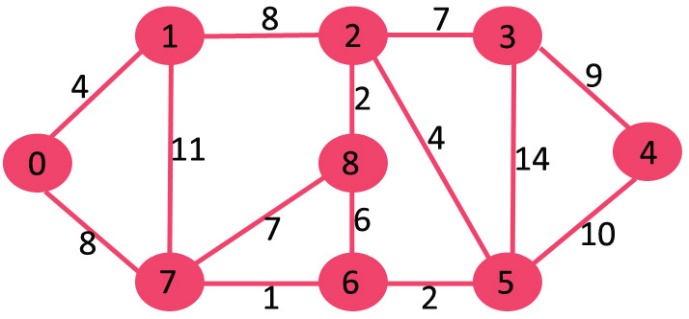
<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

Dijkstra’s shortest path algorithm | Greedy Algo-7

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/). Like Prim’s MST, we generate a*SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.  
Algorithm  
**1)** Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.  
**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.  
**3)** While *sptSet* doesn’t include all vertices  
….**a)** Pick a vertex u which is not there in *sptSet* and has minimum distance value.  
….**b)** Include u to *sptSet*.  
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with the following example:  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST1.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ2.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).  
[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ3.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ4.jpg)

We repeat the above steps until sptSet does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ5.jpg)

**How to implement the above algorithm?**

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

// A Java program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class ShortestPath {

// A utility function to find the vertex with minimum distance value,

// from the set of vertices not yet included in shortest path tree

static final int V = 9;

int minDistance(int dist[], Boolean sptSet[])

{

// Initialize min value

int min = Integer.MAX\_VALUE, min\_index = -1;

for (int v = 0; v < V; v++)

if (sptSet[v] == false && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

return min\_index;

}

// A utility function to print the constructed distance array

void printSolution(int dist[])

{

System.out.println("Vertex \t\t Distance from Source");

for (int i = 0; i < V; i++)

System.out.println(i + " \t\t " + dist[i]);

}

// Function that implements Dijkstra's single source shortest path

// algorithm for a graph represented using adjacency matrix

// representation

void dijkstra(int graph[][], int src)

{

int dist[] = new int[V]; // The output array. dist[i] will hold

// the shortest distance from src to i

// sptSet[i] will true if vertex i is included in shortest

// path tree or shortest distance from src to i is finalized

Boolean sptSet[] = new Boolean[V];

// Initialize all distances as INFINITE and stpSet[] as false

for (int i = 0; i < V; i++) {

dist[i] = Integer.MAX\_VALUE;

sptSet[i] = false;

}

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum distance vertex from the set of vertices

// not yet processed. u is always equal to src in first

// iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the

// picked vertex.

for (int v = 0; v < V; v++)

// Update dist[v] only if is not in sptSet, there is an

// edge from u to v, and total weight of path from src to

// v through u is smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] != 0 && dist[u] != Integer.MAX\_VALUE && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

// print the constructed distance array

printSolution(dist);

}

// Driver method

public static void main(String[] args)

{

/\* Let us create the example graph discussed above \*/

int graph[][] = new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },

{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },

{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },

{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },

{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },

{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },

{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },

{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },

{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

ShortestPath t = new ShortestPath();

t.dijkstra(graph, 0);

}

}

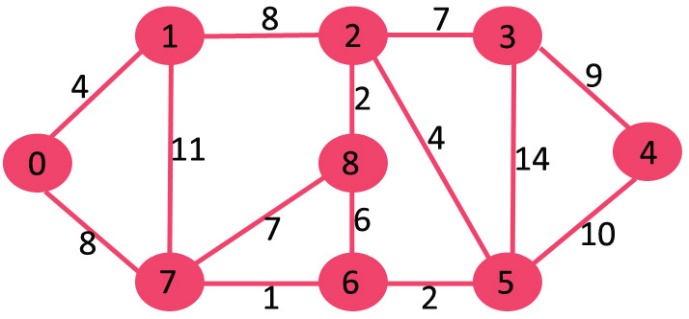
// This code is contributed by Aakash Hasija

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-in-java-using-priorityqueue/>

# Dijkstra’s shortest path algorithm in Java using PriorityQueue

Given a graph with adjacency list representation of the edges between the nodes, the task is to implement [Dijkstra’s Algorithm](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) for single source shortest path using [Priority Queue](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/) in Java.

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

Input : Source = 0

Output :

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

We have discussed Dijkstra’s shortest Path implementations like [Dijkstra’s Algorithm for Adjacency Matrix Representation (With time complexity O(v2)](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)

Below is the Java implementation of Dijkstra’s Algorithm using Priority Queue:

// Java implementation of Dijkstra's Algorithm

// using Priority Queue

import java.util.\*;

public class DPQ {

private int dist[];

private Set<Integer> settled;

private PriorityQueue<Node> pq;

private int V; // Number of vertices

List<List<Node> > adj;

public DPQ(int V)

{

this.V = V;

dist = new int[V];

settled = new HashSet<Integer>();

pq = new PriorityQueue<Node>(V, new Node());

}

// Function for Dijkstra's Algorithm

public void dijkstra(List<List<Node> > adj, int src)

{

this.adj = adj;

for (int i = 0; i < V; i++)

dist[i] = Integer.MAX\_VALUE;

// Add source node to the priority queue

pq.add(new Node(src, 0));

// Distance to the source is 0

dist[src] = 0;

while (settled.size() != V) {

// remove the minimum distance node

// from the priority queue

int u = pq.remove().node;

// adding the node whose distance is

// finalized

settled.add(u);

e\_Neighbours(u);

}

}

// Function to process all the neighbours

// of the passed node

private void e\_Neighbours(int u)

{

int edgeDistance = -1;

int newDistance = -1;

// All the neighbors of v

for (int i = 0; i < adj.get(u).size(); i++) {

Node v = adj.get(u).get(i);

// If current node hasn't already been processed

if (!settled.contains(v.node)) {

edgeDistance = v.cost;

newDistance = dist[u] + edgeDistance;

// If new distance is cheaper in cost

if (newDistance < dist[v.node])

dist[v.node] = newDistance;

// Add the current node to the queue

pq.add(new Node(v.node, dist[v.node]));

}

}

}

// Driver code

public static void main(String arg[])

{

int V = 5;

int source = 0;

// Adjacency list representation of the

// connected edges

List<List<Node> > adj = new ArrayList<List<Node> >();

// Initialize list for every node

for (int i = 0; i < V; i++) {

List<Node> item = new ArrayList<Node>();

adj.add(item);

}

// Inputs for the DPQ graph

adj.get(0).add(new Node(1, 9));

adj.get(0).add(new Node(2, 6));

adj.get(0).add(new Node(3, 5));

adj.get(0).add(new Node(4, 3));

adj.get(2).add(new Node(1, 2));

adj.get(2).add(new Node(3, 4));

// Calculate the single source shortest path

DPQ dpq = new DPQ(V);

dpq.dijkstra(adj, source);

// Print the shortest path to all the nodes

// from the source node

System.out.println("The shorted path from node :");

for (int i = 0; i < dpq.dist.length; i++)

System.out.println(source + " to " + i + " is "

+ dpq.dist[i]);

}

}

// Class to represent a node in the graph

class Node implements Comparator<Node> {

public int node;

public int cost;

public Node()

{

}

public Node(int node, int cost)

{

this.node = node;

this.cost = cost;

}

@Override

public int compare(Node node1, Node node2)

{

if (node1.cost < node2.cost)

return -1;

if (node1.cost > node2.cost)

return 1;

return 0;

}

}