An Efficient Scheme for the Single Source Shortest Path Problem based on Dijkstra and SPFA Methodologies

G. L. Prajapati, PhD
Dept. of Comp. Engg.
IET-Devi Ahilya University
Indore

Pulkit Singhal
Dept. of Comp. Engg.
IET-Devi Ahilya University
Indore

Ayush Ranjan Sharma
Dept. of Comp. Engg.
IET-Devi Ahilya University
Indore

Neelesh Chourasia
Dept. of Comp. Engg.
IET-Devi Ahilya University
Indore

ABSTRACT
This paper presents detailed comparisons and analysis of various single source shortest path algorithms. The paper proposes comparison among these algorithms on the basis of execution time taken by the algorithms to completely find the shortest path to all the nodes from a starting node. The algorithms have been analyzed on the various parameters: number of vertices, number of edges, and structure of the graph. This analysis will help in selecting the appropriate algorithm to be used in solving a particular real-life problem. This paper also proposes an algorithm that works efficiently over all types of the graph.

General Terms
Shortest Path, Algorithms, Theoretical Computer Science.

Keywords
Single Source Shortest Path, Execution Time, Performance Analysis

1. INTRODUCTION
Shortest path problem [1] is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized. Without the use of shortest path algorithms, the naïve approach for finding the shortest path between two vertices is to enumerate all possible paths between the vertices and select the shortest one i.e. brute force. However for the various domain specific applications of the shortest path problem, brute force approach is not feasible and hence require more optimal solution i.e. the shortest path algorithms. The aim of shortest path algorithms is to find the shortest path among all paths available between the pair of vertices. Some shortest path algorithms works only over the non-negative weighted graphs while some can works with negative weighted graphs too. Also the distinction between the algorithms is made by whether these are single source shortest path (SSSP) or all pair shortest path algorithm (APSP). This paper focuses on single source shortest path algorithms for the non-negative weighted graphs. Some of the shortest path algorithms are greedy in nature while some uses the dynamic programming. The shortest path algorithms works over the principle of relaxation. In such algorithms, optimization is based on the number of times relaxation is performed during the execution. Various real life applications of shortest path algorithms are finding the shortest route between the two places, social network analysis (SNA) to calculate degree of separation between two users on a social networking medium and so on. This paper compares the set of algorithms among themselves on the basis of execution time on datasets of different types.

2. SET OF SHORTEST PATH ALGORITHMS UNDER ANALYSIS

2.1 Bellman-Ford Algorithm
Bellman-Ford algorithm [2] uses the principle of edge-based relaxation. In every iteration it relaxes all the edges and these iterations are done V-1 times as the maximum number of edges in the shortest path between two vertices are V-1, where V is the number of vertices in the graph. If the relaxation can be done more than V-1 times, it indicates the presence of the negative cycles. The worst case complexity of this algorithm is O(VE).

2.2 Dijkstra Algorithm
Dijsktra algorithm [3] uses the same principle of relaxation as Bellman-Ford algorithm. It works over the graph with non-negative weighted edges only. In every iteration, it greedily chooses the vertex which is not selected before and has minimum cost. It tries to relax vertices through the selected vertex. Selection of vertex with minimum cost primarily affects the complexity of algorithm. In case of binary heap, worst case complexity of this algorithm is O(E logV). We are also considering an implementation of Dijkstra [4] which fastens the performance in case where the number of distinct weighted edges is less.

2.3 Pape-Levit Algorithm
Pape-Levit [5] is an incremental graph algorithm, where the two sets of vertices are maintained. One set of vertices contains those vertices which are scanned at least once while the second set contains those vertices which have never been scanned. The priority is given to first set for selection of vertex. The worst case complexity of this algorithm is O(VE). It works quite fast on the randomly weighted graphs.

2.4 SPFA
The Shortest Path Faster Algorithm (SPFA) [6] is an improvement of the Bellman–Ford algorithm which computes single-source shortest paths in a weighted directed graph. The algorithm is believed to work well on random sparse graphs and is particularly suitable for graphs that contain negative-weight edges. The performance of the algorithm is strongly determined by the order in which candidate vertices are used to relax other vertices.

2.5 Proposed Algorithm
As the performance of some algorithms highly depends over the order in which candidate vertices are used to relax the other vertices, we can see the linear time performance over the various type of graph structures while it also goes to O(VE) worst case time complexity. On the other hand, algorithms like the Dijkstra gives O(E logV) time complexity on every type of graph structure. This proposed algorithm harnesses the

power of both the algorithms. Proposed solution uses SPFA upto a certain limit of operations and if the shortest path is not found it moves to Dijkstra algorithm and the intermediate information produced by SPFA can be used to fasten the computation of Dijkstra algorithm. Let the transition factor be μ. Proposed solution uses SPFA till μ*(V+E) operations, after this it transits to Dijkstra. The value of transition factor (μ) will decide how fast the transition between these two algorithms happens. This algorithm will reduce the average time for finding the shortest path on a general graph.

Proposed Algorithm :

Input : Adjacency list of graph and source vertex.

Output : Distance array containing length of shortest path from source to every other vertex.

```
transitionOperationCount := μ*(V+E)
for i : = 1 to V,
        distance(i) := ∞
operationCount := 0
while operationCount <= transitionOperationCount,
        perform SPFA
        for each en-queuing operation,
                operationCount++
if shortest path is not found,
        perform Dijkstra
end
```

Description of the algorithm –

':=' denotes the assignment operation, while x++ denotes incrementing value of x by 1. This algorithm has an important factor (μ) which we termed as the transition factor. This transition factor decides how fast it moves from SPFA to the Dijkstra. If this factor is small then it moves to the Dijkstra quite fast, but if this factor is high it ends up using SPFA all the time. So a moderate value is required for this factor. Our algorithm runs SPFA algorithm for the μ*(V+E) operations, after that if it does not able to find the shortest path it moves to the Dijkstra algorithm. In best case it finds the shortest path in the time of SPFA itself, while in the worst case it ends up using both the algorithms. So asymptotically best case of this proposed algorithm is $O(\mu*(V+E))$ while worst case is $O((V+E)*(\log V+\mu))$. In general case it is assumed that μ is a small constant value. In the experimental analysis three different values of μ are considered.

3. ANALYSIS

3.1 Experiment Specifications

The tests were run on Intel Core i33217U @1.80 GHz CPU (CPU family 6, Model A, Stepping 9). Number of CPU(s) and socket is 1 having 2 Thread(s) per core and 2 Core(s) per socket. The algorithms were tested in presence of 64K L1d cache, 64K of L1i cache, 512K of L2 cache, 3072K of L3 cache, and 4GB DDR3 RAM. The system had Intel HD 4000 Graphics with 349MHz GPU clock. Byte Order of CPU used is Little Endian. The computer was running Windows 10 64-bit. All programs were written in C++ programming language with g++ compiler 4.8.4 producing x86_64 "64-bit" code.

3.2 Analysis Specification

The types of datasets used for the analysis of algorithms were: Some random datasets that were generated using the Prüfer sequence [7] consisting of various vertices and edge count. With the use of Prüfer sequence [7], generation of test data for trees can be done in equi-probable manner. In the random datasets we generated different structures like trees and graph, having vertices count from 10,000 to 10,000,000. For complete graphs, vertices count is varied from 100 to 5000. Along with random graph structures, some special graphs [8] are also generated which lead to the worst case for various algorithms. Experiments uses some datasets from DIMACS Implementation Challenge - Shortest Paths [9] (Northeast USA) containing 1,524,453 nodes and 3,897,636 edges. Along with this datasets from Stanford Large Network Dataset [10] Collection were included; one is the YouTube online social network containing 1,134,890 vertices and 2,987,624 edges, other one is the Amazon product co-purchasing network from June 1 2003 containing 403,394 vertices and 3,387,388 edges. The analysis report is based on: the execution time taken by algorithms on all the above mentioned datasets.

4. EXPERIMENTAL RESULTS

This analysis report is based on the execution time taken by algorithms on all the above mentioned parameters working with the above mentioned datasets. The resultant execution time is calculated as shown in Table 1 along with its graphical representation in Figure 1. Datasets were taken from 9th DIMACS implementation: shortest path – Northeast USA, Stanford large network dataset collection – YouTube community, Amazon sales co-purchasing network from June 1, 2003.

4.1 Analysis on Trees

Table 1 shows analysis of various algorithms on trees with different number of nodes. Figure 1 shows graph of execution time on different number of vertices in tree.

4.2 Analysis on Randomistic graphs

Table 2 shows analysis of various algorithms on randomistic graphs with different number of nodes and edges.

4.3 Analysis on Complete graphs

Table 3 shows analysis of various algorithms on complete graphs with different number of nodes. Figure 2 shows graph of execution time on different number of vertices in complete graph.

4.4 Analysis on Special graphs

Table 4 shows analysis of various algorithms on Special graphs with different number of nodes. Figure 3 shows graph of execution time on different number of vertices in special graph.

4.5 Analysis on Benchmarking Datasets

Table 5 shows analysis of various algorithms on benchmarking graphs. Figure 4 shows execution time on DIMAC distance based dataset. Figure 5 shows execution time on DIMAC time based dataset. Figure 6 shows execution time on Amazon co-purchasing dataset. Figure 7 shows execution time on YouTube community dataset.

Table 1: Showing the execution time of the algorithms on tree

Set of algorithms

Execution time (in seconds)

	Bellman-Ford	Dijkstra (Binary Heap)	Dijkstra (Edge based)	Pape-Levit	SPFA	Proposed Algorithm (μ = 1)	Proposed Algorithm (μ = 2)	Proposed Algorithm (μ = 3)
10000	0.0400	0.0350	0.1380	0.0020	0.0050	0.0060	0.0070	0.0040
50000	0.4510	0.1220	0.6020	0.0420	0.0210	0.0260	0.0240	0.0300
100000	0.7970	0.2880	0.8540	0.0880	0.0490	0.0560	0.0550	0.0620
500000	6.6220	1.6870	3.1460	0.3700	0.3280	0.3490	0.3190	0.3310
1000000	23.5760	3.5230	5.8320	0.7990	0.6770	0.7040	0.6970	0.7230
5000000	80.4500	22.8310	34.3050	5.0920	4.2270	3.6200	4.3780	4.5780
10000000	-	50.5150	-	13.5350	10.9730	13.3140	11.6240	10.6110

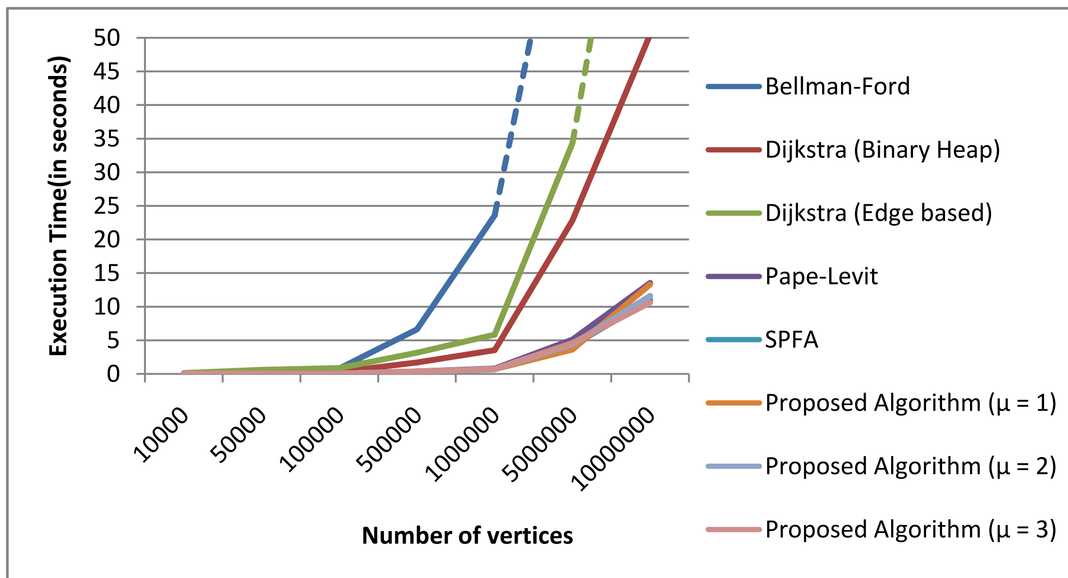

Figure 1: Graph showing execution time vs. number of vertices in tree

Table 2: Showing the execution time of the algorithms on graphs

	Set of algorithms									
Execution time (in seconds)	Number of vertices	Number of edges	Bellman-Ford	Dijkstra (Binary Heap)	Dijkstra (Edge based)	Pape-Levit	SPFA	Proposed Algorithm (μ = 1)	Proposed Algorithm (μ = 2)	Proposed Algorithm (μ = 3)
	10000	30000	0.0200	0.0770	0.3060	0.1900	0.0370	0.1150	0.1150	0.0850
	10000	40000	0.0240	0.0750	0.2620	0.2820	0.0300	0.0960	0.0820	0.0730

	10000	50000	0.0310	0.0810	0.3030	0.4980	0.0360	0.1210	0.1060	0.1000
	100000	300000	0.2110	0.8330	1.7370	15.6010	0.5730	1.2670	1.2520	1.1550
	100000	400000	0.2900	0.7560	1.9650	33.4720	1.0100	1.5580	1.6560	1.8150
	100000	500000	0.5030	1.0560	1.9810	33.9610	0.8640	1.3890	1.4270	1.3800
	1000000	3000000	2.2170	8.2320	13.1330	-	8.0270	14.5790	12.5150	15.7310
	1000000	4000000	4.5570	9.2260	14.0800	-	8.9250	15.6860	16.8670	15.9590
	1000000	5000000	4.2640	10.4900	19.8210	-	15.2440	17.3900	13.0030	13.3360

Table 3: Showing the execution time of the algorithms on complete graphs

Set of Algorithms

Execution time (in seconds)

Number of Vertices	Bellman-Ford	Dijkstra (Binary Heap)	Dijkstra (Edge based)	Pape-Levit	SPFA	Proposed Algorithm (μ = 1)	Proposed Algorithm (μ = 2)	Proposed Algorithm (μ = 3)
300	0.0160	0.0000	0.0160	0.0620	0.0310	0.0160	0.0000	0.0000
600	0.0310	0.0160	0.0780	0.7500	0.1100	0.0320	0.0250	0.0250
1000	0.1090	0.0510	0.2190	5.0910	0.5320	0.1530	0.1150	0.1150
3000	2.0720	0.4870	2.8640	134.8480	6.5170	0.5060	0.8270	0.8270
5000	4.8120	0.7860	5.9740	-	16.2580	1.8700	2.2150	2.2150

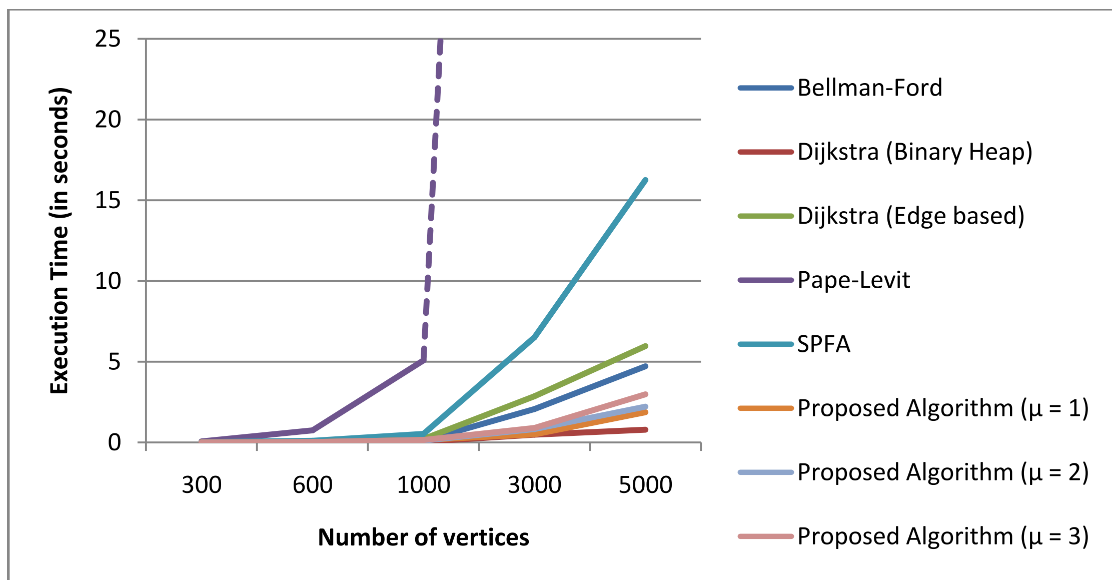

Figure 2: Graph showing execution time vs. number of vertices in complete graph

Table 4: Showing the execution time of the algorithms on special graphs

Execution time (in seconds)

	Set of Algorithms							
Number of Vertices	Bellman-Ford	Dijkstra (Binary Heap)	Dijkstra (Edge based)	Pape-Levit	SPFA	Proposed Algorithm (μ = 1)	Proposed Algorithm (μ = 2)	Proposed Algorithm (μ = 3)
100	0.0000	0.0190	0.0400	0.0200	0.0160	0.0040	0.0020	0.0010
300	0.0030	0.2260	0.6460	0.4430	0.5200	0.0400	0.0080	0.0090
600	0.0100	0.8100	4.7360	3.5010	3.4940	0.1970	0.0370	0.0470
1000	0.0340	2.3880	20.1700	18.5500	16.2490	0.5530	0.0850	0.0950
3000	0.2880	28.1930	-	-	-	4.6220	0.5950	0.7190
5000	2.7350	88.9080	-	-	-	19.7420	2.2980	2.9660

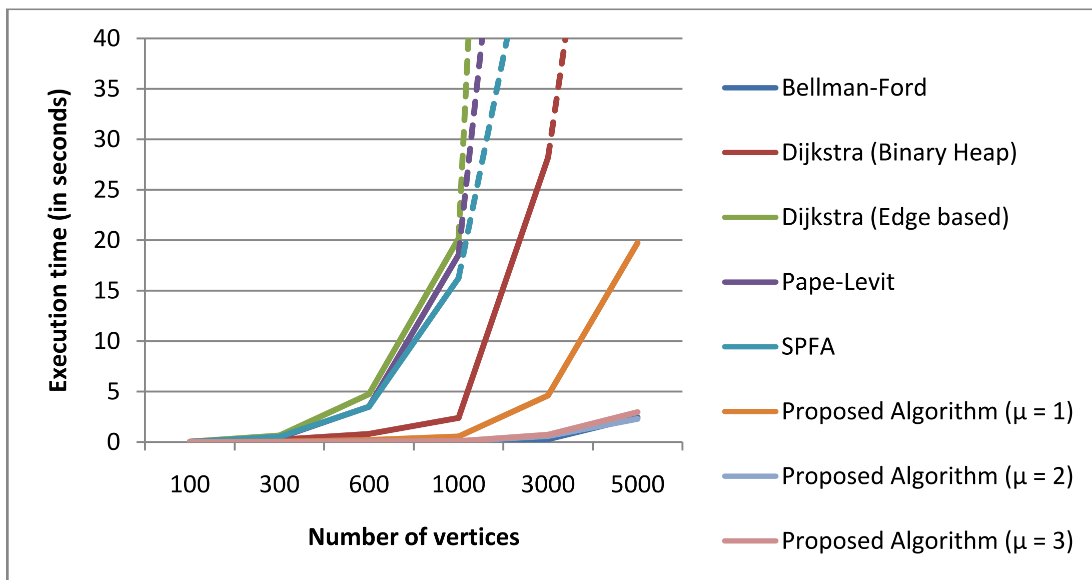

Figure 3: Graph showing execution time vs. number of vertices in special graph

Table 5: Showing the execution time of the algorithms on benchmarking datasets

	Set of algorithms							
Execution time (in seconds)	Dataset	Bellman-Ford	Dijkstra (Binary Heap)	Dijkstra (Edge based)	SPFA	Proposed Algorithm (μ = 1)	Proposed Algorithm (μ = 2)	Proposed Algorithm (μ = 3)
	DIMAC (NE USA) Distance Based	103.380	5.3560	8.0350	110.432	22.2670	22.0450	22.8000
	DIMAC (NE USA) Time Based	72.6840	6.9900	6.7190	47.7770	20.5570	23.1440	20.3710

	Youtube Communties	3.6290	12.2210	16.935	4.5170	13.1500	9.8760	9.5000
	Amazon Purchasing Network	1.2730	3.3360	4.3660	2.3650	5.9210	5.1170	4.4140

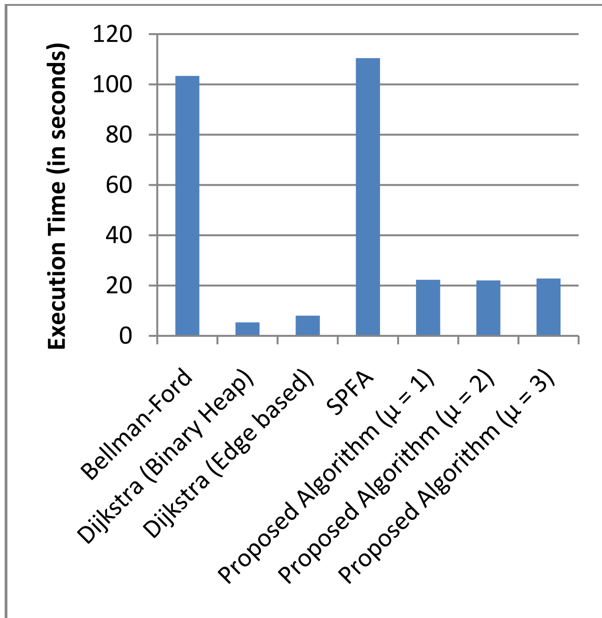

Figure 4: Graph showing execution time on the DIMAC (NE USA) distance based dataset

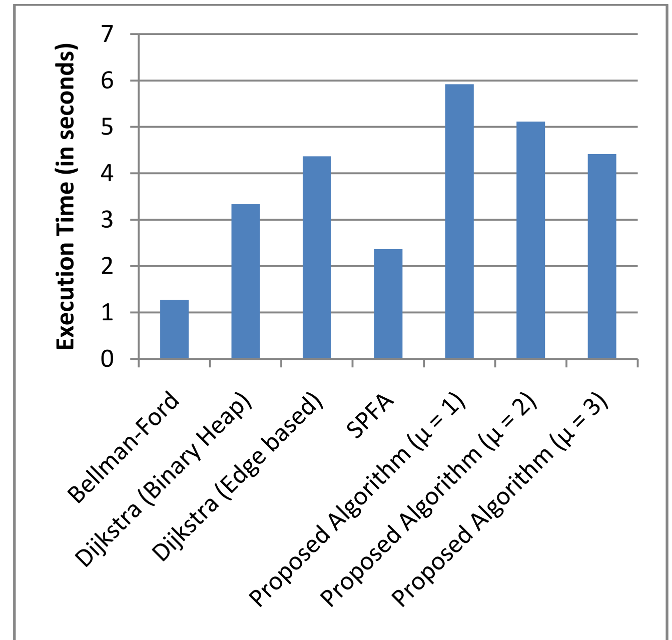

Figure 6: Graph showing execution time on the SNAP Amazon co-purchasing network based dataset

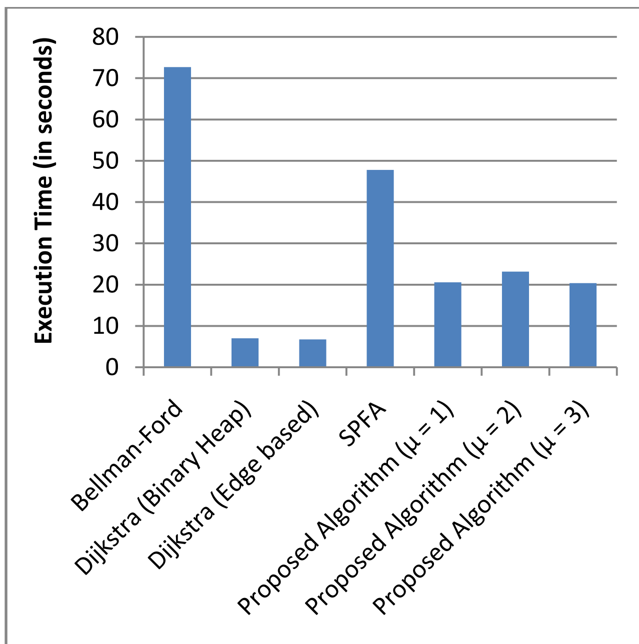

Figure 5: Graph showing execution time on the DIMAC (NE USA) time based dataset

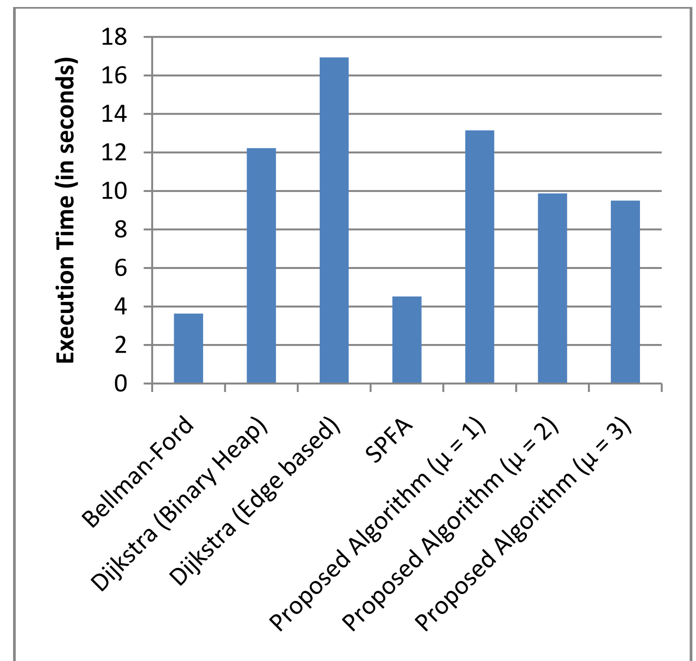

Figure 7: Graph showing execution time on the SNAP YouTube Communities based dataset

5. CONCLUSIONS

It can be seen from the aforementioned analysis that for different datasets, all algorithms under analysis perform differently i.e. certain algorithms perform better as compared to others. For random graphs, SPFA outperforms all the other algorithms while for the general graphs the performance degrades. Performance of algorithms like SPFA highly depends upon the order in which candidate vertices are used to relax other vertices. While processing a vertex, which can be relaxed further increases the time overhead of the algorithm. In case of random graph, this overhead is quite low which make the SPFA performs better in comparison to other algorithms. In case of special graph and benchmarking datasets, this overhead becomes quite high which results in poor performance of SPFA. For all type of graphs the performance of Dijkstra does not depends over the structure of graph, which makes it faster on special graphs. For harnessing the power of both the algorithms our proposed scheme proved to perform better over every kind of graph.

6. REFERENCES

[1] ShortestPathProblem,http://en.wikipedia.org/wiki/Shortest_path_problem

[2] Bellman ford (1958). “On a routing problem”. Quarterly of Applied Mathematics. 16: 87-90.Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik.

[3] Faster Dijkstra on Special Graphs, http://codeforces.com/blog/entry/43508.

[4] Boris V. Cherkassky , Andrew V. Goldberg , Tomasz Radzik, Shortest paths algorithms: theory and experimental evaluation, Mathematical Programming: Series A and B, v.73 n.2, p.129-174, May 31, 1996.

[5] Shortest Path Faster Algorithm, http://codeforces.com/blog/entry/16221.

[6] Prüfer, H. "Neuer Beweis eines Satzes über Permutationen." Arch. Math. Phys. 27, 742-744, 1918.

[7] Construction of Special Graphs for poor performance of SPFA, http://codeforces.com/blog/entry/16221?#comment-211370

[8] 9th DIMACS Implementation Challenge - Shortest Paths, http://www.dis.uniroma1.it/challenge9/download.shtml

[9] Stanford Network Analysis Project, https://snap.stanford.edu/