Proceedings of the International Conference on Industrial Engineering and Operations Management

Publisher: IEOM Society International, USA DOI: 10.46254/NA10.20250282
Published: June 17, 2025

Comparative Analysis of Shortest Path Algorithms using Graph Theory

Minjun Kwon and Dongho Shin
Student and Professor, My Paul School
12-11, Dowontongmi-gil, Cheongcheon-myeon, Goesan-gun
Chungcheongbuk-do, Republic of Korea
eavatar@hanmail.net

Jeongwon Kim
Department of Economics, College of Economics, Nihon University
3-2 Kanda-Misakicho, 1-chome, Chiyoda-ku, Tokyo, Japan
shinphys@naver.com

Abstract

This study addresses a comparative analysis of shortest path algorithms based on graph theory. The shortest path problem plays a crucial role in various application fields, with Dijkstra's algorithm and the Bellman-Ford algorithm being representative algorithms for solving it. Dijkstra's algorithm performs optimally in graphs without negative weights and efficiently explores paths using a priority queue. In contrast, the Bellman-Ford algorithm has the advantage of being able to handle negative weights, but it has a relatively high time complexity. This paper compares the time and space complexities of the two algorithms and analyzes the suitable application cases for each. Through this analysis, the importance of selecting the appropriate algorithm for solving the shortest path problem is emphasized.

Keywords

Graph theory, Shortest path, Path algorithms, Dijkstra's algorithm and Bellman-Ford algorithm.

1. Introduction

The graph theory in this study is a mathematical tool widely utilized in various fields such as computer science and computational studies, contributing to the resolution of complex problems through structures composed of vertices and edges. In particular, the shortest path problem is one of the core issues in graph theory and plays an essential role in various application areas. Shortest path algorithms are used to solve several problems, including network communication, logistics and transportation route optimization, robot path planning, and database query optimization. The purpose of this study is to compare and analyze various shortest path algorithms based on graph theory to understand the characteristics and efficiency of each algorithm. It introduces the definitions and types of graphs and explains representation methods such as adjacency matrices and adjacency lists. Additionally, it conducts an in-depth analysis of Dijkstra's algorithm and the Bellman-Ford algorithm to evaluate their performance. The algorithms are compared with a focus on time complexity and space complexity, connecting theory to practice through real-world application cases. Finally, it presents criteria for selecting the optimal algorithm in various situations.

2. Body

A graph is a mathematical entity composed of vertices (V) and edges (E), used to visually represent various relationships. Mathematically, a graph G is defined as follows: (G = (V, E)) where V is the set of vertices, and each

vertex has a unique identifier (Chang et al. 2011). E is the set of edges, which represent the relationships between two vertices. Edges are typically represented as pairs of vertices (Lee et al. 2014, Lee et al. 2023, Kim 2019, No 2013).

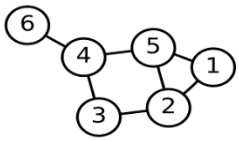

Figure 1. Vertex and Edge Graph

Graphs are categorized based on three criteria: the presence or absence of directedness, the presence or absence of weights, and the type of vertices. Graphs can be categorized into directed and undirected graphs, weighted and unweighted graphs, and homogeneous and heterogeneous graphs based on the type of vertices. Directed Graph: A directed graph is a graph with directed edges, each of which represents a flow from one vertex to another. In other words, the edges clearly distinguish between the starting and ending points. Directed graphs are useful in a variety of fields, and examples of directed graphs include social networks and web page links.

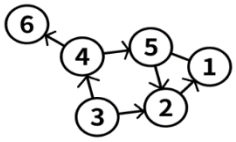

Figure 2. Directed Graph

Undirected Graph: An undirected graph is a graph in which there is no direction in the trunk line, and the connection between two vertices is bi-directional: when vertex A and vertex B are connected by a trunk line, going from A to B is treated the same as going from B to A. Examples of undirected graphs include family relationships and roads. Undirected graphs are one of the most basic forms of graph theory, and examples of undirected graphs include family relationships and roads.

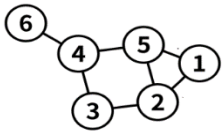

Figure 3. Undirected Graph

Weighted graph: A weighted graph is a graph in which each arterial line is assigned a weight. The weight is a number that represents the connection between two vertices, such as distance, time, or cost. Weighted graphs can be used to effectively solve problems such as finding the shortest path or minimizing cost.

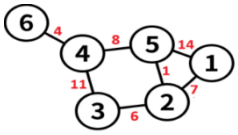

Figure 4. Weighted graph

Unweighted Graph: An unweighted graph is a graph in which the edges are not weighted, meaning that all edges are considered to have the same value. These graphs only represent the connectivity between vertices, and can be applied to both directed and undirected graphs. Unweighted graphs are used to analyze network connectivity and model social relationships. They have a simple structure, which makes them effective in identifying relationships.
Homogeneous graph: A homogeneous graph is a graph in which all nodes represent the same kind of object and all edges show the same kind of relationship. For example, in a social network, every vertex represents a person and every edge represents a friendship between people. This structure makes it easier to understand and analyze data. Homogeneous graphs are useful for representing complex networks in a simple way.

Heterogeneous Graph: A heterogeneous graph is a graph with different kinds of vertices and edges. It uses different kinds of objects (e.g., people, groups, etc.) as vertices and different kinds of relationships (e.g., friendships, affiliations, etc.) as edges. Heterogeneous graphs are great for modeling complex relationships and are used in social network analysis and more. They also play an important role in effectively representing and analyzing diverse data.

Adjacency matrix: The adjacency matrix is a two-dimensional array of size n × n (n is the number of vertices) that represents the relationship between vertices and edges in a graph. Each element of the matrix, A[i][j], is labeled 1 if a trunk line exists between vertices i and j, and 0 otherwise. In directed graphs, the value depends on the direction of the trunk, while in undirected graphs it is a symmetric matrix. In a weighted graph, it stores the weighted values of the trunk lines. This has the advantage of being very fast to check if there is a trunk between two vertices i and j, but the amount of memory used is proportional to the square of n, depending on the number of vertices n, and is inefficient for sparse graphs with few trunks. It is used in graph traversal algorithms (DFS, BFS) and shortest path algorithms (Dijkstra algorithm). Along with neighbor lists, it is one of the main graph representations (Jeong et al. 2004).Representing graphs using adjacency matrices: An adjacency matrix is a mathematical representation of the connectivity between vertices in a graph, and can be represented graphically using the following procedure. Letting the vertices be [A, B, C, D] and the edges be [A-B, A-C, B-D, C-D], the adjacency matrix is defined as follows.

Figure 5. Adjacency matrix example

Each row and column of the adjacency matrix above represents a vertex of the graph, and vertices A, B, C, and D correspond to their respective positions in the above matrix. Finally, after placing vertices A, B, C, and D in the appropriate locations and connecting the trunk lines according to the adjacency matrix, the graph can be represented as follows (Chung 2024).

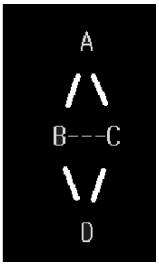

Figure 6. Graphs represented by adjacency matrices

From this simple process, we can see that adjacency matrices are a useful way to systematically represent the structure of a graph. They also allow us to clearly visualize the relationships between vertices and can be applied to a wide variety of graphs and algorithms.

Adjacency lists: Adjacency lists are an efficient way to represent graphs, maintaining a list of connected nodes for each node. This structure is good for saving memory and performs optimally on graphs with fewer edges. The advantage of using an adjacency list is that it allows you to quickly explore the neighbors of a particular node. Adjacency lists also allow you to seamlessly add or delete trunks. Additionally, when you add a new trunk, you can easily handle it by adding a node to the list for that node. These properties make neighbor lists preferred by many algorithms. Despite the advantages of neighbor lists, they do have their drawbacks. Because a list must be maintained for each node, memory usage can increase when the number of nodes is large and connectivity is complex. However, this drawback is outweighed by its memory efficiency and speed of traversal in many situations, making it a popular choice for solving graph-related problems.

Representing graphs with adjacency lists: To give an example of a graph with adjacency lists, consider a graph with vertices A, B, C, D, and E, and edges (A, B), (A, C), (B, C), (C, D), and (D, E). This graph can be represented as an adjacency list, which is the list of vertices connected to each vertex.

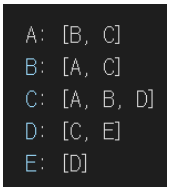

Figure 7. Adjacency list example

Vertex A is connected to vertices B and C, which means you can walk from A to B and C. Vertex B is connected to vertices A and C. So you can go from B to A and C. In this way, we can see the connectivity between vertices, and we can see that vertex C is the central vertex because it is connected to A, B, and D. This graph is undirected, meaning that there are multiple paths, and it is represented as an adjacency list, which makes it easy to see the connectivity of each vertex. This structure is useful when using a traversal algorithm. Edge List: An Edge list is one way to represent a graph, which is a list of all the trunks. Each edge is represented as an array of two vertices. For example, a graph with nodes A, B, C, D, and E would be represented as (A, B), (A, C), (B, C), (C, D), and (D, E). Edge lists are memory-efficient and simple to implement, making them ideal for graphs with a small number of trunks. Adding or deleting a edge is seamless, but it is inefficient when checking the connectivity between two vertices (Hun and Kim 2009).

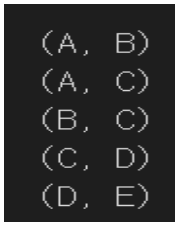

Figure 8. Example of a edge list

Shortest path algorithms include a variety of methodologies for finding the shortest path between two vertices in a graph. The algorithm is chosen based on the characteristics of the graph, with the most common being the single-start shortest path algorithm, which finds the shortest path from a specific starting vertex to all other vertices. Single starting point shortest path algorithms include the Dijkstra algorithm and the Bellman-Ford algorithm. Each algorithm has its own unique characteristics, and it is important to choose the appropriate algorithm based on the nature of the problem (Koo et al. 2016, Kim 2019, Jeong and Park 2005, Lee 2007). The Dijkstra algorithm is one of the algorithms for finding the shortest path from one vertex to another in a graph. It does this by finding all the shortest paths to each vertex, not just the arrival vertex, but also to all the other vertices, and then choosing the vertex with the shortest path and repeating the search. The sequence of steps in the Dijkstra algorithm is as follows.

The first is the initial setup phase. For each vertex, we create a list to store the shortest distance, set the distance of the starting vertex to 0, and set the distance of the remaining vertices to infinity. We also initialize a set to keep track of the visited vertices.The second step is to select the shortest path. Among the unvisited vertices, select the vertex with the smallest value in the distance array. This vertex is considered the vertex of the shortest path found so far. Add the selected vertex to the set of visited vertices so that it is not reselected later, and update the path to the selected vertex's neighbors.

The third step is to update the neighboring vertex distances. The distance value of the vertex u selected in the previous step is fixed to be the shortest path to the current vertex. This value is called d(u). Next, all vertices v neighboring u are checked. These are the vertices directly connected to u by a trunk line. For each neighboring vertex v, we compute a new distance value, which looks like this new_distance=d(u)+w(u,v) where d(u) is the shortest distance to vertex u, and w(u,v) is the weight of the trunk line from vertex u to vertex v. Thus, new_distance is the total distance to reach vertex v via vertex u. Up to this point in the process.

The remaining step is to update the distance. If the newly calculated distance value, new distance, is smaller than the currently stored distance value, d(v), update the distance array: d(v)=min(d(v), new distance), and if new distance is smaller, update d(v) to new distance. This process keeps the distance values of neighboring vertices up to date and ensures the accuracy of the algorithm. The second and third iterations are repeated, and the algorithm terminates when all vertices have been visited, or when no more distance values can be updated.

The Dijkstra algorithm is widely used in GPS navigation systems, network routing, traffic flow optimization, and many other fields. For example, it's used in map apps to calculate the best route to travel. As you can see, it's very useful because it's an algorithm that iterates over and over again to get a result. The Bellman-Ford algorithm is an algorithm for finding the shortest path from one vertex to all other vertices in a graph, and is applicable to graphs with negatively weighted edges. It is considered to be the most efficient tool for finding the shortest path from a given starting vertex. The Bellman-Ford algorithm has the following steps

The first is the initialization phase. This step creates a list to store the shortest distance to each vertex, and sets the distance of the starting vertex to zero. The remaining vertices are initialized to infinity, indicating that they have not yet been reached. With these initial settings, all subsequent calculations will be based on them.The second step is the distance update phase. The distance update is the core step of the Bellman-Ford algorithm, which iteratively checks all trunk lines to update the shortest distance. This process is repeated a total of V-1 times, where V is the number of vertices in the graph. At each iteration, all trunk lines (u, v) and their weights w(u, v) are checked, and a new distance value is computed as follows new_distance=d(u)+w(u,v) If the new distance value is less than the currently stored distance d(v), update d(v) to new_distance. This iterative process plays a big role in determining the shortest path.

The third is negative cycle checking. In this step, all trunk lines are checked again to see if there are any cases where the distance value is still updated. If such updates occur, it means that there are negative cycles in the graph, and the shortest path cannot be defined. This check is an important step to increase confidence in the algorithm.Thanks to its simplicity and efficiency, the Bellman-Ford algorithm is used in a variety of fields. It plays an important role in optimizing asset movement routes in financial networks, route planning in transportation networks, and routing problems in computer networks, among others. It is a powerful tool for finding shortest paths while allowing negative weights, contributing to the solution of optimization problems and providing a basis for solving graph-based problems.

It is important to analyze time complexity and space complexity to compare the performance of algorithms. This analysis is an important criterion for understanding the efficiency of an algorithm and choosing the right algorithm for a particular problem. In this section, we will analyze these two complexities, focusing on the Bellman-Ford algorithm and another typical shortest path algorithm, the Dijkstra algorithm (Choi et al. 2017).Time complexity is defined as a function of how the running time of an algorithm relates to the size of its input. It plays an important role in evaluating how efficiently an algorithm works for a given input by analyzing the number of operations it performs in relation to the size of the input. Typically, time complexity is described in terms of worst-case, average, and best-case, and is often expressed using big-o notation (Park et al. 2017, Choi 2004).

The time complexity of the Bellman-Ford algorithm is defined as O(VE). where V is the number of vertices in the graph and E is the number of edges. The Bellman-Ford algorithm works by iteratively inspecting all the edges in the graph V-1 times to update the shortest path. Because of this, the algorithm's running time increases when the number of trunk lines is large. The Bellman-Ford algorithm has the advantage of being able to find shortest paths in graphs with negative weights, but its relatively high time complexity can cause performance degradation on large graphs. The Dijkstra algorithm, on the other hand, has a time complexity of O((V + E) log V), which reflects its performance when implemented using a prioritized queue. The algorithm works by selecting the vertex with the shortest distance among the unvisited vertices at each step and updating the distances of the vertices neighboring that vertex. The Dijkstra algorithm works more efficiently with a larger number of trunk lines, and it performs optimally on graphs with non-negative weights. Therefore, analysis using time complexity shows that the Dijkstra algorithm can generally find shortest paths faster than the Bellman-Ford algorithm.

Space complexity is an important metric that compares the amount of memory required by an algorithm while it is running against the size of its input. It represents the total amount of memory space used by an algorithm and is mainly determined by the input data, constants, variables, and additional data structures. Like time complexity, space complexity can be described in terms of worst-case, average, and best-case scenarios, and it plays an important role in determining the memory usage of an algorithm (Jung et al. 2003).

The space complexity of the Bellman-Ford algorithm is O(V) because it requires a list to store the shortest distance to each vertex. This algorithm is efficient in terms of memory usage because it does not require any additional space beyond the array that stores the shortest distance information. However, there is a tradeoff in that as the number of vertices and trunk lines increases, the space usage increases, so memory limitations must be considered.

The Dijkstra algorithm also has a space complexity of O(V). If the algorithm uses a heap data structure to implement the prioritized queue, it requires an additional O(E) space, i.e., additional space is required to maintain the prioritized queue. Therefore, the Dijkstra algorithm will use more memory than the Bellman-Ford algorithm in some cases. In particular, for graphs with a large number of trunk lines, the space usage of the priority queue has a significant impact on the overall memory requirement. In conclusion, the time and space complexity analysis allows you to compare the characteristics of each algorithm and is an important criterion for choosing the right algorithm for a particular problem. The Bellman-Ford algorithm can be safely used even with negative weights, while the Dijkstra algorithm often performs better and is therefore more popular in real-world cases.

Item	Bellman-Ford Algorithm	Dijkstra's Algorithm
Time Complexity	$O(V \cdot E)$	$O(E + V \log V)$ (with priority queue)
Space Complexity	$O(V)$	$O(V)$
Shortest Path	Can handle graphs with negative weights	Cannot handle graphs with negative weights
Advantages	- Can handle negative weights	- Generally faster performance
	- Can compute shortest paths for all vertices	- Efficient with priority queue
Disadvantages	- Relatively high time complexity	- Cannot be used with negative weights
	- Requires repeated edge relaxation	- Performance may degrade in dense graphs

Figure 9. Algorithm comparison table

The Bellman-Ford algorithm and the Dijkstra algorithm are widely used to solve shortest path problems in a variety of fields. In particular, the Bellman-Ford algorithm has an important application in that it can be applied to graphs with negatively weighted edges. In this section, we introduce some representative applications of the Bellman-Ford and Dijkstra algorithms and analyze their efficiency in each case. Financial trading networks: In financial trading networks, the Bellman-Ford algorithm plays an important role in optimizing the paths that assets take. In particular, in transactions between multiple financial institutions, where fees can have a negative weight, the algorithm helps minimize costs by finding the optimal trade path. The Bellman-Ford algorithm has the property of reliably finding the shortest path, even in graphs with negative weights. This allows it to be flexible in situations where fees may fluctuate in financial transactions, so it can find optimal paths relatively efficiently even in large financial networks. This is an important factor in increasing the efficiency of financial transactions, contributing to cost reduction and enhancing competitiveness.

Transportation networks: In transportation systems, the Bellman-Ford algorithm is used to find the shortest route by modeling the road network as a graph and weighting the travel time on each road. The algorithm can update the optimal route in real time, especially if the weights change depending on the traffic on the road. The Bellman Ford algorithm is able to react sensitively to changes in roadway traffic, helping to improve traffic flow and reduce travel times. The algorithm's iterative updating process provides optimal routes based on real-time traffic information, which is effective in alleviating traffic congestion in cities. However, the time complexity can be relatively high in large transportation networks, so a combination with other algorithms, such as the DAEXTRAS algorithm, may be necessary in certain situations.

Robot path planning: In robotics, the Dijkstra algorithm is essential for robot path planning. It models the environment as a graph and calculates the shortest path to a goal point by exploring efficient paths. If obstacles appear along the way, the algorithm can recalculate the path in real time to avoid them. The Dijkstra algorithm is very efficient at finding optimal paths in non-negative weighted graphs. It is also used in autonomous vehicles and drones to ensure safe movement in complex environments. In particular, obstacle avoidance greatly improves the efficiency and safety of robots, giving them the ability to adapt to changing environments in real time. The Bellman-Ford and Dijkstra algorithms have contributed to solving shortest path problems in a variety of fields, each with its own characteristics and strengths. By analyzing the efficiency of these algorithms through applications in financial trading networks, transportation systems, robot route planning, and more, we can further increase the optimization potential in each field. The development of these algorithms is expected to provide innovative solutions in various industries in the future.

3. Conclusion

In this study, we conducted an in-depth comparative analysis of two of the leading shortest path algorithms in graph theory: the Dijkstra algorithm and the Bellman-Ford algorithm. We analyzed their performance, time complexity, and efficiency in networks. We found that the Dijkstra algorithm is the most efficient for positive-weighted graphs and performs well on large networks with an average time complexity of O ((V + E) log V). On the other hand, the Bellman-Ford algorithm showed efficiency in negatively weighted graphs, but performed relatively poorly in large networks with a time complexity of O(VE). In conclusion, this study clarified the unique characteristics and application scope of each algorithm through the theoretical comparison of shortest path algorithms. In the modern information and communication environment, where the complexity of networks is increasing, the selection of algorithms should go beyond simple computational efficiency and comprehensively consider the structural characteristics and requirements of the network.

Our future plans are to expand the scope of our comparative analysis to include a wider variety of shortest path algorithms. We also plan to utilize real-world traffic and network data to realistically evaluate the performance of the algorithms. We will explore new methodologies to solve the shortest path problem by developing hybrid approaches that combine the strengths of different algorithms. We will explore different techniques to optimize the performance of the algorithm to increase its real-time applicability. Finally, based on our findings, we will examine the practical applicability of our algorithms and suggest ways to apply them in various fields.

References

Chang, B. W., Park, S. Y., Jeong, J. A. and Kang, W. M., Quantitative analysis of Seoul's green space connectivity network using graph theory, *Journal of the Korean Society of Environmental Ecology*, vol.23, no.1, pp.116-129, 2011.

Lee, W. K., Park, S. H., Graph Theory and Social Networks, *Journal of the Korean Society of Information Science,* no. 1, pp.33-43, 2014.

Lee, G. Y., Park, K. S., A study on highway analysis algorithms using graph theory*, Journal of the Korean Computer Information Society*, vol.28, no.6, pp.55-62, 2023.

Jeong, H. S., Kim, C. H. and Park, C. H., Dynamic shortest path search algorithm with Dijkstra's algorithm, *Proceedings of the Korean Society of Transportation Engineers*, pp.343-348, 2004.

Chung, Y. W., Analysis of the development diagram of the cube using graph theory, *Mathematical Education,* vol. 63, no.4, pp.707-723, 2024.

Koo, B. H., Chae, C. B., Park, S. H., Park, H. S. and Ham, J. H., Graph theory-based polarization and frequency assignment algorithm*, Journal of the Korea Communications Society,* vol.41, no.8, pp.954-957, 2016.

Kim, J. S., A Database Segment Distribution Algorithm Using Graph Theory, *Journal of the Multimedia Society*, vol. 22, no.2, pp.225-230, 2019.

No, B. H., Development of wetland network construction model of Nakdong River basin using graph theory*, Journal of the Korean Wetland Society*, vol.15, no.3, pp.397-406, 2013.

Hun, J. Y., Kim, J. J., A study on pattern recognition by graph theory, *Proceedings of the Korean Society of Industrial Engineering and Technology*, pp.722-725, 2009.

Choi, J. K., Kim, C. H., Oh, D. I. and Jung, S. B., A study on efficiency comparison of urban railroads using graph theory, *Proceedings of the Korean Railway Society*, pp.621-626, 2017.

Park, H. S., Kim, H. K. and Kim, Y. S., Automatic Vehicle Delivery Algorithm to reflect time complexity for the Transportation Vulnerable, *Journal of the Electronics Engineering Society*, vol.54, no.6, pp.43-52, 2017.

Choi, T. Y., A Linear-Time Heuristic Algorithm for k-Way Network Partitioning, *Journal of the Multimedia Society*, vol.7, no.8, pp.16-16, 2004.

Jung, J. H., Lee, S. W. and Kim, H. S., Design of an efficient space complexity LFSR multiplier, *Journal of the Korean Industrial Information Society*, no.3, pp.85-90, 2003.

Jeong, Y. G., Park, C. H., Development of the Shortest Route Search Algorithm Using Fuzzy Theory, *Journal of the Korean Transportation Association*, vol.23, no.8, pp.171-179, 2005.

Lee, S. W., A point-to-point shortest path search algorithm for digraph, *Journal of the Korean Society for Intelligent Systems*, vol.17, no.7, pp.893-900, 2007.

Biographies

Minjun Kwon is student in MY PAUL SCHOOL. He is interested in artificial intelligence, deep learning, cryptography, robots, autonomous vehicles, etc., and is conducting related research.

Jeongwon Kim is graduate in College of Economics, Nihon University. She is interested in artificial intelligence, deep learning, cryptography, robots, block chains, drones, autonomous vehicles, etc., and is conducting related research.

Dongho Shin is Professor and Teacher in MY PAUL SCHOOL. He obtained his Ph.D. in semiconductor physics in 2000. He is interested in artificial intelligence, deep learning, cryptography, robots, block chains, drones, autonomous vehicles, mechanical engineering, the Internet of Things, metaverse, virtual reality, and space science, and is conducting related research.