المركز الوطني للمتميزين

NATIONAL CENTER FOR THE DISTINGUISHED

/NCDSyria

Shortest path algorithms

Research in Informatics

Eleventh grade

By: Karam Shbeb

Supervisor: Eng. Ali Jnidi

2016/2017

Abstract:

Finding the shortest path between two points or more is a very important problem for every programmer and choosing the algorithm which can solve the problem with the faster time is one of the most troubles which run across programmers.

In this search my aim is to find all of the possible cases of the shortest path problems and compare between the most used shortest path algorithm to find the most suitable algorithm for each case which can give us the correct answer with the best time complexity.

Table of Contents

1. Introduction: 3
2. Inquiry of research: 3
3. Graph theory 4
3.1 Flavors of graphs[1]: 4
3.2 Representations of graphs[1, 2]: 6
4. Dijkstra algorithm[4, 5]: 8
4.1 Dijkstra algorithm steps[2]: 8
4.2 Dijkstra algorithm Pseudo code: 8
4.3 Dijkstra algorithm example: 9
.... 9
.... 9
4.4 Run time analysis for Dijkstra algorithm: 9
5. Bellman Ford algorithm[6] 10
5.1 Bellman Ford algorithm steps[3]: 10
5.2 Bellman-Ford algorithm Example: 11
5.3 Runtime analysis for Bellman-Ford algorithm: 11
6. Floyd Warshall algorithm[7] 12
6.1 Runtime analysis for Floyd Warshall algorithm: 13
7. Johnson algorithm[3] 14
7.1 Johnson algorithm steps: 14
7.2 Johnson algorithm Runtime analysis: 14
8. Results and conclusion: 15
9. Codes: 16
10. References 20

Table of Figures

Figure (1): Simple graph 4
Figure (2): undirected graph 4
Figure (3): directed graph 4
Figure (4): Unweighted graph 5
Figure (5): Weighted graph 5
Figure (6): Graph “A” 6
Figure (7): Graph B 9
Figure (8): the output of Graph B 9
Figure (9): Graph C 11
Figure (10): the output of graph C 11

Table of Tables

Table (1): Adjacency Matrix for graph “A” 6
Table (2): Adjacency list for graph “B” 7
Table (3) Adjacency Matrix for graph “B” 9
Table (4): Edge list for graph “C” 11

1. Introduction:

If we have a map which shows many cities with the routes between it and we want to go from one to another…. If there are few cases we can enumerate all the routes between the two points, then add up the distances on each route, and select the shortest.

But if we have a lot of cases we can’t find it easily, we should use an algorithm which help us to find the shortest path and some applications need to find the shortest path and use it (maps applications, network routing….), so the programmer of this applications need to write a code which find the shortest path depending on **shortest path algorithms.**

in competitive programming: shortest path problems are one of the most important side in programming contests so every competitive programmer needs to use **shortest path algorithms.**

2. Inquiry of research:

Like we saw shortest path problems is very important for competitive programmers and other programmers which work on different projects so every programmer needs to solve a shortest path problems using one of shortest path algorithms.

And as we know there is many shortest path algorithms and solving a shortest path problem depending on the algorithm that we use.

So, what is the cases and algorithms for shortest path??

And which algorithm can we use on every case and any of them is the better in every case???

3. Graph theory

Graphs are one of the unifying themes of computer science – an abstract representation which describes the organization of transportation systems, electrical circuits, human interactions, and telecommunication networks.

Every graph is shown with many vertexes which are connected with each other by many edges.

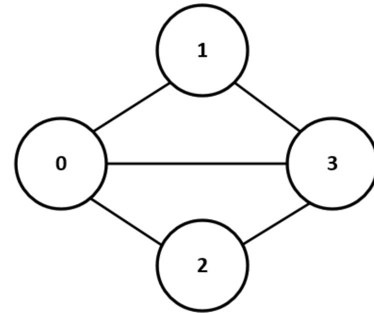

Figure (1): Simple graph

3.1 Flavors of graphs[1]:

There are several fundamental properties of graphs which impact the choice of data structures used to represent them and algorithms available to analyze them. The first step in any graph problem is determining which flavor of graph you are dealing with.

Directed vs undirected:

A graph G = (V, E) is undirected if edge (x, y) ∈ E implies that (y, x) is also in E. If not, we say that the graph is directed.

As we see in the figure 2 (undirected graph) we can move from (0) to (1) and from (1) to (0) but in the directed graph (figure 3) we can move from (0) to (1) but we can’t move from (1) to (0) and also.

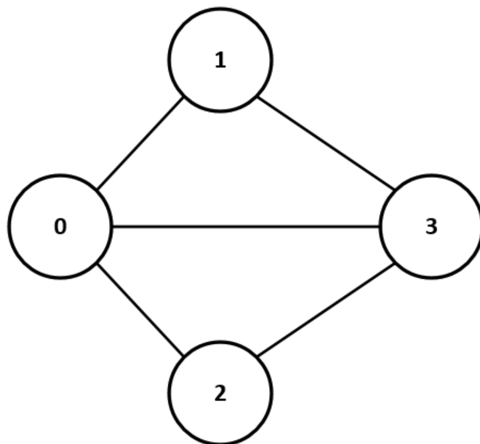

Figure (2): undirected graph

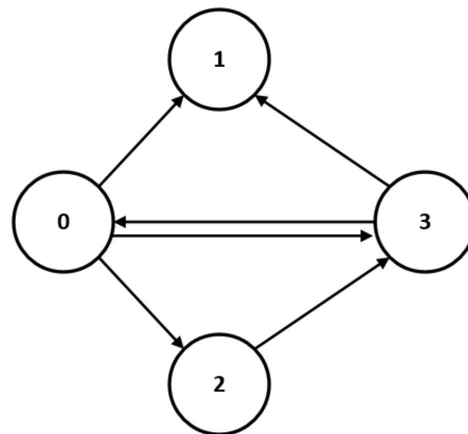

Figure (3): directed graph

weighted vs unweighted:

In weighted graphs, each edge (or vertex) of G is assigned a numerical value, or weight. Typical application-specific edge weights for road networks might be the distance, travel time, or maximum capacity between x and y. In unweighted graphs, there is no cost distinction between various edges and vertices.

The difference between weighted and unweighted graphs becomes particularly apparent in finding the shortest path between two vertices. For unweighted graphs, the shortest path must have the fewest number of edges, but in the weighted graphs we must use certain algorithm to find the shortest path.

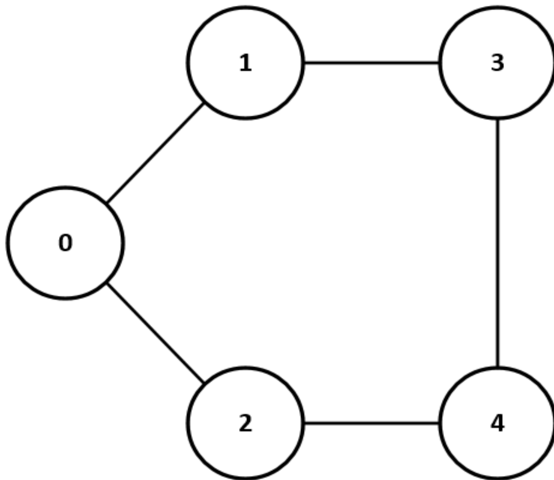

Figure (4): Unweighted graph

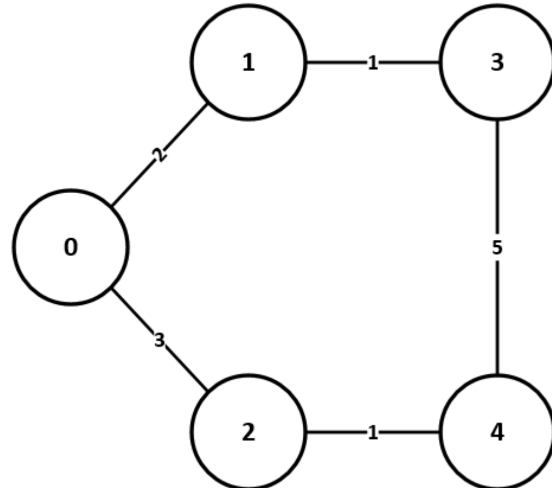

Figure (5): Weighted graph

3.2 Representations of graphs[1, 2]:

There are several possible ways to represent graphs, the most two standard ways to represent a graph G = (V, E) “a graph G with V vertices and E edges” is the adjacency lists or the adjacency matrix forms, and sometimes the edge list form is used (there are a few other rare structures).

The two ways (adjacency lists and adjacency matrix) are used in the both directed and undirected graphs and used in the both weighted and unweighted graphs.

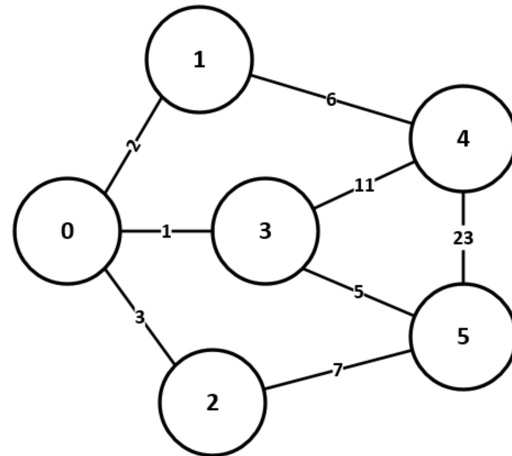

Figure (6): Graph “A”

The Adjacency Matrix: usually in the form of a 2D array (V×V)[1], For an unweighted graph, set AdjMat[i][j] to a non-zero value (usually 1) if there is an edge between vertex i-j or zero otherwise. For a weighted graph, set AdjMat[i][j] = weight (i, j) if there is an edge between vertex i-j with weight (i, j) or infinite (we can define it as INT_MAX) if the two vertex isn’t connected.

An Adjacency Matrix is a good choice if the connectivity between two vertices in a small dense graph is frequently required. However, it is not recommended for large sparse graphs as it would require too much space (O (V 2)) and there would be many blank (zeros and infinites) cells in the 2D array.

Adjacency matrix for graph A

0	2	3	1	∞	∞
2	0	∞	∞	6	∞
3	∞	0	∞	∞	7
1	∞	∞	0	11	5
∞	6	∞	11	0	23
∞	∞	7	5	23	0

Table (1): Adjacency Matrix for graph “A”

The Adjacency Lists[3]: we have a vector of vector of pairs, storing the list of neighbors of each vertex u as ‘edge information’ pairs. Each pair contains two pieces of information, the index of the neighboring vertex and the weight of the edge. If the graph is unweighted, simply store the weight as 0, 1. The space complexity of Adjacency List is O (V + E) because if there are E bidirectional edges in a (simple) graph, this Adjacency List will only store 2E ‘edge information’ pairs. As E is usually much smaller than V × (V −1)/2 = O (V 2)—the maximum number of edges in a complete (simple) graph, Adjacency Lists are often more space-efficient than Adjacency Matrices.
all of these features make the adjacency list the most common used data structure in the graph algorithms.

[1] V is the number of the vertices

Also, there is many data structure which used to represent a graph but the adjacency list and the adjacency matrix forms is the most used especially in the shortest path algorithms.

Adjacency list for graph A

0	1	2	3
1	0	4	_
2	0	5	_
3	0	4	5
4	1	3	5
5	2	3	4

Table (2): Adjacency list for graph “A”

4. Dijkstra algorithm[4, 5]:

Dijkstra's algorithm is an algorithm for finding the shortest paths from a source node to all other nodes in a graph, it was designed and published by E.W. Dijkstra[2] in 1959.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other, it can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Where does it work???

- Both directed and undirected graphs.
- Weighted graph When All edges have nonnegative weights.

How does it work???

we generate a SPT (shortest path tree) with given source as root. We maintain two sets (or vectors), one vector contains vertices included in shortest path tree, other vector includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other vector (vector of not yet included) and has minimum distance from source.

4.1 Dijkstra algorithm steps[2]:

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a vector of the unvisited nodes called the unvisited vector consisting of all the nodes.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. For example, if the current node A is marked with a distance of 8, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 8 + 2 = 10.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

4.2 Dijkstra algorithm Pseudo code:

```
1:    void Dijkstra (Graph, source):
```

[2] Edsger Wybe Dijkstra: A Dutch computer scientist (1930\2002)

```
2:      for each vertex v in Graph:                    // Initialization
3:          dist[v] := infinity                        // initial distance from source to vertex v
4:      dist[source] := 0                              // Distance from source to source
5:      Q := the set of all nodes in Graph             // the nodes in the graph
6:      while Q is not empty:                          // main loop
7:          u := node in Q with smallest dist[ ]
8:          remove u from Q
9:          for each neighbor v of u:                  // where v has not yet been removed from
10:             path := dist[u] + dist_between (u, v)
11:             if path < dist[v]                      // if there is shorter path
12:                 dist[v] := path
```

4.3 Dijkstra algorithm example:

if we have this graph (graph B) the input will be:

0	13	8	3	∞
∞	0	∞	∞	7
∞	2	0	6	5
∞	∞	∞	0	3
∞	4	1	7	0

Table (3) Adjacency Matrix for graph "B"

And when we compile the Dijkstra code with this input and we determine (0) as a source index we will have this output (shortest path from zero to all other nodes):

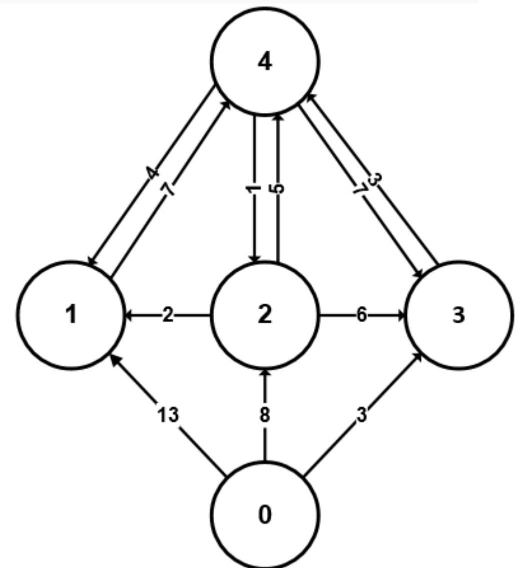

Figure (7): Graph B

```
the shortest path from the node 0 to the node 0 is 0
the shortest path from the node 0 to the node 1 is 9
the shortest path from the node 0 to the node 2 is 7
the shortest path from the node 0 to the node 3 is 3
the shortest path from the node 0 to the node 4 is 6

Process returned 0 (0x0)   execution time : 46.352 s
Press any key to continue.
```

Figure (8): the output of Graph B

4.4 Run time analysis for Dijkstra algorithm:

There is many implementation for Dijkstra but the most used implementation has $O(V^2)$ run time like the given code below and with this runtime Dijkstra can solve shortest path problems without negative weights.

5. Bellman Ford algorithm[6]

The Bellman–Ford algorithm is a dynamic algorithm that computes shortest paths from a single source vertex to all of the other vertices, it is slower than Dijkstra in the same problem but it can work on negative weights (some of the edge weights are negative numbers).

The algorithm was published by Richard Bellman and Lester Ford in 1956 and 1958.

Where does it work???

- Directed and undirected graph.
- Weighted graph (negative and nonnegative weights).
- Can detect negative weight cycle.

negative weight cycle:

A negative weight cycle is a cycle with weights that sum to a negative number it means that every time the algorithm runs into the cycle it detects a shorter path.

How does it work???

Like Dijkstra's Shortest Path, the Bellman-Ford is based on the relaxation technique, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution.

5.1 Bellman Ford algorithm steps[3]:

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Now for every edge (u-v) in the graph check if the distance to the v is less than the sum of the distance to u and the weight of the edge if it is true update the distance to v, where the number of the vertex is V we repeat this operation V-1 times.
3. The third step check if there is a negative weight cycle, the second step 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.

Pseudo code:

```
// Step 1: initialize graph
for each vertex v in vertices:
    distance[v] := inf          //  all vertices have a weight of infinity
    predecessor[v] := null      // And a null predecessor

distance[source] := 0           // Except for the Source, where the Weight is
                                   zero

// Step 2: relax edges repeatedly
```

```
for i from 1 to size(vertices)-1:
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w
            predecessor[v] := u

// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
return distance[], predecessor[]
```

5.2 Bellman-Ford algorithm Example:

If we have this directed and weighted graph (graph B) with negative weights the input (Edge list) will be:

from	To	weight
0	1	2
0	2	-2
0	3	3
1	4	-1
2	1	4
2	3	6
2	4	4
3	4	3
4	1	4
4	2	3
4	3	1

Table (4): Edge list for graph "C"

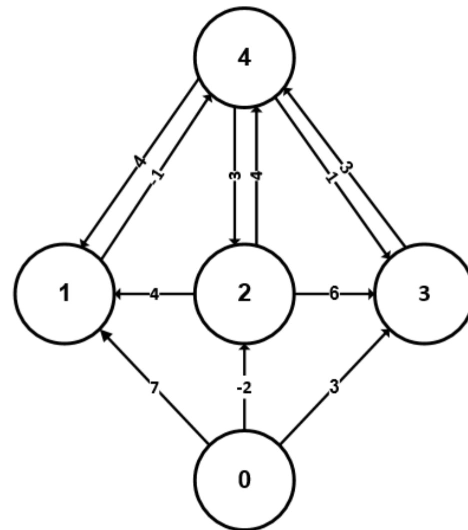

Figure (9): Graph C

And the output will be:

```
the shortest path from 0 to 0  =  0
the shortest path from 0 to 1  =  2
the shortest path from 0 to 2  =  -2
the shortest path from 0 to 3  =  2
the shortest path from 0 to 4  =  1
```

Figure (10): The output of graph C

5.3 Runtime analysis for Bellman-Ford algorithm:

Bellman-Ford algorithm is very similar to Dijkstra but it is slower (time complexity for Bellman-Ford is o(VE) when V is the number of vertex and E is the number of edges) but Bellman-Ford distinguished that it is more versatile which can work on more cases like negative weights.

6. Floyd Warshall algorithm[7]

The Floyd-Warshall algorithm is a dynamic algorithm for solving the All pairs shortest path algorithm it means it finds the shortest distance between all vertices in directed weighted graph with positive or negative weights.

If the weights are nonnegative for all edges, then we can use Dijkstra's single source shortest path algorithm for all vertices to solve the problem (because Dijkstra is faster than Floyd-Warshall in positive weights) but if we have a negative weight we should use Floyd-Warshall (because it is also faster than Bellman-Ford in all pairs shortest path algorithm).

The algorithm was published in 1962 by Robert Floyd and also by Stephen Warshall in 1962 and it is essentially the same as algorithm previously published by Bernard Roy in 1959 so it is also known as Floyd's algorithm, the Roy–Warshall algorithm, the Roy–Floyd algorithm.

Where does it work???

- To find all pairs shortest path in both directed and undirected graph.
- Weighted graphs with positive and negative weights.
- It can detect negative weight cycle.

How does it work???

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 [1 ... 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].

Pseudo code:

```
1. for i=1 to n do
2.   for j to n do
3.     dist (i, j) = weight (i, j)
4. for k= 1 to n do
5.   for i=1 to n do
6.     for j= 1 to n do
```

```
7.        if (dist (i, k) + dist (k, j) < dist (i, j)) then
8.          dist (i, j) = dist (i, k) + dist (k, j)
```

6.1 Runtime analysis for Floyd Warshall algorithm:

Like we see there is 3 "for" statements (i,j,k) so The time complexity for Floyd Warshall is $o(V^3)$ when V is the number of vertex in the graph, and we know that the time complexity of Johnson algorithm is $o(V^2+VE)$ so in small graphs V^2+VE is lower than V^3 so Johnson algorithm is faster in the sparse (small) graphs and Floyd Warshall is faster in the large graphs.

7. Johnson algorithm[3]

Johnson algorithm is the algorithm that find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative, Donald B. Johnson published it in 1977. as we see applying Dijkstra single source shortest path algorithm for every vertex is the fast way but if we have negative weight edges Dijkstra doesn't work.

Johnson algorithm is depending on Bellman-Ford and Dijkstra algorithm, the idea of this algorithm is to re-weight all edges and make them all positive (using Bellman-Ford), then apply Dijkstra's algorithm for every vertex.

Where does it work???

- Solve all pairs shortest path problems in Both directed an undirected Graph.
- Works on negative weights.
- Used in sparse graph because Floyd-Warshall algorithm is faster with $O(V^3)$ in large graphs.

7.1 Johnson algorithm steps:

1. First, a new node s is added to the graph, connected by zero-weight edges to each of the other nodes.
2. Second, the Bellman–Ford algorithm is used, starting from the new vertex s, to find for each vertex v the minimum weight h(v) of a path from s to v. If this step detects a negative cycle, the algorithm is terminated.
3. Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from u to v, having length w (u, v), is given the new length w (u, v) + h(u) − h(v).
4. Finally, q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.

7.2 Johnson algorithm Runtime analysis:

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

8. Results and conclusion:

As we saw shortest path algorithms is very important for a lot of problems such as: web graph and networks routing.

As a result of his research:

1. we find that the best algorithms to find the shortest path in unweighted and undirected graph is **BFS** or **DFS** in o(V+E) time complexity.
2. To find the single source shortest path without the negative weights **Dijkstra's** algorithm is the best way with $O(V^2)$ or O((V+E) log(V)) in other implementations.
3. To find the single source shortest path with the negative weights **Bellman-Ford** algorithm is the best way with O(V+E) time complexity.
4. To find All pairs shortest path in the sparse algorithm the best algorithm is **Johnson** algorithm with $O(V^2+VE)$ and in large graphs Floyd-Warshall is the best with $O(V^3)$ time complexity.
5. All of the mentioned algorithms before can work with negative weights but **Dijkstra's** algorithms cannot work with negative weights or detect negative weights cycle.

9. Codes:

Dijkstra code "1"

```
#include <iostream>
#include <vector>
using namespace std;
const int inf = 1<<30;
int main ()
{
   cout<<"please insert the number of nodes : ";
   int n,s,x;
   cin>>n;
   vector <vector<int> > adj;
   adj.resize(n);
   for (int i=0;i<n;i++)
      adj[i].resize(n);
   cout<<"now insert the adjacency matrix :
"<<endl;
   for (int i=0;i<n;i++)
      for (int j=0;j<n;j++)
         cin>>adj[i][j];
   cout<<"insert your source index : ";
   cin>>s;
            vector<bool> vis(n);
            vector<int> dis(n);
            for(int i=0;i<n;i++)
                        dis[i]=inf;
            dis[s]=0;
```

Dijkstra code "2"

```
for(int i=0;i<n;i++)
         {
            int cur=-1;
            for(int j=0;j<n;j++)
            {
            if(vis[j]) continue;
            if(cur==-1 || dis[j]<dis[cur])
             cur=j;
                        }
            vis[cur]=true;
                  for(int j=0;j<n;j++)
                        {
                  int path = dis[cur] + adj[cur][j];
                  if(path < dis[j])
                  dis[j]=path;
                        }     }
   cout<<"Do you want to know the shortest path
    to all nodes !!! : ";
   string f;
   cin>>f;
   if (f=="NO")
   {
      cout<<"please insert the index of the node
that";
      cout<<"you want to know the shortest path
```

Dijkstra code “3”

```
 cin>>x;
cout<<"the shortest path from the node "<<s<<" to the node ";
    cout<<x<<" is "<<dis[x];
  }
  else
  {
    for (int i=0;i<n;i++)
      cout<<"the shortest path from the node "<<s<<" to the node ";
      cout<<i<<" is "<<dis[i]<<endl;
  }
  return 0;
}
```

Bellman-Ford “1”

```
#include <iostream>

#include <vector>

#include <limits>

using namespace std;

int main()

{

   int v,e,source,src=0,dest=1,wg=2;

   cout<<"enter the number of vertex\n";

   cin>>v;

   cout<<"enter the number of edges\n";

   cin>>e;

   vector <vector<int> > graph;

   graph.resize(e);

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

      graph[i].resize(3);

   cout<<"now please insert: src dest weight : for
every edge\n";

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

      for (int j=0;j<3;j++)

         cin>>graph[i][j];

   cout<<"now insert your source index\n";

   cin>>source;

   int dist [v];

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

      dist[i]=INT_MAX;

   dist[source]=0;
```

Bellman-Ford “2”

```
   for (int i=1;i<=v-1;i++)

   {

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

      {

         int u = graph[j][src];

         int v = graph[j][dest];

         int weight = graph[j][wg];

         if (dist[u] != INT_MAX && dist[u] + weight
< dist[v])

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

      }

   }

   bool con=1;

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

   {

      int u = graph[i][src];

      int v = graph[i][dest];

      int weight = graph[i][wg];

      if (dist[u] != INT_MAX && dist[u] + weight <
dist[v])

         {

            cout<<"Graph contains negative weight
cycle";

            con=0;

         }

   }
```

Bellman-Ford “3”

```
    if (con)
      for (int i=0;i<v;i++)
        cout<<"the shortest path from "<<source<<" to "<<i<<"  =  "<<dist[i]<<"\n";
    return 0;
}
```

10. References:

1. Skiena, S.S. and M.A. Revilla, *Programming challenges: The programming contest training manual*. 2006: Springer Science & Business Media.
2. Halim, S. and F. Halim, *Competitive Programming 3: The New Lower Bound of Programming Contests: Handbook for ACM ICPC and IOI Contestants*. 2013.
3. Cormen, T.H., et al., *Introduction to algorithms second edition*. 2001, The MIT Press.
4. Chen, J.-C., *Dijkstra's shortest path algorithm.* Journal of Formalized Mathematics, 2003. **15**: p. 144-157.
5. Johnson, D.B., *A note on Dijkstra's shortest path algorithm.* Journal of the ACM (JACM), 1973. **20**(3): p. 385-388.
6. Goldberg, A.V. and T. Radzik, *A heuristic improvement of the Bellman-Ford algorithm.* Applied Mathematics Letters, 1993. **6**(3): p. 3-6.
7. Katz, G.J. and J.T. Kider Jr. *All-pairs shortest-paths for large graphs on the GPU*. in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. 2008. Eurographics Association.