Queue:**

Note: This is our Naive Priority Queue. Here, we're sorting which is O(N\*log(N)). Although we've an Optimise solution using Binary Heap.

class PriorityQueue {

constructor(){

this.values = [];

}

enqueue(val, priority) {

**// Going to accept value along with their Priority Order.**

this.values.push({val, priority});

this.sort();

};

dequeue() {

**// dequeue the Lowest priority first.**

return this.values.shift();

};

sort() {

**// Going to sort each value according to their priority. Lowest priority will come at lower index number. Like we usually sort numbers of an array.**

this.values.sort((a, b) => a.priority - b.priority);

};

}

const priorityQueue = new PriorityQueue();

priorityQueue.enqueue(val, priority);  
priorityQueue.enqueue('P', 3);  
priorityQueue.enqueue('Q', 1);  
priorityQueue.enqueue('R', 5);  
priorityQueue.enqueue('S', 4);

**/\* Output:-**

**//After Enqueue:**

values: Array(4)  
0: {val: 'Q', priority: 1}  
1: {val: 'P', priority: 3}  
2: {val: 'S', priority: 4}  
3: {val: 'R', priority: 5}

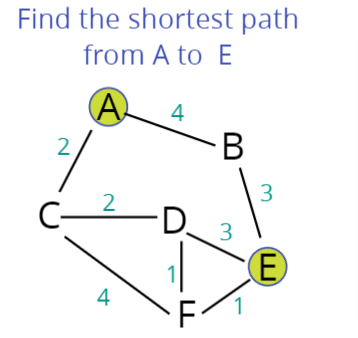
**//After Dequeue: priorityQueue.dequeue();**

{val: 'Q', priority: 1}  
{val: 'P', priority: 3}  
{val: 'S', priority: 4}  
{val: 'R', priority: 5}

\*/

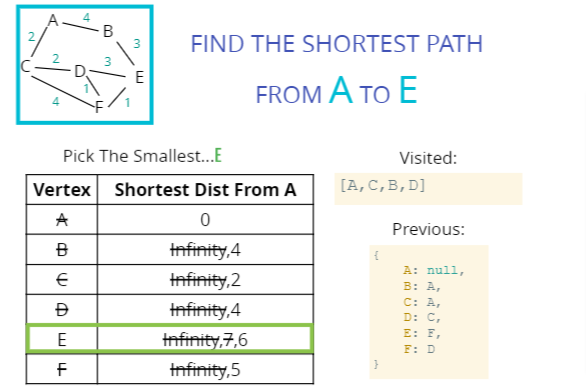
**Dijkstra's Algorithm Implementation**

**To find the shortest:**

****

We’re going to implement this Algo. Using Naïve **‘Priority Queue’.**

**Visualization:**



**Approach to understand this Algorithm:**

* Every time we look to visit a new node, we pick the node with the smallest known distance to visit first.
* Once we’ve moved to the node we’re going to visit, we look at each of its neighbors
* For each neighboring node, we calculate the distance by summing the total edges that lead to the node we’re checking from the starting node.
* If the new total distance to a node is less than the previous total, we store the new shorter distance for that node.

**Points to remember:**

* So the very first thing we do is initialize the shortest distance from ‘A’ for every vertex. And we don’t really know how to do that. We don’t’ know what the shortest distance is from A to get to F.
* So we just put **infinity** at the beginning at every vertex from ‘A’. But we know the shortest distance from ‘A’ to ‘A’ i.e. **0**.

**Note:**

While we’re implementing Dijkstra's Algorithm, we’ll have a data structure (here called ‘Previous’ – an Object list) gives use the shortest path from A to all the nodes across the graph.

**Pseudo Code:**

* This function should accept a starting and ending vertex
* Create an object (we'll call it distances) and set each key to be every vertex in the adjacency list with a value of infinity, except for the starting vertex which should have a value of 0.
* After setting a value in the distances object, add each vertex with a priority of Infinity to the priority queue, except the starting vertex, which should have a priority of 0 because that's where we begin.
* Create another object called previous and set each key to be every vertex in the adjacency list with a value of null
* Start looping as long as there is anything in the priority queue
* dequeue a vertex from the priority queue
* If that vertex is the same as the ending vertex - we are done!
* Otherwise loop through each value in the adjacency list at that vertex
* Calculate the distance to that vertex from the starting vertex
* if the distance is less than what is currently stored in our distances object
* update the distances object with new lower distance
* update the previous object to contain that vertex
* enqueue the vertex with the total distance from the start node

**Code Implementation:-**