International Journal of Research in Engineering and Innovation (IJREI)

Journal home page: https://ijrei.com/

RESEARCH PAPER

Comparative analysis of Dijkstra's and Bellman-Ford Algorithms for the metro train

Sanjay Kumar, Jitendra Kumar Saini

[1]*Research Scholar, Department of Computer science and engineering, Shobhit University, Gangoh Saharanpur, India*
[2]*Assistant Professor, Department of Computer science and engineering, Shobhit University, Gangoh Saharanpur, India*

Article Information

Received: 26 May 2026
Revised: 29 June 2026
Accepted: 10 July 2026
Available online: 15 July 2026

Keywords:

Shortest Path Algorithm
Dijkstra's Algorithm
Bellman–Ford Algorithm
Optimal Path
Routing Problems
Network Optimization

Abstract

The shortest path problem is a fundamental problem of graph theory that studies the problem of finding the lowest-cost path between nodes in a weighted graph where the edges between nodes are a positive real value. Shortest-path problems are the most fundamental and the most commonly faced problems in the study of transportation and communication networks, and In GIS applications. The problems of the shortest paths can be divided into two groups: the single-source shortest path, which is used to find the shortest path between a source node and a specific destination node, and the all-pairs shortest path, which can be used to find the shortest paths between all pairs of nodes within a network. In addition to classical shortest path problems, there are numerous related problems, such as the longest path problem, the most reliable path problem, and the maximum capacity path problem, which also highlight the importance of efficient path-finding methods. Various algorithms have been created to overcome these issues, such as Dijkstra's, Bellman-Ford, Floyd-Warshall, A, Johnson and Viterbi algorithms. In this paper, we are going to perform a comparative analysis of Dijkstra's algorithm and Bellman-Ford algorithm. Their efficiency, constraints and applicability are compared on small- and large-scale graphs, and their performance, constraints and their performance under different circumstances, including negative edge weighting. The study will be focused on providing an impression of the most appropriate algorithm to be used in the different variants of the shortest path problem.

1. Introduction

A Dijkstra Algorithm, another name of single source shortest path problem [1, 2], when we have provided a weighted graph, we will get the shortest path from the starting vertex to all the vertices. I am selecting beginning of vertex s1 then I have to find-out "shortest-path" to all the vertices may be a direct-path and I can select any one of the vertices as a source vertex as we have to find out a "shortest-path" so it's a minimization problem and we know that this problem is an "optimization problem" so these problems can be unraveled using "greedy-method", it suggest that a problem should be settled in parts by making one stride at once and considering single input at time to look for optimal solution and greedy method are already defined methods, so Dijkstra's algorithm gives a procedure for getting an optimal solution that is minimum result that is "shortest-path" in this paper we will show how our Dijkstra's will works, and Dijkstra's algorithm its works and Dijkstra's algorithm can work on a dia-graph and non-Dia-graphs and also we will show you the drawback of Dijkstra's Algorithm [3, 4]. The approach of Dijkstra's algorithm for that we will show you. The basic thing that is followed by Dijkstra's so for that first of all we will take a very small example and show you, to show the approach of Dijkstra's algorithm we have taken a very small graph here from this we will understand what is the approach of Dijkstra's [5, 6].

Corresponding author: Sanjay Kumar
Email Address: sanj.ccs@gmail.com
https://doi.org/10.36037/IJREI.2026.10401

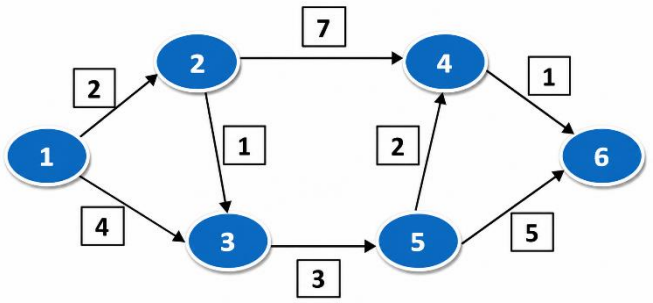

Figure 1: Weighted Graph for Dijkstra's Algorithm [7]

Figure 1 illustrates a weighted directed graph used to demonstrate the execution of Dijkstra's shortest path algorithm. The graph consists of six vertices (1–6) connected by directed edges, where each edge is assigned, a positive weight representing the cost or distance between two nodes. Node 1 is considered the source vertex, from which the algorithm computes the minimum-cost path to all other vertices. For example, the edges from node 1 to nodes 2 and 3 have weights of 2 and 4, respectively, while subsequent connections such as 2→3 (1), 2→4 (7), 3→5 (3), 5→4 (2), 4→6 (1), and 5→6 (5) define alternative routes to the destination nodes. During execution, Dijkstra's algorithm repeatedly selects the unvisited node with the smallest tentative distance and relaxes its outgoing edges to update the shortest known distances. This weighted graph serves as a standard example for illustrating how the algorithm efficiently determines the optimal paths from the source node to every other node in a network with non-negative edge weights [8]. Figure 2 presents a simplified directed weighted graph used to illustrate the basic working principle of Dijkstra's algorithm. The graph consists of three vertices (1, 2, and 3) connected by directed edges with positive weights. The edge from node 1 to node 2 has a weight of 2, while the edge from node 2 to node 3 has a weight of 4, representing the cost of traversing between the respective vertices. Starting from the source node (node 1), Dijkstra's algorithm first determines the shortest distance to node 2 as 2. It then processes node 2 and updates the shortest distance to node 3 by adding the edge weight, resulting in a cumulative distance of 6.

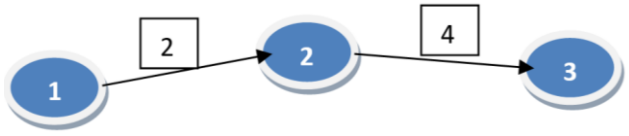

Figure 2: Dijkstra's Algorithm Dia-Graph [7]

According to relaxation algorithm

If(d[u]+c(u,v)<d[v])
D([v]=d[u]+c(u,v)

Figure 3 illustrates a directed weighted graph used to demonstrate the execution of Dijkstra's shortest path algorithm. The graph contains six vertices (1–6) connected by directed edges, where each edge is assigned, a non-negative weight representing the cost or distance between two connected nodes. Node 1 serves as the source vertex, from which the algorithm computes the shortest path to all other vertices. The graph provides multiple alternative routes to the destination nodes, allowing the algorithm to compare path costs and select the minimum-cost route. For instance, node 1 is connected to node 2 with a weight of 2 and to node 3 with a weight of 4. Additional connections include 2→3 (1), 2→4 (7), 3→5 (3), 5→4 (2), 4→6 (1), and 5→6 (5), creating several possible paths between the source and destination.

During the execution of Dijkstra's algorithm, the source node is initialized with a distance of zero, while all other vertices are assigned an infinite distance. The algorithm repeatedly selects the unvisited vertex with the smallest tentative distance, marks it as permanently visited, and relaxes its outgoing edges by updating the distances of adjacent vertices whenever a shorter path is found. This iterative process continues until the shortest distances to all reachable vertices have been determined. For the given graph, the algorithm identifies the shortest route from node 1 to node 6 as 1 → 2 → 3 → 5 → 4 → 6, with a total path cost of 9 (2 + 1 + 3 + 2 + 1 = 9), which is smaller than all other possible paths. This weighted graph effectively illustrates the greedy nature of Dijkstra's algorithm and demonstrates its capability to efficiently compute optimal shortest paths in networks with non-negative edge weights [9].

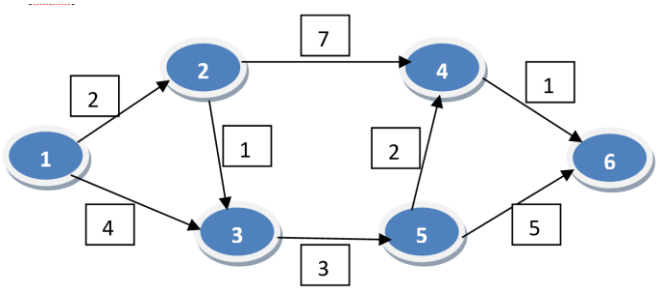

Figure 3: Directed Weighted Network Used for Dijkstra's Algorithm [7]

Now we will solve this graph problem following this procedure to find out a shortest path so we will assume that the distance of all the vertices is only single edge at first we will label the distance of starting vertex(1) as 0 and the distance between 1 and 2 as 2 and distance between 1 and 3 is 4 and between to 1 and 5 distance is infinity and between 1 and now we shall begin the repeating steps this is the steps which will be repeating first step take out a shortest path out vertices 2,4, infinity, infinity and infinity vertex 2 is the smallest having weight 2 step 1st take out this vertex now we should check this step and also we should relax this step we should not relax this step with any other step 2+7=9,9 is less than infinity this step, now second step now the step is repeating same thing I should do select the smallest one vertex three, infinity, nine and infinity which one is the smallest vertex 3 select this one now check if any vertex get relaxed so there is only one connected vertex 3 to 5 weight is 3+3=6 [10, 11] so this is infinity so change it. now select the shortest out of what 6, 9 and infinity which one is shortest vertex 5 having weight 6 select this one and see who are connected and try to relax them so 6+2=8 and this is 9 change it, 6+5 = 11 and this is infinity modified so two vertices are modified they are relaxed now who are remaining

4 & 6 who is minimum 8 select this one and check if anything gets relaxed 8+1=9, it is already 1 modified. So here we have an example: most of the time, the vertices modify their distances, though they may not. now the last one is remaining that is 6 so now we have the shortest path from starting vertex 1 to all of the vertices so what are the other vertices 2, 3, 4, 5, & 6 what are the distances I got for 2 it is 2 & for 3 it is 3 & for 4 it is 8 & for 5 it is 6 and 6 is 9 these are the distances of all the vertices so that is how a Dijkstra's algorithm works on a graph to finding the shortest path. Let us analyze and find out the time complexity of this algorithm. If you analyze what it does, it finds the shortest path to all vertices. How many vertices are there? n=|V| number of vertices, so for all vertices is finding shortest path while doing so what it is doing it is relaxing which vertexes, whether it is on 3 and it is relaxed only 5 when it was on 2 it has relaxed only 3 & 4 so total how many vertices it will be relaxing we do not know any difference on the graph ok at the most how many vertexes it may be relaxing all n vertices suppose 2 is connecting to all then it may be relaxing all of them it may be checking all of them so again it is |V| we got n=|V| |V| so it means n*n and n for vertices and n vertexes those are relaxed so when it will happen that from 2 all are connected then again 3 all are connected that will be a complete graph in case of complete graphs it will take n*n time [12].

2. Steps for Algorithm

This implementation is represented in a graph. We have given a weighted, directed graph G = (V, E) with source vertex s. S is represented as the visited vertices. And priority queue Q is represented as the set of all vertices in graph Dijkstra's (G, s).

```
For each vertex v in graph G
{
   D[s] = 0
  D[v] = ∞
}
Initialize visited vertices S in graph G
S = null
Initialize Queue Q as set of all nodes in graph G
Q = all vertices V
While Q ≠ ∅
{
        u=mind (Graph G, distance d)
            Visited vertices S = S+u
for each vertex v in neighbor[u]
 {
  If d[v] > d[u] + w(u,v)
          Then d[v] = d[u] + w (u,v)
}
Return d
}
```

To begin with initialize the graph G with list of vertices V end list of edges E. then assign the distance of all the nodes and make the current node have a 0 value, and the other unvisited nodes have ∞ a value. Then compute the distance between the current node and all the whole neighbor node of that current node and marked as visited nodes. Node Than visited cannot be reused. Its distance store and now is over and bear. Current node and. Mark unmarked node with the lesser distance as the current node and proceed as in the last step.

3. Bellman-Ford Algorithm

The Bellman-Ford algorithm is a graph relaxation algorithm that is applied to determine the shortest paths in directed graphs with a single source. And it also contains negative edges. The algorithm will also detect if there are any negative-weight cycles (such that there is no solution). Assuming that we are talking about distances on a map, then there is no negative distance. The basic structure of bellman-ford algorithm is similar to Dijkstra's algorithm. It smooths out all the edges, and does this -1 times, where V is the number of the vertices in the graph. The weight of a path is the weight of the edges in the path. srv is the source and this is a single-source shortest path problem and find distance form single source to various destinations. This is an algorithm return value indicating whether a negative cycle exists or not and also gives the shortest path. This algorithm searches the shortest path in a bottom-up fashion [13].

a. Steps for the algorithm

This implementation is represented in graph. We have given a weighted, directed graph G = (V, E). Hear "V "is represent as list of vertices and "E" is represent as list of edges. Vertex is represented as "v" and edges are represented as "e" and the source vertex is represented as "srv" and also given a weighted function w.

```
Initialize graph
Bellman-Ford (vertices V, edges E, source vertex srv)
        For each v in V
        {
                If v = srvthan d[v] =0
                Else d[v] = infinite
                Cost[v] = null
         }
         2. Relaxation of edges
        For i form 1 to IVI-1
        {
                For each [u,v] with w in E
                {
                If d[u]+w<d[v]
                {
                If d[v]=d[u]+w
                Cost[v] =u
                }
                }
        }
        3. Checking negative cycle
        For each e(u ,v) with weight win E
         {
                If d[u]+w,d[v]
                {
                 Print "graph has negative cycle"
                }
          }
```

This is because first we need to feed in input in the form of graph and a source vertex srv. On completion of such a process we will receive output as shortest path of single source to all the vertices and in case there exists negative cycle in a graph then it will be detected [14].

First step initializes graph in which we initialize distances from Figure 4 illustrates the sequential workflow of the Bellman–Ford shortest path algorithm, which is designed to determine the minimum-cost paths from a single source vertex to all other vertices in a weighted graph, including graphs containing negative edge weights. The process begins with the Initialize Distances step, where the source vertex is assigned a distance value of zero, while all remaining vertices are initialized with infinity to indicate that no paths have yet been discovered. The algorithm then proceeds to the Check Edge Condition stage, where each edge in the graph is examined to determine whether traversing that edge can produce a shorter path to the destination vertex.

If a shorter path is identified, the Update Distance step modifies the corresponding distance value by relaxing the edge. This relaxation process is repeated systematically for V − 1 iterations, where V represents the total number of vertices in the graph. Performing V − 1 iterations ensures that the shortest paths containing up to V − 1 edges are correctly identified. After completing these iterations, the algorithm performs a Final Check for Negative Cycle by relaxing all edges one additional time. If any distance can still be reduced, the graph contains a negative-weight cycle, indicating that no finite shortest path exists for the affected vertices. Otherwise, the algorithm proceeds to the Output Results stage, where the final shortest distances from the source vertex to all reachable vertices are reported. This process makes the Bellman–Ford algorithm particularly suitable for routing, network optimization, and transportation problems where negative edge weights may occur, while also providing the important capability of detecting negative-weight cycles that cannot be handled by Dijkstra's algorithm.

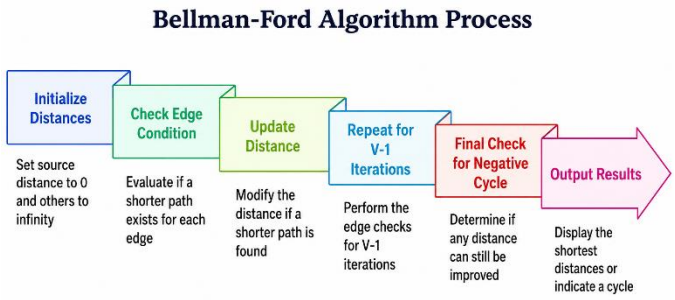

Figure 4: Process of Bellman-Ford algorithm [7]

Figure 5 presents a comparative bar chart illustrating the performance of the Bellman–Ford and Dijkstra algorithms based on the shortest-path distances computed for different nodes in a weighted graph. The horizontal axis represents the graph nodes, while the vertical axis indicates the corresponding shortest-path distance values. The blue bars denote the distances obtained using the Bellman–Ford algorithm, whereas the red bars represent those computed using Dijkstra's algorithm. The chart provides a visual comparison of the distance values generated by the two algorithms for each node.

Although both algorithms are designed to compute shortest paths from a single source vertex, they differ significantly in their underlying approach and computational characteristics. Dijkstra's algorithm employs a greedy strategy and efficiently computes shortest paths in graphs with non-negative edge weights, resulting in lower computational overhead and faster execution. In contrast, the Bellman–Ford algorithm repeatedly relaxes all edges for V − 1 iterations, making it computationally more expensive but capable of handling graphs containing negative edge weights and detecting negative-weight cycles. The graphical comparison demonstrates the variation in computed distance values across the nodes while emphasizing the trade-off between computational efficiency and algorithmic capability. Consequently, Dijkstra's algorithm is generally preferred for networks with non-negative edge weights due to its superior efficiency, whereas the Bellman–Ford algorithm is selected when negative edge weights or negative-cycle detection are required.

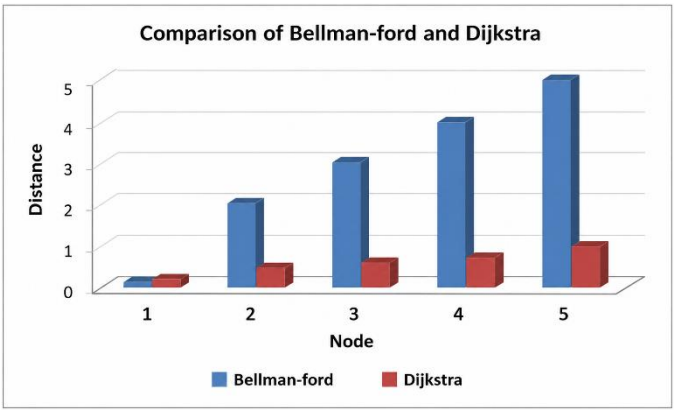

Figure 5: Graph of time complexity [4]

The simplest implementation of the Dijkstra's algorithm stores vertices in a linked list or array, a time taken by the algorithm is 0(|V|2). And time taken by bellman ford algorithm is O(| V | · | E |). So, we can clearly show that bellman ford algorithm takes more time than Dijkstra's algorithm. So, running time of bellman ford algorithm is more than Dijkstra's algorithm [5].

Implement Q using a a priority queue at most E edges in the heap. Space complexity of Dijkstra's algorithm is O (V+E). And space complexity of bellman ford algorithm is O(V). So bellman ford algorithm takes more space than Dijkstra's algorithm [15].

Table 1 presents a comparison between the Dijkstra and Bellman–Ford algorithms based on their capabilities and computational complexity for solving the single-source shortest path problem. Both algorithms are designed to determine the shortest path from a source vertex to all other vertices in a weighted graph; however, they differ significantly in terms of handling edge weights and execution efficiency.

The table shows that Dijkstra's algorithm is applicable only to graphs containing non-negative edge weights and cannot

detect negative-weight cycles. It follows a greedy approach by selecting the vertex with the smallest tentative distance at each iteration, making it highly efficient for graphs without negative edges. The algorithm has a time complexity of $O(|V|^2)$ when implemented using an adjacency matrix, where $|V|$ represents the number of vertices in the graph. When priority queues or Fibonacci heaps are used, the time complexity can be further improved for sparse graphs.

Table-1: Comparison Table [4]

Algorithm	Source	Negative edges?	Negative cycle?	Time complexity
Dijkstra	Single source	No	No	$0(\|V\|^2)$
Bellman-ford	Single source	Yes	Yes	$O(\|V\|\cdot\|E\|)$

In contrast, the Bellman–Ford algorithm supports graphs containing negative edge weights and is capable of detecting negative-weight cycles, making it more versatile than Dijkstra's algorithm. Instead of following a greedy strategy, Bellman–Ford repeatedly relaxes all edges for $|V| - 1$ iterations to ensure that the shortest paths are correctly computed. After these iterations, an additional pass is performed to determine whether any edge can still be relaxed, thereby identifying the presence of a negative-weight cycle. Due to this exhaustive relaxation process, the Bellman–Ford algorithm has a higher computational complexity of $O(|V| \times |E|)$, where $|E|$ denotes the number of edges in the graph.

4. Conclusion

After studying these two algorithms, I conclude that Dijkstra's algorithm is much faster than the Bellman-Ford algorithm. And also widely used in real time application in GIS. This is the reason why I selected the Dijkstra algorithm to find the shortest path in GIS technology. It is more workable and quicker in a very large network as well.

In this paper authors note the time of running algorithms Dijkstra and In micro seconds and compare the results with the average running time, and conclude that the Dijkstra algorithm is less efficient than Bellman Ford algorithm in the SP problem in small number of nodes and Bellman Ford algorithm is less efficient in big no. of nodes.

References

[1] Chan, S., Adnan, N., Sukri, S. S., & Zainon, W. M. (2016, August 29–30). *An experiment on the performance of shortest path algorithm*. Knowledge Management International Conference (KMICe 2016), Chiang Mai, Thailand.

[2] Dinitz, Y., & Itzhak, R. (2017). Hybrid Bellman–Ford–Dijkstra algorithm. *Journal of Discrete Algorithms, 42*, 35–44. https://doi.org/10.1016/j.jda.2017.01.001

[3] Hemalatha, S., & Valsalal, P. (2012). Identification of optimal path in power system network using Bellman–Ford algorithms. *Journal of Modeling and Simulation in Engineering*, Article 913485.

[4] Hofner, P., & Möller, B. (2012). Dijkstra, Floyd and Warshall meet Kleene. *Formal Aspects of Computing, 24*(4–6), 459–476.

[5] Huang, B., Wu, Q., & Zhan, F. B. (2007). A shortest path algorithm with novel heuristics for dynamic transportation networks. *International Journal of Geographical Information Science, 21*(6), 625–644.

[6] Hutson, K. R., Schlosser, T. L., & Shier, D. R. (2007). On the distributed Bellman–Ford algorithm and the looping problem. *INFORMS Journal on Computing, 19*(4), 542–551.

[7] Manan, A., Imran, S. and Lakyari, A., 2019. Single source shortest path algorithm Dijkstra and Bellman-Ford Algorithms: A comparative study, International journal of computer science and emerging technologies, 3(2), pp.25-28.

[8] Lacorte, A. M., & Chavez, E. P. (2018). Analysis on the use of A* and Dijkstra's algorithms for intelligent school transport route optimization system. In *Proceedings of the 4th International Conference on Human-Computer Interaction and User Experience in Indonesia (CHIuXiD '18)*. https://doi.org/10.1145/3205946.3205948

[9] Madkour, A., Aref, W. G., Rehman, F. U., Rahman, M. A., & Basalamah, S. M. (2017). *A survey of shortest-path algorithms*. arXiv. https://arxiv.org/abs/1705.02044

[10] Oyola, A., Romero, D. G., & Vintimilla, B. X. (2017). A Dijkstra-based algorithm for selecting the shortest-safe evacuation routes in dynamic environments (SSER). In *Lecture Notes in Computer Science* (pp. 131–135). Springer. https://doi.org/10.1007/978-3-319-60042-0_15

[11] Patel, V., & Baggar, C. (2014). A survey paper of Bellman–Ford algorithm and Dijkstra algorithm for finding shortest path in GIS application. *International Journal of P2P Network Trends and Technology (IJPTT), 4*(1).

[12] Permana, S. H., Bintoro, K. Y., Arifitama, B., & Syahputra, A. (2018). Comparative analysis of pathfinding algorithms A*, Dijkstra, and BFS on Maze Runner game. *International Journal of Information System & Technology (IJISTECH), 1*(2), 1–8.

[13] Sapundzhi, F. I., & Popstoilov, M. S. (2018). Optimization algorithms for finding the shortest paths. *Bulgarian Chemical Communications, 50*(Special Issue B), 115–120.

[14] Singh, J. B., & Tripathi, R. C. (2018). Investigation of Bellman–Ford algorithm and Dijkstra's algorithm for suitability of SPP. *International Journal of Engineering Development and Research (IJEDR), 6*(1).

[15] Wang, X. Z. (2018). The comparison of three algorithms in shortest path issue. *Journal of Physics: Conference Series, 1087*, 022011. https://doi.org/10.1088/1742-6596/1087/2/022011

Cite this article as: Sanjay Kumar, Jitendra Kumar Saini, Comparative analysis of Dijkstra's and Bellman-Ford Algorithms for the metro train, International Journal of Research in Engineering and Innovation, 10 (3), (2026), 130-134. https://doi.org/10.36037/IJREI.2026.10401