**Shortest Path Algorithms**

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum. This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms are discussed below depending on the use-case.

**Bellman Ford's Algorithm:**

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n−1 edges, because the shortest path couldn't have a cycle. So why shortest path shouldn't have a cycle ?  
There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

**Algorithm Steps:**   
The outer loop traverses from 0: n−1  
Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Following are the detailed steps.

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.

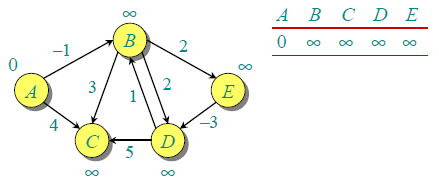
**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.  
…..**a)** Do following for each edge u-v  
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]  
………………….dist[v] = dist[u] + weight of edge uv

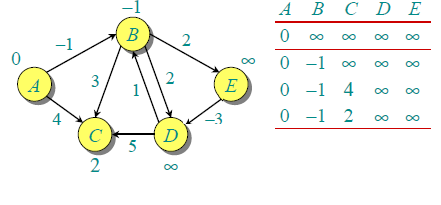
**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v  
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”  
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

***How does this work?*** Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates shortest paths with at-nost 2 edges, and so on. After the i-th iteration of outer loop, the shortest paths with at most i edges are calculated. 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

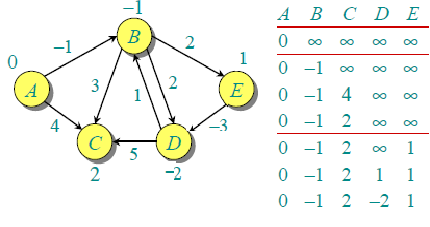
**Example**  
Let us understand the algorithm with following example graph. The images are taken from [this](http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf) source.

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

[](https://www.geeksforgeeks.org/wp-content/uploads/bellman2.png)

Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in 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.  
[](https://www.geeksforgeeks.org/wp-content/uploads/After1stIteration.png)

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

[](https://www.geeksforgeeks.org/wp-content/uploads/seconditeration2.png)

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 Dijksra’s) for distributed systems. Unlike Dijksra’s where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

// 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;

}

Below is algorithm find if there is a negative weight cycle reachable from given source.

**1)** Initialize 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.

// The main function that finds shortest distances

// from src to all other vertices using Bellman-

// Ford algorithm.  The function also detects

// negative weight cycle

bool isNegCycleBellmanFord(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])

            return true;

    }

    return false;

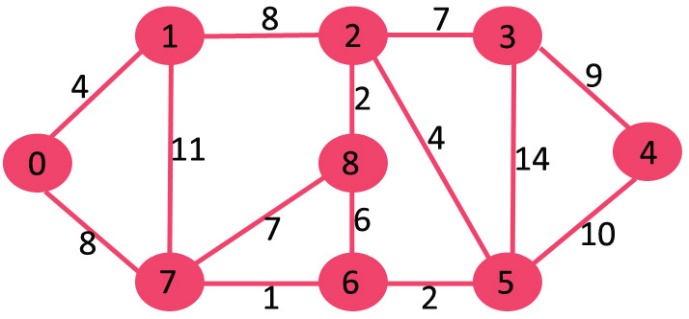
}

**(Dijkstra’s shortest path algorithm)**

Given a graph and a source vertex in 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/archives/27455). 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 minimum distance from 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 color.

[](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* doesn’t include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

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

// Funtion 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 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);

}

**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/archives/27455)) and use it show the shortest path from source to different vertices.

**2)** The code is for undirected graph, same dijkstra function can be used for directed graphs also.

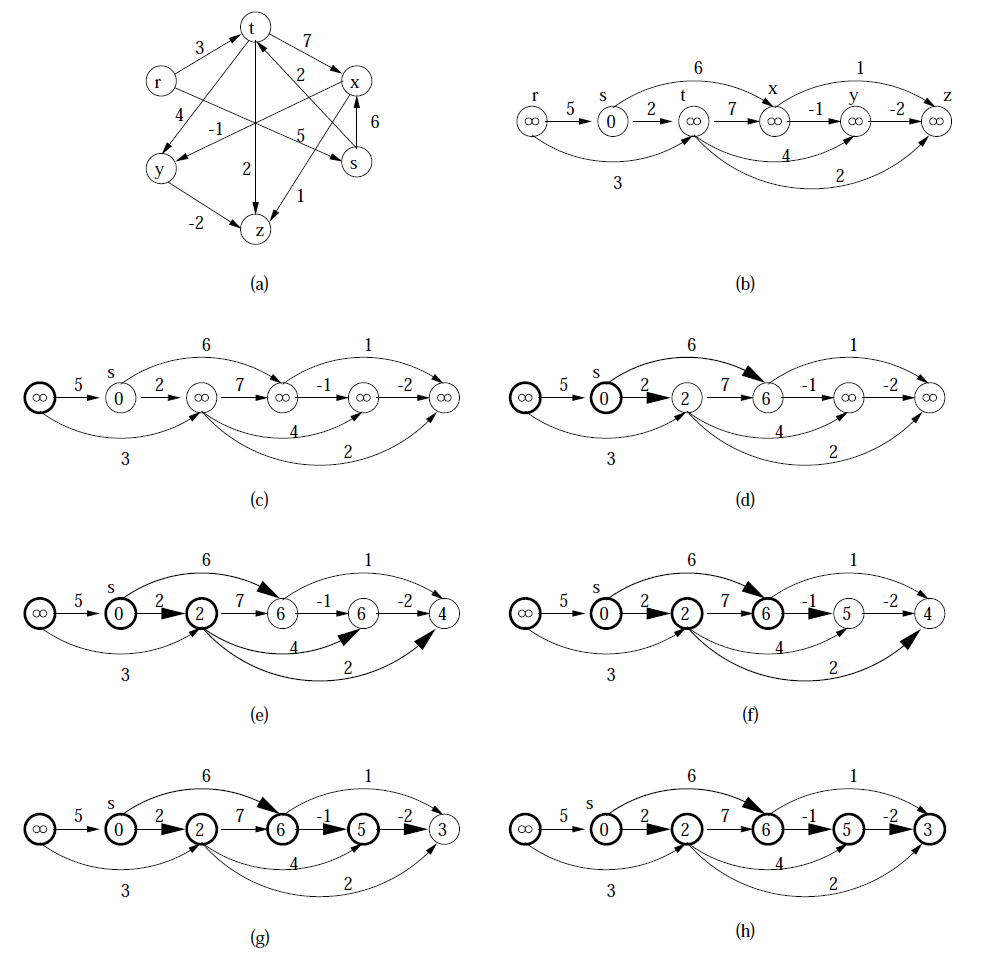
**3)** The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).

**4)** Time Complexity of the implementation is O(V^2). If the input [graph is represented using adjacency list](https://www.geeksforgeeks.org/archives/27134), it can be reduced to O(E log V) with the help of binary heap. Please see  
[Dijkstra’s Algorithm for Adjacency List Representation](https://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/) for more details.

**5)** 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, we will soon be discussing it as a separate post.

**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.  
[](https://www.geeksforgeeks.org/wp-content/uploads/TopologicalSort.png)

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 toplogical 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)

// The function to find shortest paths from given vertex. It uses recursive

// topologicalSortUtil() to get topological sorting of given graph.

void Graph::shortestPath(int s)

{

    stack<int> Stack;

    int dist[V];

    // Mark all the vertices as not visited

    bool \*visited = new bool[V];

    for (int i = 0; i < V; i++)

        visited[i] = false;

    // Call the recursive helper function to store Topological Sort

    // starting from all vertices one by one

    for (int i = 0; i < V; i++)

        if (visited[i] == false)

            topologicalSortUtil(i, visited, Stack);

    // Initialize distances to all vertices as infinite and distance

    // to source as 0

    for (int i = 0; i < V; i++)

        dist[i] = INF;

    dist[s] = 0;

    // Process vertices in topological order

    while (Stack.empty() == false)

    {

        // Get the next vertex from topological order

        int u = Stack.top();

        Stack.pop();

        // Update distances of all adjacent vertices

        list<AdjListNode>::iterator i;

        if (dist[u] != INF)

        {

          for (i = adj[u].begin(); i != adj[u].end(); ++i)

             if (dist[i->getV()] > dist[u] + i->getWeight())

                dist[i->getV()] = dist[u] + i->getWeight();

        }

    }

    // Print the calculated shortest distances

    for (int i = 0; i < V; i++)

        (dist[i] == INF)? cout << "INF ": cout << dist[i] << " ";

}

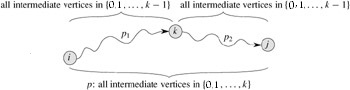
**Time Complexity:** Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, overall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

**(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.

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].

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.

[](https://www.geeksforgeeks.org/wp-content/uploads/Floyd-Warshell1.jpg)

**The Algorithm Steps:**

For a graph with N

vertices:

* Initialize the shortest paths between any 2

 vertices with Infinity.

 Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N

 vertices as intermediate nodes.

 Minimize the shortest paths between any 2

 pairs in the previous operation.

 For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j])

 dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j

for(int k = 1; k <= n; k++){

for(int i = 1; i <= n; i++){

for(int j = 1; j <= n; j++){

dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );

}

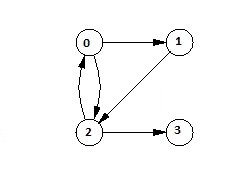
}

}

**Transitive closure of a graph**

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called transitive closure of a graph.

For example, consider below graph

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

Transitive closure of above graphs is

1 1 1 1

1 1 1 1

1 1 1 1

0 0 0 1

The graph is given in the form of adjacency matrix say ‘graph[V][V]’ where graph[i][j] is 1 if there is an edge from vertex i to vertex j or i is equal to j, otherwise graph[i][j] is 0.

[Floyd Warshall Algorithm](https://www.geeksforgeeks.org/archives/19772) can be used, we can calculate the distance matrix dist[V][V] using [Floyd Warshall](https://www.geeksforgeeks.org/archives/19772), if dist[i][j] is infinite, then j is not reachable from i, otherwise j is reachable and value of dist[i][j] will be less than V.  
Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:

**1)** Instead of integer resultant matrix ([dist[V][V] in floyd warshall](https://www.geeksforgeeks.org/archives/19772)), we can create a boolean reach-ability matrix reach[V][V] (we save space). The value reach[i][j] will be 1 if j is reachable from i, otherwise 0.

**2)** Instead of using arithmetic operations, we can use logical operations. For arithmetic operation ‘+’, logical and ‘&&’ is used, and for min, logical or ‘||’ is used. (We save time by a constant factor. Time complexity is same though)

**Time Complexity:** O(V3) where V is number of vertices in the given graph. A O(V2) solution “Transitive Closure of a Graph using DFS”.

**Johnson’s algorithm for All-pairs shortest paths**

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. We have discussed [Floyd Warshall Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/) for this problem. Time complexity of Floyd Warshall Algorithm is Θ(V3). *Using Johnson’s algorithm, we can find all pair shortest paths in O(V2log V + VE) time.* Johnson’s algorithm uses both [Dijkstra](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/) and [Bellman-Ford](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/) as subroutines.

If we apply [Dijkstra’s Single Source shortest path algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/) for every vertex, considering every vertex as source, we can find all pair shortest paths in O(V\*VLogV) time. So using Dijkstra’s single source shortest path seems to be a better option than [Floyd Warshell](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/), but the problem with Dijkstra’s algorithm is, it doesn’t work for negative weight edge.  
*The idea of Johnson’s algorithm is to re-weight all edges and make them all positive, then apply Dijkstra’s algorithm for every vertex.*

**How to transform a given graph to a graph with all non-negative weight edges?**  
One may think of a simple approach of finding the minimum weight edge and adding this weight to all edges. Unfortunately, this doesn’t work as there may be different number of edges in different paths (See [this](http://geeksquiz.com/data-structures-graph-question-31/) for an example). If there are multiple paths from a vertex u to v, then all paths must be increased by same amount, so that the shortest path remains the shortest in the transformed graph.  
The idea of Johnson’s algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be h[u]. We reweight edges using vertex weights. For example, for an edge (u, v) of weight w(u, v), the new weight becomes w(u, v) + h[u] – h[v]. The great thing about this reweighting is, all set of paths between any two vertices are increased by same amount and all negative weights become non-negative. Consider any path between two vertices s and t, weight of every path is increased by h[s] – h[t], all h[] values of vertices on path from s to t cancel each other.

How do we calculate h[] values? [Bellman-Ford algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/) is used for this purpose. Following is the complete algorithm. A new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are h[] values.

**Algorithm:**  
**1)** Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G’.

**2)** Run [Bellman-Ford algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/) on G’ with s as source. Let the distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.

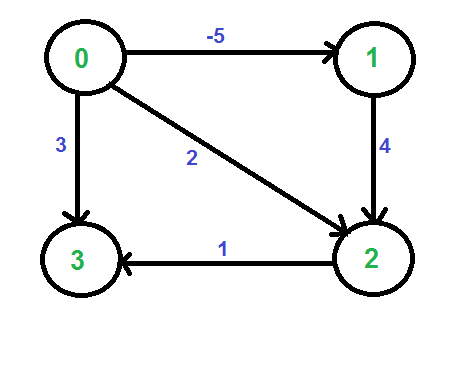
**3)** Reweight the edges of original graph. For each edge (u, v), assign the new weight as “original weight + h[u] – h[v]”.

**4)** Remove the added vertex s and run [Dijkstra’s algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/) for every vertex.

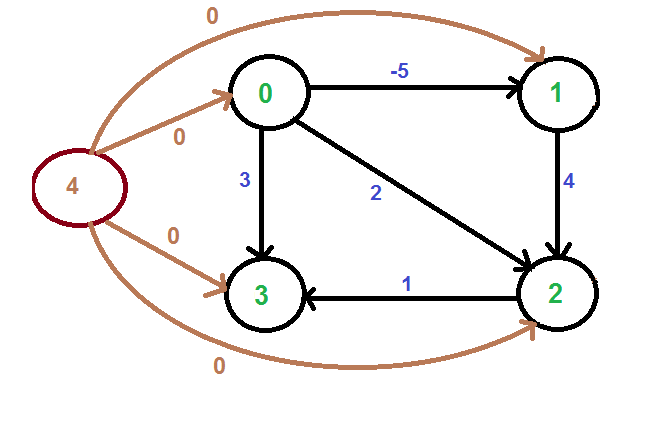
**How does the transformation ensure nonnegative weight edges?**  
The following property is always true about h[] values as they are shortest distances.

h[v] <= h[u] + w(u, v)

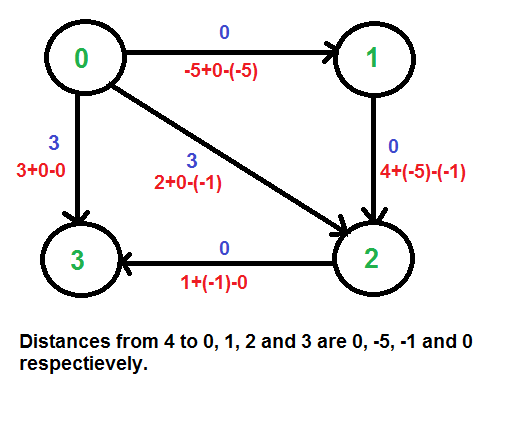
The property simply means, shortest distance from s to v must be smaller than or equal to shortest distance from s to u plus weight of edge (u, v). The new weights are w(u, v) + h[u] - h[v]. The value of the new weights must be greater than or equal to zero because of the inequality "h[v] <= h[u] + w(u, v)". **Example:**  
Let us consider the following graph.

[](https://www.geeksforgeeks.org/wp-content/uploads/Johnson1.png)

We add a source s and add edges from s to all vertices of the original graph. In the following diagram s is 4.

[](https://www.geeksforgeeks.org/wp-content/uploads/Johnson2.png)

We calculate the shortest distances from 4 to all other vertices using Bellman-Ford algorithm. The shortest distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively, i.e., h[] = {0, -5, -1, 0}. Once we get these distances, we remove the source vertex 4 and reweight the edges using following formula. w(u, v) = w(u, v) + h[u] - h[v].

[](https://www.geeksforgeeks.org/wp-content/uploads/Johnson3.png)  
Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as source.  
**Time Complexity:** The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra is O(VLogV). So overall time complexity is O(V2log V + VE).  
The time complexity of Johnson's algorithm becomes same as [Floyd Warshell](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/) when the graphs is complete (For a complete graph E = O(V2). But for sparse graphs, the algorithm performs much better than [Floyd Warshell](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/).