AN EFFECT OF USING A STORAGE MEDIUM IN DIJKSTRA ALGORITHM PERFORMANCE FOR IDEAL IMPLICIT PATH COST

Ibtusam Alashoury[1] and Mabroukah Amarif[2]

[1]Department of Computer Sciences, Faculty of Sciences, Sebha University, Sebha, Libya
[2]Department of Computer Sciences, Faculty of Information Technology, Sebha University, Sebha, Libya

ABSTRACT

The graph model is used widely for representing connected objects within a specific area. These objects are defined as nodes; where the connection is represented as arc called edges. The shortest path between two nodes is one of the most focus researchers' attentions. Many algorithms are developed with different structured approach for reducing the shortest path cost. The most widely used algorithm is Dijkstra algorithm. This algorithm has been represented with various structural developments in order to reduce the shortest path cost. This paper highlights the idea of using a storage medium to store the solution path from Dijkstra algorithm, then, uses it to find the implicit path in an ideal time cost. The performance of Dijkstra algorithm using an appropriate data structure is improved. Our results emphasize that the searching time through the given data structure is reduced within different graphs models.

KEYWORDS

Dijkstra algorithm, data structure, time complexity, implicit path, graphs

1. INTRODUCTION

The graph is a data structure of representing relationships between pairs of connected objects. Objects are defined as a set of nodes (Vertices) and the connection between these objects is denoted as edges that link these nodes. Each edge is marked with a weight value describing the cost between the connected nodes. There are two types of graph; directed graph for which each node is directed by one way to any other node and the order of vertices in the pairs is important. Undirected graph is a graph where all edges are bidirectional and it's possible to go to and back from the same way between two connected nodes. The issue of the shortest path problem is related to graph theory, which is one of the most important topics for researchers [1, 2, 3, 4, 5, 6, 7]. It concerns with finding the shortest path between two nodes, or between a node as a source to all other nodes, depending on the weights of the edges that link these nodes [8]. The graph theory and shortest path are used widely especially in the practical applications of various fields such as mapping transportation, electrical engineering and computer networks [9, 10, 11, 12].

Various algorithms have been created to find the shortest path within graphs. These algorithms depend on the graph and path type. The most common algorithm is Dijkstra algorithm which is used for finding the shortest path between source and destination [13], others are Bellman, the Johnson, and the Floyd Warshall algorithms [14].

In the last decade, the number of the rapid development and handling of the huge amount of road networks data and their development methods have been increased, researchers try to improve the

previous algorithms or develop their own in order to find out the typical shortest paths [4, 7, 9, 10, 11, 12]. They are also exploring more various ways and methods to get the lowest cost of path into a large number of nodes and speeding up their search process. Some of them are working on reconstructing the graphs in an attempt to reduce the cost of searching time either by compression, optimization or subgraphs [1, 2, 3, 4, 5, 6, 15]. Others are replacing data structure with another [18, 19, 20, 21].

Moreover, the area becomes an active filed for researchers to discover more ways and mechanisms in an attempt to get the lowest search cost within a large number of nodes and accelerate the search process which is a challenge until this time. This paper proposes an improvement of Dijkstra algorithm using a special data structure (linked hash map) for storing the solution path (shortest path given by Dijkstra algorithm) to optimize the searching time for the implicit paths. The proposed algorithm uses Dijkstra algorithm with priority queue implemented by min heap to find the solution path (shortest path). The solution path is stored into data structure of array list contains of a linked hash map elements. We have tested the proposed algorithm with different graphs sizes up to 10000 nodes of directed and undirected graphs. The following section describes the research background and literature review related to our paper while section 3 explains the algorithm description. Analysis and results are described in section 4. A discussion of this paper is explained in section 5 and the conclusion is provided in section 6.

2. RESEARCH BACKGROUND

Dutch computer scientist Edsger Dijkstra have designed the first searching algorithm for shortest path in 1956. He published his idea in 1959 [13]. His algorithm then was named by his surname and become the base in the area of finding shortest path from single source to a node destination, or multiple nodes destination within a graph [16, 17, 22]. It can also be used to find the shortest route costs from the source to the destination by stopping the algorithm once the shortest route is set to the specific target.

This algorithm has been considered by many researchers to improve the shortest path cost by minimizing the searching time (time complexity) using different data structure. Jain et. al. improve the Dijkstra algorithm by using priority queue and linked list [18]. It has been noticed that by using a graph represented by thier adjacncey lists and the priority queue implemented as a min-heap, the time efficiency is in $O(|E| \log |V|)$, where V is the number of nodes and E is the number of edges which connected these nodes. if the priority queue is implemented using an advanced data structure called the Fibonacci heap, the time becomes $O(|V| \log V + E)$, and its imporved [16].

Time efficiency could be imporved by exploit the solution path to get the implicit paths if they are queried again. From the literature, most algorithms may provide this feature, but there is no such explanation or imporvement of it. A suitable data structure could improve the time efficiency for the whole algorithm if used propably for storing the solution path and then search for the implicit path. In this paper, we propose for an improved algorithm based on Dijkstra algorithm by using a special data structure (linked hash map) to store the solution path. The proposed algorithm is tested with different number of nodes. Results are recorded to justify the Algorithm validity.

3. THE PROPOSED ALGORITHM DESCRIPTION

The idea of Dijkstra algorithm is genius and amazing. In addition, it's simple and easy to understand especially by using non complicated data structure such as priority queue implemented as a min-heap. Although Fibonacci heap has achieved greater success and better performance, however, it's complicated and often theoretical more than practical issues [16].

Based on all of that, we use Dijkstra algorithm with priority queue implemented as a min-heap. For storing the solution path, we use a linked hash map within array list. This can give us a fast searching time equal to O(1) as an efficiency time. The following steps explain the main idea of Dijkstra algorithm [16].

DIJKSTRA. (G,w,s)
1 INITIALIZE-SINGLE-SOURCE.(G, s)
2 S=Ø;
3 Q=G.V
4 while Q ≠ Ø ;
5 u=EXTRACT-MIN(Q)
6 S=SU{u}
7 foreach vertex v €G.Adj[u]
8 RELAX (u,v,w)

in step 1, the initialization of the source node s in the graph G is carried out. Step 2 initializes the set S to the empty set. The algorithm maintains the invariant that Q = V - S at the start of each iteration of the while loop of step 4 until step 8. Step 3 initializes the min-priority queue Q to contain all the vertices in V ; since S = Ø at that time, the invariant is true after step 3. Each time through the while loop of steps 4 until step 8, step 5 extracts a vertex u from Q = V - S and step 6 adds it to set S, for the first time through this loop, u = s. Vertex u, therefore, has the smallest shortest path estimate of any vertex in V - S. Then, step 7 and step 8 relax each edge (u,v). Notice that w is the wieght of the given edge.

We have built the storage data structure and identified a variant of the type of array-list which contains of a set of linked-hash-map (each value within the array-list is actually linked-hash-map includes a particular path) to store any path that has been queried .The hash-map is used to store the nodes and the distance between them. Each hash-map consists of (Key, Value). Each node and adjacent is stored in the key, while the distance between them is stored in Value. The following code describes the idea of storing and searching operations of the storage data structure.

```
//storing operation
public static  ArrayList<LinkedHashMap<Integer,Integer>> AddtoPaths(List<Integer> path,int srcc,int destt)
{
     if(path!=null)
   {
       if(AddHashMap.size()!=0)
        AddHashMap.clear();
       for (int i = 0; i < path.size() ; i++)
         {   AddHashMap.put(path.get(i),i); }
      IndexHashMap=(LinkedHashMap) AddHashMap.clone();
       AddHashMap.clear();
     if(!dublecateData(IndexHashMap,paths1,srcc,destt))
   {
       paths1.add(IndexHashMap);
      Addtolinked1(IndexHashMap,paths1);
   }
 }
       return paths1;
 }
```

```
public static boolean dublecateData(LinkedHashMap<Integer,
Integer>IndexHashMap1,ArrayList<LinkedHashMap<Integer, Integer>> paths11,int srcc1,int destt1){
    for(LinkedHashMap<Integer, Integer> p:paths11){
            if(p.equals(IndexHashMap1)){
                    return true;
            }
        }
    for(LinkedHashMap<Integer, Integer> s:paths11)
        {
         if( s.containsKey(srcc1) &&  s.containsKey(destt1) )
         {  return true;     }
        }
        return false;
    }
public static void Addtolinked1(LinkedHashMap<Integer,Integer>
path,ArrayList<LinkedHashMap<Integer,Integer>> paths1){
    int ii=1;
    int i;
    if(paths1.size()>0 ){
    i=paths1.size()-1;
    }else{i=paths1.size();}
    if(path!=null){
            comp = new ArrayList<Integer>(path.keySet());
    for (int j = 0; j < comp.size() ; j++) {
    if(Number_of_Nodes.containsKey(comp.get(j)) && Number_of_Nodes.get(comp.get(j))!=null
    &&      Number_of_Nodes.get(comp.get(j)).size()>0)
    {
      Number_of_Nodes.get(comp.get(j)).add(i);
        }else{
        Number_of_Nodes.put(comp.get(j),new ArrayList<Integer>());
         Number_of_Nodes.get(comp.get(j)).add(i);
      }
    } comp.clear();
    } }

    //searching operation
    public static ArrayList findShortestPathsDS(int src,int dest,Graph graph)
    {      start_time2 = System.nanoTime();
        ArrayList subPath = new ArrayList();
       Number_of_Nodes
      From=Number_of_Nodes.get(src);
       To=Number_of_Nodes.get(dest);
     if(MainClass.From!=null && MainClass.To!=null && MainClass.From.size()>0 &&
    MainClass.To.size()>0)
        {
            SO=FoundContains(From,To);
        } else
        {  time11=0.0;
          text = "\n The path is not implicit, So The path will be calculated using Dijkstra";
          subPath= (ArrayList)graph.findShortestPaths(src, dest);
          end_time2 = System.nanoTime();
           difference2 = (end_time2 - start_time2) / 1e6;
```

```
              time11=Graph.time;
               return subPath;
            }
       if(SO >=0)
       {
         if( (paths1.get(SO).containsKey(src)) &&  (paths1.get(SO).containsKey(dest)) )
              {
                 found=true;
                  i=SO;
              }
          if(found)
            {
               src1=paths1.get(SO).get(src);
              dest1=paths1.get(SO).get(dest);
              ss1=new ArrayList(paths1.get(SO).keySet());
           if (src1 <= dest1)
           {
              for (ii = src1; ii <= dest1; ii++ )
              { subPath.add(ss1.get(ii));  }
          }else{
               for (ii = src1; ii >= dest1; ii--)
                  {
                    if (ii >=0)
                    subPath.add(ss1.get(ii));
                  }
                }
            end_time2 = System.nanoTime();
            difference2 = (end_time2 - start_time2) / 1e6;
        } else{
                 time11=0.0;
                 text = "\n The path is not implicit,So The path will be calculated using Dijkstra";
                 subPath= (ArrayList)graph.findShortestPaths(src, dest);
                 end_time2 = System.nanoTime();
                difference2 = (end_time2 - start_time2) / 1e6;
                time11=Graph.time;
            }
             return subPath;
          } else{
                 time11=0.0;
                text = "\n The path is not implicit,So The path will be calculated using Dijkstra";
                subPath= (ArrayList)graph.findShortestPaths(src, dest);
                end_time2 = System.nanoTime();
               difference2 = (end_time2 - start_time2) / 1e6;
               time11=Graph.time;
       }
           return subPath;
    }
public static Integer FoundContains(ArrayList<Integer> From11,ArrayList<Integer> To11)

{
     int SO1=-1;
     To12.clear();
```

```
 From12.clear();
   if(From11.size()>0 && To11.size()>0)
  {
     if(From11.size()>=To11.size())
      {
        for (int i = 0; i <To11.size(); i++)
          { To12.put(To11.get(i), i); }
        for (int i = 0; i <From11.size(); i++)
         {
           if(To12.containsKey(From11.get(i)))
             {
               SO1=From11.get(i);
               break;
             }
         }
        }else{
          for (int i = 0; i <From11.size(); i++)
           {   From12.put(From11.get(i), i); }
         for (int i = 0; i < To11.size(); i++)
          {
          if (From12.containsKey(To11.get(i)))
           {
             SO1=To11.get(i) ;
             break;
           }
         }
        } return SO1;
   } return SO1;   }
```

4. THE ALGORITHM ANALYSIS AND RESULTS

The time complexity of hash-map, linked-hash-map and array-list are different according to the kind of operations. Table 1 and 2 describes the time for each given data structure.

Table 1. The time complexity of hash-map and linked-hash-map

	get	Contains Key	next	note
Hash-map	O(1)	O(1)	O(h/n)	h is the table capacity
Linked-hash-map	O(1)	O(1)	O(1)	

Table 2. The time complexity of array-list

	get	add	contains	next	remove	Iterator_remove
Array-list	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)

According to the previous tables, n means the number of nodes (V), and m means the number of edges (E). The total time for putting the new solution path in the given data structure is:
$O(n\log n) + O(n) + O(n\log n) = O(n) + O(2n\log n) = O(2n\log n) = O(n\log n)$

In the other side, the total time complexity of the query from the data structure is:
$O(\log n)+O(\log n)+O(n\log n)+O(\log n)+O(\log n)$
$=O(4\log n)+O(n\log n)$

=O(logn)+O(nlogn)
=O((1+n)logn) =O(nlogn)

We run the original Dijkstra algorithm and the proposed algorithm with different type of graph; directed graph and undirected graph. We also run both of them with different number of nodes in order to get the shortest path between a given source node and destination. After the solution path is found, it's stored in the given data structure (array-list of linked-hash-map). Then, an inquiry for the implicit path is taken place. If the implicit path is found within the stored solution path in the given data structure, the time is calculated and recorded, else; the searching using the original Dijkstra algorithm is started again and the total time is recorded. The averages of each recorded times are calculated. These operations are repeated many times with different number of nodes.

Table 3 and 4 contain the averages values of the time when the required path is the implicit path of the stored solution path and the graph is directed, and figures 1 and 2 shows the results analysis.

Table 3. The run time of the implicit path (directed graph)

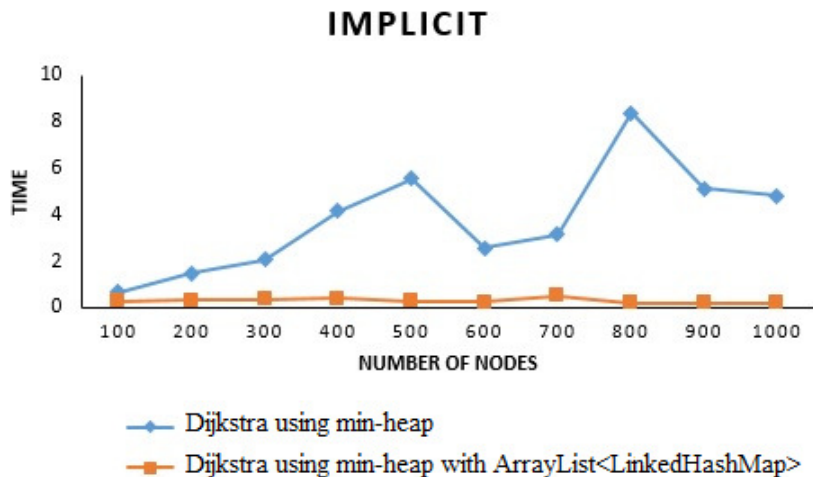| No. of nodes	Dijkstra using only min-heap Time average	Dijkstra using min-heap with ArrayList<LinkedHashMap> Time average
100	0.66256	0.28489
200	1.47381	0.3076
300	2.06859	0.34078
400	4.15724	0.38133
500	5.56397	0.28971
600	2.552	0.24657
700	3.15004	0.51842
800	8.35776	0.22292
900	5.11631	0.21436
1000	4.81021	0.2146

Figure 1. The run time of 100-1000 number of nodes (directed graph)

From figure 1, we notice that the time decreases as the number of nodes increases using the proposed algorithm. Time average is equal to 0.28489 when the number of nodes is 100, then

time increases with unobserved amount. It gives the highest value when 700 nodes and starts to decrease at 800 nodes and above.

Table 4. The run time of the implicit path (directed graph)

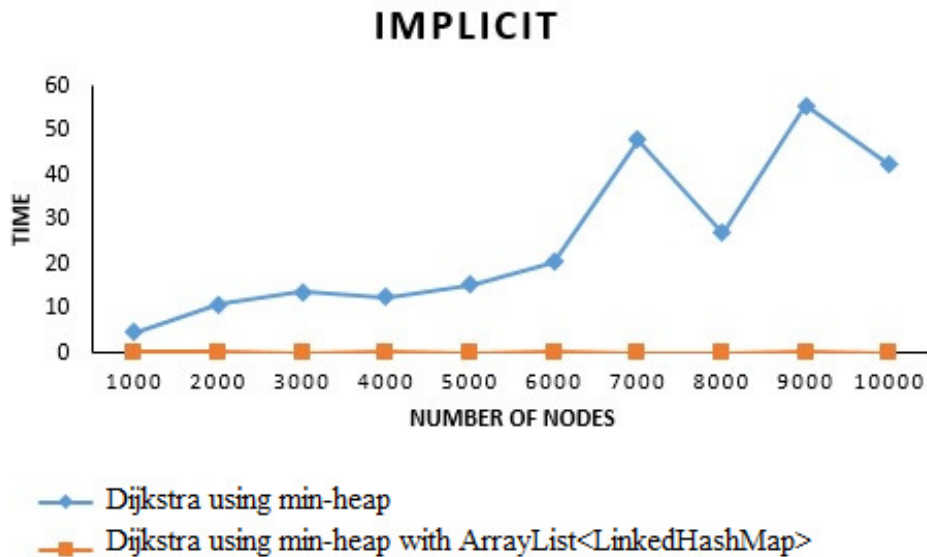| No. of nodes	Dijkstra using only min-heap Time average	Dijkstra using min-heap with ArrayList<LinkedHashMap> Time average
1000	4.81021	0.2146
2000	11.03552	0.18769
3000	13.76008	0.17272
4000	12.66174	0.19816
5000	15.29122	0.16887
6000	20.46555	0.18637
7000	47.87112	0.1797
8000	27.05117	0.17704
9000	55.55694	0.18778
10000	42.26821	0.17102

Figure 2. The run time of 1000-10000 numbers of nodes (directed graph)

Looking at figure 2, nodes numbers are various from 1000 to 10000 nodes. We realize that the time decreases as the number of nodes increase and that improved the proposed algorithm validity. We observe a linear time for the given data structure; ArrayList<LinkedHashMap>.

Table 5 and 6 contain the averages values of the time when the required path is not the implicit path of the stored solution path and the graph is also directed, and figure 3 and 4 show the results analysis.

Table 5. The run time of the non-implicit path (directed graph)

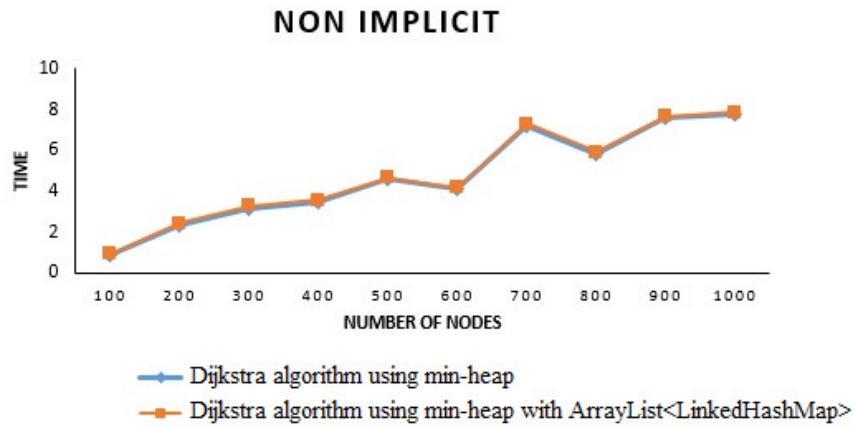| No. of nodes	Dijkstra using min-heap	Dijkstra using min-heap with ArrayList<LinkedHashMap>
100	0.86681	0.92647
200	2.30324	2.41939
300	3.15914	3.27483
400	3.45716	3.54365
500	4.5522	4.64095
600	4.08183	4.17757
700	7.1498	7.26413
800	5.8222	5.90921
900	7.55844	7.65247
1000	7.73631	7.84603

Figure 3. The run time of 100-1000 number of nodes (directed graph)

Figure 3 describes the run time of non-implicit path if it's not included within the stored solution shortest path. The relationship between run time and nodes numbers is shown. It is observed that the time for the proposed algorithm is almost equal to the original Dijkstra algorithm with min-heap. There is a slightly fixed difference as the nodes numbers increases.

Table 6. The run time of the non-implicit path (directed graph)

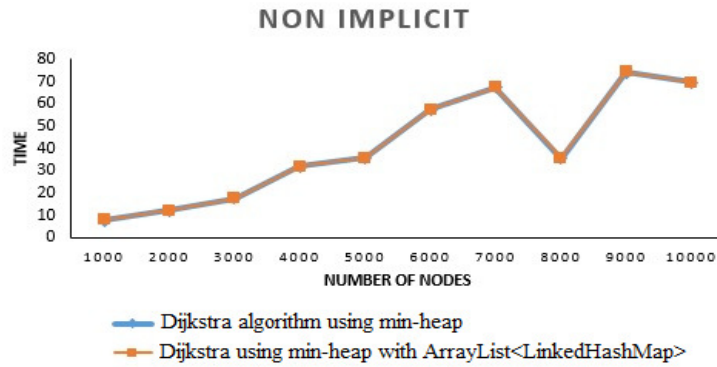| No. of nodes	Dijkstra using min heap	Dijkstra using min heap with ArrayList<LinkedHashMap>
1000	7.73631	7.84603
2000	11.90012	12.02769
3000	17.38345	17.50731
4000	31.68681	31.79614
5000	35.59574	35.68865
6000	57.20521	57.31768
7000	67.07719	67.16318
8000	35.34394	35.4589
9000	73.9846	74.109
10000	69.43308	69.54271

NON IMPLICIT

Figure 4. The run time of 1000-10000 number of nodes (directed graph)

From figure 3 and 4, we notice that almost the same time is observed for searching for a path which is not a sub path from the solution shortest path.

With respective to an undirected graph, table 7 and 8 contain the averages values of the time when the required path is the implicit path of the stored solution path and figure 5 and 6 show the results analysis.

Table 7. The time complexity of the implicit path (undirected graph)

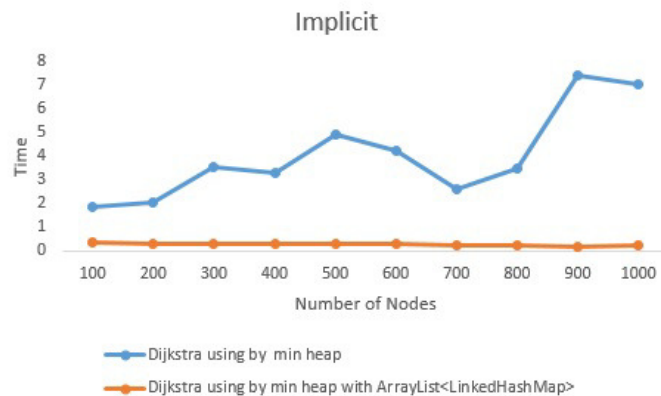| No. of nodes	Dijkstra using min heap	Dijkstra using min heap with ArrayList<LinkedHashMap>
100	1.84947	0.33223
200	2.05475	0.28836
300	3.52372	0.30657
400	3.28394	0.27613
500	4.92279	0.2832
600	4.20319	0.29005
700	2.61191	0.24252
800	3.49764	0.21569
900	7.41144	0.2038
1000	7.00862	0.23575

Implicit

Figure 5. the chart of time complexity of 100-1000 number of nodes (undirected graph)

From figure 5, we notice that the time decreases as the number of nodes increases. It's almost same as the directed graph. We also notice that the time start to decrease as the number of nodes increase. There is a slightly difference between the directed graph and undirected graph but undirected graph is a bit better.

Table 8. The time complexity of the implicit path (undirected graph)

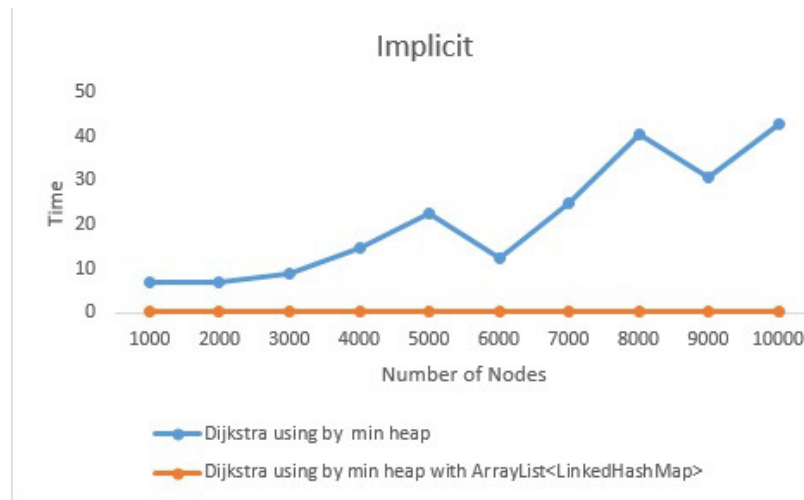| No. of nodes	Dijkstra using min heap	Dijkstra using min heap with ArrayList<LinkedHashMap>
1000	7.00862	0.23575
2000	7.0446	0.19993
3000	8.87373	0.20988
4000	14.7859	0.19355
5000	22.37447	0.17785
6000	12.27828	0.20967
7000	24.78728	0.18866
8000	40.37087	0.18704
9000	30.50427	0.17007
10000	42.68629	0.19351

Figure 6. the chart of time complexity of 1000-10000 number of nodes (undirected graph)

Figure 6, shows the average time of nodes from 1000 to 10000 nodes number. It's also clear that as the number of nodes increase, time is decrease.

Table 9 and 10 contain the averages values of the time when the required path is not the implicit path of the stored solution path and the graph is undirected, and figure 7 and 8 show the results analysis.

Table 9. The time complexity of the non-implicit path (undirected graph)

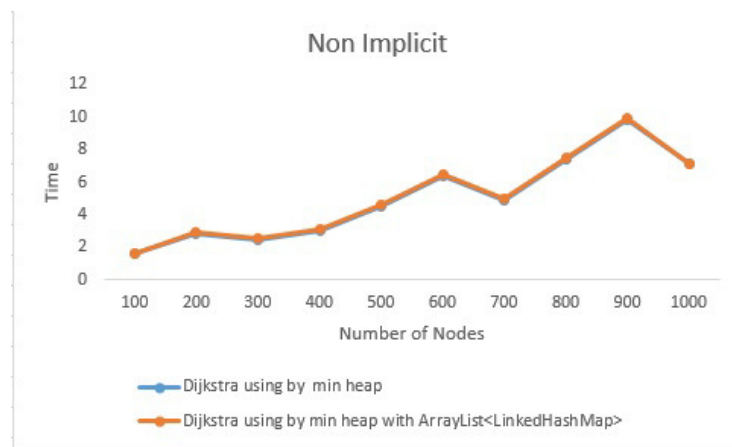| No. of nodes	Dijkstra using min heap	Dijkstra using min heap with ArrayList<LinkedHashMap>
100	1.60807	1.6634
200	2.87129	2.93079
300	2.47767	2.56841
400	3.056	3.15386
500	4.51615	4.61892
600	6.38728	6.46438
700	4.84882	4.93895
800	7.36835	7.48686
900	9.83962	9.94413
1000	7.09856	7.17292

Figure 7. the chart of time complexity of 100-1000 number of nodes (undirected graph)

Figure 7 and 8 describes the run time of non-implicit path if it's not included within the stored solution shortest path and the graph is an undirected. The relationship between run time and nodes numbers is shown. It is observed that the time for the proposed algorithm is almost equal to the original Dijkstra algorithm with min-heap. There is a slightly fixed difference as the nodes numbers increases.

Table 10. The time complexity of the non-implicit path (undirected graph)

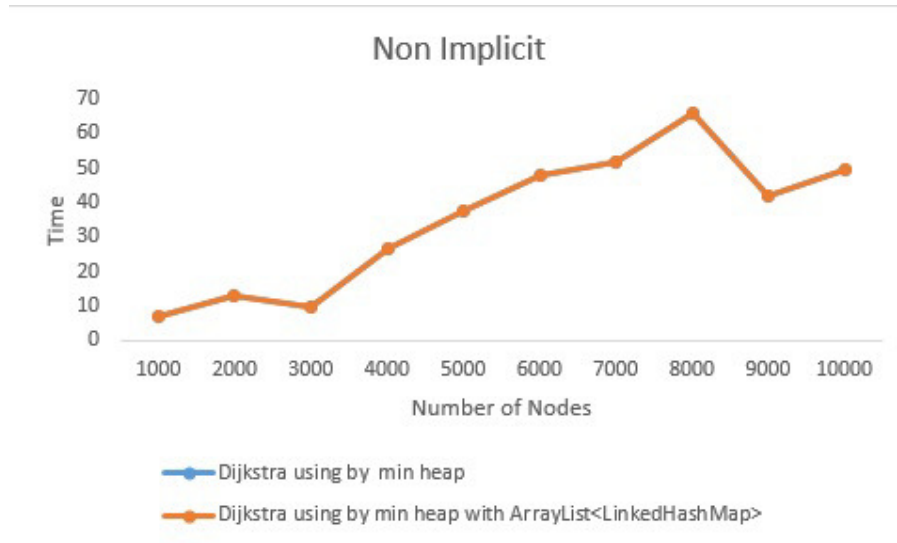| No. of nodes	Dijkstra using min heap	Dijkstra using min heap with ArrayList<LinkedHashMap>
1000	7.09856	7.17292
2000	12.75358	12.84875
3000	9.63923	9.73673
4000	26.47921	26.57234
5000	37.46062	37.53992
6000	47.93749	48.00873
7000	51.68053	51.78028
8000	66.05719	66.12449
9000	42.02263	42.11391
10000	49.61217	49.69425

Figure 8. the chart of time complexity of 1000-10000 number of nodes (undirected graph)

5. DISCUSSION

According to our case study of random directed graph and undirected graph, with different number of nodes, we have noticed that our proposed algorithm works better if the graph is undirected. The same case study has been given to the original Dijkstra algorithm with priority queue implemented as a min-heap. The results of both algorithms have been recorded and analysed.

Comparisons between these results have shown that the proposed algorithm is almost the best. Although more data structures have been used within the proposed algorithm, however, the enlarged storage is available for all of the current devices, even for the smallest one. We argue that data storages aren't problem if the performance of the given algorithm is higher and success.
It has been observed that the searching time of the original algorithm is almost the same as the time of the proposed algorithm if the required path is not an implicit path within the solution path. Regarding of the results in the literature, run time is various depending on the type of used data structure and also CPU speed. Although our results give time average equal to approximately 0.1 for searching within a solution path, we realize that the greater the number of nodes, time becomes less and less. This can work in a large storage of nodes especially in a huge road network.

6. CONCLUSION AND FUTURE WORK

The analysis of searching time for implicit path seems to be rare and less concern. Most available algorithms expect the nature of easy time searching for sub path within the solution path. Well, data structure plays main role in the whole procedures and operations. We realize this point when we start proposing our idea for improvement of the available Dijkstra algorithm. We consider improving the time complexity of implicit path within the solution path as a first starting point. According to our results, the improvement of the proposed algorithm is achieved with regard of directed graph and undirected graph. In fact, most practical applications involve both directed and undirected graphs. The proposed algorithm achieves the best result especially for a large number of nodes and if the graph is undirected. We also plan to apply our proposed algorithm to a real network road map to show the performance and improve the validity from a real point of view.

REFERENCES

[1]   Arman, N., & Khamayseh, F., (2015) "A Path-Compression Approach for Improving Shortest-Path Algorithms," *International Journal of Electrical and Computer Engineering,* vol. 5, p. 772,.

[2]   Broumi, S., Bakal, A., Talea, M., Smarandache, F., & Vladareanu, L., "Applying Dijkstra algorithm for solving neutrosophic shortest path problem," *International Conference on Advanced Mechatronic Systems (ICAMechS), 2016*, pp. 412-416.

[3]   Gao, J., Zhao, Q., Ren, W., Swami, A., Ramanathan, R., & Bar-Noy, A., (2015) "Dynamic shortest path algorithms for hypergraphs," *IEEE/ACM Transactions on Networking,* vol. 23, pp. 1805-1817.

[4]   Gutenschwager, K., Völker, S., Radtke, A., & Zeller, G., "The shortest path: Comparison of different approaches and implementations for the automatic routing of vehicles," in *Simulation Conference (WSC), Proceedings of the 2012 Winter*, 2012, pp. 1-12.

[5]   Khamayseh, F., & Arman, N., (2015) "Improvement of Shortest-Path Algorithms Using Subgraph's Heuristics". *Journal of Theoretical & Applied Information Technology,* vol. 76.

[6]   Niemeyer, K., E., & Sung, C.-J., (2016) "On the importance of graph search algorithms for DRGEP-based mechanism reduction methods," *Combustion and Flame,* vol. 158, pp. 1439-1443.

[7]   Shu-Xi, W., (2012) "The improved dijkstra's shortest path algorithm and its application," *Procedia Engineering,* vol. 29, pp. 1186-1190.

[8]   Douglas,W., B., (2001) *Introduction to Graph Theory*, Pernice Hall.

[9]   Chandra, "Shortest Path Problem for Public Transportation Using GPS and Map Service," 2012.

[10]   Saab, Y., & VanPutte, M., (1999) "Shortest path planning on topographical maps," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans,* vol. 29, pp. 139-150.

[11]   Xing, S., & Shahabi, C., "Scalable shortest paths browsing on land surface," in *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2010, pp. 89-98.

[12]   Faro, A., & Giordano, D., (2016) "Algorithms to find shortest and alternative paths in free flow and congested traffic regimes," *Transportation Research Part C: Emerging Technologies,* vol. 73, pp. 1-29.

[13]   Dijkstra., E., W., (1959) "A note on Two Problems in Connexion with graphs", Numerische Mathematik, 1, 269-271.

[14]   Thorat, S., & Rahane, S., (2016) "Review of Shortest Path Algorithm", IRJET, vol 3, issue 8.

[15]   Yao, B., Yin, J., Zhou, H., & Wu, W., (2016) "Path Optimization Algorithms Based on Graph Theory," *International Journal of Grid and Distributed Computing,* vol. 9, pp. 137-148.

[16]   Cormen, T., Leiserson, C., Rivest, R., & Stein, C., (2009) *Introduction to Algorithms*, 3rd. ed., MIT Press, London.

[17]   Levitin, A., (2012) *Introduction to the Design and Analysis of Algorithms*, 3rd ed., Pearson Education, Inc., Addison-Wesley.

[18]   Jain, A., Datta, U., & Joshi, N., (2016) "Implemented modification in Dijkstra" s Algorithm to find the shortest path for N nodes with constraint," *International Journal of Scientific Engineering and Applied Science,* vol. 2, pp. 420-426.

[19] Xie, D., Zhu, H., Yan, L., Yuan, S., & Zhang, J., "An improved Dijkstra algorithm in GIS application," in *World Automation Congress (WAC),* 2012, pp. 167-169.

[20] Lu, J., & Dong, C., "Research of shortest path algorithm based on the data structure," *in the 3rd International Conference of Software Engineering and Service Science (ICSESS), 2012 IEEE*, 2012, pp. 108-110.

[21] Kong, D., Liang, Y., Ma, X., & Zhang, L., "Improvement and Realization of Dijkstra Algorithm in GIS of Depot," *in the International Conference on Control, Automation and Systems Engineering (CASE), 2011*, 2011, pp. 1-4.

[22] Deng, Y., Chen, Y., Zhang, Y., & Mahadevan, S., (2012) "Fuzzy Dijkstra algorithm for shortest path problem under uncertain environment," *Applied Soft Computing,* vol. 12, pp. 1231-1237

AUTHOR

Ibtusam Alashoury: received her BSc degree in Computer Science from University of Sebha, Libya. She is currently doing her MSc in computer Sciences at Sebha University of Libya. She interests in the area of Software Engineering, System Analysis and Web design. she is currently working as a technical Engineer in the information development project of Sebha University of Libya.

Mabroukah Amarif: received her BSc degree in Computer Science from University of Sebha, Libya, MSc in Computer Science from Universiti Sains Malaysia, and PhD in Software Engineering from Universiti Kebangsaan Malaysia. Her interests span a wide range of topics in the area of Software Engineering, Networking, Computer Security, Visual Informatic, Computer Education and programming languages. She is currently working as Assistant Professor at the departement of computer science, Faculty of Information Technology in Sebha University of Libya.