**⭐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?"

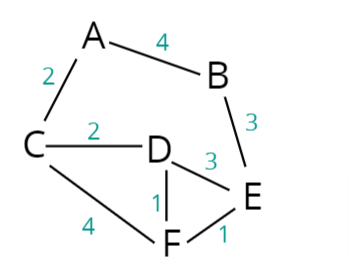
**Edsger Dijkstra** was a Dutch programmer, who has discovered this Algorithm.

**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}

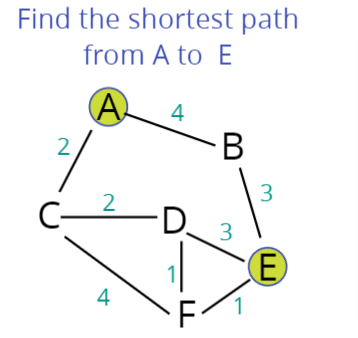
**//After Dequeue: priorityQueue.dequeue();**

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

\*/

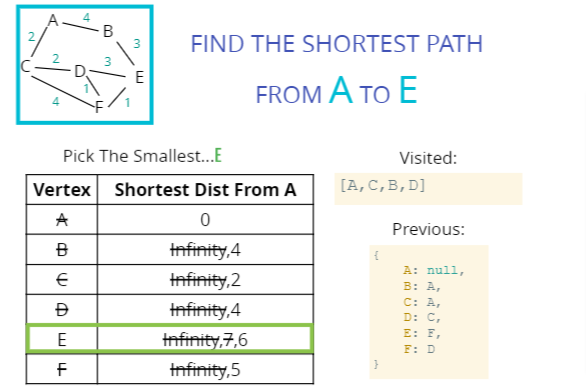
**Dijkstra's Algorithm Implementation**

**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

**Approach to understand the Code:**

* Prerequisites: Priority Queue & Weighted Graph
* Priority Queue:
* Help to dequeue the value having Minimum distance from the starting vertex.
* Store/Enqueue all vertices along with distance from the starting vertex in 'values' array
* Store/Enqueue vertex with new Minimum distance from the starting vertex
* Sort all the vertex as per Minimum distance after storing each vertex to 'values' array.
* Weighted Graph:
* Construct a network of vertices called Graph.
* Store each vertex in the adjacencyList (Object) and connect each vertex by edge with other vertices along with some weight (i.e. distance between
* vertex) as an Array of Object for each vertex having neighboring vertex.
* Call Dijkstra Algorithm inside this weightedGraph to find the shortest path from starting vertex to finish vertex.
* Note:
* The difference between adjacencyList{} & distance{} is that:
* adjacencyList{}: Distance associated with the vertex as weight is the distance from its neighboring vertex. This distance is set as weight while connecting the vertices by edges to form a graph. Here, distance will not get updated.
* distance{}: Distance associated with the vertex is the distance from the starting vertex. This distance will set as we start finding the shortest path between start & finish vertex. It will get update for every vertex as we got the minimum distance from the starting vertex.
* Dijkstra Algorithm: Find the Shortest path from Starting Vertex to Finish Vertex.
* Accept 'start' and 'finish' vertex as argument.
* Create an Object of PriorityQueue as 'nodes' for the sake of convinience to easily access its properties & methods.
* Initialize a 'distance' object to store Minimum distance from start vertex to each vertex.
* Initialize a 'previous' object to store a vertex that tell from where it came from for each corresponding vertex.
* Declare a variable called 'smallest' to store the vertex having smallest distance from starting vertex, by dequeue from the Priority Queue.
* BUILD UP INITIAL STATE to start Dijkstra Algorithm:
* Till now, adjacencyList of weightedGraph consisting all the vertices along with their corresponding neighbors and their weights.
  + Loop over the adjacencyList to get each and every vertex from their key.
    - Set the value (Initla distance) of starting vertex as 0 and all other vertices as Infinity in the distance object. (Because Initally, the distance from starting to itselft would be 0 & we don't know about other vertices where they are. So we've to assume, the distance from starting to other vertices would be Infinity).
    - Enqueue each vertex as 'val' to the priorityQueue along with their initial distance as 'priority', which would going to store and sort (as per distance) in an array called 'values'.
    - Set the value of every vertex as null to the previous object. (Because initially, every vertex haven't reached by any vertex yet).
* As long as their is something in the 'values' array of priorityQueue, start a loop.

Assign a smallest vertex to the 'smallest' variable by dequeue from the Priority Queue. That vertex would be the value of that vertex having smallest distance from the starting vertex.

* + Start a loop over the neighboring vertices (stored as an object means neighboring vertex would be store as an object consisting its value in 'node' key & weight in 'weight' key from the smallest vertex) of that smallest vertex in the adjacencyList[smallest].
    - Store that neighboring vertex as 'nextNode' (Neighboring vertex would be in the form of an object)
    - Calculate a new distance from smallest to that neighboring vertex and Assign it in a variable called 'newDistance'. It might be Minimum or Maximum distance from the previous distance of that neighboring vertex.
    - Store the value of nextNode to 'nextNeighbor'.
    - if(newDistance less than the previous distance of neighboring vertex distance[nextNeighbor])
      * Updating the previous distance of neighboring vertex with new smallest distance by (distance[nextNeighbor] = newDistance;)
      * Update the previous reached vertex to neighboring vertex with the smallest vertex in the previous object as (previous[nextNeighbor] = smallest;)
      * Enqueue that neighbor vertex to the priorityQueue with Minimum newDistance (as new priority). That would store in the 'values' array.
    - if(smallest vertex === finish vertex), then we've to stop the loop by returning the final shortest path because we reached the finish vertex.
      * Initialize an empty 'path' array.
      * if(the value smallest vertex would be there something in the previous object)
        + push that smallest vertex to the path array.
        + Update the smallest variable with the value of current smallest in the previous object
        + As the value of current smallest would be null in the previous object move out of this block.
    - return the path array by concatinating with the start vertex (basically pushing to the path array) and reverse it.

**Code Implementation:  
✨ with Naive(Unoptimized) Priority Queue ✨**

class PriorityQueue {

constructor(){

// work as the visited array

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);

};

}

// Weighted Graph

class WeightedGraph {

constructor() {

this.adjacencyList = {};

}

// Add Vertex

addVertex(vertex){

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

// Add Edge between the two Vertices along with weight

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});

}

// Dijkstra Algorithm

Dijkstra(start, finish){

const nodes = new PriorityQueue();

// Store the shortest distances from Vertex 'A'or starting Vertex to each vertex

const distances ={};

// Store the vertex that tell from where it came from for each corresponding vertex.

const previous = {};

// Store the vertex having smallest distance from starting vertex, by dequeue from the Priority Queue

let smallest;

// build up initial State

for(let vertex in this.adjacencyList){

if(vertex === start){

distances[vertex] = 0;

// enqueue of vertex in Priority Queue with distance as priority.

nodes.enqueue(vertex, 0);

}

else{

distances[vertex] = Infinity;

nodes.enqueue(vertex, Infinity);

}

previous[vertex] = null;

}

//as long as there is something to visit

while(nodes.values.length){

// get the smallest node(Smallest Node will whatever the actual node we're visiting is.)

smallest = nodes.dequeue().val;

if(smallest === finish){

//We're DONE;

// BUILD UP PATH TO RETURN AT END

let path = [];

while(previous[smallest]){

path.push(smallest);

// updating smallest till starting vertex(having value = null);

smallest = previous[smallest];

}

return path.concat(smallest).reverse();

}

if(smallest || distances[smallest]!==Infinity){

for(let neighbor in this.adjacencyList[smallest]){

let nextNode = this.adjacencyList[smallest][neighbor];

//Calculate new distances to neighboring node from the starting vertex

let newDistance = distances[smallest] + nextNode.weight,

nextNeighbor = nextNode.node;

// if the newDistance is Minimum then only, update the distances & previous of

// nextNeighbor and enqueue with that new Minimum distance as new Priority.

if(newDistance < distances[nextNeighbor]){

// updating new smallest distance to neighbor

distances[nextNeighbor] = newDistance;

// updating previous - How we got to neighbor

previous[nextNeighbor] = smallest;

//enqueue in priority queue with new priority (Minimum newDistance)

nodes.enqueue(nextNeighbor, newDistance);

}

}

}

}

}

}

let graph = new WeightedGraph();

//Adding vertex;

graph.addVertex('A');

graph.addVertex('B');

graph.addVertex('C');

graph.addVertex('D');

graph.addVertex('E');

graph.addVertex('F');

//Adding Edge between Vertices along with weight

graph.addEdge('A', 'B', 4);

graph.addEdge('B', 'E', 3);

graph.addEdge('A', 'C', 2);

graph.addEdge('C', 'D', 2);

graph.addEdge('C', 'F', 4);

graph.addEdge('D', 'F', 1);

graph.addEdge('D', 'E', 3);

graph.addEdge('E', 'F', 1);

// **INPUT**: Find the shortest path from vertex 'A' to 'E'

graph.Dijkstra('A', 'E');

\*/

// //**OUTPUT**: shortest path from vertex 'A' to 'E'

// ['A', 'C', 'D', 'F', 'E']

**Code Implementation:  
✨ with Binary Heap(an Optimized) Priority Queue ✨**

// Weighted Graph

class WeightedGraph {

constructor() {

this.adjacencyList = {};

}

// Add Vertex

addVertex(vertex){

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

// Add Edge between the two Vertices along with weight

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});

}

// Dijkstra Algorithm

Dijkstra(start, finish){

const nodes = new PriorityQueue();

// Store the shortest distances from Vertex 'A'or starting Vertex to each vertex

const distances ={};

// Store the vertex that tell from where it came from for each corresponding vertex.

const previous = {};

// Store the vertex having smallest distance from starting vertex, by dequeue from the Priority Queue

let smallest;

// build up initial State

for(let vertex in this.adjacencyList){

if(vertex === start){

distances[vertex] = 0;

// enqueue of vertex in Priority Queue with distance as priority.

nodes.enqueue(vertex, 0);

}

else{

distances[vertex] = Infinity;

nodes.enqueue(vertex, Infinity);

}

previous[vertex] = null;

}

//as long as there is something to visit

while(nodes.value.length){

// get the value of smallest node(Smallest Node will whatever the actual node we're visiting is.)

smallest = nodes.dequeue().val;

if(smallest === finish){

//We're DONE;

// BUILD UP PATH TO RETURN AT END

let path = [];

while(previous[smallest]){

path.push(smallest);

// updating smallest till starting vertex(having smallest vertex's value == null);

smallest = previous[smallest];

}

return path.concat(smallest).reverse();

}

if(smallest || distances[smallest]!==Infinity){

for(let neighbor in this.adjacencyList[smallest]){

let nextNode = this.adjacencyList[smallest][neighbor];

//Calculate new distances to neighboring node from the starting vertex

let newDistance = distances[smallest] + nextNode.weight,

nextNeighbor = nextNode.node;

// if the newDistance is Minimum then only, update the distances & previous of

// nextNeighbor and enqueue with that new Minimum distance as new Priority.

if(newDistance < distances[nextNeighbor]){

// updating new smallest distance to neighbor

distances[nextNeighbor] = newDistance;

// updating previous - How we got to neighbor

previous[nextNeighbor] = smallest;

//enqueue in priority queue with new priority (Minimum newDistance)

nodes.enqueue(nextNeighbor, newDistance);

}

}

}

}

}

}

// An Optimized Priority Queue

class PriorityQueue {

constructor() {

this.value = [];

}

// Adding to create MinBinary Heap

enqueue(value, priority){

let newNode = new Node(value, priority);

this.value.push(newNode);

this.bubbleUp();

}

// Bubbling to adjust as MinBinary Heap

bubbleUp(){

let idx = this.value.length-1,

element = this.value[idx];

while(idx>0){

let parentIdx = Math.floor((idx-1)/2),

parent = this.value[parentIdx];

if(element.priority >= parent.priority) break;

this.value[parentIdx] = element;

this.value[idx] = parent;

idx = parentIdx;

}

}

// Removing from MinBinary Heap

dequeue(){

const min = this.value[0],

end = this.value.pop();

if(this.value.length>0){

this.value[0] = end;

this.sinkDown();

}

// return the minimum distance vertex in the form of object

return min;

}

//BubblingDown or say sinkDown to adjust as MinBinary Heap

sinkDown(){

let idx = 0,

element = this.value[0],

length = this.value.length;

while(true){

let leftChildIdx = 2\*idx +1,

rightChildIdx = 2\*idx +2,

LeftChild, RightChild,

swap = null;

if(leftChildIdx<length){

LeftChild = this.value[leftChildIdx];

if(element.priority > LeftChild.priority){

swap = leftChildIdx;

}

}

if(rightChildIdx<length){

RightChild = this.value[rightChildIdx];

if(swap===null && RightChild.priority<element.priority || swap!==null && RightChild.priority<LeftChild.priority){

swap = rightChildIdx;

}

}

if(swap===null) break;

this.value[idx] = this.value[swap];

this.value[swap] = element;

idx = swap;

}

}

}

class Node {

constructor(val, priority) {

this.val = val;

this.priority = priority;

}

}

let graph = new WeightedGraph();

//Adding vertex;

graph.addVertex('A');

graph.addVertex('B');

graph.addVertex('C');

graph.addVertex('D');

graph.addVertex('E');

graph.addVertex('F');

//Adding Edge between Vertices along with weight

graph.addEdge('A', 'B', 4);

graph.addEdge('B', 'E', 3);

graph.addEdge('A', 'C', 2);

graph.addEdge('C', 'D', 2);

graph.addEdge('C', 'F', 4);

graph.addEdge('D', 'F', 1);

graph.addEdge('D', 'E', 3);

graph.addEdge('E', 'F', 1);

\*/

// **INPUT**: Find the shortest path from vertex 'A' to 'E'

graph.Dijkstra('A', 'E');

\*/

// //**OUTPUT**: shortest path from vertex 'A' to 'E'

// ['A', 'C', 'D', 'F', 'E']