A Path-Compression Approach for Improving Shortest-Path Algorithms

Nabil Arman*, Faisal Khamayseh**
* Department of Computer Science and Engineering, Palestine Polytechnic University, Palestine
** Department of Information Technology, Palestine Polytechnic University, Palestine

Article Info

Article history:

Received Mar 2, 2015
Revised Apr 23, 2015
Accepted May 18, 2015

Keyword:

Communication network
Graph Representation
Path Compression
Reverse Representation
Shortest Path

ABSTRACT

Given a weighted directed graph G=(V;E;w), where w is non-negative weight function, G' is a graph obtained from G by an application of path compression. Path compression reduces the graph G to a critical set of vertices and edges that affect the generation of shortest trees. The main contribution of this paper is finding shortest path between two selected vertices by applying a new algorithm that reduces number of nodes that needs to be traversed in the graph while preserving all graph properties. The main method of the algorithm is restructuring the graph in a way that only critical/relevant nodes are considered while all other neutral vertices and weights are preserved as sub paths' properties. Our algorithm can compress the graph paths into considerable improved percentage especially when the graph is sparse and hence improves performance significantly.

Corresponding Author:

Faisal T. Khamayseh,
Departement of Information Technology,
College of InformationTechnology and Cimputer Engineering
Palestine Polytechnic University,
Hebron, Palestine.
Email: faisal@ppu.edu

1. INTRODUCTION

Efficient approach of the single source shortest path problem on communication or transportation networks is extremely important requirement for real-world applications.

Since finding shortest paths over network topology is expensive, it is worthy to consider various techniques and heuristics that can help in improving the existing algorithms. The most well-known algorithm for finding a single-source shortest path is Dijkstra's algorithm [1]. A large variety approaches have been proposed attempting to improve the performance of shortest path algorithms using different assumptions and graph representations [2]-[10]. This paper addresses the value of the graph representation - in the form of compression graph- to improve the performance of the shortest path algorithm.

Finding the shortest path varies in time complexity upon the constraints is to be applied. Such examples are finding the single-source shortest path, single-source shortest path with the possibility of negative weights, k-shortest paths, single-pair using heuristics, all-pairs shortest paths, etc. These assumptions and constraints may require applying simple minimum spanning tree procedures to effectively find the shortest path, while other assumptions may require advanced algorithms such as Dijkstra's algorithm. Some variations and improvements based on tree structures have been presented in the literature. Example of such variations is the running time based on Fibonacci-heap min-priority queue which is O(|V|log|V|+|E|) assuming that w(e) is a nonnegative weight [1].

Recent research attempt to improve shortest path algorithm based on search strategy by introducing a constraint function with weighted value and ignoring the large number of irrelevant nodes during shortest path finding [11]. Some researchers have focused more on overcoming the network structure rather than the algorithm itself. References [12]-[14] presented an algorithm to find the shortest path through graph partitioning. They took an advantage of road network features to improve the search. The main feature is the possibility of partitioning the graph into a set of components or clusters. They focused on simplifying the detailed graph by clustering nodes that are near each other. In the final generated graph, the search is conducted near the start of the destination of the path and among the components on the transit edges.

The first section of this paper describes an existing technique of graph representation and how this technique works on path existence in directed weighted graphs [2]. The remaining sections present our algorithm and its improvements. This work presents a new improved variation of finding single source-destination shortest path by focusing on compression graph.

Let G=(V;E;w) be a directed graph, and G'=(V; E';w) the compressed graph which formed from G, where V is a set of vertices, E is a set of edges , E' a set of edges formed using compression function and w is the weight function, where w(e) > 0 for each edge e $\in$ E. Let each edge e has a non-negative weight. Assume <s> and <t> are given vertices where <s> and <t>$\in$V, <s> is the source vertex and <t> is the destination. The single pair source-destination shortest path is to find the path with the minimum cost sum of edges from source <s> to destination <t>.

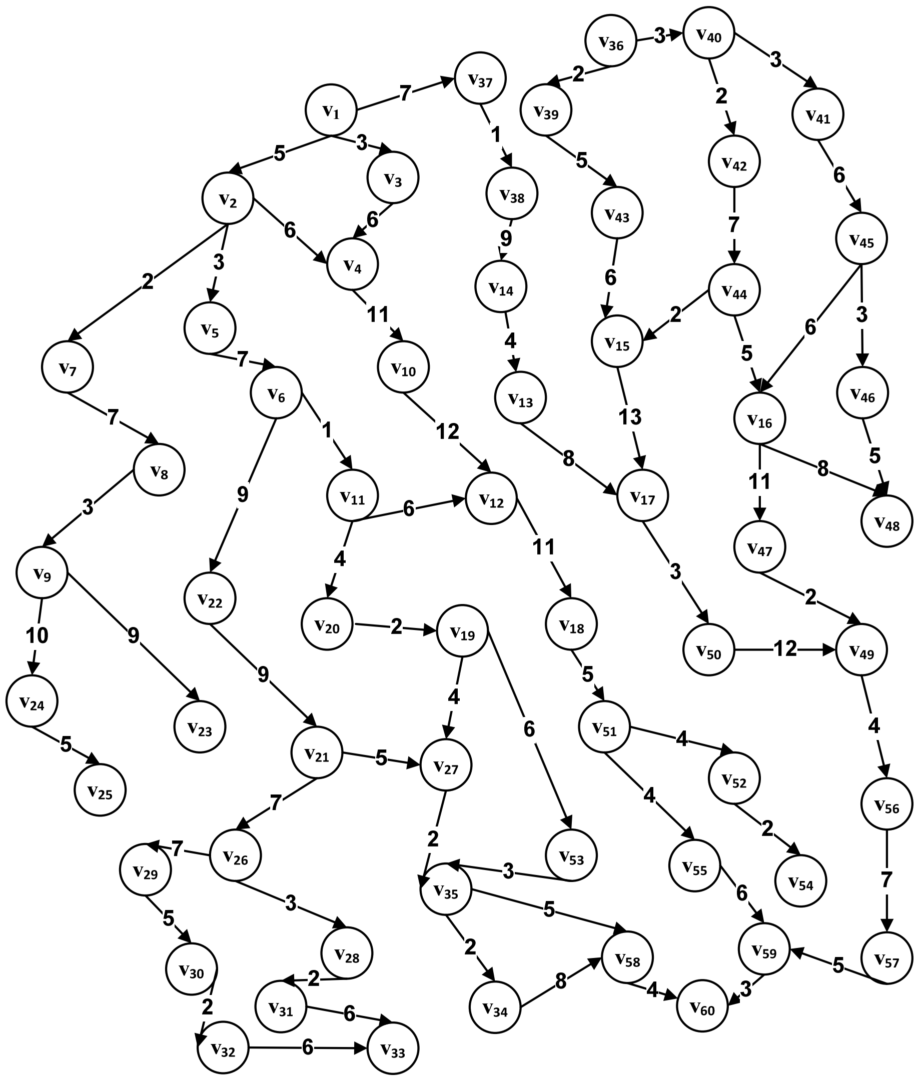

Figure 1. Graph G=(V;E;w)

2. REPRESENTATION AND DATA STRUCTURE

A graph G=(V;E;w) consisting of a set of |V| non-repeatable vertices, requires two matrices with maximum $|V|^2$ elements to represent the graph in normal and reverse representations [2]. Figure 1 depicts graph G=(V;E;w) which is represented in the matrix structure and linear array as shown in Table 1 and Table 3 respectively. These representations were used in developing parallel algorithms for the generalized same generation queries in deductive databases [4].

Table 1. Graph Matrix Representation for Graph G

--	0	1	2	3	4	5	6	7	8	9	10
0	**v1**	**v2**	**v4**	**v10**	**v12**	**v18**	**v51**	**v52**	**v54**		
1								**v55**	**v59**	**v60**	
2			**v5**	**v6**	**v11**	**0,4**					
3						**v20**	**v19**	**v27**	**v35**	**v58**	**1,9**
4									**v34**	**3,9**	
5								**v53**	**3,8**		
6					**v22**	**v21**	**v26**	**v28**	**v31**	**v33**	
7								**v29**	**v30**	**v32**	**6,9**
8							**3,7**				
9			**v7**	**v8**	**v9**	**v23**					
10						**v24**	**v25**				
11		**v3**	**0,2**								
12		**v37**	**v38**	**v14**	**v13**	**v17**	**v50**	**v49**	**v56**	**v57**	**1,8**
13	**v36**	**v39**	**v43**	**v15**	**12,5**						
14		**v40**	**v41**	**v45**	**v46**	**v48**					
15					**v16**	**v47**	**12,7**				
16						**14,5**					
17			**v42**	**v44**	**13,3**						
18					**15,4**						

The reverse matrix representation of the graph G is depicted in Table 2. This structure helps in finding all possible ancestor vertices starting from a destination node <ti>. For example, vertex <v60> is reachable from node <v1> via <v2, v4, v5, v6, v11, v20, v19, v53, v35, v34, v58>.

For efficient implementation and to save storage, the matrix can also be represented as a linear array with |E| entries.

The advantage of representing the graph in graph matrix is storing all paths from every possible vertex to all reachable vertices in the graph. This way of representation provides a set of benefits. It takes a linear time to check the path existence. It also helps in finding simple sub baths of vertices of in-degree and out-degree of 1. This representation also shows all graph roots and paths' ends in reference to a given source vertex is. Zero in-degree vertices are shown in the first column (the sources column). Paths are represented in as a depth first search traversal order while common parts of the paths are stored only once using array indexing to avoid duplications of subpaths. For example, if paths p1 is represented as vertices <v1,v2,vi,…,vn-1,vn> and p2 is <v1,v2,..,vi,..,vm>, P1 and P2 are present in the graph, then p2 is stored in the next row of p1 starting from the column (i+1) representing the rest of the p2 as <vi+1, vi+2, …> with empty i+1 entries.

Table 2. Reverse Matrix Representation for Graph

--	0	1	2	3	4	5	6	7	8	9	10	11
0	**v23**	**v9**	**v8**	**v7**	**v2**	**v1**						
1	**v25**	**v24**	**0,1**									
2	**v33**	**v31**	**v28**	**v26**	**v21**	**v22**	**v6**	**v5**	**0,4**			
3		**v32**	**v30**	**v29**	**2,3**							
4	**v48**	**v16**	**v44**	**v42**	**v40**	**v36**						
5			**v45**	**v41**	**4,4**							
6		**v46**	**5,2**									
7	**v54**	**v52**	**v51**	**v18**	**v12**	**v10**	**v4**	**0,4**				
8							**v3**	**0,5**				
9						**v11**	**2,6**					
10	**v60**	**v58**	**v34**	**v35**	**v53**	**v19**	**v20**	**9,5**				
11					**v27**	**10,5**						
12						**2,4**						
13			**10,3**									
14		**v59**	**v55**	**7,2**								
15			**v57**	**v56**	**v49**	**v50**	**v17**	**v13**	**v14**	**v38**	**v37**	**0,5**
16								**v15**	**v43**	**v39**	**4,5**	
17									**4,2**			
18						**v47**	**4,1**					

Table 3. Linear array representation for graph

Node	Reference	Node	Reference	Node	Reference
0,0	**v1**	**5,8**	**3,8**	**12,6**	**v50**
0,1	**v2**	**6.4**	**v22**	**12,7**	**v49**
0,2	**v4**	**6.5**	**v21**	**12,8**	**v56**
0,3	**v10**	**6.6**	**v26**	**12,9**	**v57**
0,4	**v12**	**6.7**	**v28**	**12,10**	**1,8**
0,5	**v18**	**6.8**	**v31**	**13,0**	**v36**
:	**:**	**:**	**:**	**:**	**:**
:	**:**	**:**	**:**	**:**	**:**
4,8	**v34**	**12,3**	**v14**	**18,4**	**15,4**
4,9	**3,9**	**12,4**	**v13**		
5,7	**v53**	**12,5**	**v17**		

Table 4. Compressed-weighted graph representation

--	0	1	2	3	4	5	6	7	8	9	10
0	**v1 -**	**v2 5**	**v4**	**v10**	**v12**	**v18**	**v51**	**v52**	**v54**		
		v1									
1								**v55**	**v59**	**v60**	
2			**v5**	**v6 15**	**v11 16**	**0,4**					
			v2	**v5**	**v6**						
3						**v20 20**	**v19 22**	**v27 26**	**v35 28**	**v58 33**	**1,9 37**
						v11	**v20**	**v19**	**v27**	**v35**	**v58**
4									**v34**	**3,9**	
5								**v53**	**3,8**		
6					**v22**	**v21**	**v26**	**v28**	**v31**	**v33**	
7								**v29**	**v30**	**v32**	**6,9**
8							**3,7**				
:	:	:	:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:	:	:	:
18					**15,4**						

3. PATH COMPRESSION

Figure 2 shows the compressed graph G'=(V,E,w') after doing some processing/compression on graph G=(V,E,w) to reduce the number of nodes and edges needed to be considered in computing paths as shown table 4. For example, the shortest path from vertex <v1> to vertex <v60> on Graph G in Figure 1 is 37 and the path is <v1, v2, v5, v6, v11, v20, v19, v27, v35, v58, v60>. And also the shortest path from vertex <v1> to vertex <v60> on G' is 37 but the path is compressed as <v1, v2, v6, v11, v19, v35, v60>.

4. SHORTEST PATH USING COMPRESSION

This optimized technique may exclude huge parts of the graph and hence saves the cost and improves performance of the graphs. The technique is summarized as:

1) Construct the matrix to represent the graph with inner structure that includes the Vertex, Dist, Pred Node, DirectPath that have a 0/1 flag, GoalNode, and accumulated weights from Direct Path. Dist[v] maintains the minimum distance to <v> via Pred Node.
2) Traverse the graph G to store direct paths for nodes where outDegree[Node] equal 1 and inDegree[Node] equal 1 as:
   - Store the Node parent, where outDegree[parent] equal 1.
   - Find the GoalNode, the first node we reached it that has outDegree[Node] not equal 1 in the same path.
   - Store Direct Path between the parent and the GoalNode, and the accumulated weight.
3) Construct the Reverse Matrix to represent the graph rooted with destinations.

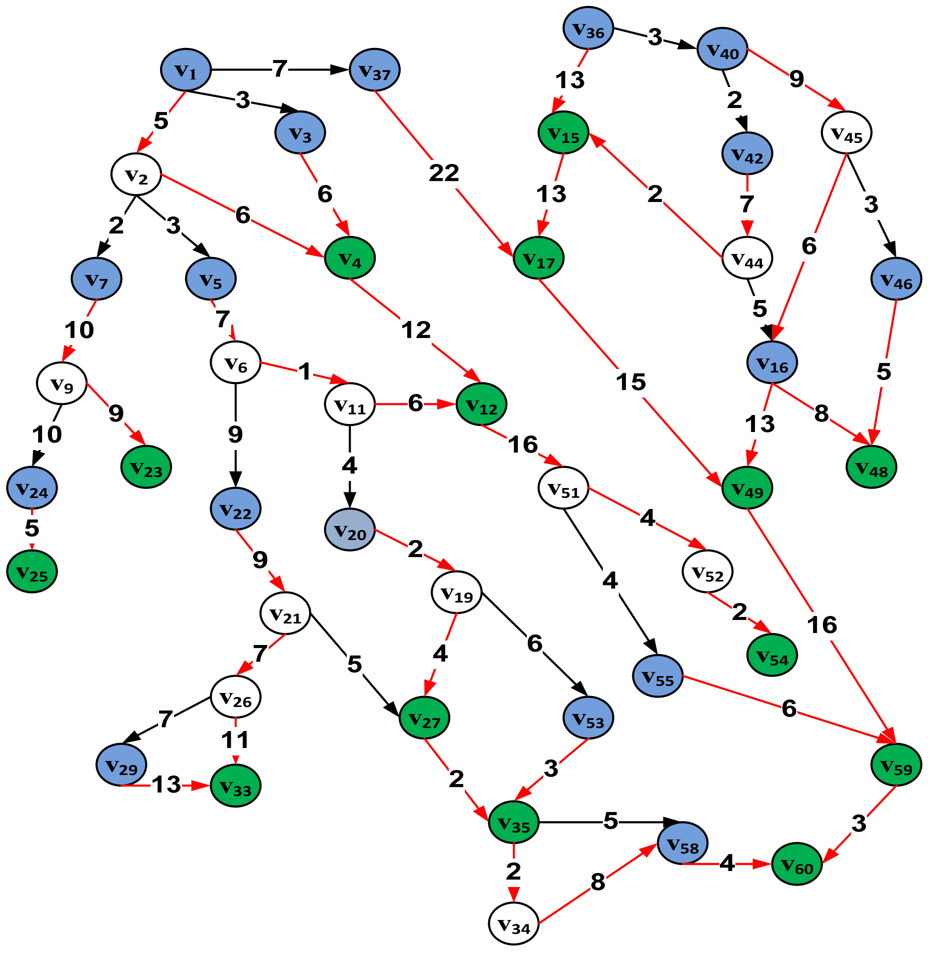

Figure 2. Compressed graph

4) Traverse the graph G starting by the given destination to mark all candidate nodes in the main matrix representation. This is possible using Reverse Matrix marking all candidate nodes as discussed in the algorithm. This is also possible in different ways as preferred by the programmer, e.g., copying the candidate nodes to a different reduced matrix, having a mark flag in the node structure, or by changing the weights of the excluded nodes to infinity in the main graph matrix. The preferred way is to have a 0/1 flag in a corresponding coordinate linear array representation.
5) After marking the candidate subgraph in the main matrix, and starting from the given source s, the algorithm check if there exist a DirectPath.

If there exists a Direct Path, we directly take the direct paths; else we add all neighbor edges by visiting all nodes listed in the next column (breadth fashion) of the current node (vertex). In this case, we always accumulate the subpath weight by adding the current vertex weight to accumulated path weight (dist).

Whenever we read the coordinates (i,j) of any vertex, it means that we revisit the node using another edge e with new weight w(e). In this case we directly jump to coordinates' pointer (i,j) in the main graph matrix as represented in Table 1 and compare the new weights and hence we keep the minimum path distance with updated predecessor nodes.

The algorithm Shortest Path Using Path Compression finds the shortest path based on reducing the number of edges and nodes that make the search take less time than searching in the entire graph. This efficient procedure saves much work comparing to the functionality of known conventional algorithms.

```
***Algorithm Shortest_Path_Using_Candidates (Graph, Mark, Reverse_Matrix)***
{initialize Mark[i] to 0
Node=t
Mark[Node]=1
for every vertex next to Node in Reverse_Matrix
  if vertex != coordinates_pointer
            Node=vertex
     else
     Node=ReverseMatrix[coordinates_pointer]
     Mark[Node]=1
  end if
end for
dist= FindShortest(GraphMatrix, Mark, s)
}

***Function FindShortest (GraphMatrix,Mark,s)***
 {
for each vertex v in GraphMatrix
     dist[v]=infinity;
     pred[v]=undefined;
end for
dist[s] := 0 ;
MarkQ=set of Marked nodes in GraphMatrix ordered as of depth first search visits.
while MarkQ is not empty and u != t:
     u= vertex in MarkQ with smallest distance in dist[ ];
     remove u from MarkQ;
     if (u == t ||dist[u] == infinity)
            break ;
     end if
  find coordinates  for v
  if ( there is a direct path for node )
    for each goalNode in GraphMatrix  and
     goalNode  is in MarkQ
      p=dist[u] + dist_between (u ,goalNode);
    if( p <dist[ goal Node]  )
            dist[goalNode]=p ;
            pred[goalNode]=u ;
     end if
    end for
  end if
   for each neighbor v of u and v is in MarkQ
     if v is coordinate pointer <a,b>
       v=GraphMatrix[a,b]
     end if
   p=dist[u] + dist_between(u, v) ;
   ifp<dist[v]:
     dist[v]=p ;
     pred[v]=u ;
     update  v in MarkQ;
   end if
  end for
end while
return dist;
     }
```

5. PERFORMANCE IMPROVEMENT

An algorithm that finds the shortest path P(s,t) between two given vertices <s> and <t> in a directed weighted graph G(V,E,w) is presented. It clearly determines the compressed graph G'(V',E',w). This obvious improvement reduces the number of edges |E| and vertices |V| in the graph which lowers the cost. Equation1 DC(E) calculates the degree of compression in terms of vertices and edges.

$$\mathrm{DC(E)} = \frac{(\text{\#of Edges in Compressed Graph})}{(\text{Total \# of Edges})} * 100\%$$

$$\mathrm{DC(E)} = \frac{|E'|}{|E|} * 100\% \qquad (1)$$

Having Graph G dipected in Figure 1 which has a direct path P from v37 to v17 where p={v37, v38, v14, v13, v17} and P has 4 edges, as improvement on this path, we have a direct edge from v37 to v17. This means that we reduced the number of edges from 4 edges to one edge. If the degree of compression is used, we get 75% time saving. If we look at the entire graph G, it has a total number of edges equals to 71 edges. After compressing it to graph G′ as illustrated in Figure 2, the new number of edges is 46. Applying the compression equation, the graph size, the saving ratio is 36%. This tangible saving is demanding in real-world applications and networks.

Each shortest path requires O((|V|+|E|)log |V|) using traditional algorithms making the cost extremely high. The proposed algorithm introduces more improvements comparing with some late studies such as the improvements introduced in [5]-[7]. Moreover, our algorithm works in better and improved performance on sparse and dense networks.

The experimental phase provides evidence that the proposed heuristic outperforms the conventional algorithms. The performance of the algorithm is compared with that of the conventional procedure and shows a considerable cost saving in random generated graphs with different sizes range from 30 to 300 nodes. Savings in performance occur in dense graphs and more in sparse ones in most of the trials. Table 5 shows the performance saving ratios as a result of the experiment.

Table 5. Saving ratio of performance in sparse and dense graphs

Nodes	Sparse (%)	Dense (%)
30	**0.1366846**	**0.4460346**
60	**0.4913375**	**0.5619978**
90	**0.4481861**	**0.4249575**
120	**0.6437499**	**0.2406172**
150	**0.3121945**	**0.1788921**
180	**0.369958**	**0.1761222**
210	**0.2612862**	**0.1249683**
240	**0.1970283**	**0.2412186**
270	**0.2618782**	**0.2114776**
300	**0.2410497**	**0.3028802**

Figures 3 and 4 show the average performance of applying the improved algorithm on set of randomly generated graphs with different density degrees. This shows that the proposed algorithm outperforms the standard depth-first search procedures on the original graph. Specifically, the conventional algorithm applied is Dijkstra’s.

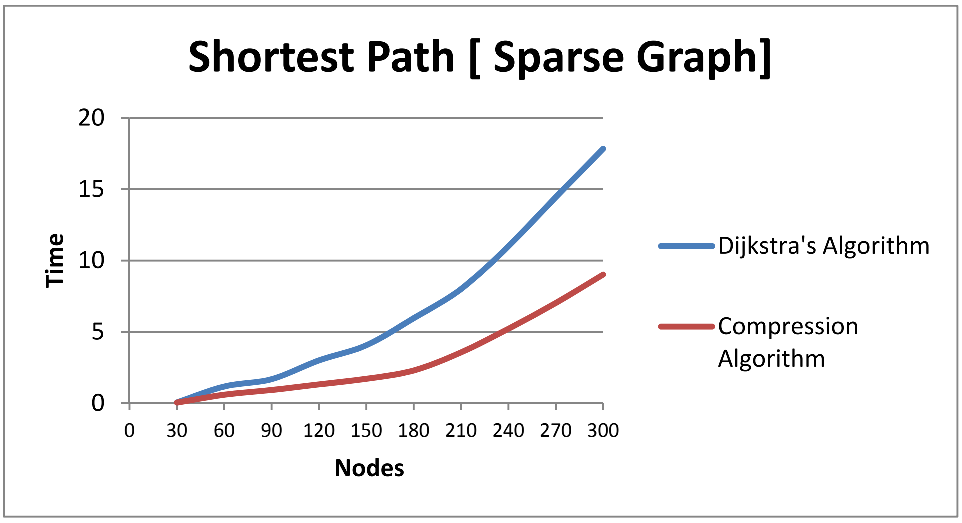

Figure 3. Performance of proposed algorithm on sparse graph

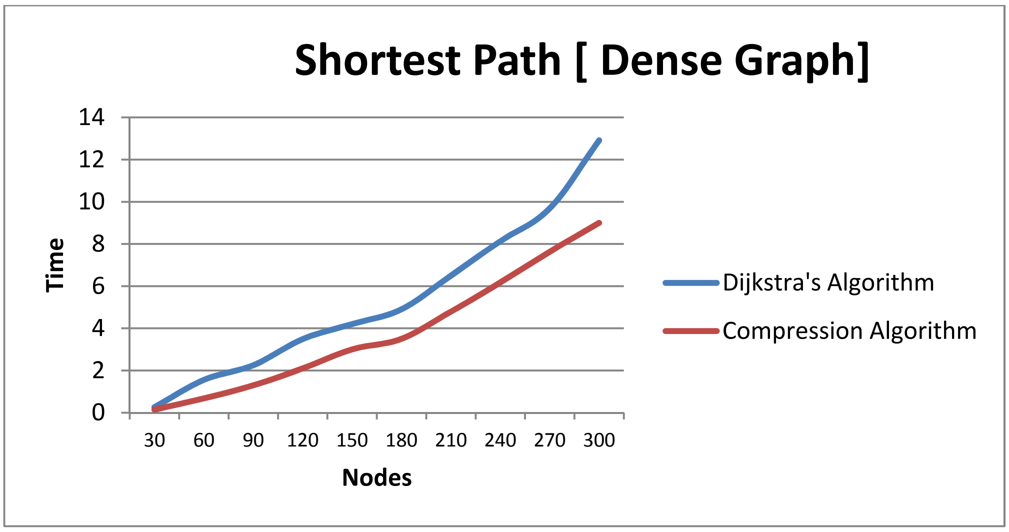

Figure 4. Performance of proposed algorithm on dense graph

6. CONCLUSION

For a given weighted graph G(V,E,w), with weights as a function of |w(e)|, an efficient and improved algorithm for finding shortest paths between a given source <s> and destination <t> using generated-compressed graph G' have been presented. In the practical phase, the algorithm outperforms the performance of improved algorithm. It shows obvious improved performance in set of random general applied graphs. As a heuristic algorithm, the complexity will always be bounded by the complexity of known algorithms, ie., it will not exceed O((|V|+|E|)log |V|).

ACKNOWLEDGEMENTS

This research is funded by The Scientific Research Council, Ministry of Education and Higher Education, State of Palestine under a project number of 01/12/2013, and Palestine Polytechnic University. The authors would like to thank the research assistants Ms. Walaa Naser Idin and Ms. Salma Dirbashi for their help in implementing the algorithms.

REFERENCES

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Dijkstra's Algorithm. Introduction to Algorithms", (Second ed.). Section 24.3: pp. 595–601. *MIT Press and McGraw-Hill*. ISBN 0-262-03293-7.

[2] F. Khamayseh and N. Arman, "An Efficient Multiple Source Single Destination (MSSD) Heuristic Algorithm Using Nodes Exclusions", *International Journal of Soft Computing*, Vol. 10, 2015.

[3] H.N. Djidjev, G.E. Pantziou, and C.D. Zaroliagis, "Improved Algorithms for Dynamic Shortest Paths", *Algorithmica* (2000) 28: 367–389.

[4] N. Arman, "Parallel Algorithms for the Generalized Same Generation Query in Deductive Databases", *Journal of Digital Information Management*: 4(3), 192- 196, 2006, ISSN 0972-72.

[5] J.B. Orlin, K. Kamesh Madduri, K. Subramani, and M. Williamson, "A faster algorithm for the single source shortest path problem with few distinct positive lengths". *J. of Discrete Algorithms*, 8, 2 (June 2010), 189-198

[6] L. Xiao, L. Chen, and J. Xiao, "A new algorithm for shortest path problem in large-scale graph". *Appl. Math*, 6(3), (2012), 657-663.

[7] F. Zhang, A. Qiu, and Q. Li, "Improve on Dijkstra Shortest Path Algorithm for Huge Data". *Chinese academy of surveying and mapping*: China, 2005.

[8] F. Khamayseh and N. Arman, "An Efficient Heuristic Shortest Path Algorithm Using Candidate Subgraphs". *International Conference on Intelligent Systems and Applications*. Hammamet, Tunisia. 22-24 March, 2014.

[9] F. Khamayseh and N. Arman. "Improvement of Shortest-Path Algorithms Using Subgraphs' Heuristics", *Journal of Theoretical and Applied Information Technology*, 2015. Vol. 76.

[10] E.W. Dijkstra, "A note on Two Problems in Connexion with Graphs", *Numerische Mathematik*, 1: 269 271. doi:10.1007/BF01386390.

[11] Y. Huang, Q. Yi, and M. Shi, "An Improved Dijkstra Shortest Path Algorithm". Proceedings of the 2nd *International Conference on Computer Science and Electronics Engineering* (ICCSEE 2013). Hangzhou, China, Paris: Atlantis Press, March 2013: 226-229.

[12] F. Simek and I. Simecek, "Improvement of Shortest Path Algorithms through Graph Partitioning". *International Conference Presentation of Mathematics*. Liberec, Czech Republic, 2011.

[13] W. Yahya1, A. Basuki2, J. Jiang. "The Extended Dijkstra's-based Load Balancing for Open Flow Network", *International Journal of Electrical and Computer Engineering* (IJECE), Vol. 5, No. 2, April 2015, pp. 289~296.

[14] J. Zhang, J. Li, X. Fan, Z. Deng, "Research on Real-Time Optimal Path Algorithm of Urban Transport", *TELKOMNIKA Indonesian Journal of Electrical Engineering*, Vol.12, No.5, May 2014, pp. 3515 ~ 3520.

BIOGRAPHIES OF AUTHORS

Dr. Nabil Arman is a Computer Science professor at Palestine Polytechnic University. He received his BS in Computer Science with high honors from Yarmouk University, Jordan in 1990 and an MS in Computer Science from The American University of Washington, DC USA in 1997, and his PhD from the School of Information Technology and Engineering, George Mason University, Virginia, USA in 2000. At Palestine Polytechnic University, he worked as the MS Informatics Program Coordinator and the head of the Department of Mathematics and Computer Science. Currently, he is the Dean of the College of Information Technology and Computer Engineering. Dr. Arman is interested in Database and Knowledge-Base Systems, Algorithms, and Automated Software Engineering. He has published more than thirty refereed conference and journal papers.

Dr. Faisal Khamayseh is a Computer Science assistant professor. He received his BS in Computer Information – Advanced Computer Careers, from Southern Illinois University, USA 1992, and MS in Computer Science from same university in 1995, and his PhD in Computers and Information Systems from the College of Computers and Information, Helwan University, Egypt, in 2009.Currently working at Palestine Polytechnic University as instructor and head of Dept. of Information Technology and as instructor of MS in Informatics. Dr. Khamayseh is a researcher in software engineering research unit at college of Information Technology and Computer Engineering. He is interested in Computer Algorithms, Software Engineering and E-learning.