**Dijkstra’s shortest path algorithm**

**Dijkstra's algorithm works on undirected, connected, weighted graphs.**

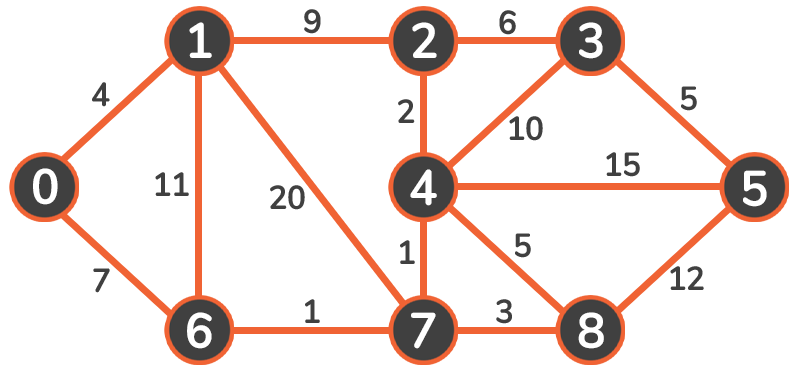
In the beginning, we'll want to create a set of visited vertices, to keep track of all of the vertices that have been assigned their correct shortest path. We will also need to set "costs" of all vertices in the graph (lengths of the current shortest path that leads to it).

All of the costs will be set to ***'infinity'*** at the beginning, to make sure that every other cost we may compare it to would be smaller than the starting one. The only exception is the cost of the first, starting vertex - this vertex will have a cost of 0, because it has no path to itself - marked as node s.

Then, we repeat two main steps until the graph is traversed (as long as there are vertices without the shortest path assigned to them):

* We pick a vertex with the shortest current cost, visit it, and add it to the visited vertices set.
* We update the costs of all its adjacent vertices that are not visited yet. For every edge between n and m where m is unvisited - if the cheapestPath(s, n) + cheapestPath(n,m) < cheapestPath(s,m), update the cheapest path between s and m to equal cheapestPath(s,n) + cheapestPath(n,m).

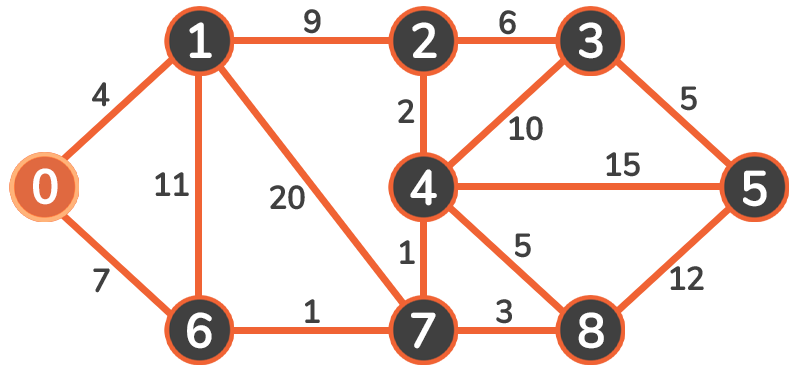
Since this might be a bit difficult to wrap one's head around - let's visualize the process before implementing it! Here's an undirected, weighted, connected graph:



Let's say that *Vertex 0* is our starting point. We are going to set the initial costs of vertices in this graph to infinity, except for the starting vertex:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | inf | inf | inf | inf | inf | inf | inf | inf |

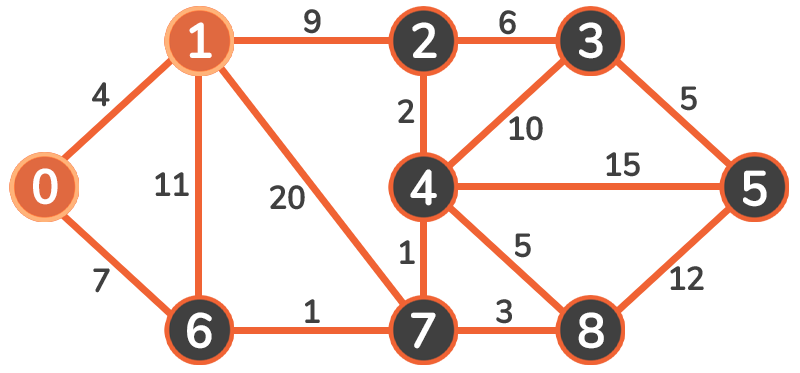
We pick the vertex with a minimum cost - that is *Vertex 0*. We will mark it as visited and add it to our set of visited vertices. The starting node will *always* have the lowest cost so it will always be the first one to be added:



Then, we will update the cost of adjacent vertices (1 and 6). Since 0 + 4 < infinity and 0 + 7 < infinity, we update the costs to these vertices:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | 4 | inf | inf | inf | inf | 7 | inf | inf |

Now we visit the next smallest cost vertex. The weight of *4* is lower than *7* so we traverse to *Vertex 1*:



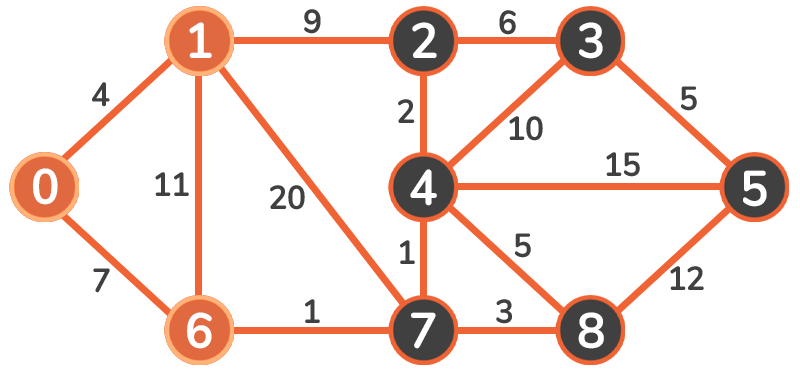
Upon traversing, we mark it as visited, and then observe and update the adjacent vertices: 2, 6, and 7:

* Since 4 + 9 < infinity, new cost of vertex 2 will be 13
* Since 4 + 11 > 7, the cost of vertex 6 will remain 7
* Since 4 + 20 < infinity, new cost of vertex 7 will be 24

These are our new costs:

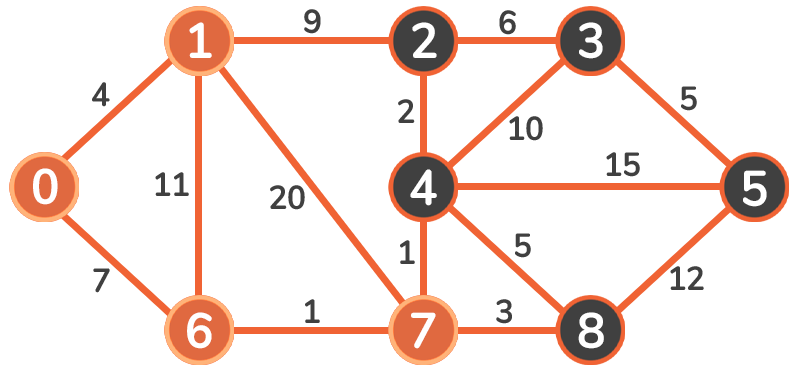
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | 4 | 13 | inf | inf | inf | 7 | 24 | inf |

The next vertex we're going to visit is *Vertex 6*. We mark it as visited and update its adjacent vertices' costs:



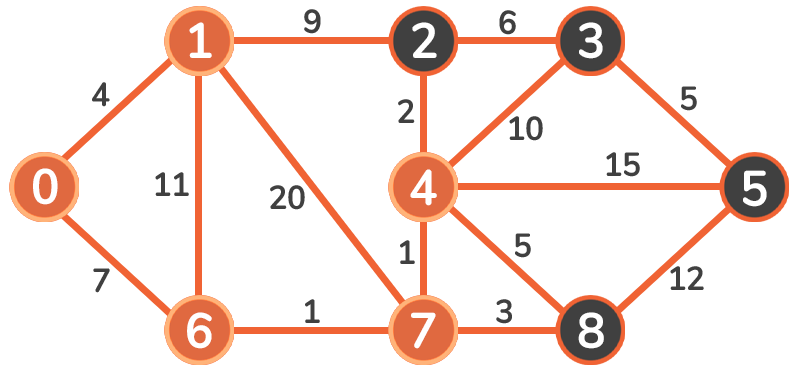
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | 4 | 13 | inf | inf | inf | 7 | 8 | inf |

The process is continued to *Vertex 7*:



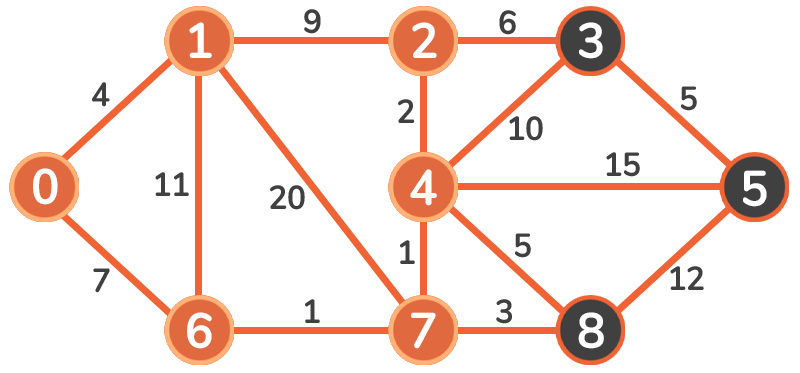
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | 4 | 13 | inf | 9 | inf | 7 | 8 | 11 |

And again, to *Vertex 4*:



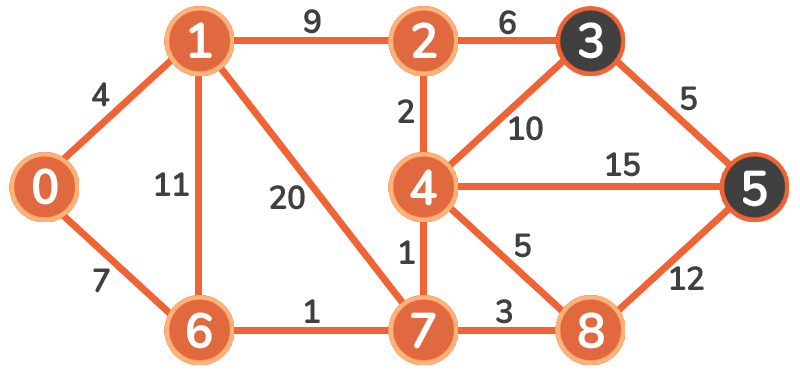
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | 4 | 11 | 19 | 9 | 24 | 7 | 8 | 11 |

And again, to *Vertex 2*:



The only vertex we're going to consider is *Vertex 3*. Since 11 + 6 < 19, the cost of vertex 3 is updated.

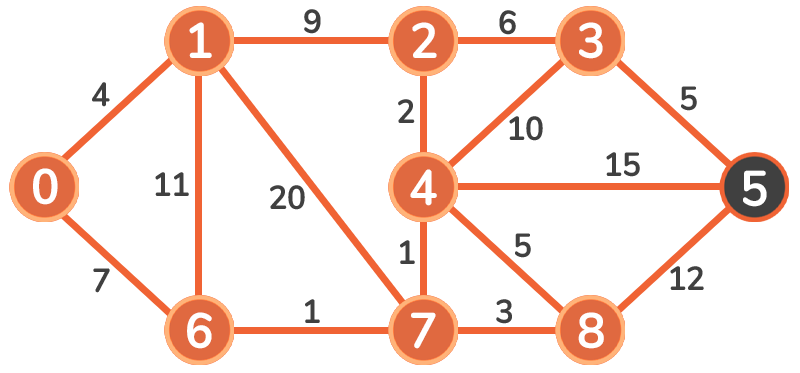
Then, we proceed to *Vertex 8*:



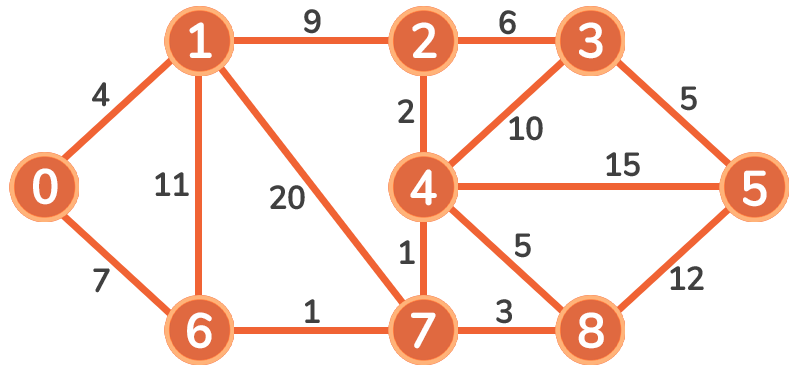
Finally, we're updating the *Vertex 5*:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | 4 | 11 | 17 | 9 | 24 | 7 | 8 | 11 |

We've updated the vertices in the loop-like structure in the end - so now we just have to traverse it - first to *Vertex 3*:



And finally, to the *Vertex 5*:



There are no more unvisited vertices that may need an update. Our final costs represent the shortest paths from node 0 to every other node in the graph:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Cost | 0 | 4 | 11 | 17 | 9 | 24 | 7 | 8 | 11 |

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/DijkstrasShortestPath.java>

**Dijkstra’s Algorithm using Adjacency List Representation**  
  
**Following are the detailed steps.**1) Create a priority queue of size V where V is the number of vertices in the given graph

2) Initialize priority queue with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).   
3) While Min Heap is not empty, do following   
 a) Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.

b) For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of u-v plus distance value of u, then update the distance value of v.

**Time Complexity:**  overall time complexity is O(E+V)\*O(LogV) which is O((E+V)\*LogV) = O(ELogV)   
Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) time.  
**Notes:**

1. The code calculates shortest distance, but doesn’t calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim’s implementation](https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/)) and use it to show the shortest path from source to different vertices.
2. The code is for undirected graph, same dijekstra function can be used for directed graphs also.
3. The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
4. Dijkstra’s algorithm doesn’t work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](http://en.wikipedia.org/wiki/Bellman-Ford_algorithm) can be used

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/DijkstrasAdjacencyListRepresenation.java>

**Bellman–Ford Algorithm**

Given a graph and a source vertex *src*in graph, find shortest paths from *src*to all vertices in the given graph. The graph may contain negative weight edges.   
*Time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*   
   
***How does this work?***

There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

**Example**   
Let us understand the algorithm with following example graph. Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

Bellman–Ford Algorithm Example Graph 1

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed. 

Bellman–Ford Algorithm Example Graph 2

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values). 

Bellman–Ford Algorithm Example Graph 3

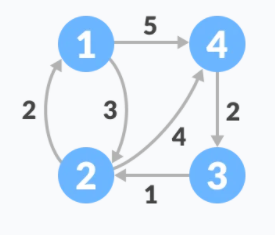
The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

**Notes**   
**1)**Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.  
**2)** Bellman-Ford works better (better than Dijkstra’s) for distributed systems. Unlike Dijkstra’s where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.                                                                    
**3)**Bellman-Ford does not work with undirected graph with negative edges as it will declare as negative cycle.

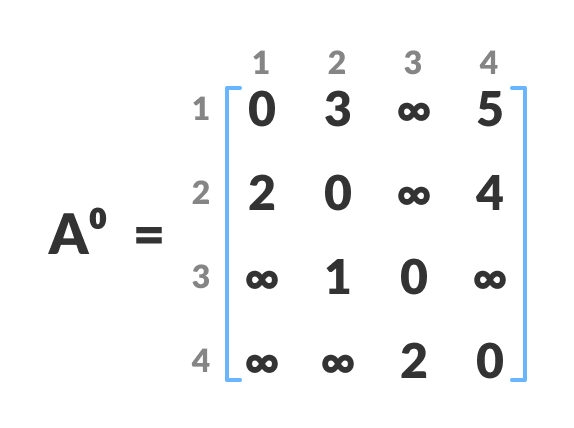
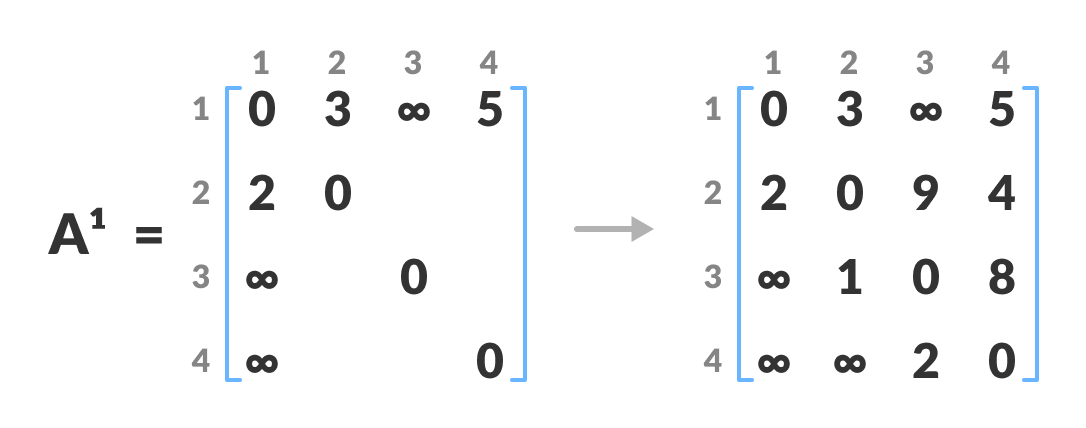
<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/BellmanFordAlgorithm.java>

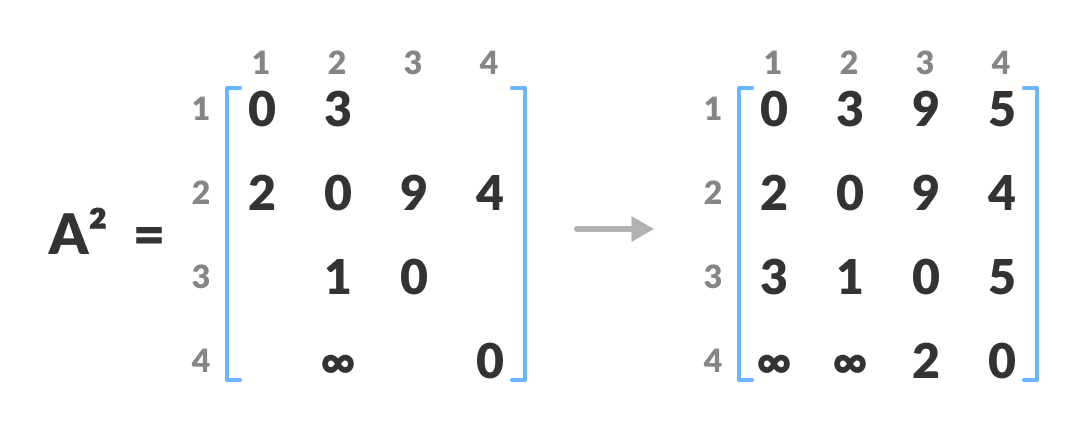
**Floyd Warshall Algorithm**

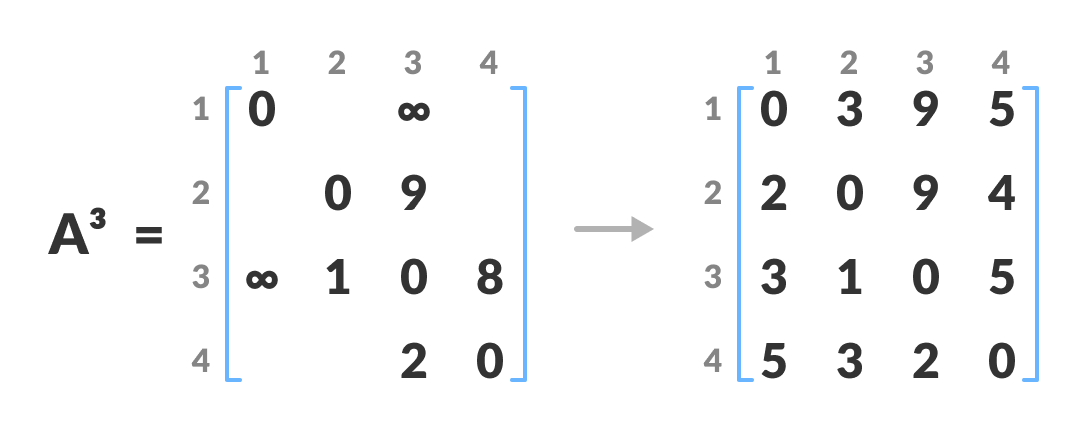
The [Floyd Warshall Algorithm](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.   
**Example:**

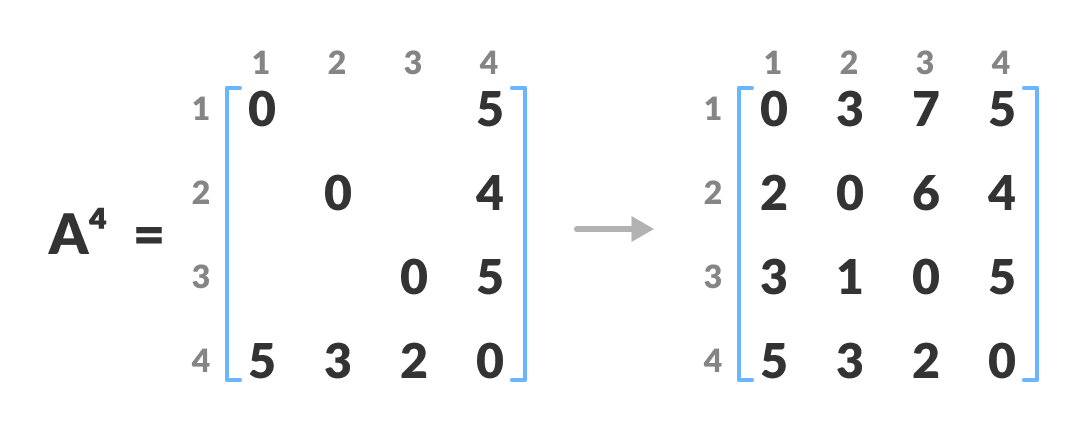


Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A0 of dimension n\*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.  
     
   Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.
2. Fill each cell with the distance between ith and jth vertex
3. Now, create a matrix A1 using matrix A0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.  
     
   Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).  
     
   That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].  
     
   In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.Calculate the distance from the source vertex to destination vertex through this vertex k  
   For example: For A1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4.
4. Similarly, A2 is created using A1. The elements in the second column and the second row are left as they are.  
     
   In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.



1. Calculate the distance from the source vertex to destination vertex through this vertex 2
2. Similarly, A3 and A4 is also created.Calculate the distance from the source vertex to destination vertex through this vertex 3



Calculate the distance from the source vertex to destination vertex through this vertex 4

1. A4 gives the shortest path between each pair of vertices.

The following matrix shows the shortest distances between every pair of vertices

0 3 7 5

2 0 6 4

3 1 0 5

5 3 2 0

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/FloyedWarshallAlgorithm.java>

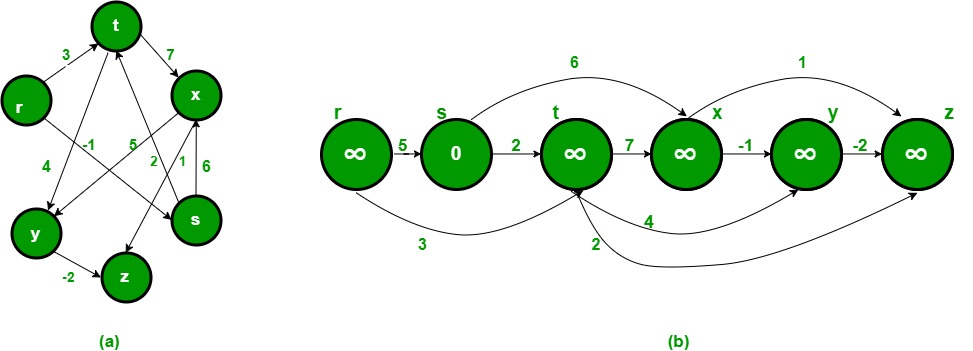
**Shortest Path in Directed Acyclic Graph**

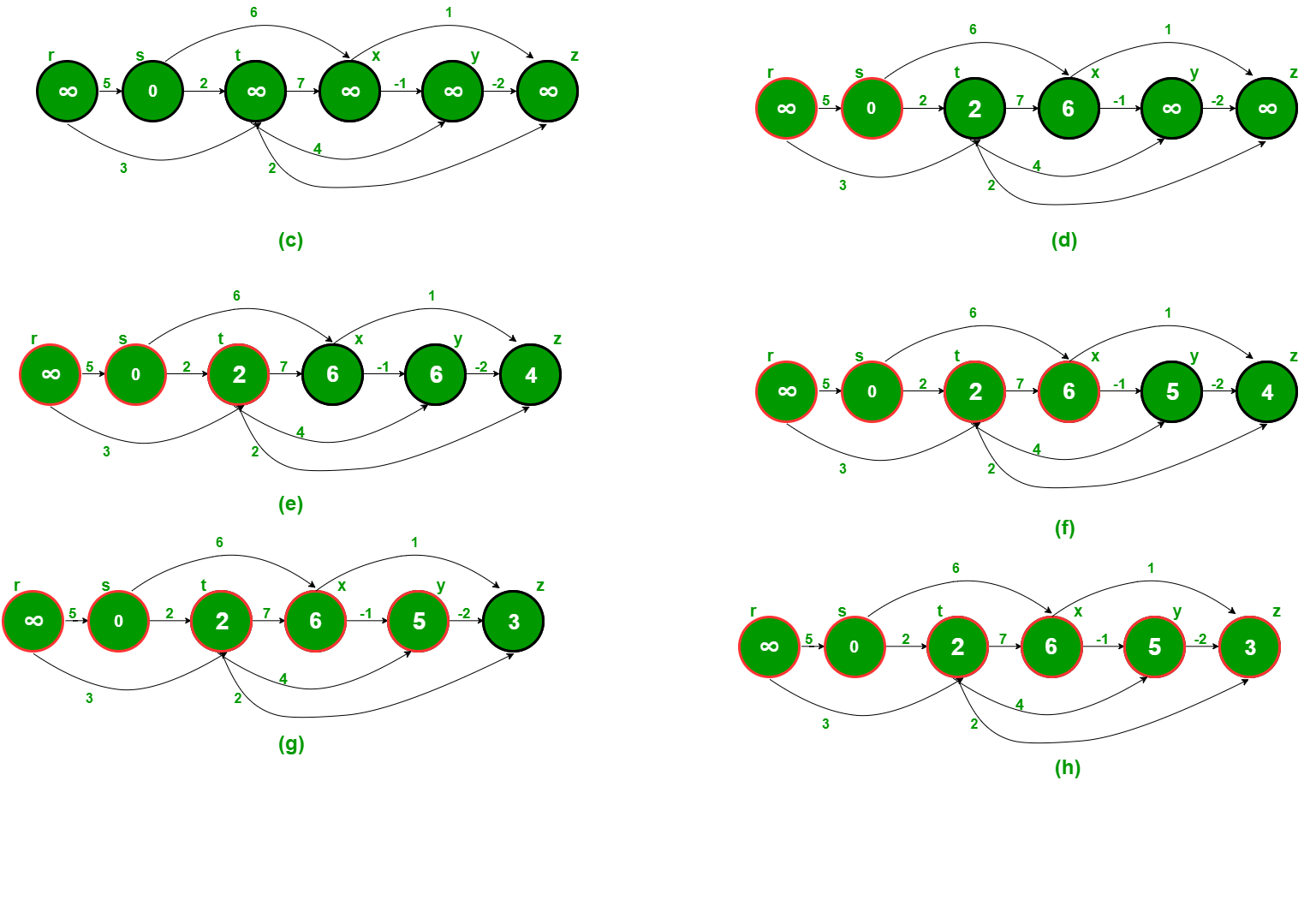
**Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.**

For a general weighted graph, we can calculate single source shortest distances in O(VE) time using [Bellman–Ford Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/). For a graph with no negative weights, we can do better and calculate single source shortest distances in O(E + VLogV) time using [Dijkstra’s algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/)**. Can we do even better for Directed Acyclic Graph (DAG)?**

We can calculate single source shortest distances in O(V+E) time for DAGs. The idea is to use [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/).

We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/) of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a) ). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.  
Following figure is taken from [this](http://www.utdallas.edu/~sizheng/CS4349.d/l-notes.d/L17.pdf)source. It shows step by step process of finding shortest paths. 





Following is complete algorithm for finding shortest distances.   
**1)** Initialize dist[] = {INF, INF, ….} and dist[s] = 0 where s is the source vertex.   
**2)** Create a topological order of all vertices.   
**3)**Do following for every vertex u in topological order.   
………..Do following for every adjacent vertex v of u   
………………if (dist[v] > dist[u] + weight(u, v))   
………………………dist[v] = dist[u] + weight(u, v)

**Time Complexity:** Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, overall time complexity of this algorithm is O(V+E).

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/ShortestPathInDirectedACyclicGraph.java>

**Printing Paths in Dijkstra’s Shortest Path Algorithm**

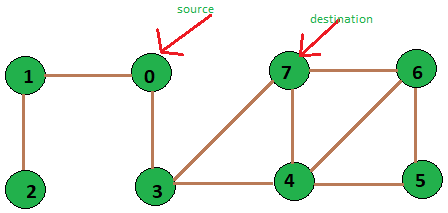
The idea is to create a separate array parent[]. Value of parent[v] for a vertex v stores parent vertex of v in shortest path tree. Parent of root (or source vertex) is -1. Whenever we find shorter path through a vertex u, we make u as parent of current vertex.

Once we have parent array constructed, we can print path using below recursive function.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/PrintAllDijkstrahShortestPath.java>

**Shortest path in an unweighted graph**

**Given an unweighted graph, a source, and a destination, we need to find the shortest path from source to destination in the graph in the most optimal way.**

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/exampleFigure-1.png)

*unweighted graph of 8 vertices*

Input: source vertex = 0 and destination vertex is = 7.

Output: Shortest path length is:2

Path is::

0 3 7

Input: source vertex is = 2 and destination vertex is = 6.

Output: Shortest path length is:5

Path is::

2 1 0 3 4 6

Since the graph is unweighted, we can solve this problem in O(V + E) time. The idea is to use a modified version of [Breadth-first search](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) in which we keep storing the predecessor of a given vertex while doing the breadth-first search.

We first initialize an array dist[0, 1, …., v-1] such that dist[i] stores the distance of vertex i from the source vertex and array pred[0, 1, ….., v-1] such that pred[i] represents the immediate predecessor of the vertex i in the breadth-first search starting from the source.

Now we get the length of the path from source to any other vertex in O(1) time from array d, and for printing the path from source to any vertex we can use array p and that will take O(V) time in worst case as V is the size of array P. So most of the time of the algorithm is spent in doing the Breadth-first search from a given source which we know takes O(V+E) time. Thus the time complexity of our algorithm is O(V+E).   
Take the following unweighted graph as an example:

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/shortestpath/ShortestPathInUnweightedGraph.java>