**Dijkstra’s Algorithm:**

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. 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.

**// A C++ program for Dijkstra's single source shortest path algorithm.**

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

**#include <stdio.h>**

**#include <limits.h>**

**// Number of vertices in the graph**

**#define V 9**

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

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

**int minDistance(int dist[], bool sptSet[])**

**{**

**// Initialize min value**

**int min = INT\_MAX, min\_index;**

**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**

**int printSolution(int dist[], int n)**

**{**

**printf("Vertex Distance from Source\n");**

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

**printf("%d tt %d\n", i, dist[i]);**

**}**

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

**// for a graph represented using adjacency matrix representation**

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

**{**

**int dist[V]; // The output array. dist[i] will hold the shortest**

**// distance from src to i**

**bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest**

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

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

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

**dist[i] = INT\_MAX, 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 the 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] && dist[u] != INT\_MAX**

**&& dist[u]+graph[u][v] < dist[v])**

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

**}**

**// print the constructed distance array**

**printSolution(dist, V);**

**}**

**// driver program to test above function**

**int main()**

**{**

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

**int graph[V][V] = {{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}**

**};**

**dijkstra(graph, 0);**

**return 0;**

**}**

**Dijkstra’s Implementation Using Min Heap:**

Following are the detailed steps.

1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.

2) Initialize Min Heap 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).  
(we have all of them in min heap. Source nodes’ distance is updated to 0, all other nodes’ distance field are initially set to INF)

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.

**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.

We have discussed Dijkstra’s algorithm for this problem. Dijksra’s algorithm is a Greedy algorithm and time complexity is O(ELogV) (with the use of Fibonacci heap).

**Advantages:**

Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs. (if their is a negative edge, Bellman Ford can detect it. It wont calculate the shortest path in that case) Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems.

**Disadvantages:**But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

**Input:** Graph and a source vertex src

**Output:** Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

…..a) Do following for each edge u-v

………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]

………………….dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

**// A C++ program for Bellman-Ford's single source**

**// shortest path algorithm.**

**#include <bits/stdc++.h>**

**// a structure to represent a weighted edge in graph**

**struct Edge**

**{**

**int src, dest, weight;**

**};**

**// a structure to represent a connected, directed and**

**// weighted graph**

**struct Graph**

**{**

**// V-> Number of vertices, E-> Number of edges**

**int V, E;**

**// graph is represented as an array of edges.**

**struct Edge\* edge;**

**};**

**// Creates a graph with V vertices and E edges**

**struct Graph\* createGraph(int V, int E)**

**{**

**struct Graph\* graph = new Graph;**

**graph->V = V;**

**graph->E = E;**

**graph->edge = new Edge[E];**

**return graph;**

**}**

**// A utility function used to print the solution**

**void printArr(int dist[], int n)**

**{**

**printf("Vertex Distance from Source\n");**

**for (int i = 0; i < n; ++i)**

**printf("%d \t\t %d\n", i, dist[i]);**

**}**

**// The main function that finds shortest distances from src to**

**// all other vertices using Bellman-Ford algorithm. The function**

**// also detects negative weight cycle**

**void BellmanFord(struct Graph\* graph, int src)**

**{**

**int V = graph->V;**

**int E = graph->E;**

**int dist[V];**

**// Step 1: Initialize distances from src to all other vertices**

**// as INFINITE**

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

**dist[i] = INT\_MAX;**

**dist[src] = 0;**

**// Step 2: Relax all edges |V| - 1 times. A simple shortest**

**// path from src to any other vertex can have at-most |V| - 1**

**// edges**

**for (int i = 1; i <= V-1; i++)**

**{**

**for (int j = 0; j < E; j++)**

**{**

**int u = graph->edge[j].src;**

**int v = graph->edge[j].dest;**

**int weight = graph->edge[j].weight;**

**if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])**

**dist[v] = dist[u] + weight;**

**}**

**}**

**// Step 3: check for negative-weight cycles. The above step**

**// guarantees shortest distances if graph doesn't contain**

**// negative weight cycle. If we get a shorter path, then there**

**// is a cycle.**

**for (int i = 0; i < E; i++)**

**{**

**int u = graph->edge[i].src;**

**int v = graph->edge[i].dest;**

**int weight = graph->edge[i].weight;**

**if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])**

**printf("Graph contains negative weight cycle");**

**}**

**printArr(dist, V);**

**return;**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**/\* Let us create the graph given in above example \*/**

**int V = 5; // Number of vertices in graph**

**int E = 8; // Number of edges in graph**

**struct Graph\* graph = createGraph(V, E);**

**// add edge 0-1 (or A-B in above figure)**

**graph->edge[0].src = 0;**

**graph->edge[0].dest = 1;**

**graph->edge[0].weight = -1;**

**// add edge 0-2 (or A-C in above figure)**

**graph->edge[1].src = 0;**

**graph->edge[1].dest = 2;**

**graph->edge[1].weight = 4;**

**// add edge 1-2 (or B-C in above figure)**

**graph->edge[2].src = 1;**

**graph->edge[2].dest = 2;**

**graph->edge[2].weight = 3;**

**// add edge 1-3 (or B-D in above figure)**

**graph->edge[3].src = 1;**

**graph->edge[3].dest = 3;**

**graph->edge[3].weight = 2;**

**// add edge 1-4 (or A-E in above figure)**

**graph->edge[4].src = 1;**

**graph->edge[4].dest = 4;**

**graph->edge[4].weight = 2;**

**// add edge 3-2 (or D-C in above figure)**

**graph->edge[5].src = 3;**

**graph->edge[5].dest = 2;**

**graph->edge[5].weight = 5;**

**// add edge 3-1 (or D-B in above figure)**

**graph->edge[6].src = 3;**

**graph->edge[6].dest = 1;**

**graph->edge[6].weight = 1;**

**// add edge 4-3 (or E-D in above figure)**

**graph->edge[7].src = 4;**

**graph->edge[7].dest = 3;**

**graph->edge[7].weight = -3;**

**BellmanFord(graph, 0);**

**return 0;**

**}**

Now, note the following things:

**Calculate shortest path from source to other vertices take O(VE)**

for (int i = 1; i <= V-1; i++)

{

for (int j = 0; j < E; j++)

{

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

}

}

**Now, detecting the presence of negative edge weight cycle takes O( E )**for (int i = 0; i < E; i++)

{

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

printf("Graph contains negative weight cycle");

}

**Floyd Warshall Algorithm**The Floyd Warshall 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.

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

# 0-1 BFS (Shortest Path in a Binary Weight Graph)

Given a graph where every edge has weight as either 0 or 1. A source vertex is also given in the graph. Find the shortest path from source vertex to every other vertex.

In normal BFS of a graph all edges have equal weight but in 0-1 BFS some edges may have 0 weight and some may have 1 weight. In this we will not use bool array to mark visited nodes but at each step we will check for the optimal distance condition. We use double ended queue to store the node. While performing BFS if a edge having weight = 0 is found node is pushed at front of double ended queue and if a edge having weight = 1 is found, it is pushed at back of double ended queue.