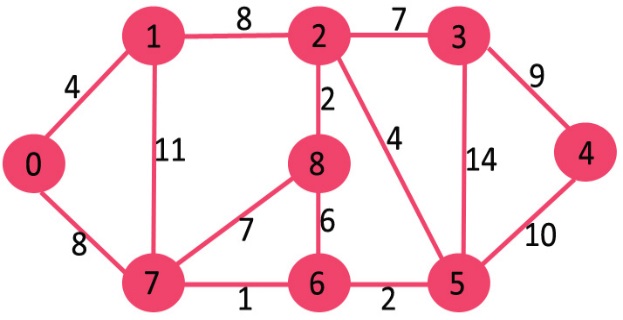
**Algorithm Implementation: Djikstra’s Algorithm for Shortest Path of a Weighted Graph Represented by an Adjacency Matrix**

**Part 1 – Weighted Graph Representation Using Adjacency Matrix:**

We can represent the following graph using the 2-D matrix below. The 2-D array consists of all the vertices, each with an array of its adjacency list. For non-weighted graph, the adjacency list consists of only 1’s and 0’s (for is connected or isn’t connected). For weighted graphs, a zero indicates that the vertices are not connected . A non-zero integer indicates that the vertices are connected. For example vertex 0 has the adjacency list {0,4,0,0,0,0,0,8,0}. This means it is connected to vertices 1 and 7 (since there indexes are non-zero) and that the distances are 4 and 8 respectively.

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

|  |
| --- |
| **public** **static** **void** main (String[] args){  **int** graph[][] = **new** **int**[][]{  {0, 4, 0, 0, 0, 0, 0, 8, 0}, //adjacency list for vertex 0  {4, 0, 8, 0, 0, 0, 0, 11, 0}, //adjacency list for vertex 1  {0, 8, 0, 7, 0, 4, 0, 0, 2}, //adjacency list for vertex 2  {0, 0, 7, 0, 9, 14, 0, 0, 0}, //adjacency list for vertex 3  {0, 0, 0, 9, 0, 10, 0, 0, 0}, //adjacency list for vertex 4  {0, 0, 4, 14, 10, 0, 2, 0, 0}, //adjacency list for vertex 5  {0, 0, 0, 0, 0, 2, 0, 1, 6}, //adjacency list for vertex 6  {8, 11, 0, 0, 0, 0, 1, 0, 7}, //adjacency list for vertex 7  {0, 0, 2, 0, 0, 0, 6, 7, 0} //adjacency list for vertex 8  };  **int** sourceVertex = 0;  **int** destinationVertex = 5;    **int** shortestPath = DjikstraAlgorithm.*getShortestPath*(graph, sourceVertex, destinationVertex);  System.*out*.println(shortestPath);  } |

**Part 2 – Important Variables and Initialization:**

There are two arrays that we will be working with to solve this problem.

1. The first is an integer array called “shortest paths”. At each index ‘i’ in this array, the integer value represents the distance between vertex ‘i’ and the source vertex (in the paramter). This array is constantly being updated when the algorithm finds faster paths to vertices.
2. The second is a boolean array called “solved vertex”. It indicates whether the vertex at index ‘i’ has been solved. If a vertex has been solved, it means its integer value in the shortest paths array is final and cannot be further optimized.

**Initialization:** The program is initialized when the boolean variable is initialized and all values are set to false (since nothing has been solved). In addition, the shortest path array is initialized with all values set to infinity EXCEPT the shortest path to start index. This is initialized to zero since the shortest path from the source (or start) index to itself is zero.

|  |
| --- |
| **public** **static** **int** getShortestPath(**int** graph[][], **int** sourceVertex, **int** destinationVertex){  **int**[] shortestPathDistances = **new** **int**[*NUM\_OF\_VERTICES*];  Boolean[] vertexSolved = **new** Boolean[*NUM\_OF\_VERTICES*];  *initialize*(shortestPathDistances, vertexSolved, sourceVertex);  . . . . . . . . . to be continued. . . . . . . . . . . . .  }  **public** **static** **void** initialize(**int** shortestPaths[], Boolean[] vertexSolved, **int** sourceVertex){  **for** (**int** vertex = 0; vertex < *NUM\_OF\_VERTICES*; vertex++){  shortestPaths[vertex] = Integer.*MAX\_VALUE*;  vertexSolved[vertex] = **false**;  }  shortestPaths[sourceVertex] = 0;  } |

**Part 3 –Deciding Which Node to Solve Next:**

|  |
| --- |
| <http://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg> |

The algorthm starts at the starting vertex (which in this case was zero). This node gets solved first and so it gets set to solved. The next node to solve is “*the closest unsolved vertex from the orgin*”. The reason we are going to solve this vertex next is because we have solved all the vertices before it and sine it is the closest to the origin, we can leverage this information to solve it.

For example, look at the image above. From the source vertex zero, 1 is 4 away, the 7 is 8 away, the 6 is 9 away, the 2 is 12 away. So it will solve these nodes in that order. So to repeat, the next vertex to solve is “*the closest unsolved vertex from the orgin*”.

**Part 4 – Low Level Method to Get Next Vertex To Solve:**

|  |
| --- |
| **public** **static** **int** getNextVertexToSolve(**int** shortestPaths[], Boolean[] solvedVertex){  **int** shortestDistanceFromSource = Integer.*MAX\_VALUE*;  **int** nextNodeToSolve = -1;  **for** (**int** vertex = 0; vertex < *NUM\_OF\_VERTICES*; vertex++){// for every other vertex  **if** (!solvedVertex[vertex]){// if vertex is not solved  **int** distanceFromSource = shortestPaths[vertex]; //get the shortest path to this vertex  **if**(distanceFromSource <= shortestDistanceFromSource){  shortestDistanceFromSource = distanceFromSource;  nextNodeToSolve = vertex;  }  }  }  **return** nextNodeToSolve ;  } |

**Summary:** What you are doing is you are looking through all the nodes to determine which ones are not solved yet. Of the ones that are not solved, you check to see their shortest distance from the source. Whichever of these unsolved nodes that is the closest to the source will be solved next.

**Volatile Component:** If you notice, we are relying on the “shortest paths” array to determine which is the closest non-solved vertex from the source node. The “shortest paths” array specifies the (shortest) distance from source. However, this array is constantly being updated as you go right? Won’t this cause unpredictable behaviour? Just ignore the volatile component for now.

**Part 5 – High Level Solution:**

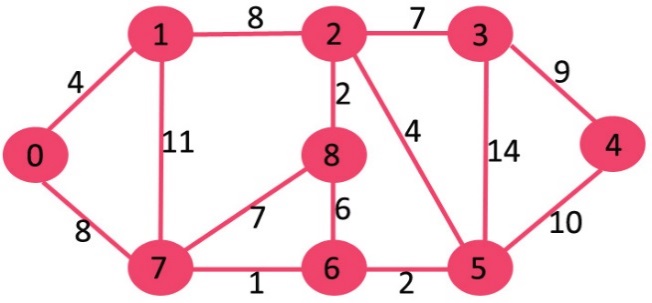
[](http://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

Figure 5A) - Initial Graph with edge distances on edges

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

Figure 5B) – Solution with Shortest Path from source to vertex on the vertex

* You start at the source node. You know that the shortest path from (source -> source) is 0.
* You analyze the next node to Solve which is the “closest non-solved node from source”. Since the source itself has not been solved, the next node to solve is the source. The shortest paths array tells us it zero distance away and the other nodes are infinity away.
* You solve the source. Then you try to update this solved node’s neighbour paths. For example, if you solve zero, and vertex 1 is a neighbour of zero, you can update vertex 1’s shortest path to (zero) + (the edge between zero and 1).
* You might be thinking “what if it is faster to go from 0 -> 7 -> 1 then it is to go from 0 -> 1? Eventually it will solve node 7. Once it does this, it will update node 7’s neighbor paths which will include 1 also.
* When the node 1 gets updated twice (it is the neighbor of 7 and the neighbor of 0) it will leave the shortest path. So for examle, node 1’s shortest path will be analyzed through all its neighbors (0,7, 2) and the one that can provide the shortest path will be left in the shortest paths array.

**Volatile Component:**

1. Each time you solve a node, you update the shortest paths of all its neughbours.
2. When your finding the next node to solve, it will be guaranteed be a neighbour of a solved node. This means that this solved node’s neighbours have their shortest paths updated.
3. The last statement allows you to be certain that it is safe to use the “shortest paths array” since what we are looking for will have its shortest path updated.

**Edge Case:** At each iteration, you are using information from the previous iteration. On the first iteration, you won’t know the shortest paths however. To get through this edge case, you initialize all shortest paths to infinity and the start node’s shortest path to zero. In addition, you set all nodes to unsolved. This initial state will make the program get through the edge case correctly by returning the source node (since it is unsolveed and it has a value of zero which is less than infinity).

**Part 6 – Update Neighbors Method:**

|  |
| --- |
| **public** **static** **void** updateNeighborPaths(**int** solvedVertex, **int**[][] graph, **int**[] shortestPaths){  **for** (**int** v = 0; v < *NUM\_OF\_VERTICES*; v++){ // for every vertex v  **if**(graph[solvedVertex][v] != 0){ //if this vertex is a neighbor  **int** alternatePath = shortestPaths[solvedVertex] + graph[v][solvedVertex];  **if**(alternatePath < shortestPaths[v]){  shortestPaths[v] = alternatePath;  }  }  }  } |

**Part 7 – Getting the Shortest Paths from Source to All Vertices:**

|  |
| --- |
| **public** **static** **int**[] getAllShortestPaths(**int** graph[][], **int** sourceVertex){  **int**[] shortestPaths = **new** **int**[*NUM\_OF\_VERTICES*];  Boolean[] verticesSolved = **new** Boolean[*NUM\_OF\_VERTICES*];  *initialize*(shortestPaths, verticesSolved, sourceVertex);  **int** currentVertex;  **while**(!*allVerticesSolved*(verticesSolved)){  currentVertex = *getNextVertexToSolve*(shortestPaths, verticesSolved);  verticesSolved[currentVertex] = **true**;  *updateShortestPaths*(currentVertex, graph, shortestPaths);  }  *printSolution*(shortestPaths);  **return** shortestPaths;  } |

**Part 8 - Getting the Shortest Path from Source Vertex to Destination Vertex:**

|  |
| --- |
| **public** **static** **int** getShortestPath(**int** graph[][], **int** sourceVertex, **int** destinationVertex){  **int**[] shortestPaths = **new** **int**[*NUM\_OF\_VERTICES*];  Boolean[] verticesSolved = **new** Boolean[*NUM\_OF\_VERTICES*];  *initialize*(shortestPaths, verticesSolved, sourceVertex);  **int** currentVertex;  **while**(!*allVerticesSolved*(verticesSolved)){  currentVertex = *getNextVertexToSolve*(shortestPaths, verticesSolved);  verticesSolved[currentVertex] = **true**;  *updateShortestPaths*(currentVertex, graph, shortestPaths);  **if**(currentVertex == destinationVertex){  **return** shortestPaths[currentVertex];  }  }  **return** -1;  } |

* Remember that in “update shortest paths” you prepare the shortest paths array for the upcoming “get next vertex to Solve” call. In the first iteration, you prepare the shortest path yourself in initialization.

**Part 9 - Utility Methods:**

|  |
| --- |
| **public** **static** **boolean** allVerticesSolved(Boolean[] solvedVertex){  **for**(Boolean vertexSolved : solvedVertex){  **if**(vertexSolved == **false**){  **return** **false**;  }  }  **return** **true**;  } |

**Part 10 – Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Method** | **Time Complexity** | **Reason** |
| Initialize() | O(V) | You have to iterate through the “shortest paths” and “vertex solved” arrays to initialize their values to infinitiy and false respectively. Both these arrays are of size V. |
| allVerticesSolved() | O(V) | You have to iterate through the “vertex solved” array to check if there are any unsolved vertices. This array is of size V. |
| getNextVertexToSolve() | O(V) | You have to iterate through all the vertices to find which ones are not solved. There are V vertices. |
| updateNeighborPaths() | O(V) | You have to iterate through all the vertices and check to see which vertices are neighbours. There are V vertices. |
| getAllShortestPaths() | O(V2) | The while loop including the while condition costs you O(3 V). However, each vertex must be solved. So you go through the while loop V times. |
| getShortestPath() | O(V2) | See “get all shortest paths” explanation. Assuming the destination node is halfway deep in the graph it will still be the same time complexity as “get all shortest paths”. |