COURSE ASSIGNMENT
Problem 11

Draghici Bogdan
The Faculty of ACE, Craiova
First Year, Calculatoare romana
CR 1.2.A

June 2, 2019

Contents

1 Problem statement

George is a truck driver, he needs to transport cargo from Craiova to Paris in a given limited amount of time. In order to avoid traffic jams he needs to determine two alternative routes between every two intermediary consecutive points on his route.
Write an application that allows George to establish the shortest route (the least amount of intermediary points), with given source and destination points, such that between two consecutive intermediary points there are two alternative routes.
Also, all the connections between the points on the map are given. The application should implement two different algorithms to solve the above problem.

Input: Read the number of intermediary points (nodes) and the number of routes (edges) of the map, then the starting and the end points, and last the displacement of the routes.

Output: Write the length of the shortest route and the path from the starting point to the ending point.

2 Pseudo-code description of the algorithms used

2.1 Creating and adding the edges of graph

In order to create the graph, I used used in one of the two algorithms three structures: structure graph having the number of vertexes *no-nodes* and array of pointers to the structure AdjList that is memorizing for each node the beginning of the list of neighbors, having as parameter the structure AdjListNode that is having as parameters the value of adjacent node and a pointer to the next adjacent one. In order to create the graph, I used the next functions:

Function CreateGraph(no-nodes)

1. **for** i=1, no-nodes, **execute**
2. head of array[i] of graph= NULL
3. **return** graph

Function AddRoute(graph, src, dest)

1. adress of newNode = newAdjListNode(dest)
2. next of newNode = head of array[src] of graph
3. head of array[src] of graph = newNode
4. adress of newNode = newAdlListNode(src)
5. next of newNode = head of array[dest] of graph
6. head of array[dest] of graph = newNode

Function AdjListNode(info)

1. dest of newNode = info
2. next of newNode = NULL
3. textbfreturn node

The function *AdjListNode* is adding the info of a new node to the adjacency list of a node. The function *AddRoute* is using the previous function to point to it's values' address and at it to the array of the graph. The function *CreateGraph* just creates the graph, memorize the number of nodes of the graph and initialize the adjacency list with NULL.

2.2 The DFS implementation

In order to find how many alternative routes are between two consecutive intermediary points, I implemented the Depth First Search Algorithm.
FunctionDFS(graph, source, target, route[])

1. counter = 0
2. **if** source == target
3. **return** 1
4. route[source] = 1
5. aux = head of adjacency list of source
6. **while** aux has value
7. **if** route[value of aux] **not visited**

8. counter = counter + DFS(graph, value of aux, target, route)
9. aux = next element in the adjacency list
10. **return** counter

In the variable counter I memorize the number of paths found that go to the target. If the starting point *source* is equal with the ending point *target*, then we found a new route and return 1(lines 2 and 3). Next we mark node source as visited and initialize the aux with the head of adjacency list of source node(lines 4 and 5). Till all the adjacent nodes are visited, every time we find one that was not visited, I recall the function from the current node in the adjacency list(lines 6, 7 and 8). In the end, return the value of counter.

2.3 The main algorithms

The algorithms I used to solve this problem are Dijkstra's algorithm using adjacency matrix and Bellman-Ford algorithm using adjacency matrix.

2.3.1 Dijkstra's Algorithm

Function Dijkstra(input, output)

1. **read** no-nodes, no-edges, source, finish
2. **for** i=0, no-edges, **execute**
3. **read** x, y
4. AddRoute(graph,x,y)
5. **initialize** dist[] = no-nodes, visited[] = 0, path[] = 0, prev[i] = -1
6. current-node = source
7. mark source as visited and set distance of source as 0
8. ok = no-nodes
9. **while** target not visited and ok !=0 **do**
10. min = no-nodes, m = -1, ok = ok - 1
11. aux = head of adjacency list of current-node
12. **while** value of aux **do**
13. **if** value of aux not visited
14. **initialize** route[]= 0
15. **if** dist[current-node] + 1 ¡ dist[value of aux] **and** DFS(graph, current-node, value of aux, route) ¿ = 3
16. dist[value of aux] = dist[current-node] + 1
17. prev[value of aux] = current-node
18. **if** min ¿ dist[value of aux]

```
19.                     min = dist[value of aux]
20.                     m = value of aux
21.             aux = next of aux
22.         if m == -1
23.             ok = 0, break
24.         current-node = m, mark m as visited
25. if ok
26.         current-node = target, i = 0
27.         while current-node != -1
28.             path[i++] = current-node
29.             current-node = prev[current-node]
30.         write" The shortest path has 'dist[target]' edges and the path is :"
31.         while i
32.             i = i - 1, write 'path[i]'
33. else
34.         write"Then path can not be found with the given input"
```

First we read the number of nodes and edges, then the starting and ending nodes and the pairs of nodes that represent the edges and memorize them in the adjacency lists of the graph (lines 1-4). Next I initialize the distance to each node with a number bigger than the possible one, in this case no-nodes, because the path that uses all the nodes has no-nodes - 1 length, and the vectors that store either the element was visited or not and the path taken by the truck driver with 0, and the vector that memorizes the order of the nodes with -1. The current-node will become the starting point, we will mark it as visited and set it's distance at 0 (lines 6 and 7). The repetitive structure from line 9 will repeat while the target node was not visited and if it was not made more than no-nodes times. Set the minimum distance and no-nodes and the next element in the path as -1. The aux will take the position of the first element in the adjacency list of the current-node. While there are nodes to visit, we see it has already been visited or not(lines 12 and 13), and in case it was not, we reinitialize the vector route used in DFS with 0 and see if the distance from the current-node +1 compared with the one from it's adjacency list's node is smaller and if there are at least 3 alternative routes from one to the other. After this condition, we will have memorized the shortest distance from the two of them. Using the conditional clause from line 18 we are able to memorize the next node to use in m. On line 21 we go to the next node in the adjacency list. If there are no nodes in the adjacency list, m will remain -1, in this case exiting the algorithm(lines 22 and 23). In the case m was modified, the current-node takes its value and mark it as visited. After exiting the repetitive structure, if the counter ok is not null, we recreate the path found

using the information stored in the vector prev, and then print the number of edges of the path and the nodes of the path output file (lines 25-32). In case ok is null, we print that it can not be found the path in the output file.

2.3.2 Bellman-Ford Algorithm

Function BellmanFord(input, output)

```
1.  read no-nodes, no-edges, source, finish
2.  for i = 0, no-edges, execute
3.      read edges[i].src, edges[i].dest
4.      graph[edges[i].src][edges[i].dest] = graph[edges[i].dest][edges[i].src]
    = 1
5.  initialize distance[] = no-nodes
6.  for i = 0, no-nodes, execute
7.      for j = 0, no-edges, execute
8.          if distance[edges[j].src] + 1 ¡ distance[edges[j].dest]
9.              initialize route[]= 0
10.             if DFS(graph, current-node, value of aux, route) ¿ = 3
11.                 distance[edges[j].dest] = distance[edges[j].src] + 1
12.                 prev[edges[j].dest] = edges[j].src;
13. if distance[finish] == no-nodes
14.     write"The destination can't be reached with the given input"
15. else
16.     current-node = finish, path[distance[finish] = finish
17.     for i = distance[finish]-1], 0, execute
18.         path[i] = prev[current-node]
19.         current-node = prev[current-node]
20.     write" The vertex distance is 'distance[finish]' and the path is :"
21.     for i = 0, distance[finish], execute
22.         write 'path[i]'
```

First we read the number of nodes and edges, then the starting and ending nodes and the pairs of nodes that represent the edges and memorize them in a vector type structure with tow parameters: src and dest, and in an adjacency matrix (lines 1-4). On line 5 is initialized the distance to every node with no-nodes, because it can be maximum no-nodes - 1 when the path contains all nodes, the same principle being applied also on line 6. On line 7 is a repetitive structure that verifies each edge from the vector edges. If the distance in the destination node is bigger than the one in the source node + 1 (line 8), then the vector route is reinitialized with 0 and is used to verify if there are at least 3 routes from source node to destination node using the

DFS function(lines 9 and 10). In case there are, the distance from destination node takes the value of the source node + 1 and source node is also added in the vector prev on the position of destination node (lines 11 and 12). In case the distance for finish is equal with the number of node, then it means there is not even one path that fulfill the conditions and we print in the output file this conclusion (lines 13 and 14). In other case, I recreate the path using the vectors path and prev in a repetitive structure with the number of steps equal with the distance to the finish point. After this, I print in the output file the distance from starting point to the ending one and the path taken.

3 Algorithms' memory and computational complexity

Supposing that all nodes of the graph are connected by at least one route, the DFS function will have O(no-nodes) complexity, and this will repeat for each node in the adjacency list of the current node. Because the graph is not oriented, the medium value for the number of the adjacent nodes of a node is no-edges * 2 / no-nodes.
For **Dijkstra's algorithm**, for each visited node (repetitive structure from line 12) we will have O(no-nodes * no-edges * 2 / no-nodes) = O(no-edges * 2) complexity. The total complexity depends on how fast the we visit the target node. The memory allocated for this algorithm is 2 * no-edges + no-nodes for memorizing the graph, and 5 * no-nodes for the vectors used in the algorithm, in total 2 * no-edges + 6 * no-nodes memory allocated.
For **Bellman-Ford algorithm**, the total complexity of the algorithm is O(no-nodes * no-edges * 2 / no-nodes * no-nodes) = O(2 * no-nodes * no-edges). This function does not have best and worst case scenarios, it is executed for an exact number of steps. For this algorithm is allocated no-nodes * no-nodes memory for the graph, 2 * no-edges for the array that contains the edges of the graph and 4 * no-nodes for the vectors used for solving the problem, in total no-nodes * (no-nodes + 4) + 2 * no-edges memory allocated.

3.1 Best case scenario

For Dijkstra's algorithm, the best case scenario will be to visit the target node right after the first search. In this case the complexity would be O(1 *no-edges * 2) = O (2 * no-edges).

3.2 Worst case scenario

For Dijkstra's algorithm, the worst case scenario would be to visit the last time the target node from all the no-nodes nodes. In this case, the complexity of the algorithm is O(2 * no-edges * no-nodes), equal with the complexity of the Bellman-Ford algorithm.

4 Experimental data

4.1 Data generation function

For the generation of the input data sets I used the rand() function from the library math.h, and also an function implemented by myself, that generates a random number from a given range.
Function random(min, max)

1. counter = max - min, number = 0
2. **while** counter != 0 **do**
3. number = number * RAND_MAX + rand()
4. counter = counter RAND_MAX
5. return number % (max - min + 1) + min

RAND_MAX is a constant which represents the biggest value that can be generated using the function rand(), and it's value is 32767.
On line 2 I initialized counter with the length of the range and the variable in which the number will be created with 0. On line 3 is a repetitive structure that will be executed for log(RAND_MAX) counter times, in order to obtain a number in the range. On line 4, the number is multiplied by RAND_MAX and also added a new random value generated by the rand() function. On line 5 the counter is divided by the maximum value that can be taken by a number generated in rand().
So, when the number is multiplied by RAND_MAX, the counter is divided by RAND_MAX, in order to get to the given range as quickly as possible. After the repetitive structure ends, the number created might exceed the range length, so we must do the rest of the division with the range length+1, then add the first value of the range in order to obtain a number between min and max.
For the input, I created a function called generator that creates a given number of input tests using the function random previously explained.
Function generator(tests)

```
1. for i = 1, tests, execute
2.     no-nodes = random(min1, max1)
3.     no-edges = random(min1, max2) %(no-nodes at power 3/2)
4.     start = random(min1, max1) % no-nodes
5.     finish = random(min1, max1) % no-nodes
6.     write in test i no-nodes, no-edges, start, finish
7.     for j=1, no-edges, execute
8.         source = random(min1, max1) % no-nodes
9.         do
10.            destination = random(min1, max1) % no-nodes
11.        while source == destination
12.        write in test source, destination
```

First I use an a repetitive structure in order to write in the specific file the input test. After this I generate the number of nodes and number off edges of the graph(lines 2 and 3). I chose that the number of edges to be maximum no-nodes at power 3/2 because this way there are more chances to exist a route, and not exist one with distance 1. Next I generate the starting and the ending point of the route(lines 4 and 5). After this I generate the edges of the graph, making sure it is not an edge to the same node, and write them in the input file(lines 7-12).

4.2 Data interpretation and graphics

I used for both algorithms as abscise axis the produce between no-edges and no-nodes. It can be seen a similarity between the time needed to execute a certain test, and the produce between the number of nodes and number of edges of the graph.
After calculating the times to solve the problem with the 2 algorithms in an excel worksheet, I found that the Dijkstra's algorithm implemented with adjacency lists is approximate 10 times faster than the Bellman-Ford algorithm implemented with adjacency matrix.

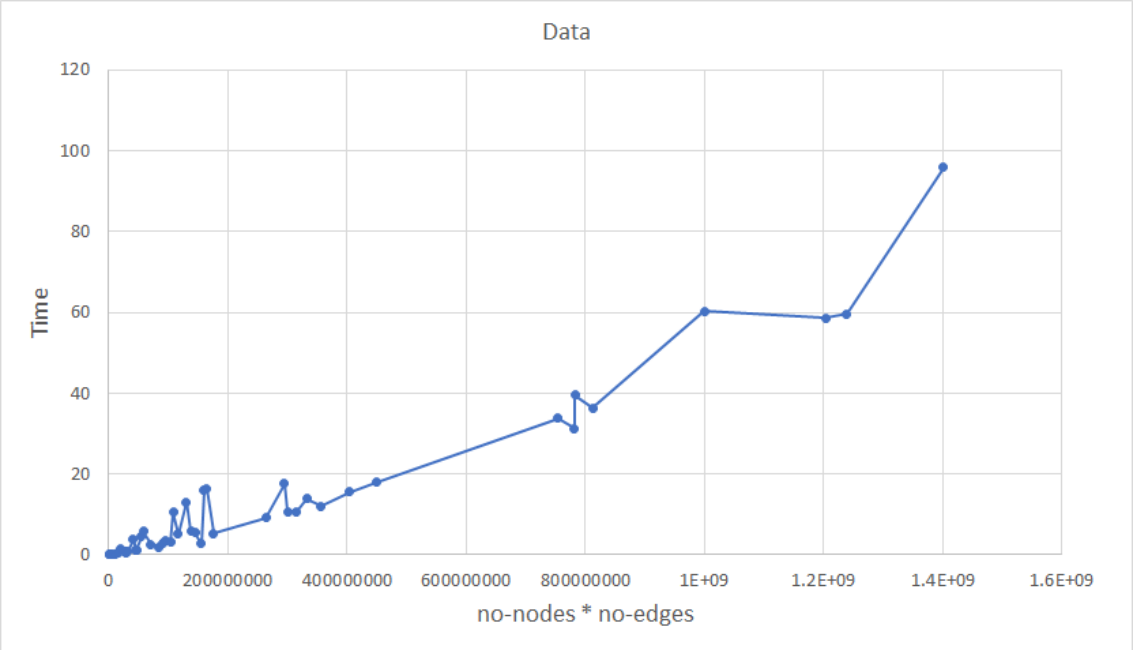

Figure 1: Dijkstra's algorithm

Figure 2: Bellman-Ford algorithm

5 Application design

5.1 The high level architectural overview of the application

The architecture is, in my opinion, quite complex for the Dijkstra's algorithm, and an easier to understand architecture in the Bellman-Ford algorithm, this one having only 4 functions in the second c file: the generator of the numbers previously explained, the generator of the tests, the DFS function and the main function that contains the algorithm. The Dijkstra's algorithm implementation contains, besides the previously mentioned functions, the functions specific to the graph represented with adjacency lists: newAdjListNode, AddRoute, CreateGraph, all already described in the second section.

5.2 The specification of the input

The input is the same for both algorithms, the input files containing the following in this exact order: number of nodes, number of edges, starting node and target node on the first line, and on all other lines are exactly 2 nodes that represent an edge of the graph.

5.3 The specification of the output

In the output file it is written the length of the path found on the first line, and on the second line is shown the path from the starting point to the ending one. In case the path was not found, the output will display instead this affirmation.

5.4 The list of all the modules in the application and their description

All modules used have already been described in the section 2 or 4 of this assignment. I will only enumerate them here: newAdjListNode, AddRoute, CreateGraph, generator, DFS, Dijsktra, AdjListNode, AdjList, Graph, BellmanFord, edge.

6 Results and Conclusion

6.1 Summary of achievements

By making this problem, I learned how to implement algorithms that find the shortest paths and a function similar with DFS that finds how many routes are between 2 nodes, and also how to create a python project and also **only** the basics of the python program.

6.2 Brief description of the most challenging achievement

The most challenging achievement was to successfully implement, after many tries, the Dijkstra's algorithm using an adjacency list for the graph.

6.3 Future directions for extending the lab homework

I hope that in the future I will be able to implement correctly the algorithm in python language.

7 References

1. https://cs.stackexchange.com/questions/109768/shortest-path-with-a-given-condition
2. https://www.codewithc.com/dijkstras-algorithm-in-c/
3. https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/
4. https://www.geeksforgeeks.org/graph-and-its-representations/
5. https://docs.microsoft.com/en-us/visualstudio/python/tutorial-working-with-python-in-visual-studio-step-01-create-project?view=vs-2019step-1-create-a-new-python-project
6. https://gist.github.com/econchick/4666413
7. https://markhneedham.com/blog/2013/01/18/bellman-ford-algorithm-in-python/