

Johnson's Algorithm

Introduction:

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 for this problem. Time complexity of Floyd Warshall Algorithm is $\Theta(V^3)$. Using Johnson's algorithm, we can find all pair shortest paths in $O(V^2 \log V + VE)$ time. Johnson's algorithm uses both Dijkstra and Bellman-Ford as subroutines.

If we apply Dijkstra's Single Source shortest path algorithm for every vertex, considering every vertex as source, we can find all pair shortest paths in $O(V * V \log V)$ time. So, using Dijkstra's single source shortest path seems to be a better option than Floyd Warshall, but the problem with Dijkstra's algorithm is, it doesn't work for negative weight edge.

Theory & Concept:

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.[1]

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. 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 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:

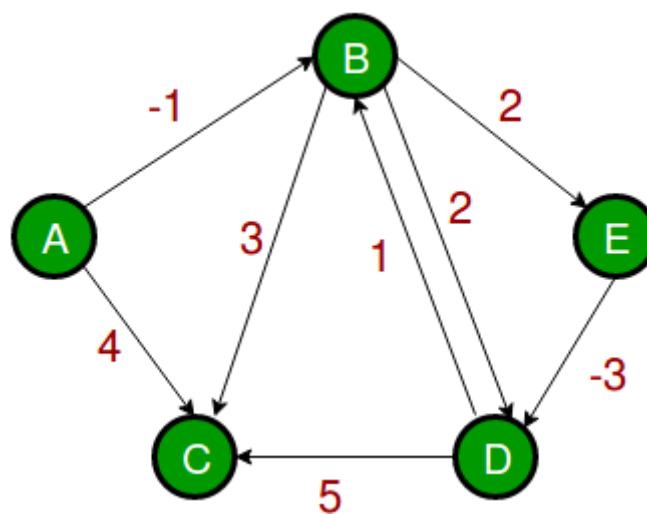
1. Let the given graph be G . Add a new vertex s to the graph, add edges from new vertex to all vertices of G . Let the modified graph be G' .

2. Run Bellman-Ford algorithm on G' with s as source. Let the distances calculated by Bellman-Ford be $h[0], h[1], \dots, h[V-1]$. 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 for every vertex.

The following property is always true about $h[]$ values as they are shortest distances:

$$h[v] \leq h[u] + w(u, v)$$

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 " $h[v] \leq h[u] + w(u, v)$ ".



I have taken the following graph as an example to implement the code below.[2] The stdin is empty.

Code Implemented:

```

/*A C++ program for Johnson's shortest path algorithm.*/
#include <bits/stdc++.h>
using namespace std;
/* 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;

int weight_vertex[10];

// 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+V];

    return graph;

}

// A utility function used to print the solution

void printSolution(int dist[], int n, int src)

{

    cout<<"Vertex \t\t Distance from Vertex "<<src<<endl;

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

    {

        cout<<i<<"\t\t\t";

        (dist[i]==INT_MAX)?cout<<"INF\n":cout<<dist[i]<<endl;

    }

}

/* 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) + 1;

    int E = (graph->E) + V - 1;

    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])
    {
        cout << "Graph contains negative weight cycle.\n";
        return; // If negative cycle is detected, simply return
    }
}

for (int i = 0; i < (V - 1); i++)
{
    graph->weight_vertex[i] = dist[i];
}

for (int i = 0; i < (E - V + 1); i++)

```

```

{
    int u = graph->edge[i].src;
    int v = graph->edge[i].dest;
    int weight = graph->edge[i].weight;
    graph->edge[i].weight = weight + dist[u] - dist[v];
}
return;
}

/* 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[], int V)
{
    // 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;
}

void dijkstra(struct Graph* graph, int src)
{
    int V = (graph->V);
    int E = (graph->E);
    int dist[V]; /* The output array. dist[i] will hold the shortest distance from src to i*/

    bool sptSet[V];

    /*sptSet[i] will be 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, V);
    // Mark the picked vertex as processed
    sptSet[u] = true;
    // Update dist value of the adjacent vertices of the picked vertex.
    for (int i = 0; i < E; i++)
    {
        int s = graph->edge[i].src;
        int e = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        /* Update dist[e] only if is not in sptSet, there is an edge from u to e, and total weight
        of path from src to e through u is smaller than current value of dist[v] */
        if(!sptSet[e] && s==u && dist[u] != INT_MAX && dist[u] + weight < dist[e])
            dist[e] = dist[u] + weight;
    }
}
// print the constructed distance array
printSolution(dist, V, src);
}

// Driver program to test above functions
int main()
{
    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;
```

```
/*A new vertex is added to the graph and connected to all existing vertices with edges from this vertex and have weight as 0*/
```

```
for(int i=8;i<(V+8);i++)
```

```
{
```

```
graph->edge[i].src = 5;
```

```
graph->edge[i].dest = i-8;
```

```
graph->edge[i].weight = 0;
```

```
}
```

```
BellmanFord(graph, 5);
```

```
/* Run Dijkstra's shortest path algorithm for every vertex as source*/
```

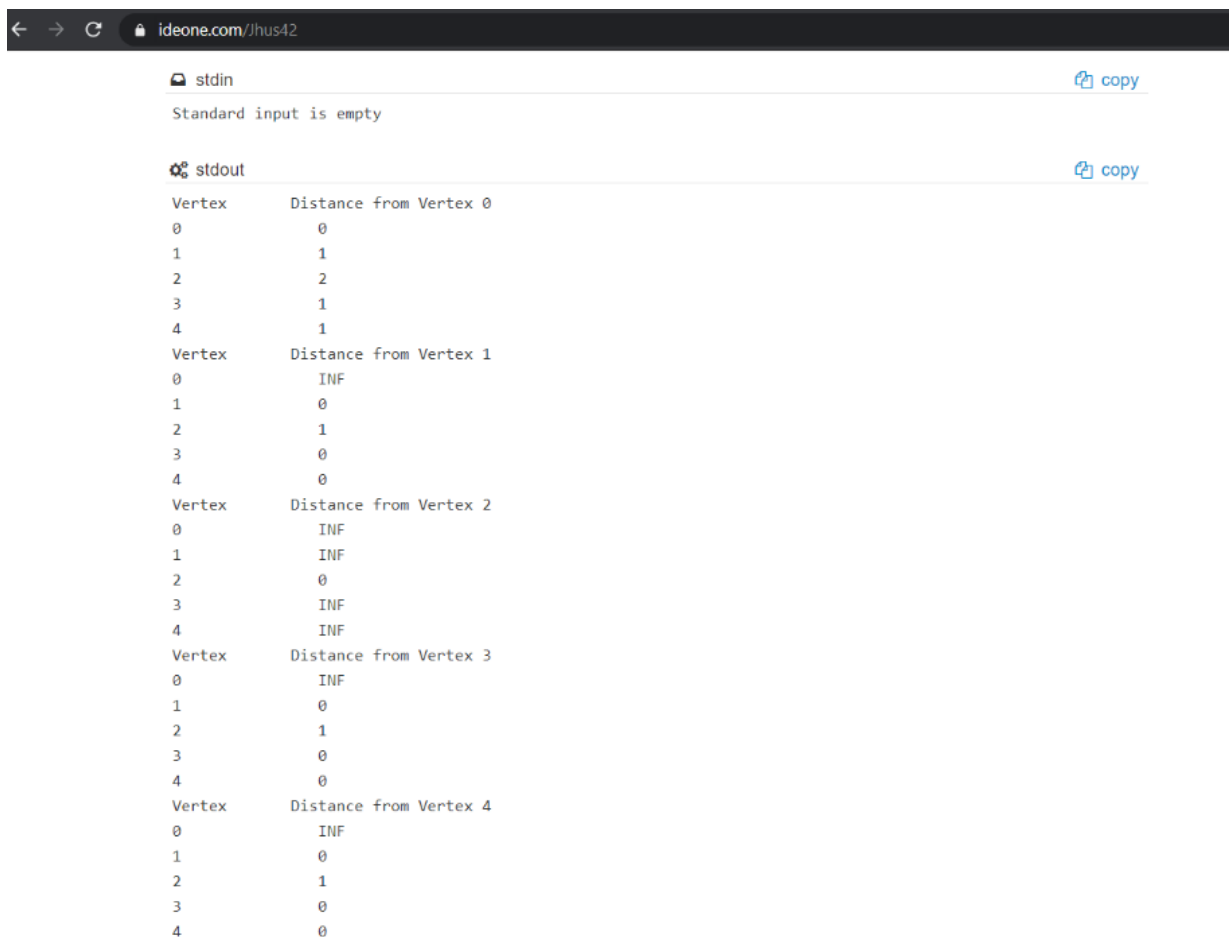
```
for(int i=0; i<V; i++)
```

```
dijkstra(graph, i);
```

```
return 0;
```

```
}
```

Output:



The screenshot shows a web-based IDE interface with a browser address bar at the top displaying "ideone.com/jhus42". Below the browser, there are two main sections: "stdin" and "stdout".

The "stdin" section shows "Standard input is empty" with a "copy" icon to its right.

The "stdout" section displays the program's output, which consists of five tables. Each table shows the distance from a specific vertex (0, 1, 2, 3, and 4) to all other vertices (0, 1, 2, 3, 4). The output uses "INF" to represent infinity.

Vertex	Distance from Vertex 0
0	0
1	1
2	2
3	1
4	1

Vertex	Distance from Vertex 1
0	INF
1	0
2	1
3	0
4	0

Vertex	Distance from Vertex 2
0	INF
1	INF
2	0
3	INF
4	INF

Vertex	Distance from Vertex 3
0	INF
1	0
2	1
3	0
4	0

Vertex	Distance from Vertex 4
0	INF
1	0
2	1
3	0
4	0

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 $O(V\log V)$. So overall time complexity is $O(V^2\log V + VE)$.

The time complexity of Johnson's algorithm becomes same as Floyd Warshall when the graphs is complete (For a complete graph $E = O(V^2)$). But for sparse graphs, the algorithm performs much better than Floyd Warshall.

References:

1. Theory & Concept <https://www.geeksforgeeks.org/johnsons-algorithm/>
2. Graph Example from GeeksforGeeks

Code is my own. I typed and implemented it myself.