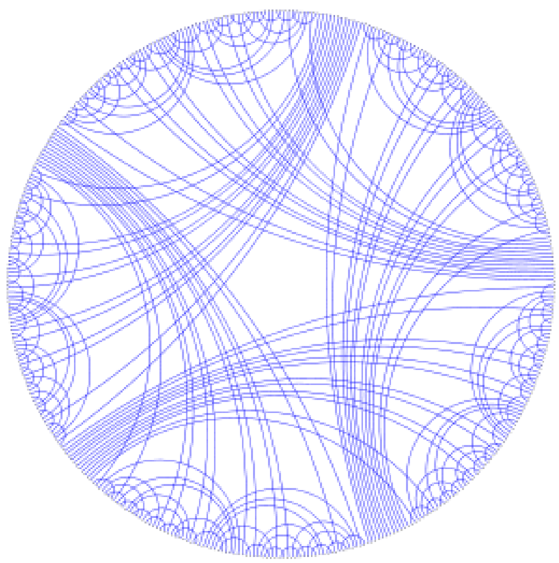
Universitatea din Craiova
Facultatea de Automatică,Calculatoare şi Electronică

04 June 2018



Project : Algorithm Design
Title: Minimum Length Path
Teachers: Becheru Alex & Bădică Costin
Student: Amzuloiu Andrei-Ciprian
Section : Calculatoare Română
Year I
Group: CR 1.1

Contents

1 Problem Statement

1.1 Title

Minimum Length Path.

1.2 Description

My assignment requires the implementation of two different algorithms to determine the minimum path between two vertexes in weighted directed graphs.

To done that I implemented the Bellman Ford algorithm and the Dijkstras algorithm in C.

2 Pseudocode

2.1 convert_mat_array(

$$INT coordinates_1, INT coordinates\_2, INT maximum\_size$$

)

```
return coordinate_1 * maximum_size + coordinates_1
```

2.2 random_graph(struct graph* graph)

```
INT iterator_1
INT iterator_2
INT flag
graph-> no_elems = rand()
graph-> no_edges = 0
graph-> ad_matrix = calloc(graph-> no_elems * graph > no_elems, sizeof(int))
for iterator_1 = 0 to graph-> no_elems do
   for iterator_2 = 0 to graph-> no_elems do
      if (iterator_1 == iterator_2) then
         continue
      end if
      flag = rand()
      if (flag) then
         graph-> no_edges++
         if graph-> ad_matrix[convert_mat_array(iterator_2, iterator_1, graph->
         no_elems)] == 0) then
            graph-> ad_matrix[convert_mat_array(iterator_1, iterator_2, graph->
            no_elems)] = rand()
         end if
      end if
   end for
end for
```

2.3 create_edges(struct graph* graph)

```
INT iterator
INT iterator_edge = 0
graph-> edge = (structedge)malloc(graph-> no_edgessizeof(structedge))
for iterator = 0 to graph-> no_elemsgraph-> no_elems do
   if (graph-> ad_matrix[iterator]! = 0) then
      graph-> edge[iterator_edge].source = iterator/graph-> no_elems
      graph-> edge[iterator_edge].destination = iterator%graph->
      no_elems
      graph-> edge[iterator_edge].weight = graph-> ad_matrix[iterator]
      iterator_edge++
```

```
  end if
end for
```

2.4 negative_cycle_check(struct graph graph, INT dist[ ])

```
INT iterator
INT source
INT destination
INT weight
for iterator = 0 to graph.no_edges do
  source = graph.edge[iterator].source;
  destination = graph.edge[iterator].destination
  weight = graph.edge[iterator].weight
  if (dist[source]! = INFINITY && dist[source]+weight < dist[destination])
  then
    print("Graph contains negative weight cycleBellman Ford algorithm is unavailable")
    return 1
  end if
end for
return 0
```

2.5 init_dijkstra (struct graph graph, INT value_mat[ ], INT distance[ ], INT pred[ ], int start_node)

```
INT iterator_1
INT iterator_2
for iterator_1 = 0 to graph.no_elems do
  for iterator_2 = 0 to graph.no_elems do
    if ( then
      graph.ad_matrix[convert_mat_array(iterator_1, iterator_2, graph.no_elems)] ==
      0)value_mat[convert_mat_array(iterator_1, iterator_2, graph.no_elems)] =
      INFINITY
    else
      value_mat[convert_mat_array(iterator_1, iterator_2, graph.no_elems)] =
      graph.ad_matrix[convert_mat_array(iterator_1, iterator_2, graph.no_elems)]
    end if
  end for
end for
FOR iterator_1 = 0 to graph.no_elems
distance[iterator_1] = value_mat[convert_mat_array(start_node, iterator_1, graph.no_elems)]
pred[iterator_1] = start_node
```

2.6 print_results_dijkstra(INT dist[ ], INT path[ ][ ], INT no_elems)

```
INT iterator
```

```
if (final_node != start_node) then
   if (distance[final_node] < INFINITY) then
      print("Distance between %d and %d : %d", start_node, final_node, distance[final_node])
      print("Path : %d", final_node)
      iterator = final_node
      while (iterator != start_node) do
         iterator = pred[iterator]
         print(" <-%d", iterator)
      end while
   else
      print("No valid path between the two vertexes.");
   end if
else
   print("Distancebetween%dand%d : %d", start_node, final_node, distance[final_node])
   print("Path : %d", start_node)
end if
```

2.7 init_bellman_ford(int dist[ ], INT path[ ][ ], INT no_elems)

```
INT iterator
for iterator = 0 to no_elems do
   dist[iterator] = INFINITY
end for
for iterator = 0 to no_elems do
   path[iterator][0] = 0
end for
```

2.8 print_results_bellman_ford(INT dist[ ], INT path[ ][ ], INT no_elems)

```
INT iterator
if dist[dest] == INFINITY then
   print("No valid path between the two vertexes.")
else
   print("Distance between%d and%d : %d", src, dest, dist[dest])
   print("Path : %d", dest)
   for iterator = path[dest][0]to0 do
      print(" <-%d", path[dest][iterator])
   end for
end if
```

2.9 dijkstra(struct graph graph, INT start_node, INT final_node)

```
INT *value_mat = (int*)malloc(graph.no_elems*graph.no_elems*sizeof(int))
```

```
INT *distance = (int*)malloc(graph.no_elems * sizeof(int))
INT *pred = (int*)malloc(graph.no_elems * sizeof(int))
INT *visited = (int*)calloc(graph.no_elems, sizeof(int))
INT count
INT minimum_distance
INT next_node
INT iterator_1
INT iterator_2
init_dijkstra(graph, value_mat, distance, pred, start_node)
distance[start_node] = 0
visited[start_node] = 1
count = 1
while (count < graph.no_elems − 1) do
    minimum_distance = INFINITY
    for iterator_1 = 0 to graph.no_elems do
        if (distance[iterator_1] < minimum_distance!visited[iterator_1]) then
            minimum_distance = distance[iterator_1]
            next_node = iterator_1
        end if
    end for
    visited[next_node] = 1
    for iterator_1 = 0 to graph.no_elems do
        if (!visited[iterator_1]) then
            if (minimum_distance+value_mat[convert_mat_array(next_node, iterator_1, graph.no_elems)] <
            distance[iterator_1]) then
                distance[iterator_1] = minimum_distance+value_mat[convert_mat_array(next_node, iterator_1, g
                pred[iterator_1] = next_node
            end if
        end if
    end for
    count + +
end while
print_results_dijkstra(start_node, final_node, distance, pred)
free(value_mat)
free(distance)
free(pred)
free(visited)
```

2.10 bellman_ford(struct graph graph, INT start_node, INT final_node)

```
INT *dist = (int*)malloc(graph.no_elems * sizeof(int))
INT iterator_1
INT iterator_2
INT iterator_3
INT weight
```

```
INT source
INT destination
INT **path = (int**)malloc(graph.no_elems * sizeof(int*))
for iterator_1 = 0 to iterator_1 < graph.no_elems do
   path[iterator_1] = (int*)malloc(graph.no_elems * sizeof(int))
end for
create_edges(graph)
init_bellman_ford(dist, path, graph.no_elems)
dist[start_node] = 0
for iterator_1 = 1 to graph.no_elems − 1 do
   for iterator_2 = 0 to graph.no_edges do
      source = graph.edge[iterator_2].source
      destination = graph.edge[iterator_2].destination
      weight = graph.edge[iterator_2].weight
      if (dist[source]! = INFINITY dist[source]+weight < dist[destination]) then
         dist[destination] = dist[source] + weight
         path[destination][0] = path[source][0] + 1
         for iterator_3 = 1 to path[destination][0] do
            path[destination][iterator_3] = path[source][iterator_3]
         end for
         path[destination][path[destination][0]] = source
      end if
   end for
end for
if (!negative_cycle_check(graph, dist)) then
   print_results_bellman_ford(start_node, final_node, dist, path)
end if
free(dist)
for iterator_1 = 0 to graph.no_elems do
   free(path[iterator_1])
end for
free(path)
free(graph.edge)
```

2.11 print_ad_mat(struct graph graph)

```
FILE *f_write
INTiterator
f_write = open("graph.txt", "w")
print(f_write, "Number of vertexes : %dAdjacency matrix with cost on each edge : NEW LINE", graph.no_elems)
for iterator = 0 to graph.no_elems * graph.no_elems do
   print(f_write, "%d", graph.ad_matrix[iterator])
   if (iterator%graph.no_elems == graph.no_elems − 1) then
      print(f_write, NEWLINE)
```

  end if
end for
$close(f\_write)$

2.12 read_input(INT var$_1, INT var_2$)

$FILE * f\_read$
$f\_read = open("input.txt", "r")$
$scan(f\_read, "\%d\%d", var_1, var_2)$
$close(f\_read)$

3 Application Design

3.1 Main

This module has the user interaction part and some functions calls (random_graph(), print_ad_mat(), read_input(), bellman_ford() and dijkstra() functions).

I used the random_graph() function to generate a random weighted directed graph. The graph is stored in *graph*, a structure variable. After that, the adjacency matrix with cost edges is printed in *graph.txt* using the print_ad_mat() function, because it's easier for user to check it in file then checking from the console screen. I used the *keep_open* variable to put a pause in the running to allow the user to check the graph in file and choose the source vertex and destination vertex. The source and destination vertex are written by the user in the input.txt file.

After all, I called the bellman_ford() function and the dijkstra() function and put another pause in the console screen to allow the user to check the results.

3.2 dijkstra() function

Dijkstra algorithm is also called single source shortest path algorithm. It is based on greedy technique. The algorithm maintains a list visited[ ] of vertices, whose shortest distance from the source is already known. If visited[1], equals 1, then the shortest distance of vertex i is already known. Initially, visited[i] is marked as, for source vertex. At each step, we mark visited[v] as 1. Vertex v is a vertex at shortest distance from the source vertex. At each step of the algorithm, shortest distance of each vertex is stored in an array distance[ ].

The steps for Dijkstra algorithm is:

1. Create cost matrix (value_mat[ ], in our case) from adjacency matrix (graph.ad_matrix). value_mat[i, j, graph.no_elems)] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then value_mat[i, j, graph.no_elems)] is infinity.

I used array instead of matrices. To convert the coordinates of a matrix in array position I used the function convert_mat_array(). It returns the converted array position.

2. Array visited[ ] is initialized to zero. To done that I allocated memory using function calloc.

3. If the vertex 0 is the source vertex then visited[0] is marked as 1.

4. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to graph.no_elems - 1 from the source vertex 0. Initially, distance of source vertex is taken as 0. i.e. distance[0] = 0.

I done the step 1, 3 and 4 in the function init_dijkstra().

5. Iterate from 1 to graph.no_elems

Choose a vertex next_node, such that distance[next_node] is minimum and visited[next_node] is 0. - Mark visited[next_node] as 1.

Recalculate the shortest distance of remaining vertices from the source.

Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance.

At the end I called the function print_results_dijkstra() to print the distance and path in the console scree. I used the *time.h* library to find the execution time.

The program contains two loops each of which has a complexity of O(n). n is number of vertices. So the complexity of algorithm is O(n^2).

3.3 bellman_ford() function

Dijkstra and Bellman-Ford Algorithms used to find out single source shortest paths. i.e. there is a source node, from that node we have to find shortest distance to every other node. Dijkstra algorithm fails when graph has negative weight cycle. But Bellman-Ford Algorithm wont fail even, the graph has negative edge cycle. (using negative_cycle_check() function in our case) If there any negative edge cycle it will detect and say there is negative edge cycle. If not it will give answer to given problem.

Bellman-Ford Algorithm will work on logic that, if graph has n nodes, then shortest path never contain more than n - 1 edges. This is exactly what Bellman-Ford do. It is enough to check each edge (graph.no_edges - 1) times to find shortest path. But to find whether there is negative cycle or not we again do one more relaxation. If we get less distance in n^{th} relaxation we can say that there is negative edge cycle. Reason for this is negative value added and distance get reduced.

Following are the detailed steps.

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 graph.no_elems with all values as infinite except dist[start_node] where start_node is source vertex. This step is included in the init_bellman_ford() function. This algorithms works with edges so I used the create_edges() function to find all edges (with source, destination and weight) from adjacency matrix.

2) This step calculates shortest distances. Do following (graph.no_edges - 1) times where graph.no_edges is the number of vertices in given graph. If a better way is found the distance is added in the distance array and the path in the path matrix (path[ ][ ]) which is a matrix where all paths with minimum distances are stored.

3) This step reports if there is a negative weight cycle in graph. To done that I used the negative_cycle_check() function.

The idea of step 3 is, step 2 guarantees shortest distances if graph does not 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 time complexity is (v-1) (E) O(1) = O(VE).

3.4 random_graph() function

The graph is a non-trivial concept, so I used a generator to generate a graph with big number of elements (up to 500). The function uses the rand() function to generate the number of nodes. After that, each element of the matrix is iterated. When flag == 1 there will be an edge, else the position in ad_matrix will be 0. The rand() is used again to change the value of flag (1 or 0) and to give a value to each edge of the graph (a adjiancy matrix position).

3.5 Input//Output

The generated graph is printed in file graph.txt. The only input of the program are the two vertexes (source and destination) from the input.txt. After the user check the graph, he can choose two vertexes to find the minimum distance and path of them.

The results (distance, path and time execution of each algorithm) will be printed in the console screen.

3.6 Modules

I used two header files.

1) *algorithms.h*, a C library implementation for Dijkstra algorithm and Bellman Ford algorithm. This library has the following functions:

- void dijkstra(struct graph graph, int start_node, int final_node);

This function is used to find the minimum path between two vertexes and print them in console screen using the Dijkstra algorithm.

Parameters:

graph - A structure variable that contains the details about the graph that will be used.

start_node - The source vertex.

final_node - The destination vertex.

- void bellman_ford(struct graph graph, int start_node, int final_node);

This function is used to find the minimum path between two vertexes and print them in console screen using the Bellman Ford algorithm.

Parameters:

graph - A structure variable that contains the details about the graph that will be used.

start_node - The source vertex.

final_node - The destination vertex.

2) *tools.h*, a C library implementation for some functions used in algorithms or used for input/output. This library has the following functions:

- int convert_mat_array(int coord_1, int coord_2, int max_size);

This function converts the coordinates of a matrix in array position and return the value of that position.

Parameters:

coord_1 - The column coordinate of a matrix.

coord_2 - The row coordinate of a matrix.

max_size - The size of a row/column.

- void create_edges(struct graph* graph);

This function use the ad_matrix to create an array with all edges of the graph. This function is used in the Bellman Ford algorithm because it works with the edges of the graph.

Parameters:

*graph - The graph that we use in our algorithms.

- void random_graph(struct graph* graph);

This function is used to generate a random graph.

Parameters:

*graph - A struct variable that we use to generate the graph.

- void print_ad_mat(struct graph graph);

This function is used to print the adjacency matrix in graph.txt.

Parameters:

graph - A structure variable that contains the details about the graph that will be used.

- void read_input(int* var_1, int* var_2);

This function is used to read the source vertex and the destination vertex from input.txt.

Parameters:

*var_1 - The source vertex.

*var_2 - The destination vertex.

- void init_dijkstra(struct graph graph, int* value_mat, int* distance, int* pred, int start_node);

This function is used to initialize the value_mat, distace and pred arrays for the Dijkstra algorithm.

Parameters:

graph - A structure variable that contains the details about the graph that will be used.

*value_mat - A copy of *ad_matrix modified for dijkstra algorithm.

*distance - An array used to store the distance between source vertex and any other vertex in graph.

*pred - An array used to store the predecessor of each node.

- void print_results_dijkstra(int start_node, int final_node, int* distance, int* pred);

This function is used to print the minimum distance and path between two vertexes after the implementation of the Dijkstra algorithm.

Parameters:

start_node - The source vertex.

final_node - The destination vertex.

*distance - An array used to store the distance between the source vertex and any other vertex.

*pred - An array used to store the predecessor of each node.

- void init_bellman_ford(int* dist, int path, int no_elems);**

This function is used to initialize the dist array and the path matrix for the Bellman Ford algorithm.

Parameters:

*dist - An array used to store the distance between the source vertex and any other vertexes.

**path - A matrix used to store the path between source vertex and any other vertexes.

no_elems - The number of vertexes.

- void print_results_bellman_ford(int src, int dest, int* dist, int path);**

This function is used to print the minimum distance and path between two vertexes after the implementation of the Bellman Ford algorithm.

Parameters:

src - The source vertex.

dest - The destination vertex.

*dist - An array used to store the distance between the source vertex and any other vertexes.

**path - A matrix used to store the path between the source vertex and any other vertex of the graph.

- int negative_cycle_check(struct graph graph, int* dist);

This function is used to check if the graph has negative cycles. I used this function in the Bellman Ford algorithm.

Parameters:

graph - A structure variable that contains the details about the graph that will be used.

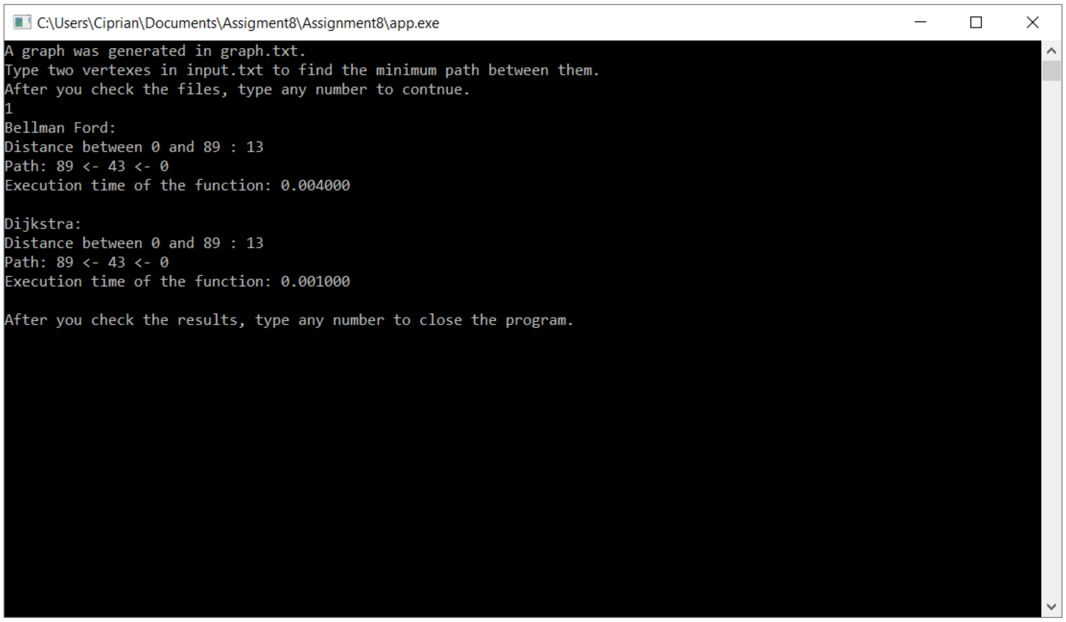
*dist - An array used to store the distance between the source vertex and any other vertexes.

Also, in the tools.h library I declared two structures. First is the structure *edge* which contain details about an edge (source vertex, destination vertex, the weight of the edge). The second is the structure *graph* which contain details about the used grah (number of nodes, number of edges, adjacency matrix and an array of all edges stored in graph).

3.7 Experiments

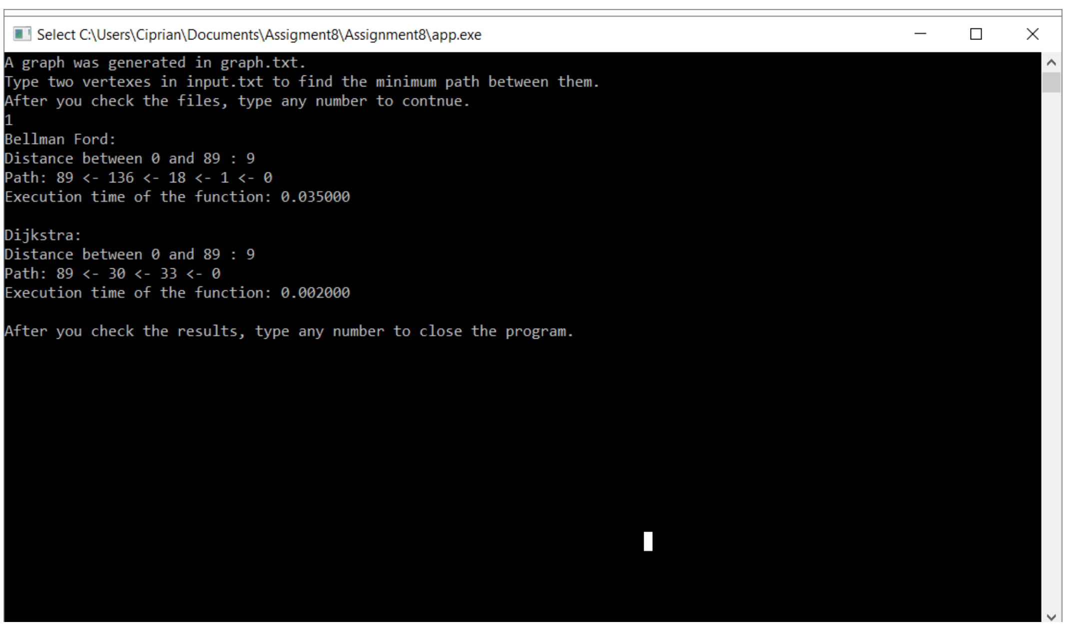
I ran the program few times to see the difference between the execution time of the two algorithms.

For a random generated graph with 102 vertexes:



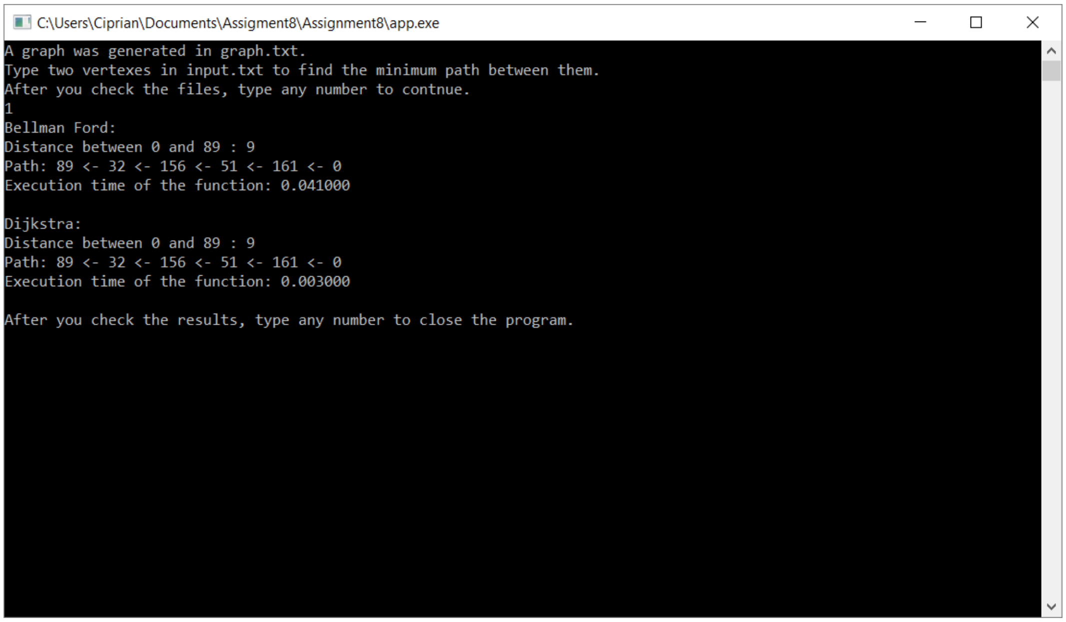
```
C:\Users\Ciprian\Documents\Assigment8\Assignment8\app.exe
A graph was generated in graph.txt.
Type two vertexes in input.txt to find the minimum path between them.
After you check the files, type any number to contnue.
1
Bellman Ford:
Distance between 0 and 89 : 13
Path: 89 <- 43 <- 0
Execution time of the function: 0.004000

Dijkstra:
Distance between 0 and 89 : 13
Path: 89 <- 43 <- 0
Execution time of the function: 0.001000

After you check the results, type any number to close the program.
```

For a random generated graph with 217 vertexes:

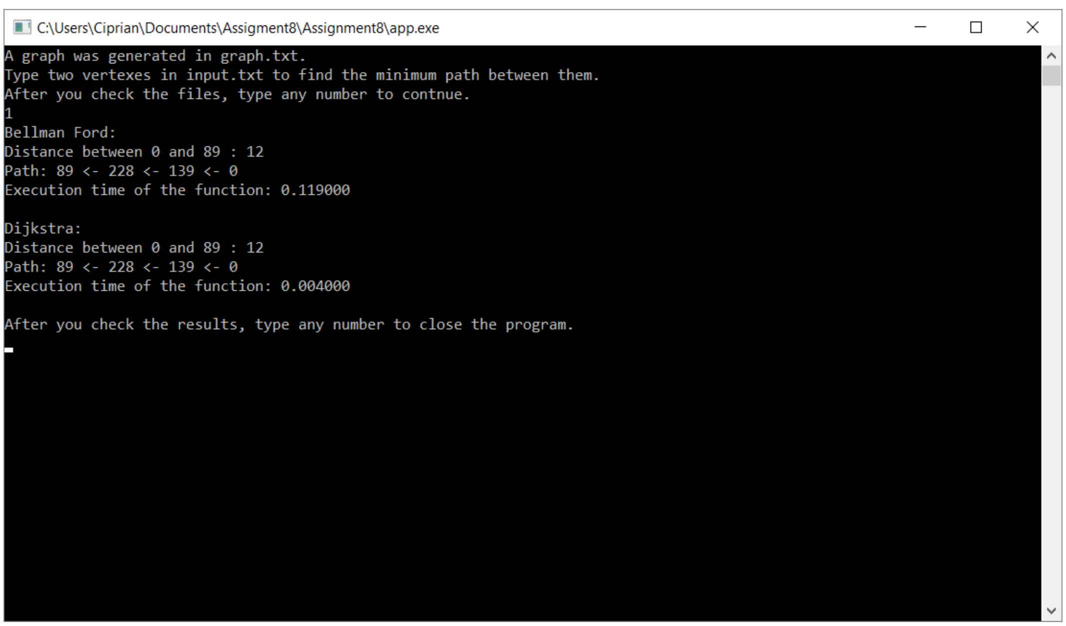


```
Select C:\Users\Ciprian\Documents\Assigment8\Assignment8\app.exe
A graph was generated in graph.txt.
Type two vertexes in input.txt to find the minimum path between them.
After you check the files, type any number to contnue.
1
Bellman Ford:
Distance between 0 and 89 : 9
Path: 89 <- 136 <- 18 <- 1 <- 0
Execution time of the function: 0.035000

Dijkstra:
Distance between 0 and 89 : 9
Path: 89 <- 30 <- 33 <- 0
Execution time of the function: 0.002000

After you check the results, type any number to close the program.
```

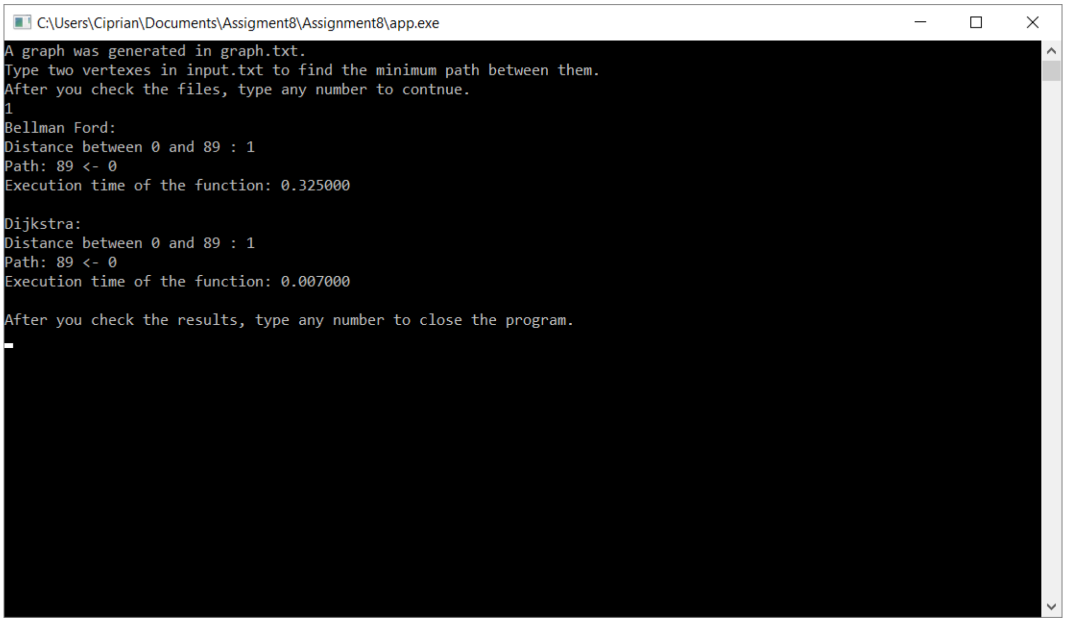
For a random generated graph with 269 vertexes:



For a random generated graph with 341 vertexes:



For a random generated graph with 478 vertexes:



From the above pictures and other tests I ran You can observe easily the difference between the execution time of the two algorithms, namely Dijkstra is faster than the Bellman-Ford implementation because Dijkstra implementation has a greedy programming approach and Bellman-Ford implementation is a dynamic approach which offers an optimal solution, but it requires a higher execution time.

4 Conclusion

After this project I gained a better understanding of the graph theory.

The most challenging part of the assignment was the adapting of the algorithms to store and print the path between the two vertexes.

I would like to use one on this algorithms in future for a video game with a minimap or an app based in gps locations. Also, I am interested in the implementation of graphs and this algorithms in other programming languages.

5 References

Book:

Name : Totul despre C si C++
Year of publication :2005
Publisher :Teora
Author :Dr. Kris Jamsa Lars Klander

Web references:

1.$http://www.geeksforgeeks.org$
2.$https://www.sharelatex.com/learn/Main_Page$
3.$https://www.thecrazyprogrammer.com$