Execution Performance of Dijkstra's Algorithm: A Cross-Language Analysis

Muhammad Hamza[1], Hassan Raza[2]

Abstract

This present study looks at the intensive performance analysis of the algorithm of DIJKSTRA executive in four programming languages: Python, Java, C++, and MATLAB. This study explains how the performance of the DIJKSTRA algorithm changes across these four languages, which is often used to find the shortest distance. It considers key performance factors and different types of graphs. The researchers analyzed various graph complications to mimic real-world problems, including small, large, rare, and dense examples. They tested how well the algorithm worked in terms of time to run, memory usage, complexity, and efficiency in situations using the selected language. The results show how the output changes with the choice of programming languages, and have a big effect on the performance of the algorithms, with each language having its own strengths and weaknesses. Python takes less memory but a long time to run, while C++ is faster but uses more memory for execution in dense graphs. MATLAB is strong in computation but uses a lot of memory, and Java offers a good balance between time and memory. These results show the importance of choosing the right languages depending on the specific task we need. This study adds to the discussion about adopting algorithms. It helps programmers make informed choices about which language to use for implementing the Dijkstra algorithm based on their needs and challenges.

Keywords: *Dijkstra's Algorithm; Graph Types; Execution Time; Memory Usage; Efficiency; Programming Paradigms; Computational Capabilities.*

Introduction

Computer science is a rapidly developing field. The algorithm invented by Edsger W. Dijkstra in 1956, and known as Dijkstra's algorithm, is one of the most significant graph algorithms because it efficiently calculates the shortest path. This algorithm has found widespread application because of its ability to identify the shortest path in a graph with non-negative edge weights from a source node to all other nodes (Makariye, 2017). The Dijkstra algorithm is useful in many areas beyond

[1] M.Phil from Department of Criminology, Lahore Garrison University. Email: muhammad.hamza@lgu.edu.pk
[2] M.Phil from The University of Lahore, Lahore. Email: hassan.raza@cs.uol.edu.pk

https://doi.org/10.58932/MULL0003

computer science. Many modern tools, such as GPS-based navigation systems in vehicles, rely on this algorithm. These systems help users find the shortest route while considering traffic conditions, alternative paths, and other factors. The algorithm's robustness and adaptability also extend to warehouse logistics, where automated guided vehicles (AGVs) use it for optimal route planning (Sun et al., 2021).

Despite its widespread adoption, several enhanced versions have been proposed, such as the hybrid Bellman–Ford–Dijkstra algorithm, demonstrating continuous efforts to improve the algorithm's efficiency and adaptability by combining the strengths of classical Dijkstra's and Bellman–Ford algorithms (Dinitz & Itzhak, n.d.). Furthermore, the algorithm has been applied in disaster management and building evacuation models to improve evacuation procedures under strict time constraints (Mirahadi & McCabe, 2021). These applications are implemented in different programming environments. The programming language selected may influence the algorithm's performance because each language has its own syntax, execution model, and optimization techniques (Makariye, 2017). Other factors, such as graph size, edge weights, and graph complexity, may also affect performance. Therefore, this study aims to analyze the performance of Dijkstra's algorithm across different programming environments.

This study compares the execution time, memory consumption, and performance of Dijkstra's algorithm using Python, C++, Java, and MATLAB across graphs of different sizes and densities. The findings will help programmers and engineers select the most suitable programming language according to their requirements. This research also contributes to the ongoing discussion on algorithm optimization and performance improvement (Dinitz & Itzhak, n.d.; Makariye, 2017; Mirahadi & McCabe, 2021; Sun et al., 2021).

Literature Review

The different scholarly works offer an extensive understanding of the execution and application of Dijkstra's algorithm across different programming environments. Each study contributes to the literature and provides the foundation for our research, which aims to compare the algorithm's execution time and memory consumption across multiple programming languages. Ridlo Al-Hakim et al. (2020) implemented Dijkstra's algorithm using React Native in a real-time COVID-19 tracking application. Their work demonstrated both the accuracy and adaptability of the algorithm across modern software platforms. He (2019) applied Dijkstra's algorithm to self-driving systems and public transportation, demonstrating its effectiveness in solving complex route optimization problems. Fitriansyah et al. (2018) developed a geographical mapping system that uses Dijkstra's algorithm to determine the shortest path between tourist destinations in Bali. Their work illustrates the algorithm's effectiveness in real-world navigation systems. Wu et al. (2019) implemented Dijkstra's algorithm in Java for parking berth optimization. Their work highlights the usefulness of the algorithm in solving spatial optimization problems.

Both Eneh and Arinze (2017) and Kumar et al. (2016) demonstrated the effectiveness of Dijkstra's algorithm in emergency response systems, logistics planning, and supply chain optimization. Their findings emphasize the algorithm's suitability for dynamic and time-sensitive environments. Mustapha et al. (2019) analyzed the cognitive complexity of Dijkstra's algorithm implemented in Python and Java. Their study showed that programming language selection influences software maintainability and algorithm implementation. Urubkin et al. (2021) implemented Dijkstra's algorithm using Transact-SQL and relational algebra to improve database query optimization. Similarly, Ismailov and Mirzakhalilov (2018) applied the algorithm in geolocation systems, while Yurchak and Dudas (2019) demonstrated its implementation within SQLite databases. These studies collectively illustrate the algorithm's effectiveness in database and location-based applications. Wang et al. (2019) proposed a three-dimensional Dijkstra optimization algorithm for ship route planning, achieving fuel savings of approximately 5% while maintaining optimal routing. Ojekudo and Akpan (2019) applied Dijkstra's algorithm to optimize transportation routes for product distribution in Dominion Paints Nigeria Ltd., demonstrating improvements in logistics efficiency. Salem et al. (2021) employed Dijkstra's algorithm for flight route planning, successfully identifying cost-effective and time-efficient routes with a reported success rate of approximately 95%. Gunawan et al. (2019) developed a healthcare application that uses Dijkstra's algorithm to determine the shortest route to professional doctors in Bandar Lampung, demonstrating the algorithm's flexibility in healthcare services. Alshammrei et al. (2019) proposed an enhanced Dijkstra algorithm for mobile robot path planning and obstacle avoidance. Their approach continuously updates the graph whenever obstacles are detected, allowing the robot to identify a new optimal path in real time. Liu et al. (2020) integrated Dijkstra's algorithm with deep learning techniques for food delivery path planning. Their work demonstrates the algorithm's compatibility with Artificial Intelligence applications. Galib et al. (2018) proposed a modified version of Dijkstra's algorithm to identify alternative routes for remote communities, improving route accessibility under varying conditions. Sembiring et al. (2017) demonstrated an unconventional application of Dijkstra's algorithm by using it to solve Sudoku puzzles, highlighting its flexibility beyond traditional shortest-path problems.

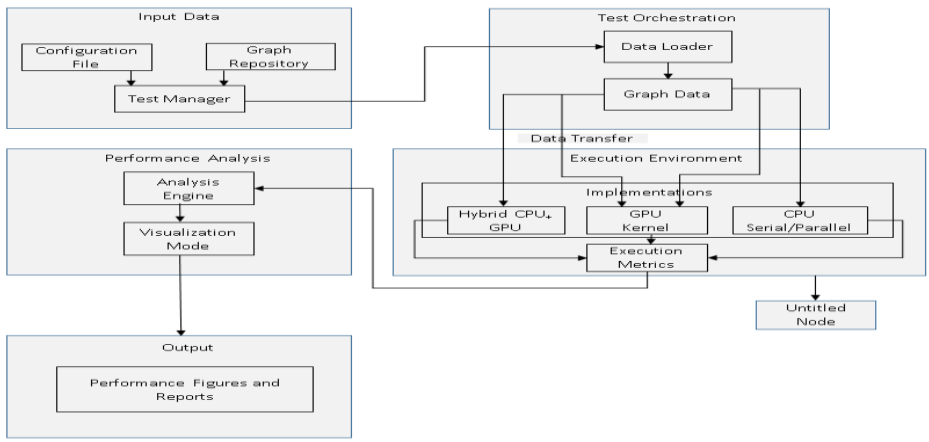
In summary, previous studies have demonstrated the broad applicability of Dijkstra's algorithm in transportation, logistics, healthcare, robotics, databases, geolocation, and artificial intelligence. These studies provide the foundation for the present research, which evaluates the algorithm's performance across different programming languages.

Methodology

The approach utilized in this research study is developed to extensively examine Dijkstra's algorithm by implementing it across numerous programming languages and diverse chart types. The goal is to acquire essential insights into the algorithm's performance, performance, and practical applicability in different situations. The method incorporates 3 key actions: Implementation Details, Test Environment Setup, and Data Collection and Analysis.

Figure 1:

Model of Proposed System



Implementation Details

The foundation of our efficiency research study relies on the precise application of Dijkstra's algorithm within each of the four selected program languages: Python, Java, MATLAB, and C++. Digging much deeper into the rationale behind our choice of programming languages, we not only considered their universality in the existing technological landscape but also their distinguishing characteristics, which play a substantial role in shaping the execution of Dijkstra's Algorithm. Python is often misunderstood due to simplicity, but powerful to build application quickly. Java is statically typed, Object Oriented language that is platforms-independent and very strong.

MATLAB, made by MathWorks, is special language for technical computing. It good for matrix operations, which are important in algorithms and graph theory. Lastly, C++, a general-purpose programming language with low-level memory adjustment functions, uses performance and control over system resources, providing a distinct environment for algorithm execution. By running Dijkstra algorithm in mentioned languages, we can compare and test its efficiency, performance and use. Each language offers special characteristics and idiosyncrasies, and it is within these

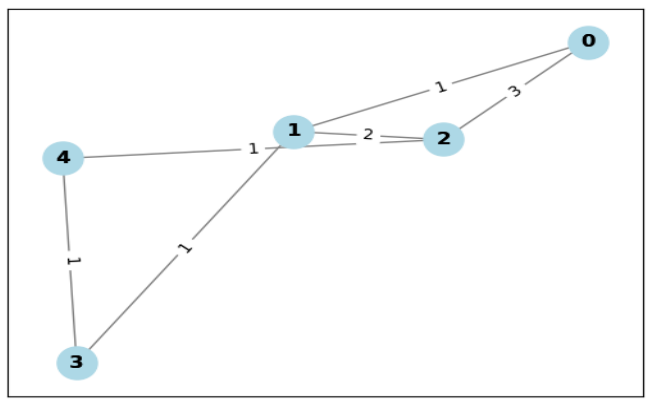
distinctions that we intend to discover how the algorithm's execution is influenced. This choice, for that reason, ensures that our research study findings will resonate with a wide array of specialists and experts. To make sure about study completion, we tested many kinds of graphs and grouped them based on their uses and features. These are simple graphs with only a small number of nodes and edges.

Small Graphs:

These are simple graphs with only a small number of edges and nodes. The decision to consist of such graphs was driven by the need to understand the algorithm's efficiency in circumstances with smaller-sized datasets representing basic or localized situations.

Figure 2:

The Graphical Representation of Small Graph

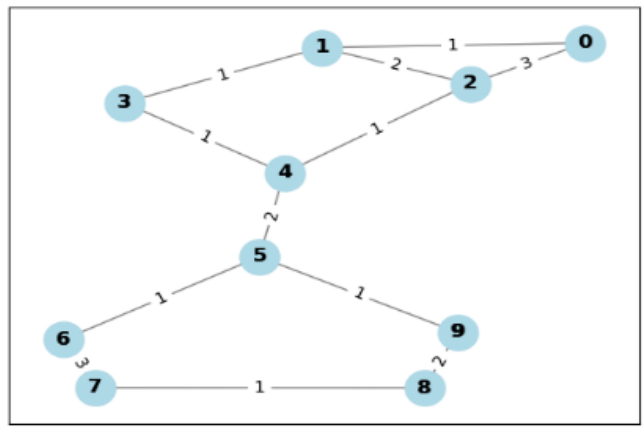


Large Graphs:

They show complex cases with many edges and nodes. We use these graphs to test the algorithm performance and see how it works in heavy computing situations.

Figure 3:

The Graphical Representation of Large Graph

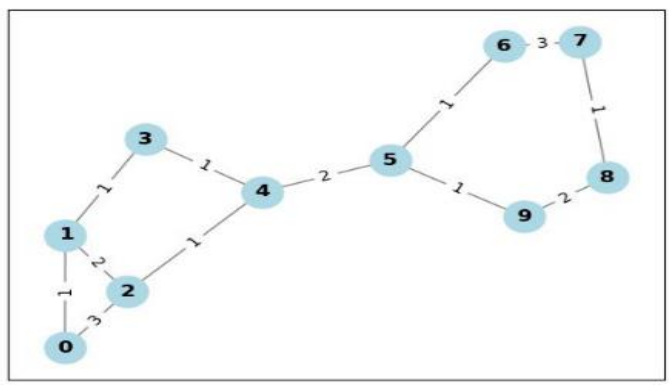


Sparse Graphs:

These graphs, defined by having fewer edges relative to the number of nodes, represent circumstances where the connection is restricted. We utilized sparse graphs to assess the algorithm's performance in discovering the quickest path when the options are not easily evident or when the offered courses are few.

Figure 4:

The Graphical Representation of a Sparse Graph

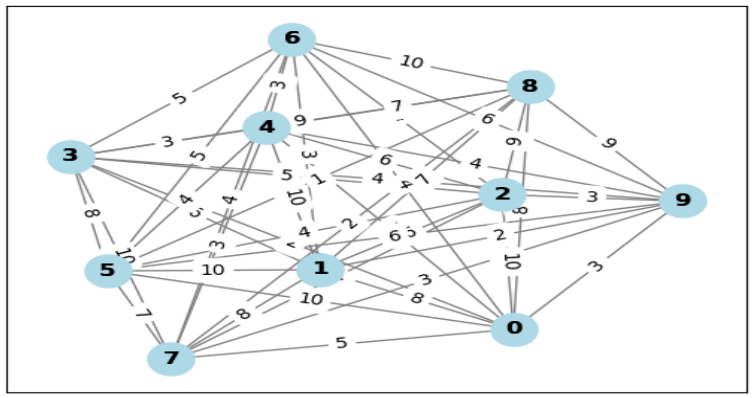


Dense Graphs:

Unlike sporadic graphs, dense graphs have a higher ratio of edges to nodes, symbolizing scenarios with abundant connections. Such conditions allow us to inspect the algorithm's effectiveness and performance under a high computational load.

Figure 5:

The Graphical Representation of a Dense Graph



Using a structured technique to carry out Dijkstra's algorithm on various graph types, our objective is to replicate a wide spectrum of real-life scenarios. By considering different conditions and environments, we aim to develop a comprehensive and robust assessment of the algorithm's efficiency, guaranteeing its applicability throughout diverse situations. Using different types of graphs help us make a balance and complete study. To test the algorithm, we use execution time, efficiency, complexity and memory key performance. By comparing execution time in different languages and graph types, we can better understand the fastness of algorithm.

Memory usage: We measured memory usage by tracking the peak memory utilization throughout the execution of the algorithm. This gives us insights into the memory effectiveness of the algorithm, a critical factor in efficiency, especially in resource-limited environments. This type of analysis helps us to understand algorithm in detail. Such a thorough investigation will further work as a detailed guide for software application designers and engineers seeking to perform Dijkstra's Algorithm optimally in various situations.

In addition, our research study structure goes beyond the mere execution of the algorithm. We have created an extensive testing environment, carried out robust information collection techniques, and prepared thorough analysis methods. Each aspect of our study process is structured to guarantee our findings, highest precision and dependability. Furthermore, we plan to give thoughtful analyses of our results, going over the suggestions and future applications in real-word circumstances. By embracing this multi-pronged technique, we intend to paint a detailed photo of Dijkstra's algorithm's performance, covering every possible angle. We believe this extensive exploration will contribute significantly to Understanding Dijkstra's

Algorithm's performance will work as an important resource for professionals in the field. Our methodical application of Dijkstra's Algorithm across various program languages and graph types uses an in-depth exploration of the algorithm's efficiency in many situations. This comprehensive technique is developed to provide researchers, practitioners, and developers with a thorough understanding of the algorithm's habits across various contexts, informing more effective applications of Dijkstra's algorithm. This method is made to helps developers, researchers engineers, and practitioners fully understand how the algorithm works in different circumstances, helps them to use it more efficiently.

3.2 Test Environment Setup

The tests were done in an organized environment to keep results reliable. All tests used the same computer configuration with an Intel i7 processor, 20GB RAM, and %00 GB SSD, with Windows 11. This setup made sure hardware differences did not affect the results. The latest stable versions of Python, Java, C++ and MATLAB were installed and used for the experiments. This confirmed that we tested the most updated versions of these languages, which most developers would use in the field. To escape intrusion, all the needless applications are closed. Tests were run once at a time, which helped us collect accurate and reliable data.

3.3 Code Implementation

Writing and running code is vital in all languages like Python, MATLAB, C++ and Java. The process means turning an algorithm or idea into working code. No matter the language, coding desires careful thinking, problem-solving, and knowledge of the language's rules. It contains breaking a problem into small steps, making an algorithm, and then written it as code.

Figure 6: *Code Implementation of Dijkstra in Python*

```python
def create_adjacency_matrix(V, G):
    adjacency_matrix = [[0 for _ in range(V)] for _ in range(V)]

    for i in range(V):
        for j in range(V):
            if i == j:
                adjacency_matrix[i][j] = 0
            else:
                adjacency_matrix[i][j] = sys.maxsize

    for i, j, weight in G:
        adjacency_matrix[i][j] = weight

    return adjacency_matrix

def dijkstra(V, G, start_vertex):
    D = {v:float('infinity') for v in range(V)}
    D[start_vertex] = 0

    queue = []
    heapq.heappush(queue, (0, start_vertex))

    while queue:
        (dist, current_vertex) = heapq.heappop(queue)
        if D[current_vertex] < dist:
            continue

        for neighbor, weight in enumerate(G[current_vertex]):
            old_cost = D[neighbor]
            new_cost = D[current_vertex] + weight
            if new_cost < old_cost:
                D[neighbor] = new_cost
                heapq.heappush(queue, (new_cost, neighbor))

    return D
```

The code includes two functions: create adjacency matrix creates an adjacency matrix from a list of edges, appointing weights and representing graph connections. The code determines execution time and memory usage, supplying results for different graph types (small, large, sparse, and dense). Efficiency graphs are produced to visualize the outcomes.

Figure 7: *Code Implementation of Dijkstra in C++.*

```cpp
vector<vector<pair<int, int>>> create_adjacency_list(int V, vector<vector<int>>& G) {
    vector<vector<pair<int, int>>> adjacency_list(V);

    for (auto& edge : G) {
        int i = edge[0];
        int j = edge[1];
        int weight = edge[2];
        adjacency_list[i].emplace_back(j, weight);
    }

    return adjacency_list;
}

vector<int> dijkstra(int V, vector<vector<pair<int, int>>>& G, int start_vertex) {
    vector<int> D(V, INF);
    D[start_vertex] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push(make_pair(0, start_vertex));

    while (!pq.empty()) {
        int dist = pq.top().first;
        int vertex = pq.top().second;
        pq.pop();

        if (D[vertex] < dist)
            continue;

        for (const auto& edge : G[vertex]) {
            int neighbor = edge.first;
            int weight = edge.second;

            if (D[vertex] + weight < D[neighbor]) {
                D[neighbor] = D[vertex] + weight;
                pq.push(make_pair(D[neighbor], neighbor));
            }
        }
    }

    return D;
}
```

The given code snippet consists of 2 functions: create adjacency list and Dijkstra. The create adjacency list function develops an adjacency list representation of a graph based on the supplied edges and weights. The Dijkstra function implements Dijkstra's algorithm to calculate the fastest paths from a provided start vertex to all other vertices in the graph. These functions are helpful for graph-related computations and pathfinding in algorithms and applications.

Figure 8:

Code Implementation of Dijkstra in Java

```java
public class App {
    static final int INF = Integer.MAX_VALUE;

    static int[][] createAdjacencyMatrix(int V, List<int[]> G) {
        int[][] adjacencyMatrix = new int[V][V];

        for (int i = 0; i < V; i++) {
            Arrays.fill(adjacencyMatrix[i], INF);
            adjacencyMatrix[i][i] = 0;
        }
        for (int[] edge : G) {
            int i = edge[0], j = edge[1], weight = edge[2];
            adjacencyMatrix[i][j] = weight;
        }
        return adjacencyMatrix;
    }

    static int[] dijkstra(int V, int[][] G, int startVertex) {
        int[] D = new int[V];
        Arrays.fill(D, INF);
        D[startVertex] = 0;

        PriorityQueue<int[]> queue = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
        queue.add(new int[]{0, startVertex});

        while (!queue.isEmpty()) {
            int[] array = queue.poll();
            int dist = array[0], currentVertex = array[1];

            if (D[currentVertex] < dist) continue;

            for (int neighbor = 0; neighbor < V; neighbor++) {
                int weight = G[currentVertex][neighbor];
                if (weight == INF) continue;  // Skip if there is no direct edge
                int oldCost = D[neighbor];
                int newCost = D[currentVertex] + weight;
                if (newCost < oldCost) {
                    D[neighbor] = newCost;
                    queue.add(new int[]{newCost, neighbor});
                }
            }
        }
        return D;
    }
```

The code executes Dijkstra's algorithm to find the fastest paths in a weighted graph utilizing an adjacency matrix representation. It efficiently computes the quickest distances from a start vertex to all other vertices by iteratively picking the vertex with the smallest tentative distance and upgrading the ranges of neighboring vertices. Here explores the interaction between the algorithm's performance, the nature of the graph, and the homes of the shows' language in use. Starting with Python, the data shows that Python's memory consumption across various graphs is nearly minimal. This element of Python could be attributed to the language's high-level and dynamic nature that allows it to manage memory resources effectively. Dijkstra's algorithm, in essence, relies greatly on passing through nodes and edges, and Python's capability to manage these operations with a minimal memory footprint makes it a feasible choice for implementation. Nevertheless, Python's execution time presents a somewhat varied story. While the distinction between execution times for numerous graphs is very little, Python shows a greater execution time relative to the

other languages evaluated. This observation aligns with the commonly understood fact that Python, being an interpreted language, is typically slower than its compiled counterparts, and the results here bear that out.

Table 1

Table Representation of the Dijkstra algorithm in Python

	Memory Usage	**Execution Time**
Small Graph	0.0 MB	0.101466 seconds
Large Graph	0.00390625 MB	0.108255 seconds
Sparse Graph	0.0 MB	0.109025 seconds
Dense Graph	0.0078125 MB	0.095101 seconds

Figure 9:

Memory Usage and Execution Time

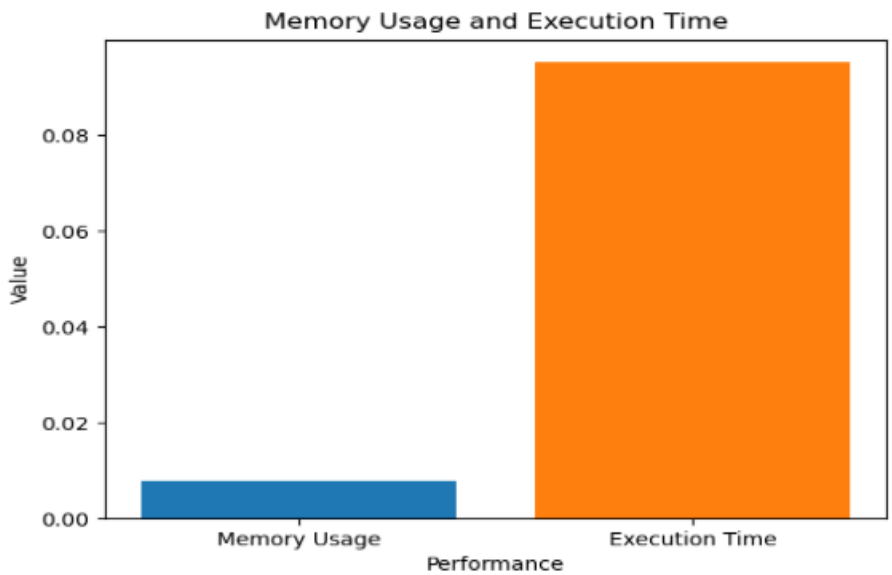


Figure 10:

Code Implementation of Dijkstra in MATLAB.

```
function [D, memory_used] = dijkstra_algo(V, adjacency_list, start_vertex)
    D = inf(1, V);
    D(start_vertex) = 0;
    Q = 1:V;

    memory_used = 0; % Initialize memory usage

    while ~isempty(Q)
        [~, min_index] = min(D(Q));
        u = Q(min_index);
        Q(min_index) = [];
        neighbors = adjacency_list{u};
        for i = 1:size(neighbors, 1)
            v = neighbors(i, 1);
            weight = neighbors(i, 2);
            alt = D(u) + weight;
            if alt < D(v)
                D(v) = alt;
            end
        end

        % Measure memory usage at each iteration
        current_memory = memory;
        memory_used = max(memory_used, current_memory.MemUsedMATLAB);
    end

    D = [D, memory_used]; % Append memory usage to the result
end
```

The code carries out Dijkstra's algorithm utilizing an adjacency list representation to find the fastest courses in a weighted graph. It iteratively updates the distances of neighboring vertices based on the current fastest path. The algorithm tracks memory usage and returns the resulting variety of quickest distances, along with the maximum memory used throughout execution.

Discussion

Delving much deeper into the study results, observing the performance of Dijkstra's algorithm throughout the spectrum of programming languages in the context of nu- numerous graph types is interesting. The analysis presented

Figure 11:

The Graphical Behavior of Dense Graph in Python

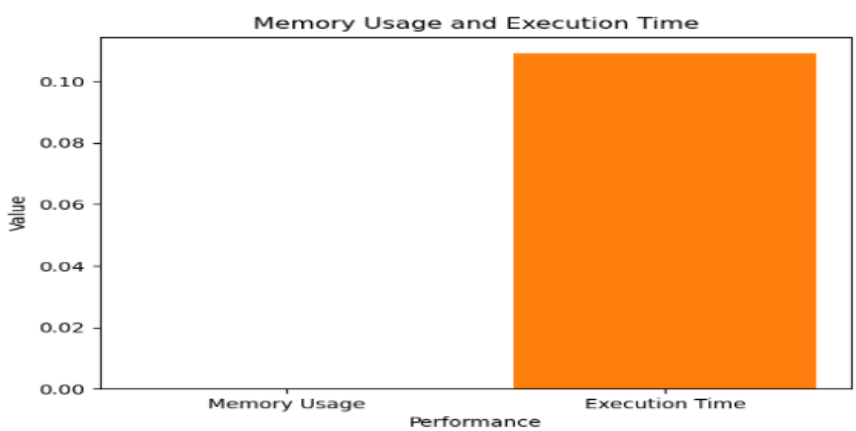


Figure 12:

The Graphical Behavior of Spars Graph in Python

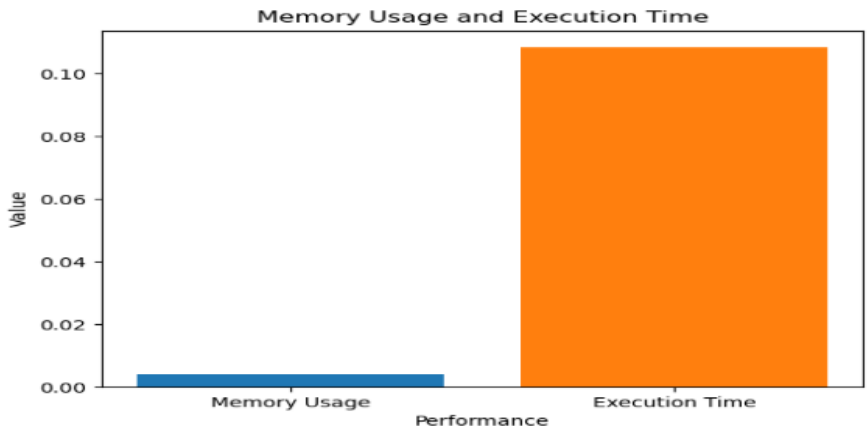


Figure 13:

The Graphical Behavior of Dense Graph in C++

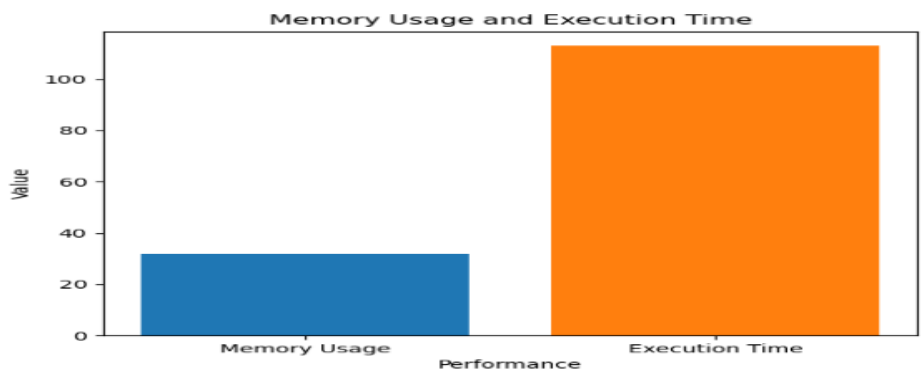
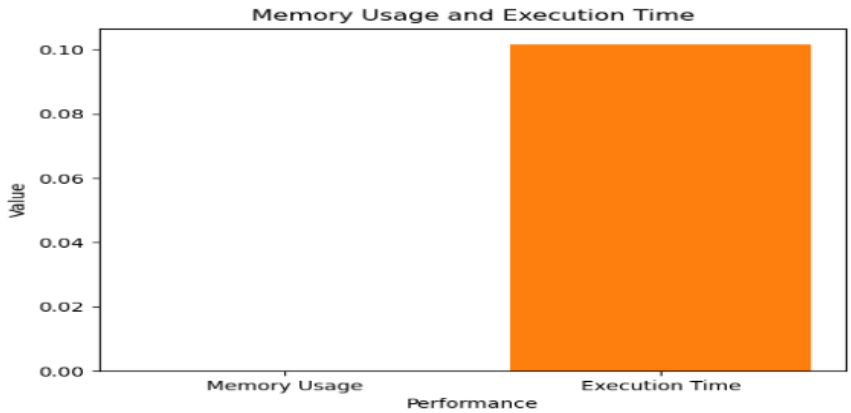


Figure 14:

The Graphical Behavior of Small Graph in Python



Proceeding to C++, there is an obvious divergence in performance compared to Python. While the memory use stays low, it is not as minimal as Python, showing that the more explicit and lower-level memory management in C++ might lead to a somewhat greater memory footprint. Nevertheless, the significant point of divergence is the execution time. Unlike Python, C++ reveals a considerable increase in execution time for all types of graphs, with dense graphs experiencing the most dramatic boost. This observation might be surprising given C++'s credibility for speed and efficiency. Nevertheless, it is important to consider that C++'s low-level nature, while using higher control and capacity for optimization, likewise places more responsibility on the programmer for efficient memory management and optimization. The longer execution times for dense graphs, in particular, suggest that these complexities can accumulate in C++, especially for more complex data structures like dense graph.

Table 2

Table Representation of the Dijkstra algorithm in C++

	Memory Usage	**Execution Time**
Small Graph	0 Bytes	22 seconds
Large Graph	32 Bytes	36 seconds
Sparse Graph	16 Bytes	41 seconds
Dense Graph	32 Bytes	113 seconds

Figure 15:

The Graphical Behavior of Sparse Graph in C++

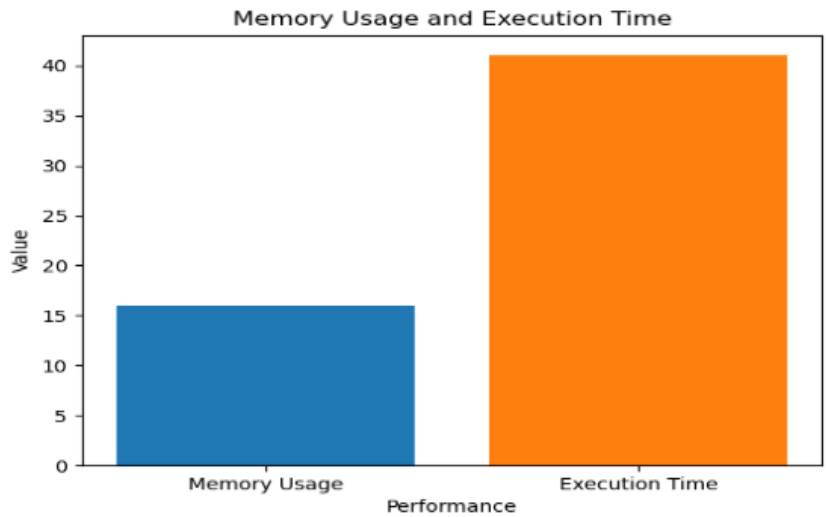


Figure 16:

The Graphical Behavior of Large Graph in C++

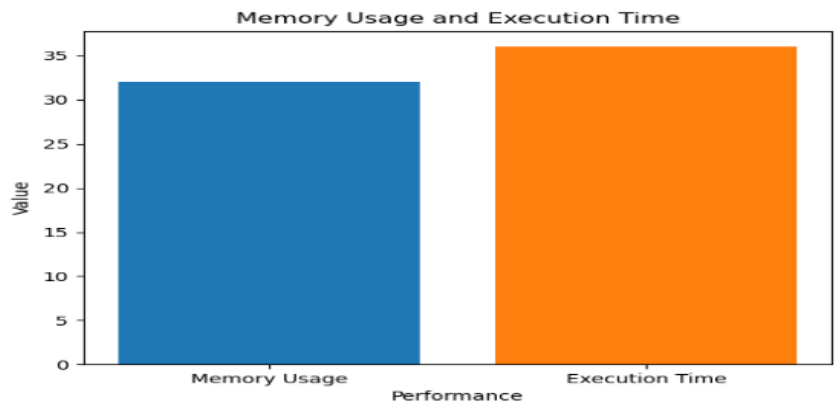
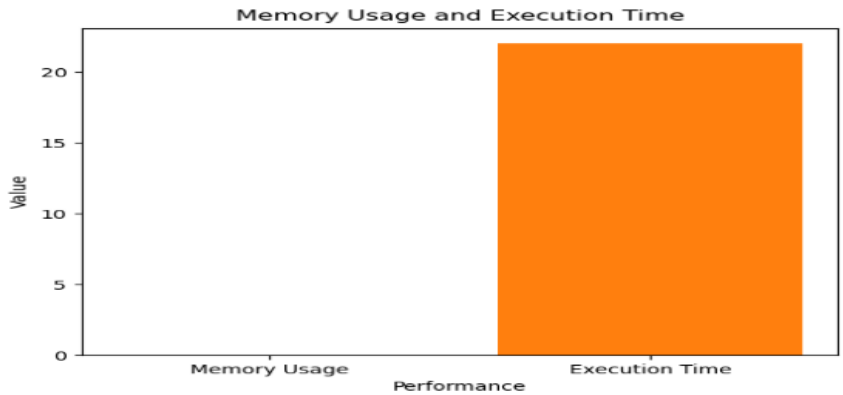


Figure 17:

The Graphical Behavior of Small Graph in C++



Next, we focus on MATLAB, a language created mostly for numerical and matrix calculations. MATLAB reveals a significantly higher memory use for all kinds of graphs.

This is perhaps because of how MATLAB grips data using arrays which use a lot of memory but are enhanced for the math operations that Dijkstra algorithm needs. On the other hand, MATLAB speed is good, meaning that the way its optimized for numbers helps makeup the extra memory it uses. Remarkably, dense graphs do not show a significantly higher execution time than other graph types, indicating that MATLAB's optimizations may especially shine when handling these sorts of complex mathematical structures.

Table 3:

Representation of the Dijkstra Algorithm in MATLAB

Graphs	Memory Usage	Execution Time
Small Graph	2.489 MB	0.36894 seconds
Large Graph	2.639 MB	0.53290 seconds
Sparse Graph	2.297 MB	0.66297 seconds
Dense Graph	2.592 MB	0.63247 seconds

Figure 18:

The Graphical Behavior of Dense Graph in MATLAB.

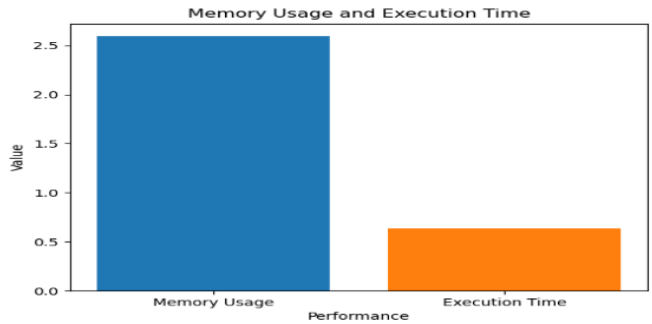


Figure 19:

The Graphical Behavior of the Sparse Graph in MATLAB.

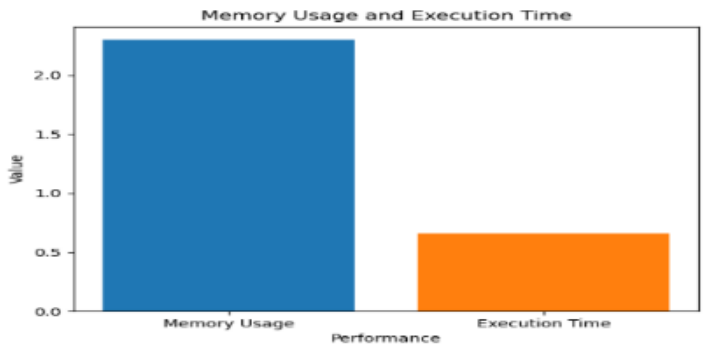
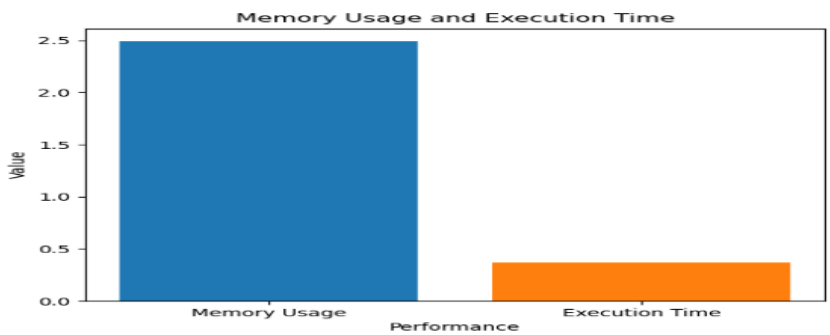


Figure 20:

The Graphical Behavior of Small Graph in MATLAB.



Finally, Java's performance reveals a great balance between memory use and execution time, recommending that it effectively manages Dijkstra's algorithm's complexities throughout various graph types. This might be due to Java's robust memory management and Just-In-Time (JIT) collection that optimizes code execution at runtime. Interestingly, the execution time is really low for large graphs, even though one may expect these more complicated structures to need more processing time. This may suggest that Java's abstractions and optimizations supply particular advantages when managing more complex data structures. To this end, it is clear that there is no" one-size-fits-all" service. Python might be the language of choice for scenarios where memory use is an issue and where execution speed is less of an issue.

Table 4:

Representation of the Dijkstra Algorithm in Java

	Memory Usage	**Execution Time**
Small Graph	0.08625 MB	0.01891 seconds
Large Graph	0.0 MB	3.837E-5 seconds
Sparse Graph	0.08634 MB	0.01859 seconds
Dense Graph	0.08634 MB	0.02075 seconds

Figure 21:

The graphical behavior of Large Graph in MATLAB.

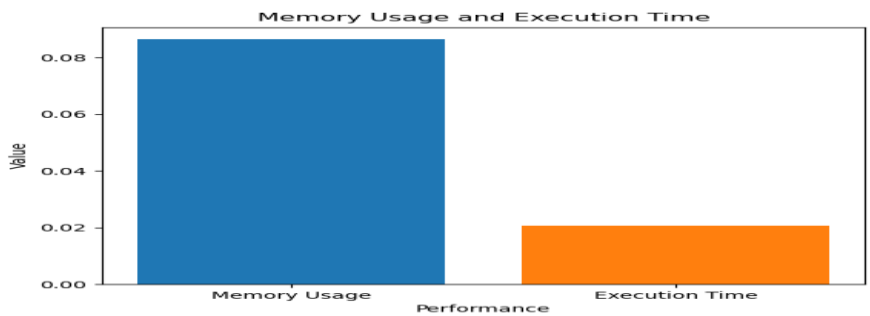


Figure 22:

The Graphical Representation of a Dense Graph in Java.

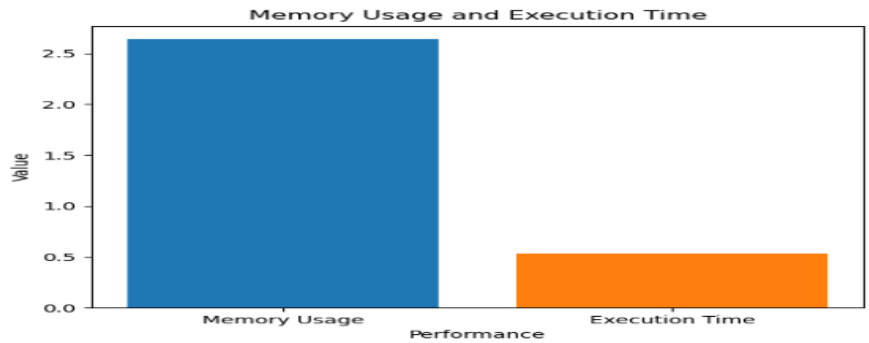


Figure 23:

The Graphical Representation of a Sparse Graph in Java.

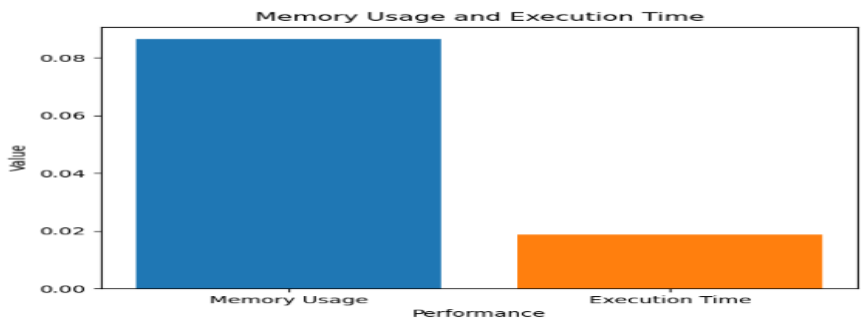


Figure 24:

The Graphical Representation of a Large Graph in Java.

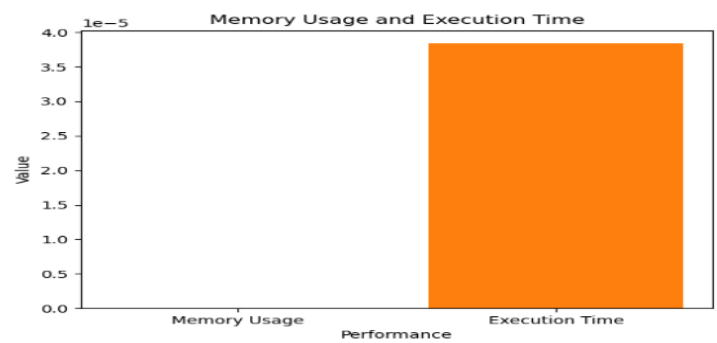
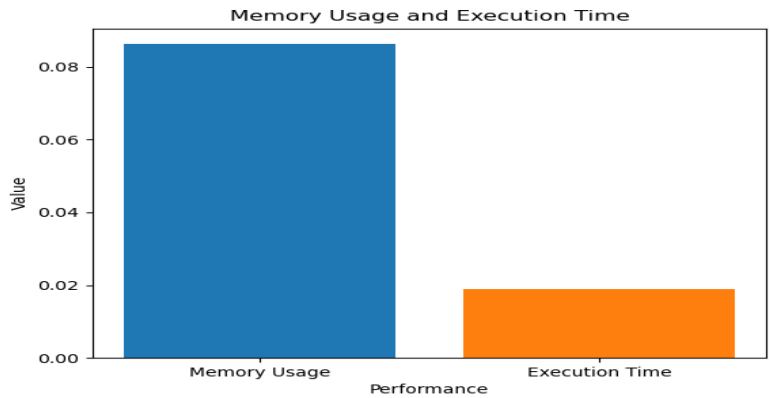


Figure 25:

Small Graph in Java.



From the above conversations, it is clear that the choice of program language can substantially affect the performance of Dijkstra's algorithm. Furthermore, different graphs can act differently under the same language due to their structural distinctions and the varying methods by which the languages manage these structures. On the other hand, if you need fast performance and have enough memory, Java is better than others. MATLAB will be good for mathematically problem s even though it uses more memory because it run quickly. C++ could be best pick when you need detailed control. The study results show that choosing the right programming language depends on thinking about what the task really needs.

Conclusion

Our study considerably deepened our understanding of the efficiency characteristics of Dijkstra's algorithm through numerous programs, languages, and graph structures. We found out the programming language you choose affects how well Dijkstra algorithm works, when it comes to using memory and how long it takes to implement. Python demonstrated effective memory monitoring, albeit with longer execution times, revealing a compromise with its simplicity and ease of use. On the other hand, C++ had relatively low memory use but longer execution times, especially with dense graphs. MATLAB is powerful but uses extra memory, while Java is fat and uses memory efficiently. This study shows that best programing language depend on the task. For example, Python may be good when memory is restricted, and Java may be better when speed is vital. But as the languages enhance and faster methods are developed, performance can change, so more research is needed. Other Things like data size, algorithm complexity, hardware and program skill also affect how well Dijkstra algorithm works. In short, our study shows that choosing of right language is very important. Knowing the weakness and strengths of each language can help

engineers and developers make better choices and guide future research. Future study could also analyze at how easy language is to maintain, debug and connect with other systems.

References:

Alshammrei, A., et al. (2019). Enhanced Dijkstra algorithm for mobile robot path planning and obstacle avoidance. In Proceedings of the 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)

Dinitz, Y., & Itzhak, R. (n.d.). Hybrid Bellman–Ford–Dijkstra algorithm. Department of Computer Science, Ben-Gurion University of the Negev.

Eneh, A., & Arinze, O. (2017). Implementation of Dijkstra algorithm in finding shortest path for emergency response. In Proceedings of the International Conference on Cybersecurity, Communication, and Computing (IC4).

Fitriansyah, F., et al. (2018). Development of geographical mapping using Dijkstra algorithm to determine shortest path of tourist locations in Bali. In Proceedings of the International Seminar on Application for Technology of Information and Communication (iSemantic).

Galib, M., et al. (2018). Shortest path calculation for remote community using Dijkstra algorithm. In Proceedings of the International Conference on Informatics, Electronics and Vision (ICIEV).

Gunawan, G., et al. (2019). Implementation of Dijkstra algorithm for shortest path to find professional doctors in Bandar Lampung. In Proceedings of the International Conference on Information Management and Technology (ICIMTech).

He, X. (2019). Research on the application of Dijkstra algorithm in self-driving and public transportation. In Proceedings of the 2019 International Conference on Computer Information and Big Data Applications (CIBDA).

Ismailov, I., & Mirzakhalilov, M. (2018). Optimal routes generation in geolocation systems using Dijkstra algorithm. In Proceedings of the International Conference on Information Science and Communications Technologies (ICISCT).

Kumar, A., et al. (2016). Application of Dijkstra algorithm in logistics and supply chain management. International Journal of Latest Engineering and Management Research, 1(6).

Liu, Y., et al. (2020). Integration of Dijkstra's algorithm with deep inverted support for food delivery path planning. In Proceedings of the IEEE 5th International Conference on Signal and Image Processing (ICSIP).

Mirahadi, F., & McCabe, B. Y. (2021). EvacuSafe: A real-time model for building evacuation based on Dijkstra's algorithm. Department of Civil and Mineral Engineering, University of Toronto.

Mustapha, M., et al. (2019). A cognitive complexity analysis of Dijkstra's algorithm using Python and Java. International Journal of Computer Science and Information Security, 17(4).

Ojekudo, N., & Akpan, E. (2019). Optimal routing of goods and services using Dijkstra's algorithm: A case study of Dominion Paints Nig. Ltd. Nigerian Journal of Technology, 38(1).

Ridlo Al-Hakim, R., et al. (2020). Real-time COVID-19 tracking application using Dijkstra's algorithm in React Native. International Journal of Advanced Computer Science and Applications, 11(9).

Salem, M., et al. (2021). Organizing flights between cities using Dijkstra algorithm. In Proceedings of the International Conference on Computer Applications and Information Security (ICCAIS).

Sembiring, R., et al. (2017). Implementation of Dijkstra algorithm for solving Sudoku game. In Proceedings of the 1st International Conference on Informatics and Computational Sciences (ICICoS).

Sun, Y., et al. (2021). AGV path planning based on improved Dijkstra algorithm. Journal of Physics: Conference Series, 1746.

Urubkin, A., et al. (2021). Using Dijkstra's algorithm in relational algebra to optimize SQL queries. Information Technology and Computer Science, 6(1).

Wang, H., et al. (2019). Three-dimensional Dijkstra algorithm based on ship trip optimization. In Proceedings of the International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM).

Wu, X., et al. (2019). Optimization of parking lot berth assistance using Dijkstra algorithm. In Proceedings of the 2nd International Conference on Intelligent Transportation Engineering (ICITE).

Yurchak, O., & Dudas, V. (2019). Application of Dijkstra's algorithm in SQLite database. In Proceedings of the IEEE 10th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS).