Online-Journals.org

iJIM International Journal of Interactive Mobile Technologies

iJIM | eISSN: 1865-7923 | Vol. 20 No. 13 (2026) | OPEN ACCESS

https://doi.org/10.3991/ijim.v20i13.61989

PAPER

Evaluating the Influence of Graph Density on the Efficiency of Shortest Path Algorithms Using Different Data Structures

Velin Kralev (✉), Radoslava Kraleva, Aleksandra Popova

South West University, Blagoevgrad, Bulgaria

velin_kralev@swu.bg

ABSTRACT

This paper presents a comparative analysis of two variants of a classical algorithm for finding the shortest path in a connected graph. The first variant uses an adjacency matrix (AM) to verify the existence of an edge (arc) between two vertices, while the second variant performs the same verification using an adjacency list (AL). The objective of this study is to examine how graph density affects the performance of the two algorithmic modifications depending on the data structure used. A total of 95 graphs were analyzed, grouped into five sets from 100 to 500 in increments of 100. For each group, 19 graphs were generated with densities ranging from 5% to 95% in increments of 5%. The methodology includes analyzing the number of iterations, assignments, and comparisons executed by the algorithms for all graphs. The initial hypothesis assumed that the total number of operations would always be lower when using an AL instead of an adjacency matrix, regardless of graph density. The results demonstrate that this assumption is incorrect: for densities above 82%, the total number of operations is lower when using an adjacency matrix, whereas the AL is more efficient for densities below 82%, with its efficiency increasing as density decreases. These findings are particularly important for mobile technologies, as they support the design of more efficient pathfinding solutions that optimize performance and energy consumption in mobile applications.

KEYWORDS

graph theory, shortest path, adjacency matrix (AM), adjacency list (AL), graph density, data structures

1 INTRODUCTION

Graph theory, as an essential part of discrete mathematics, has significantly evolved over the past decades [1], offering abstract non-linear data structures that enable the visual and highly effective formalization and solution of complex applied

Kralev, V., Kraleva, R., Popova, A. (2026). Evaluating the Influence of Graph Density on the Efficiency of Shortest Path Algorithms Using Different Data Structures. *International Journal of Interactive Mobile Technologies (iJIM)*, 20(13), pp. 84–98. https://doi.org/10.3991/ijim.v20i13.61989

Article submitted 2026-03-15. Revision uploaded 2026-05-06. Final acceptance 2026-05-13.

© 2026 by the authors of this article. Published under CC-BY.

problems across various fields through appropriate algorithms. This development presupposes determining precise structural and numerical characteristics of real-world objects represented by graph structures, which is crucial for their modeling and algorithmic processing [2]. Numerous applied problems such as transportation, allocation, optimization, route planning, service center placement, scheduling, and timetabling, as well as modeling in the fields of learning and assessment, are described by graphs and solved through appropriate algorithms, enabling the representation and study of various systems and processes [3, 4]. Nevertheless, solving these problems without a computer is impossible, and even with software some remain unsolvable within acceptable time limits, particularly those belonging to the NP-complete and NP-hard complexity classes, for which no solution with complexity other than exponential has yet been found [5]. Some problems are linear and solvable through polynomial algorithms, but real-world problems with large input sizes require the search for efficient algorithms that generate approximate solutions, with graph theory offering significant opportunities in this direction [6, 7]. Therefore, it is essential to develop interactive applications for working with graphs that visualize the modeled problems, test the corresponding algorithms, and enable the analysis of results through graphical representation.

Finding shortest paths in graphs is a fundamental problem with critical importance for optimizing transportation and logistics networks, routing in communication systems, and navigation in geographic information systems. The efficient solutions reduce costs, improve performance, and ensure reliability in critical infrastructures, while, from a scientific perspective, contributing to the advancement of graph theory, algorithmic complexity, and optimization methods [8]. The primary approach to solving this problem is Dijkstra's algorithm, which enables the efficient determination of the shortest path from a single source vertex to all other vertices in a given weighted graph. A specific case of this algorithmic strategy involves specifying a target vertex, in which the algorithm terminates once the shortest path to the target vertex has been found, regardless of whether shortest paths to other vertices remain unresolved. Numerous modifications of this algorithm have been proposed in the scientific literature, addressing various aspects of its optimization. These include path optimization under equal edge weights and generation of multiple alternative routes, as well as enhancing security in communication networks [9–11].

The practical application of the algorithm is associated with optimizing the functions of various robotic systems [12], as well as improving traffic flow in transportation networks, where input data can be obtained both from traffic control systems and from different navigation systems. This enables the modeling of additional constraints related to road infrastructure, as well as the specification of dynamic conditions for vehicle movement in real time [13–15]. When real-world systems employing the algorithm are very large, additional optimization approaches have been developed and applied, targeting both the data structures used for system modeling and methods for improving the algorithm's execution, which significantly enhance its performance compared to its classical implementation [16–20]. In the educational domain, where systems and their models are smaller in scale, the algorithm is most often applied in the learning process focused on modeling, using, and analyzing the applicability of various data structures and approaches, such as arrays, priority queues, trees, and dynamic programming [21]. This application of the algorithm is primarily oriented toward teaching rather than modeling the educational system itself, as the emphasis lies on how and through which data structures the algorithm's efficiency can be improved by reducing its overall

execution time, rather than on using it to model the interrelations within the learning process [22, 23].

Another aspect of applying Dijkstra's algorithm to optimization problems in large and clustered systems arises when this exact algorithm is combined with other approaches. These approaches may rely on approximate methods for solving optimization tasks, incorporating local search and genetic operators for crossover and mutation, as well as hybrid techniques based on metaheuristic and memetic strategies [24, 25]. The integration of classical shortest-path algorithms with other methods, both hybrid and evolutionary, enables the development of efficient solutions for a wide range of problems, including those related to transportation and industrial networks, as well as educational strategies and methodologies [26, 27]. These approaches represent a promising foundation for future research and development in the optimization of various algorithms for analyzing and exploring graph structures [28–32].

The primary objective of this study is to conduct a comparative analysis of two modifications of the classical Dijkstra algorithm for finding shortest paths, with the comparison based on their efficiency across different types of graphs with varying density and the use of different data structures for storing the required graph-related information. The number of iterations, comparisons, and assignments for both modifications is evaluated, and based on the experimental results, recommendations are formulated for selecting an appropriate structure depending on the graph's density. The outcomes of this study are particularly relevant for mobile technologies, where selecting appropriate graph representations can significantly improve the efficiency, responsiveness, and energy consumption of mobile applications involving pathfinding and network optimization.

2 MATERIAL AND METHODS

This section presents the implementations of two modifications of Dijkstra's algorithm for finding minimal paths in graphs. The program codes for each modification are provided, including additional instructions that allow counting and subsequent analysis of the number of iterations, assignments, and comparisons executed in both algorithm variants. The analysis of the obtained data will address the question of which modification is more suitable for different types of graphs, depending on their density, with the goal of minimizing the number of operations performed as well as the memory required during the computational process. The comparison between the two algorithmic strategies aims to highlight differences in efficiency and applicability when using different data structures to represent graphs that vary in density. The core idea of this algorithmic strategy is to iteratively select the vertex with the minimal current distance value from the starting vertex and update the distances of its adjacent vertices. By relaxing the edges incident to the currently analyzed vertex, it is ensured that the distances are updated optimally until all vertices have been fully processed. This approach is effective for graphs with non-negative edge weights.

The first modification of the algorithm uses a vertex list, an edge list, and an adjacency matrix (AM) to represent the graph. Additionally, a linear structure is used to store information about the nearest neighboring vertices for each vertex in the graph during the computational process. At each iteration, the vertex with the smallest distance value to its adjacent vertices is selected, which is implemented through a linear search among all unprocessed vertices. In the next step, the distances to the

neighbors of the analyzed vertex are updated, but only if a shorter path is found that passes through the current vertex and leads to its adjacent vertex. The main advantage of this approach is its simplicity; however, the trade-off is a higher time complexity, respectively $O(V^2)$, since each iteration involves a linear search for the minimum value across all vertices of the graph. The code for this implementation of the algorithm is presented in Algorithm 1.

Algorithm 1: Source Code of the Dijkstra-AM Algorithm Uses the AM Data Structure

```
var Iterations := 0; var Comparisons := 0; var
Assignments := 0;
var Start := StrToInt(EditStart.Text);
for var I := 1 to VCount do begin
  SetUsed(I, 0); SetPrev(I, 0); SetDist(I, INF);
  Inc(Iterations); Inc(Assignments, 3);
end;
SetDist(Start, 0); SetPrev(Start, 0);
Inc(Assignments, 2);
repeat
  Inc(Iterations);
  var J := 0; var D := INF; Inc(Assignments, 2);
  for var I := 1 to VCount do begin
    Inc(Iterations); Inc(Comparisons);
    if (GetUsed(I) = 0) then begin
      Inc(Comparisons);
      if (GetDist(I) < D) then begin
        D := GetDist(I); J := I; Inc(Assignments, 2);
      end;
    end;
  end;
  Inc(Comparisons); if (J = 0) then Break;
  SetUsed(J, 1); Inc(Assignments);
  for var I := 1 to VCount do begin
    Inc(Iterations); Inc(Comparisons);
    if (GetUsed(I) = 0) then begin
      Inc(Comparisons);
      if (GetWMatrixValue(J, I) > 0) then begin
        Inc(Comparisons);
        if (GetDist(I)>
          (GetDist(J)+GetWMatrixValue(J, I))) then
        begin
          SetDist(I, (GetDist(J)+
            GetWMatrixValue(J, I)));
          SetPrev(I, J); Inc(Assignments, 2);
        end;
      end;
    end;
  end;
until (False);
ShowShortestPathTree(Start);
```

At the beginning, three counters: Iterations, Comparisons, and Assignments, are defined to measure the primary operations (line 01). The starting vertex is read from the user interface (line 02). The state structures for all vertices are then initialized: each vertex is marked as unused (SetUsed(I, 0)), its predecessor is reset (SetPrev(I, 0)), and its distance is set to infinity (SetDist(I, INF)), with each loop pass incrementing the corresponding counters (lines 03–06). The starting vertex is initialized with zero distance and a null predecessor (line 07), which matches the standard setup of Dijkstra's algorithm. These counters enable subsequent empirical analysis of the operational complexity depending on the graph's size and density.

The main loop is controlled by a repeat ... until (False) statement (lines 08 and 36), with termination achieved through break when no candidate remains (line 20). At the beginning of each iteration, local variables are initialized: J:= 0 and D:= INF (line 10). A linear search is performed to find the "next" vertex with the smallest current distance among all yet-unprocessed vertices: the inner loop traverses all vertices (lines 11–19), checks whether a vertex is unused (GetUsed(I) = 0, lines 13–18), and, if so, compares its current distance with the temporary best D (lines 15–17). Upon finding a smaller distance, it updates D and selects J:= I as the candidate (line 16). If, after the full linear search, no candidate is found (J = 0), the algorithm terminates (line 20); otherwise, the chosen vertex is marked as "used" (SetUsed(J, 1), line 21), and the process proceeds to relaxation. This selection mechanism implements the "classical" $O(V^2)$ variant of Dijkstra, where each iteration performs a linear minimum search across all vertices.

After marking the selected vertex J (line 21), relaxation toward all of its neighbors follows (lines 22–35). In the inner loop, already processed vertices are skipped (GetUsed(I) = 0, lines 24–34). The existence of an edge between J and I is checked through the AM value GetWMatrixValue(J, I) (lines 26–27); a positive value indicates an edge, while 0 indicates none. If the edge exists, the standard relaxation test is performed: the current distance to I is compared with the potential new distance that passes through J (lines 28–32). When a shorter path is discovered, the distance is updated via SetDist(I, GetDist(J) + GetWMatrixValue(J, I)), and the predecessor is set with SetPrev(I, J) (lines 30–31). Finally, after the main loop exits (by break when no unprocessed vertex with finite distance remains), the shortest-path tree from the starting vertex is displayed (line 37). Using an AM yields constant-time edge existence checks but can be memory-intensive for sparse graphs; coupled with linear selection, this implementation is typical for educational purposes and for analyzing counts of iterations, comparisons, and assignments.

The second variant of the algorithm uses a vertex list, an edge list, and an adjacency list (AL) instead of an AM to represent the graph. Similar to the previous version, it also uses a linear structure to store information about the closest adjacent vertices for each vertex in the graph. After initializing the distances, the algorithm maintains a set of already processed vertices, and at each iteration, it again selects the vertex with the smallest distance value through a linear search among all unprocessed vertices. The distances to its adjacent vertices are then updated if a shorter path is found. The program block from the source code of the algorithm that uses an AL is shown in Algorithm 2. Since the remaining part of the code related to the initialization of structures and initial values is identical, only the block of program code that differs from the first variant of the algorithm is presented for brevity.

Algorithm 2: Source Code of the Dijkstra-AL Algorithm Uses the AL Data Structure

```
22 | for var Col := 1 to GetVertexDegree(J) do begin
23 | | Inc(Iterations);
24 | | var Adj := GetAListValue(J, Col);
25 | | Inc(Assignments); Inc(Comparisons);
26 | | if (GetUsed(Adj) = 0) then begin
27 | | | Inc(Comparisons);
28 | | | if (GetDist(Adj)>
          (GetDist(J)+GetWMatrixValue(J,Adj))) then
29 | | | begin
30 | | | | SetDist(Adj, (GetDist(J) +
            GetWMatrixValue(J, Adj)));
31 | | | | Inc(Assignments);
32 | | | | SetPrev(Adj, J); Inc(Assignments);
33 | | | end;
34 | | end;
35 | end;
```

The loop iterates over the adjacent vertices of the currently selected vertex J, ranging from Col = 1 to GetVertexDegree(J) (line 22). Each pass increments the Iterations counter (line 23). The algorithm filters out already processed vertices using if (GetUsed(Adj) = 0) (lines 26–27). This ensures that subsequent relaxation work applies exclusively to vertices that have not yet been finalized. Adjacency is implicitly guaranteed by enumeration from the AL. The algorithm directly evaluates the standard Dijkstra relaxation predicate: GetDist(Adj) > GetDist(J) + GetWMatrixValue(J, Adj) (line 28). When the predicate holds (lines 29–33), it updates the tentative distance to Adj with SetDist(Adj, GetDist(J) + GetWMatrixValue(J, Adj)) (line 30), increments Assignments (line 31), sets the predecessor SetPrev(Adj, J) (line 32), and increments Assignments again (line 32). The loop continues until all adjacent vertices of J have been processed (line 35).

In the adjacency-list variant (lines 22–35), the relaxation loop iterates over the true neighbors of the selected vertex J using the AL and the degree of J. For each adjacent vertex Adj, already processed vertices are skipped, and a single relaxation condition comparing the current distance to Adj against the path via J using the edge weight from the weight matrix is evaluated. On success, the algorithm updates both the distance and the predecessor while explicitly tracking assignments and comparisons. This design yields efficient neighbor traversal (bounded by deg(J) per iteration) and maintains constant-time access to edge weights.

The implementations of Dijkstra's algorithm using an AM and an AL exhibit significant structural differences that directly affect the number of fundamental operations: iterations, comparisons, and assignments. In the matrix-based representation, the relaxation phase scans all vertices, resulting in a higher number of checks but a simpler control logic. In contrast, the adjacency-list representation iterates only over actual neighbors, which substantially reduces the number of iterations and comparison operations in sparse graphs. However, in highly dense graphs (above 70%), the matrix-based implementation proves more efficient, an interesting and somewhat unexpected outcome related to the overhead of accessing ALs and managing additional structures. These observations emphasize that the choice of graph representation is not universal but depends on the characteristics of the input data, particularly the graph's density. The following section describes the experimental methodology used to validate this conclusion and presents quantitative results for different scenarios.

3 EXPERIMENTAL PROCEDURE AND METHODOLOGY

The analysis focuses on the execution of Dijkstra's algorithm using two different graph representations: an AM and an AL. The objective is to measure the number of basic operations (iterations, assignments, and comparisons) performed by the algorithm under each structure. The algorithm employs an array for storing distances and does not include a priority queue, which allows emphasizing its classical implementation and the impact of the chosen representation on efficiency. In the modified versions of Dijkstra's algorithm, additional code has been inserted at points where these operations occur to calculate their counts for graphs of varying sizes. The analysis is conducted through experimental measurements using real tests. For this purpose, graphs with different densities are utilized, and the results of the algorithm's execution are compared when using an AM versus an AL. For the purposes of the experimental analysis, an application (named GraphExplorer) was developed that implements Dijkstra's algorithm in two versions: one using an AM and the other using an AL. A working session with the GraphExplorer application is shown in Figure 1.

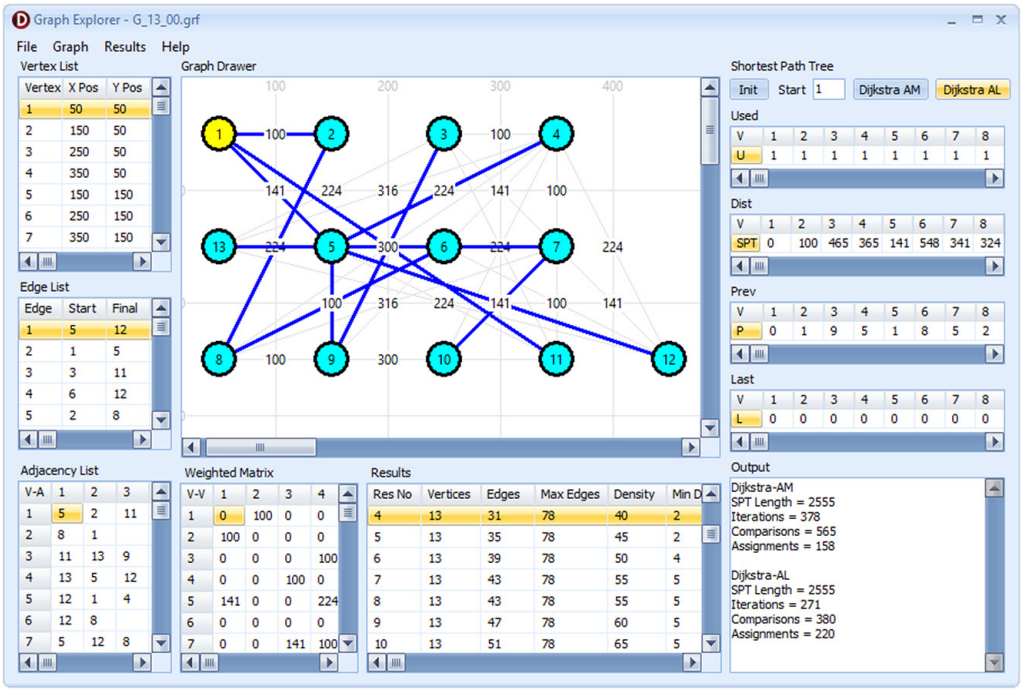

Fig. 1. A working session with the GraphExplorer application

The application allows automatic generation of graphs with varying numbers of vertices and edges, where the number of edges is determined by a predefined value for graph density. This approach enables the creation of both sparse and dense graphs. A key functionality of the application is the precise counting of fundamental operations: iterations, assignments, and comparisons performed during the execution of the algorithm. This is achieved through built-in counters that are updated at each corresponding operation. In addition to measuring operations, the application provides functionality for graph visualization and for exporting results in tabular format for subsequent analysis. The tool was developed with an emphasis on transparency and reproducibility of experiments, allowing configuration of parameters such as graph size, density, and the type of result representation. This ensures a reliable basis for comparing the algorithm's efficiency across different data structures.

The user interface of the application is designed to provide complete visualization of the data and the algorithm's results. It displays the list of vertices and edges of the graph, as well as the AL and AM for the selected representation. In addition, the interface includes a table with the analysis results and a graph designer that interactively visualizes the graph and the changes after the algorithm's execution. An output window is provided to display details of the computational process, along with visualization of the arrays used during execution. This organization enables the user to monitor both the input data and the intermediate and final results in real time. For the purposes of the experimental analysis, a total of 95 graphs with varying numbers of vertices and densities are generated using the developed application. Specifically, for each of the following graph sizes: 100, 200, 300, 400, and 500 vertices, 19 graphs are created with densities ranging from 5% to 95% at intervals of 5%.

4 EXPERIMENTAL RESULTS

The analysis covers the number of fundamental operations: iterations, assignments, and comparisons, depending on the size and density of the analyzed graphs. The summary results are organized in both tabular and graphical form, allowing a comparison between the use of an AM and an AL. The experimental results are obtained using the GraphExplorer application, which is run on a computer with Windows 11 OS and the following hardware configuration: CPU: Intel Core i7-8700 Processor (12M Cache, 4.60 GHz); RAM memory: 32 GB.

Table 1. Summary results by graph density

Density	Avg AM Operations	Avg AL Operations	Difference
5	560,276.8	301,922.4	258,354.4
10	563,355.8	321,415.8	241,940.0
15	566,181.4	340,695.8	225,485.6
20	569,098.4	360,068.4	209,030.0
25	571,764.2	379,189.8	192,574.4
30	574,566.6	398,446.6	176,120.0
35	577,204.6	417,539.0	159,665.6
40	579,919.6	436,709.6	143,210.0
45	582,667.4	455,913.0	126,754.4
50	585,358.2	475,058.2	110,300.0
55	587,959.8	494,114.2	93,845.6
60	590,664.0	513,274.0	77,390.0
65	593,452.2	532,517.8	60,934.4
70	596,066.6	551,586.6	44,480.0
75	598,692.6	570,667.0	28,025.6
80	601,346.0	589,776.0	11,570.0
85	604,083.8	608,969.4	–4,885.6
90	606,723.0	628,063.0	–21,340.0
95	609,386.6	647,181.0	–37,794.4

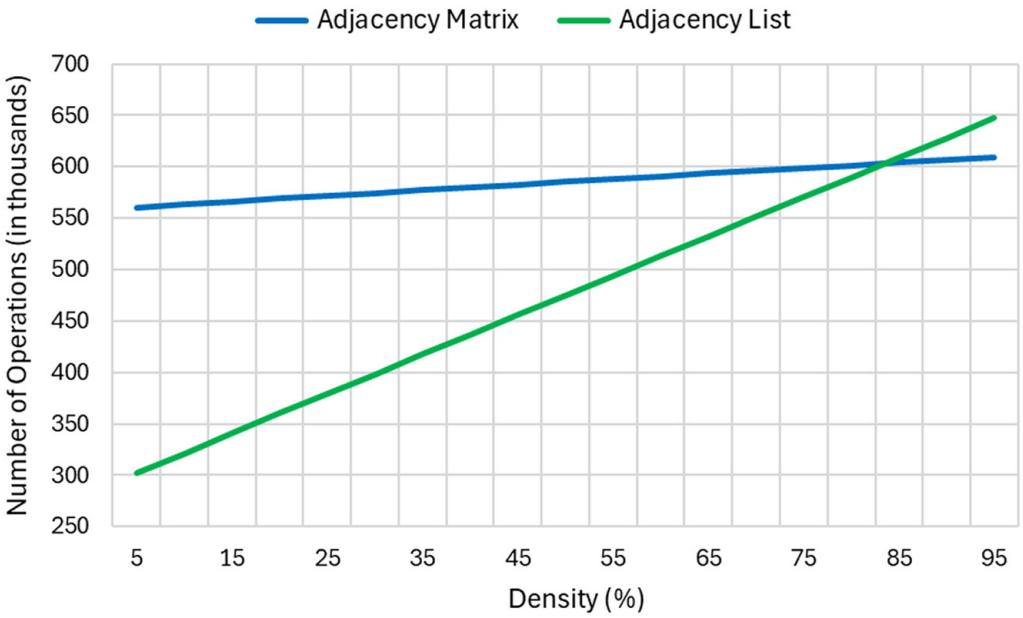

Fig. 2. Impact of density on number of operations

The data in Table 1 and the chart in Figure 2 show a monotonic increase in the average number of operations as the graph density grows, but the rate of increase differs significantly in the two cases. A linear approximation indicates that when using an AL, the increase is approximately 3,833 operations for each additional 1% of density, whereas when using an adjacency matrix, the growth is much slower: about 542 operations for every additional 1% of density. Therefore, at low graph densities, the AL has a significant advantage; for example, at 5% density, it requires approximately 301,922 operations compared to the adjacency matrix, which needs about 560,277 operations for the same density. This is roughly 46.1% fewer operations. This linear decrease in the difference leads to a clear intersection of the two curves, which equalize at approximately 83% density; beyond this point, the AM becomes more efficient in terms of the total number of operations, contradicting the initial hypothesis of the study. For instance, at 95% density, the AM requires 609,387 operations, while the AL requires 647,181, which is about 6.2% fewer. This means that for sparse and moderately dense graphs, such as those in the range between 5% and 80%, the AL is preferable because it minimizes the number of operations, whereas for very dense graphs, for example, above 80%, the AM is the better option.

Table 2. Difference (AM–AL) vs. density for different vertices

Graph Density	Vertices 100	Vertices 200	Vertices 300	Vertices 400	Vertices 500
5	23,262	93,930	211,398	375,860	587,322
10	21,980	87,960	197,940	351,920	549,900
15	20,498	81,990	184,482	327,980	512,478
20	19,010	76,020	171,030	304,040	475,050
25	17,522	70,050	157,578	280,100	437,622
30	16,040	64,080	144,120	256,160	400,200
35	14,558	58,110	130,662	232,220	362,778

(Continued)

Table 2. Difference (AM–AL) vs. density for different vertices *(Continued)*

Graph Density	Vertices 100	Vertices 200	Vertices 300	Vertices 400	Vertices 500
40	13,070	52,140	117,210	208,280	325,350
45	11,582	46,170	103,758	184,340	287,922
50	10,100	40,200	90,300	160,400	250,500
55	8,618	34,230	76,842	136,460	213,078
60	7,130	28,260	63,390	112,520	175,650
65	5,642	22,290	49,938	88,580	138,222
70	4,160	16,320	36,480	64,640	100,800
75	2,678	10,350	23,022	40,700	63,378
80	1,190	4,380	9,570	16,760	25,950
85	–298	–1,590	–3,882	–7,180	–11,478
90	–1,780	–7,560	–17,340	–31,120	–48,900
95	–3,262	–13,530	–30,798	–55,060	–86,322

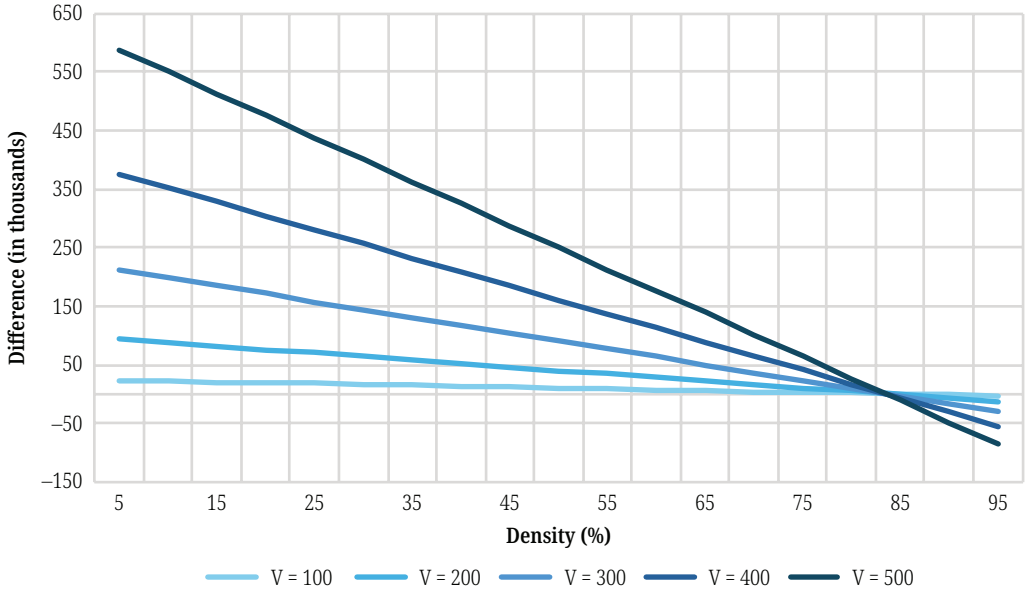

Fig. 3. Difference (AM–AL) vs. density for different vertices

The data in Table 2 and the chart in Figure 3 show that as graph density increases, the difference in the number of operations between the AM and the AL decreases almost linearly for all examined graphs (with 100, 200, 300, 400, and 500 vertices). This means that, with respect to the number of vertices as well, the AL is more efficient for sparse graphs, while the AM becomes more efficient for very dense graphs. Illustratively, at 5% density the differences are positive and grow with the number of vertices: 23,262 (V = 100); 93,930 (V = 200); 211,398 (V = 300); 375,860 (V = 400); and 587,322 operations (V = 500), which demonstrates a significant advantage of using the AL for sparse graphs. At 95% density the differences are negative: –3.262, –13.530, –30.798, –55.060, and –86.322, respectively, indicating that for dense graphs

the AM outperforms the AL in terms of the number of operations. From these results, it can be concluded that, considering the number of vertices as well, for graphs with densities below 83% the AL should be the preferred option, whereas for graph densities above 83% the AM becomes more efficient. The data show that the crossover point stabilizes at approximately 83.5% for graphs with 200 or more vertices.

Table 3. Operations vs. numbers of vertices at key densities

Vertices	AM 5%	AL 5%	AM 5%	AL 5%	AM 5%	AL 5%
100	51,965	28,703	54,947	44,847	56,910	60,172
200	205,601	111,671	214,876	174,676	223,519	237,049
300	459,580	248,182	480,157	389,857	499,860	530,658
400	814,474	438,614	851,174	690,774	885,772	940,832
500	1,269,764	682,442	1,325,637	1,075,137	1,380,872	1,467,194

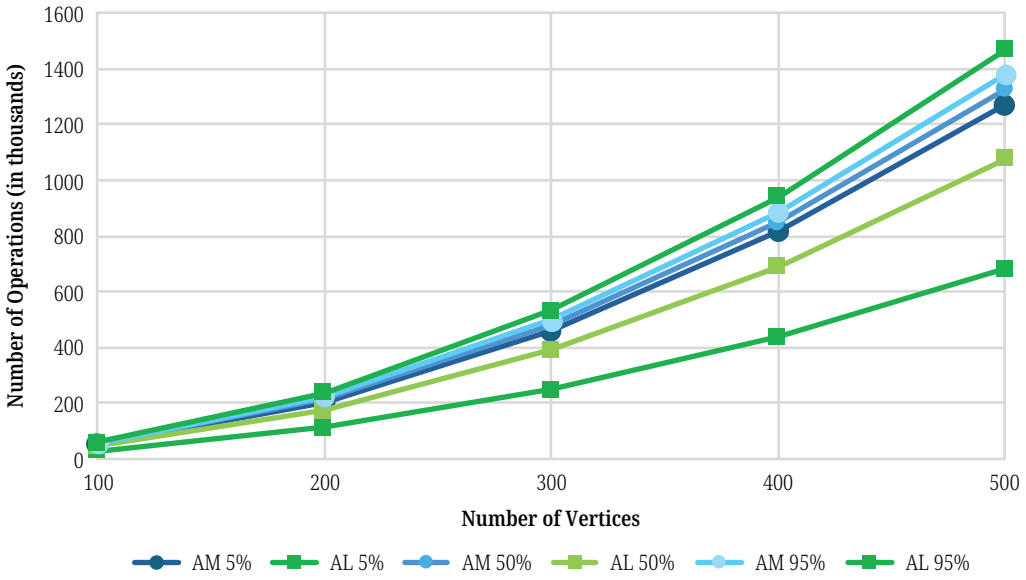

Fig. 4. Operations vs. numbers of vertices at key densities

The data in Table 3 and the chart in Figure 4 indicate that, for the three examined densities (5%, 50%, and 95%), the number of operations for both graph representations increases as the number of vertices grows. At 5% density, the AL is significantly more efficient than the AM: for $|V| = 100$, the difference AM–AL is 23,262 operations, while for $|V| = 500$, it reaches 587,322 operations. At 50% density, AL maintains a consistent advantage: the difference grows from 10,100 at $|V| = 100$ to 250,500 operations at $|V| = 500$, with the relative difference remaining almost constant at approximately 18–19% for all vertex counts. At 95% density, the trend reverses: the AM gradually requires fewer operations than the AL, with the difference being negative and approximately stable in magnitude (around –6.2%). For example, at $|V| = 100$ the difference is –3,262, and at $|V| = 500$ it is –86,322. These patterns indicate that for sparse and moderately dense graphs, the AL dominates regardless of the number of vertices, whereas for very dense graphs, the AM is the more efficient option, with the effect only slightly influenced by the vertex count.

5 SUMMARY AND CONCLUSIONS

This study presents a comparative analysis of two different implementations of Dijkstra's algorithm for finding the shortest paths in connected graphs: one based on an AM and the other based on an AL. The motivation for this study stems from the fundamental trade-off between time complexity and memory complexity when selecting data structures for representing various graphs. Although the algorithmic strategy is consistent in both cases, the choice of data representation plays a critical role in performance, particularly under varying graph sizes and densities.

The experimental methodology involved generating graphs of different sizes in terms of the number of vertices, ranging from 100 to 500 (in increments of 100), and varying densities from 5% to 95% (in increments of 5%). For each graph in the respective group, key performance indicators were measured, such as the number of iterations, comparisons, assignments, and the total number of all these operations executed by each of the two algorithm variants. Additionally, a memory requirement analysis was conducted, highlighting the quadratic memory size when using an AM compared to the linear memory size when using an AL, which is proportional to the number of edges in the graph for directed graphs and twice the number of edges for undirected graphs.

The results reveal clear performance trends. For sparse and moderately dense graphs (with density below 82%), the AL implementation consistently outperforms the AM approach in terms of the total number of operations executed by both algorithm variants. Conversely, for dense graphs (density above 82%), the AM becomes more efficient, performing fewer operations overall. This threshold remains relatively stable across different graph sizes, with only a slight decrease as the number of vertices increases. Regarding memory analysis, it confirms that ALs are more space-efficient for sparse and moderately dense graphs, while adjacency matrices become competitive as graph density increases. For dense graphs, adjacency matrices offer better performance compared to ALs, with comparable memory consumption. Therefore, future research directions will focus on investigating and modeling hybrid approaches that adaptively switch between different data structures depending on the density of the graphs. This study demonstrates that the algorithm's performance is a function of graph density, the type of data structure used, and the kind of operations executed during the computational process.

6 REFERENCES

[1] H. W. Y. Adoni, T. Nahhal, M. Krichen, B. Aghezzaf, and A. El Byed, "A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems," *Distributed and Parallel Databases*, vol. 38, no. 2, pp. 495–530, 2020. https://doi.org/10.1007/s10619-019-07276-9

[2] C. H. Tognon and R. A. Kharabsheh, "Some properties of the formal local cohomology module and application in the theory of graphs," *Applied Mathematics and Information Sciences*, vol. 16, no. 1, pp. 45–49, 2022. https://doi.org/10.18576/amis/160105

[3] S. Balaji, M. S. Obaidat, S. Suthir, M. Rajesh, and K. C. Suresh, "Selection of intermediate routes for secure data communication systems using graph theory application and grey wolf optimization algorithm in MANETs," *IET Networks*, vol. 10, no. 5, pp. 246–252, 2021. https://doi.org/10.1049/ntw2.12026

[4] H. Yue, H. Lin, Y. Jin, H. Zhang, and K. Cai, "Opening knowledge graph model building of artificial intelligence curriculum," *International Journal of Emerging Technologies in Learning*, vol. 17, no. 14, pp. 64–77, 2022. https://doi.org/10.3991/ijet.v17i14.32613

[5] C. M. H. De Figueiredo, "The P versus NP-complete dichotomy of some challenging problems in graph theory," *Discrete Applied Mathematics*, vol. 160, no. 18, pp. 2681–2693, 2012. https://doi.org/10.1016/j.dam.2010.12.014

[6] N. Gruttemeier, P. H. Keßler, C. Komusiewicz, and F. Sommer, "Efficient branch-and-bound algorithms for finding triangle-constrained 2-clubs," *Journal of Combinatorial Optimization*, vol. 48, no. 3, 2024. https://doi.org/10.1007/s10878-024-01204-z

[7] T. Li, Y. Su, Z. Yang, and S. Zhang, "Quantum approximate optimization algorithms for maximum cut on low-girth graphs," *Physical Review Research*, vol. 7, no. 3, 2025. https://doi.org/10.1103/jypq-v1fn

[8] A. Tilantera, A. Korhonen, O. Seppälä, and T. Taivainen, "Investigating students' misconceptions of Dijkstra's algorithm: Exploration of algorithm simulation traces," in *Annual Conference on Innovation and Technology in Computer Science Education*, *ITiCSE*, vol. 1, 2025, pp. 674–680. https://doi.org/10.1145/3724363.3729081

[9] O. Asher, S. Dolev, and L.-O. Raviv, "SSPT: Shallowest shortest path tree (Short Paper)," in *Lecture Notes in Computer Science*, vol. 16244, 2026, pp. 358–371. https://doi.org/10.1007/978-3-032-10759-6_25

[10] B. Riabenko, O. Martynova, Y. Boyarinova, and A. Krainosvit, "Finding optimal routes in internal routing networks based on a modified Dijkstra's algorithm," *International Journal of Modern Education and Computer Science*, vol. 17, no. 4, pp. 37–47, 2025. https://doi.org/10.5815/ijcnis.2025.04.03

[11] K. Obelovska, O. Tkachuk, and Y. Snaichuk, "Minimizing the number of distrustful nodes on the path of IP packet transmission," *Computation*, vol. 12, no. 5, p. 91, 2024. https://doi.org/10.3390/computation12050091

[12] M. Zhang, M. Sutcliffe, P. I. Nicholson, and Q. Yang, "Efficient autonomous path planning for ultrasonic non-destructive testing: A graph theory and K-dimensional tree optimisation approach," *Machines*, vol. 11, no. 12, p. 1059, 2023. https://doi.org/10.3390/machines11121059

[13] Z. Grujic and B. Grujic, "Optimal routing in urban road networks: A graph-based approach using Dijkstra's algorithm," *Applied Sciences (Switzerland)*, vol. 15, no. 8, p. 4162, 2025. https://doi.org/10.3390/app15084162

[14] D. Ouyang, D. Wen, L. Qin, L. Chang, X. Lin, and Y. Zhang, "When hierarchy meets 2-hop-labeling: Efficient shortest distance and path queries on road networks," *VLDB Journal*, vol. 32, no. 6, pp. 1263–1287, 2023. https://doi.org/10.1007/s00778-023-00789-x

[15] L. Jiang, Y. Lai, Q. Chen, W. Zeng, F. Yang, and F. Fan, "Shortest path distance prediction based on CatBoost," in *Lecture Notes in Computer Science*, vol. 12999, 2021, pp. 133–143. https://doi.org/10.1007/978-3-030-87571-8_12

[16] M. Joswig and B. Schröter, "Parametric shortest-path algorithms via tropical geometry," *Mathematics of Operations Research*, vol. 47, no. 3, pp. 2065–2081, 2022. https://doi.org/10.1287/moor.2021.1199

[17] T. Iwata, K. Kitaura, R. Matsuo, and H. Ohsaki, "A solution for finding quasi-shortest path with graph coarsening," in *International Conference on Information Networking*, 2022, pp. 215–219. https://doi.org/10.1109/ICOIN53446.2022.9687193

[18] Sunita and D. Garg, "A retroactive approach for dynamic shortest path problem," *National Academy Science Letters*, vol. 42, no. 1, pp. 25–32, 2019. https://doi.org/10.1007/s40009-018-0674-6

[19] H. Arslan and M. Manguoğlu, "A hybrid single-source shortest path algorithm," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 27, no. 4, pp. 2636–2647, 2019. https://doi.org/10.3906/elk-1901-23

[20] X. Gao, Y. Xianzang, X. You, Y. Dang, G. Chen, and X. Wang, "Reachability for airline networks: Fast algorithm for shortest path problem with time windows," *Theoretical Computer Science*, vol. 749, pp. 66–79, 2018. https://doi.org/10.1016/j.tcs.2018.01.016

[21] M. A. Qureshi, M. F. Hassan, S. Safdar, R. Akbar, A. Ali, and M. K. Ehsan, "Dijkstra's algorithm–A case study to understand how algorithms are improved," *VFAST Transactions on Software Engineering*, vol. 9, no. 3, pp. 48–56, 2021. https://doi.org/10.21015/vtse.v9i3.706

[22] S. Atalig, A. Hickerson, A. Srivastav, T. Zheng, and M. Chrobak, "Lower bounds for adaptive relaxation-based algorithms for single-source shortest paths," in *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 322, 2024. https://doi.org/10.4230/LIPIcs.ISAAC.2024.8

[23] G. Brodal, "Priority queues with decreasing keys," *Theoretical Computer Science*, vol. 1000, p. 114563, 2024. https://doi.org/10.1016/j.tcs.2024.114563

[24] H. T. T. Binh, T. B. Thang, N. D. Thai, and P. D. Thanh, "A bi-level encoding scheme for the clustered shortest-path tree problem in multifactorial optimization," *Engineering Applications of Artificial Intelligence*, vol. 100, p. 104187, 2021. https://doi.org/10.1016/j.engappai.2021.104187

[25] V. Kralev and R. Kraleva, "Combining genetic algorithm with local search method in solving optimization problems," *Electronics (Switzerland)*, vol. 13, no. 20, p. 4126, 2024. https://doi.org/10.3390/electronics13204126

[26] H. T. T. Binh, P. D. Thanh, and T. B. Thang, "New approach to solving the clustered shortest-path tree problem based on reducing the search space of evolutionary algorithm," *Knowledge-Based Systems*, vol. 180, pp. 12–25, 2019. https://doi.org/10.1016/j.knosys.2019.05.015

[27] K. Ibrahim Mohammad Ata, A. Che Soh, A. J. Ishak, and H. Jaafar, "Guidance system based on Dijkstra-ant colony algorithm with binary search tree for indoor parking system," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 24, no. 2, pp. 1173–1182, 2021. https://doi.org/10.11591/ijeecs.v24.i2.pp1173-1182

[28] V. Kralev, R. Kraleva, and V. Ankov, "An interactive application for learning and analyzing different graph vertex cover algorithms," *International Journal of Engineering Pedagogy*, vol. 13, no. 1, pp. 4–19, 2023. https://doi.org/10.3991/ijep.v13i1.35661

[29] Y. Li, "Robot path selection based on path planning algorithms in traffic situations," *AIP Conference Proceedings*, vol. 3144, no. 1, 2024. https://doi.org/10.1063/5.0214289

[30] W. Liu *et al.*, "Towards geodesic ridge curve for region-wise linear representation of geodesic distance field," *Computer Aided Geometric Design*, vol. 111, p. 102291, 2024. https://doi.org/10.1016/j.cagd.2024.102291

[31] H. Yang, "Analysis and study on path planning algorithms in the further mobile action," *Journal of Physics: Conference Series*, vol. 2824, no. 1, p. 012006, 2024. https://doi.org/10.1088/1742-6596/2824/1/012006

[32] Wamiliana, R. P. Sari, A. Reformasari, J. Suparman, and A. Junaidi, "Solving the shortest total path length spanning tree problem using the modified sollin and modified Dijkstra algorithms," *Science and Technology Indonesia*, vol. 8, no. 4, pp. 684–690, 2023. https://doi.org/10.26554/sti.2023.8.4.684-690

7 AUTHORS

Velin Kralev is an Associate Professor of Computer Science at the South-West University "Neofit Rilski" in Blagoevgrad, Bulgaria. He defended his Ph.D. Thesis in 2010. His research interests include graph algorithms, optimization problems in graphs, and component-oriented software engineering (E-mail: velin_kralev@swu.bg).

Radoslava Kraleva is an Associate Professor of Computer Science at the South-West University "Neofit Rilski" in Blagoevgrad, Bulgaria. She defended her Ph.D.

Thesis in 2014. Her research interests include speech recognition, mobile app development, and computer graphics.

Aleksandra Popova is a student in Information Systems and Technologies at the Faculty of Natural Sciences and Mathematics at South-West University "Neofit Rilski" in Blagoevgrad, Bulgaria. Her research interests focus on graph algorithms, database systems, web design, and graphic design, as well as machine learning.