JESICA

JESICA: Journal of Enhanced Studies in Informatics and Computer Applications
Journal homepage: jesica.itsk-soepraoen.ac.id
ISSN: e-ISSN: 3046-6938 p-ISSN: 3046-6997
Vol. 2, No. 2, July 2025, PP. 43-51

Comparison of Breadth-First Search (BFS) and Depth-First Search (DFS) Algorithms for Shortest Search in Campus Maze

Ardhan Aghsal Dwi Putra*[1], Andhika Nur Maulana[2], Shasha Billa Febrianti[3], Nabila Camelia[4], Sabastian Kaka Hutagalung[5], Ahsanun Naseh Khudori[6]
1, 2, 3, 4, 5, 6 Institut Teknologi Sains dan Kesehatan RS.DR. Soepraoen Kesdam V/BRW, Indonesia

***Corresponding author**
Email address:
ardhanag9@gmail.com

Keywords:
Campus navigation, shortest path search, BFS, DFS, graphs, mazes

Abstract
Finding the shortest path in a complex campus environment is a challenge, especially for new students who are not familiar with the layout of buildings and available paths. Efficient path finding can help improve mobility on campus, especially in areas with many branching paths and possible dead ends. In this study, an analysis of the shortest path search was conducted by comparing the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms in a campus environment represented as a maze-shaped graph. The research methods include literature study, simulation design, data collection, algorithm implementation, and performance evaluation based on execution speed, memory usage, and processor efficiency. Data were obtained from field surveys and secondary studies on campus layout. Simulations were conducted by implementing BFS and DFS in a graph model to measure the effectiveness of both algorithms. The results show that DFS has advantages in execution speed and lower memory usage, while BFS is more consistent in finding optimal solutions. DFS is more suitable for scenarios with fast search time requirements, while BFS is more effective in ensuring the shortest path in an environment with a complex graph structure. The conclusion of this study emphasizes that the selection of algorithms must be adjusted to the specific needs of navigation applications.

1. Introduction
Navigating a campus environment is often a challenge for students, especially new students unfamiliar with the building layout and available routes. This challenge is compounded on campuses with large areas and numerous buildings connected by small, labyrinthine pathways. In this context, finding the quickest or shortest route from one location to another becomes a crucial need.

In general, the concept of shortest path finding can be applied in various graph-based systems, both in theoretical contexts and in real-world applications. In graph theory, vertices and edges are used to represent points and relationships between locations, which are then analyzed to determine the most efficient route [1]. As a visual representation of graph structure, mazes are often used as visual simulations that reflect real-world challenges, such as branching paths, many turns, and the possibility of paths that do not lead to the fastest route or end in dead ends that do not reach the final destination [2]. Both of these representations provide an interesting framework for testing and evaluating various shortest path finding algorithms.

Mazes used in simulations often reflect characteristics of real-world problems, such as many turns and the possibility of dead ends, making them ideal for testing navigation algorithms [3]. In practical contexts, efficient pathfinding has many applications, such as robot navigation, campus route guidance systems, and the development of maze-based games [4]. However, determining the optimal algorithm for finding the shortest path with high efficiency in terms of speed, memory usage, and processor efficiency remains a relevant challenge. This challenge is interesting to discuss because the performance of the search algorithm directly impacts the efficiency of the system, especially in applications that require fast response and resource constraints, such as new real-time navigation [5].

Various algorithms have been designed to address this problem, including BFS and DFS. The Breadth-First Search (BFS) algorithm works by exploring vertices in a level-by-level fashion, making it particularly suitable for finding shortest paths in unweighted graphs [6]. The process begins by visiting one vertex, then continues by visiting all its neighbors before moving on to the next vertex [7]. In a study conducted by Elkari et al., BFS and DFS were applied to maze path mapping, and both were able to find the target, but showed significant differences in terms of complexity,

completeness, optimality, and time efficiency of finding the best solution [8]. In its implementation, BFS uses a queue data structure to store visited nodes, which are then used as references to explore neighboring nodes [9].

In contrast, DFS is a graph traversal algorithm that works by exploring paths in depth until reaching the final node before returning to the previous node to check for other unvisited paths [10]. The Depth-First Search (DFS) algorithm, when implemented in a programming language, uses a stack data structure . This algorithm works based on the Last In First Out (LIFO) principle, where the last element added to the stack will be the first element taken out [11].

Previous research has demonstrated the advantages of each algorithm in different contexts. One study [12] found that BFS was effective in helping players find their way through mazes in game-based applications. Other research has shown that BFS and DFS yield different results in solving pathfinding problems, such as in Sudoku. BFS tends to be more systematic but slower, while DFS is faster but can miss optimal solutions. This study compared the results of both algorithms with human solutions and showed that DFS has faster execution time, while BFS offers a more comprehensive search structure within the context of a given number of clues in Sudoku [13]. Čarapina et al. [14] also compared the two algorithms in the context of a large-scale maze and found that BFS is more efficient in finding shortest paths, while DFS tends to be less optimal due to systematic undirected path exploration. However, no comparative study has directly evaluated the performance of BFS and DFS in terms of speed, memory efficiency, and processor utilization in a real-world environment such as a campus maze.

This study offers a solution by comparing the performance of the BFS and DFS algorithms through a maze simulation built based on a campus route plan. Campus paths are modified and arranged to resemble a maze structure to represent the complexity of a real-world environment. Evaluation is conducted by reviewing three main parameters: execution speed, memory usage, and processor efficiency. Unlike previous studies that generally use artificial environments or logic games, this study presents a more realistic simulation scenario that utilizes the spatial structure of the campus. Furthermore, the empirical analysis will demonstrate the advantages and limitations of each algorithm in more detail than previously reported approaches.

2. Research methods

This study applies a quantitative approach to evaluate the performance of the BFS and DFS algorithms in finding the shortest path. The analysis is conducted through a graph-based simulation depicting the campus maze layout, using a directed and unweighted graph according to the characteristics described in the previous reference. This study consists of five stages that must be carried out: literature study, simulation design, data collection, algorithm implementation, and evaluation of results, as visualized in Figure 1.

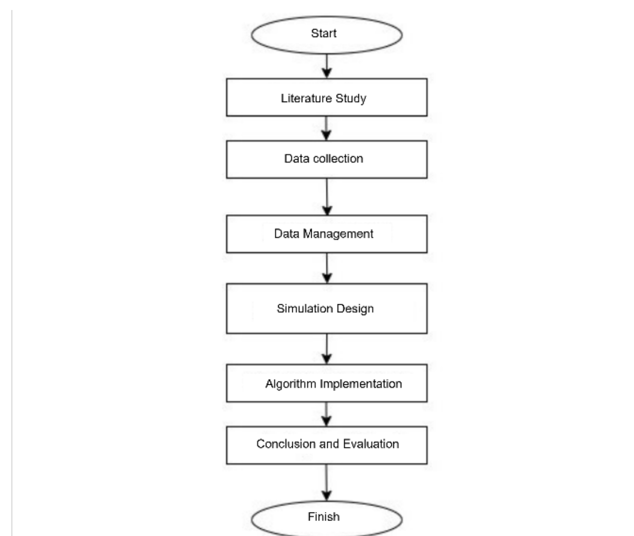

Figure 1. Research stages

2.1. Literature Study

A literature review was conducted to gather information from various relevant sources as a basis for supporting research on algorithm comparison. The referenced scientific articles discuss graph exploration methods, search algorithm efficiency, and the implementation of BFS and DFS algorithms in various contexts, such as mazes, campus

© 2025 Putra et al. Published by ITSK Soepraoen
This is an open-access article under the CC BY SA license. (https://creativecommons.org/licenses/by-sa/4.0/)

environments, and robotics applications. Several studies have shown that algorithm selection is strongly influenced by graph structure, navigation objectives, and device limitations.

For example, Hsaine and Sekkat [15] compared the BFS, DFS, and A* algorithms in maze navigation using Python, and found that DFS was superior in speed, while BFS was more consistent in generating shortest paths. Güllü and Kuşçu, [16] in the context of robotic exploration, confirmed that DFS was faster in initial path identification, while BFS provided a more stable path representation for complex navigation. Mochammad Darip et al. [3] compared the BFS and DFS algorithms in the context of a historical-cultural tourism route in Banten Province. The results showed that BFS tended to produce more efficient paths in terms of distance traveled, while DFS was able to optimize the number of locations visited through in-depth path exploration. The graph model used represented a real tourist location network. This demonstrates how each algorithm responds to complexity in geographic simulations. This approach emphasizes the importance of considering the application context and user goals, whether distance efficiency or exploration level is prioritized, in selecting the appropriate pathfinding algorithm.

Sugianti et al. [17] compared BFS, A*, and Greedy BFS in a 2D virtual environment, and found that BFS provides stable performance in determining the shortest path, although it is less time-efficient. Elkari et al. [15] compared BFS and DFS in path mapping on a complex grid map. The results showed that BFS excels in completeness and robustness to map changes, while DFS is faster but less optimal in determining the shortest path. This study emphasized that the effectiveness of the algorithm depends on the application context, as well as the priority between accuracy and search speed. Mustaqim et al. [7] optimized the implementation of the BFS and DFS algorithms in the development of a web crawler on the Kumparan news site. This study showed that BFS is able to index more files thoroughly, but requires a longer execution time. In contrast, DFS produces faster search times, even though the number of indexed files is smaller. For example, at search depth level 4, BFS successfully indexed 949 files in 886.94 seconds, while DFS indexed 470 files in only 233.02 seconds. These findings have implications for algorithm selection based on the need for a trade-off between depth of coverage and search speed in indexing web content.

Mat Diah et al. [13] conducted a comparative analysis between the Breadth-First Search (BFS), Depth-First Search (DFS), and human approaches in solving Sudoku puzzles. The results showed that DFS has a faster execution time than BFS, especially on puzzles with low to medium difficulty levels. However, BFS produces more structured solutions and is closer to human solving patterns. This study emphasizes that algorithm selection needs to be tailored to the context of the complexity and accuracy requirements of the solution.

All of this literature reinforces the relevance of this research, particularly in the context of a campus environment simulated as a maze-like graph. This research addresses the gap in the direct application of BFS and DFS algorithms to an actual campus map, as well as its evaluation based on three key indicators: execution speed, memory usage, and processor efficiency, which have not been comprehensively addressed in previous studies.

2.2. Data collection

Data collection in this study was conducted through two types of sources: primary and secondary data. Primary data was obtained directly through field research, which included surveys and observations around the Institut Teknologi, Sains, dan Kesehatan RS dr. Soepraoen Malang. This activity aimed to gather detailed and up-to-date information regarding the floor plan, connecting routes between buildings, and the physical condition of the campus environment.

In secondary data collection, specific information was extracted from three primary sources. The campus map retrieved spatial information in the form of building locations and names to serve as nodes, as well as visualizations of connecting paths that serve as edges in graph modeling. Meanwhile, the data sought from scientific journal articles and previous research focused on theoretical foundations, relevant performance measurement methodologies, and the results of previous comparative studies between the BFS and DFS algorithms, which can be used as references and comparisons in the analysis.

2.3. Data Management

Data obtained through field surveys was analyzed to develop an accurate campus plan. The attributes analyzed included building locations, types of facilities available, and the distribution of space within the campus. This analysis aimed to produce a clear map of the campus layout, which could serve as the basis for developing a more efficient campus navigation system.

2.4. Simulation Design

At this stage, a campus layout mapping algorithm was designed so that the Breadth-First Search (BFS) and Depth-First Search (DFS) methods could be implemented effectively. This study used data from 20 room location points in

the campus environment of the Institut Teknologi, Sains, dan Kesehatan RS dr. Soepraoen Malang as listed in Table 1. Each point was labeled with the letters A to T to facilitate the identification process. Some of the rooms recorded included the Rectorate Building (A), Parking (B), Field (C), UPTI (D), RMIK (E), to the Mushola (T), as well as various academic rooms and laboratories such as Informatics, Pharmacy, Nursing, and Physiotherapy Laboratories.

Table 1. List of rooms on campus

Room initialization	Room name
A	Rectorate Building
B	Parking
C	Field
D	UPTI
E	RMIK
F	Dental Lab
G	Canteen
H	Informatics Lab 1
I	Informatics Lab 2
J	Research Lab
K	SmartClass
L	Midwifery
M	Nursing
N	D3 Nursing
O	Physiotherapy
P	Pharmacy
Q	D3 Pharmacy
R	Anesthesia
S	Volleyball Court
T	Mushola

Next, the 20 locations listed in Table 1 were modeled into a graph structure to visualize the connectivity relationships between rooms on campus. As shown in Figure 2, each room is represented as a node, while the connecting paths between rooms are represented as weighted edges. These weights indicate the estimated physical distance between rooms.

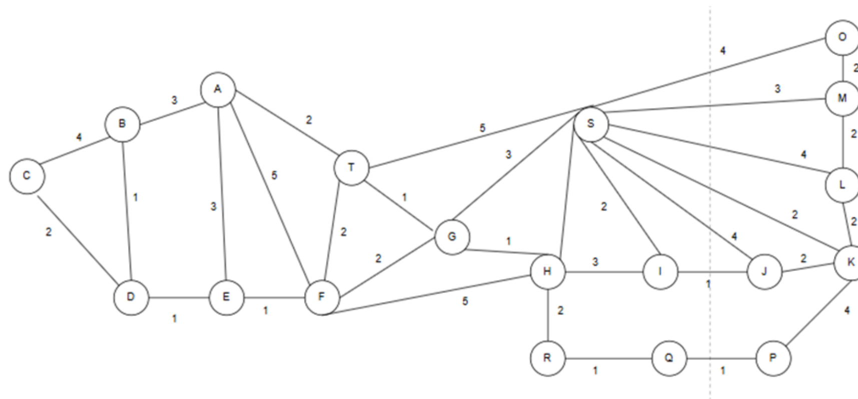

Figure 1. Visual representation graph of the room plan

This graph is used as the basis for implementing pathfinding algorithms such as BFS and DFS to analyze connectivity patterns between spaces, optimize travel routes, and efficiently use space on campus. This approach is expected to make navigation within the campus environment more systematic and structured.

© 2025 Putra et al. Published by ITSK Soepraoen
This is an open-access article under the CC BY SA license. (https://creativecommons.org/licenses/by-sa/4.0/)

2.5. Algorithm Implementation

In the implementation stage, a graph is represented as a collection of nodes and edges with certain weights arranged using data structures such as matrices or adjacency lists, where the weights on the edges reflect the distances between nodes to support the calculation of the shortest route. Breadth-First Search Algorithm (BFS) utilizes a queue data structure to perform graph exploration incrementally at each level, by starting at the initial vertex, adding unvisited neighboring vertices to the queue, and processing the closest vertices first. In contrast , the DFS algorithm uses a stack data structure or a recursive approach to perform in-depth exploration, by sequentially exploring neighboring vertices and backtracking when all neighbors have been visited.

2.5.1. Breadth-First Search (BFS) pseudocode

```
BFS algorithm(graph, start, goal)
start_time ← current_time()
memory_before ← used_memory()
cpu_before ← cpu_usage()

Q ← empty queue
visited ← empty set
Enqueue(Q, (start, [start]))
Add start to visited

while Q is not empty do
(current, path) ← Dequeue(Q)

if current = goal then
end_time ← current_time()
memory_after ← used_memory()
cpu_after ← cpu_usage()

execution_time ← end_time - start_time
memory_used ← memory_after - memory_before
cpu_used ← cpu_after - cpu_before

Output execution_time, memory_used, cpu_used
return path
end if

for each neighbor in Neighbors(graph, current) do
if neighbor not visited then
Add neighbors to visit
Enqueue(Q, (neighbor, path + [neighbor]))
end if
end for
end while

return "No path found"
```

The Breadth-First Search (BFS) algorithm begins by recording three main evaluation parameters: execution time, memory usage, and estimated processor usage before the search begins. For node exploration, the algorithm utilizes a FIFO (First In First Out) queue data structure and a visited set to ensure each node is visited only once. The starting node is inserted into the queue along with the path traveled so far. As long as the queue is not empty, the algorithm will select an entry from the queue to examine. If that node is the goal node, the search is stopped. The system then records the end time, post-process memory usage, and CPU usage, then calculates the difference with the initial data to obtain the values of the three evaluation parameters: execution speed, memory consumption, and processor efficiency.

The results of these measurements are displayed as performance information for the algorithm, and the path to the goal node is returned as output. If the current node is not the goal, all unvisited neighboring nodes are marked and

queued with the updated path. This process continues until the goal node is found or all reachable nodes have been explored. If no path exists between the starting and goal nodes, the algorithm returns a path not found error.

2.5.2. Pseudocode Depth First Search (DFS)

```
DFS algorithm(graph, start, goal):
start_time ← current_time()
memory_before ← used_memory()
cpu_before ← cpu_usage()

create an empty stack S
create an empty set visited
push (start, [start]) into S
mark start as visited

while S is not empty:
(current, path) ← S.pop()

if current = goal then
end_time ← current_time()
memory_after ← used_memory()
cpu_after ← cpu_usage()

execution_time ← end_time - start_time
memory_used ← memory_after - memory_before
cpu_used ← cpu_after - cpu_before

print execution_time, memory_used, cpu_used
return path

for each neighbor in graph.neighbors(current):
if neighbor not visited then
mark neighbor as visited
push (neighbor, path + neighbor) into S

return "No path found"
```

The Depth-First Search (DFS) algorithm begins by recording the initial time, memory usage, and estimated processor usage to measure the algorithm's performance in the path finding process. DFS uses a deep path exploration approach, utilizing a stack data structure with a LIFO (Last In First Out) behavior and a visited set to ensure that each node is not visited more than once. The starting node is placed in a stack along with an initial path that includes only that node. The algorithm then iterates as long as the stack contains data. At each step, the top element of the stack is checked. If the current node is the goal node, the search process is stopped. The system records the ending time, the memory used after processing, and the estimated CPU, then calculates the difference to obtain values for the three evaluation parameters: execution time, memory consumption, and processor efficiency. The path from the starting node to the goal is returned as the result.

If the retrieved node is not the destination node, all of its unvisited neighbors are added to the stack along with the updated path and marked as visited. This approach allows the algorithm to traverse the path as deeply as possible before backtracking to try alternative paths. The process continues until the destination node is found or all reachable nodes have been explored. If no valid path is found, the algorithm returns a statement stating that there is no available path to the destination node.

2.6. Conclusion and Evaluation

The evaluation phase was conducted by comparing the execution results of the two algorithms on several different graph scenarios. Each algorithm was tested on a non-uniformly weighted graph scenario to assess its performance under various conditions. In each scenario, execution speed, memory usage, and processor efficiency were

© 2025 Putra et al. Published by ITSK Soepraoen
This is an open-access article under the CC BY SA license. (https://creativecommons.org/licenses/by-sa/4.0/)

quantitatively measured using a profiler. In this study, we used Java (VisualVM) as the profiler. The following is the method used to calculate each parameter:

1 . Execution Speed (T)

Execution speed measures the time it takes the algorithm to complete the route search. The value (T) is measured in milliseconds (ms). Execution speed is calculated by:

$$T = t_{end} - t_{start} \quad (1)$$

t_{start} is the start time of the algorithm, and t_{end} is the finish time. The value (T) provides direct information about the time performance of the algorithm.

2 . Memory usage (M)

Memory usage measures the amount of memory used during the search process. This value depends on the number of nodes explored and the memory required to store the algorithm's data structures, such as queues in BFS or heaps in DFS. The formula for calculating memory usage is:

$$M = N \times S \quad (2)$$

N is the number of nodes explored during the search process, S is the amount of memory required to store data for each node (expressed in kilobytes). Memory usage becomes important in the case of graphs or mazes with a large number of nodes, where BFS tends to require more memory than DFS due to its exploratory nature.

3 . Processor Efficiency

Processor efficiency (E) describes the extent to which an algorithm utilizes computing time to complete a task. This parameter is calculated as the inverse of the execution time, which is expressed as:

$$E = \frac{1}{T} \quad (3)$$

The smaller the T value and the greater the E value indicates better processor efficiency. This value is important for assessing an algorithm's ability to complete searches with minimal time usage.

Using these three parameters, an evaluation was conducted to compare the strengths and weaknesses of each algorithm. Execution speed measures the time required, memory usage determines data storage efficiency, and processor efficiency evaluates the utilization of computing resources. This approach provides a comprehensive overview of the algorithm's performance on the problem of finding the shortest route in a campus maze.

3. Results and Discussion

Based on the tests conducted to compare the BFS and DFS algorithms in finding the shortest route in the campus maze. carried out based on three main parameters: execution speed (T), memory usage (M), and processor efficiency ($\bar{E}$). Tests were conducted five times for each algorithm to obtain representative data without causing redundancy. This number was chosen because the simulation scenario used is fixed and controlled, so five trials are sufficient to show the performance pattern and consistency of the results. The following evaluation results are shown in the table of tests that have been carried out:

Table 2. Test results

Testing	Algorithm	Execution speed (T) (ms)	Memory usage (M)	Processor fficiency ($\bar{E}$)
1	BFS	15	80	0.0667
	DFS	10	50	0.1
2	BFS	17	85	0.0588
	DFS	11	55	0.0909
3	BFS	16	82	0.0625
	DFS	12	52	0.0833
4	BFS	14	78	0.0714
	DFS	11	53	0.0909
5	BFS	15	80	0.0667
	DFS	12	51	0.0833

The results of the BFS algorithm implementation show that this method is able to explore graphs with a consistent and systematic exploration pattern. In five tests, the BFS execution speed ranged from 14 ms to 17 ms, with memory usage between 78 and 85 memory units. This indicates that BFS tends to require quite a large memory space because

it stores all nodes at the same level before proceeding to the next level. The processor efficiency ($\bar{E}$) obtained ranged from 0.0588 to 0.0714, which reflects moderate computational requirements. BFS is suitable for use in situations where the guarantee of finding the shortest path is highly desirable and memory resources are sufficiently available.

Meanwhile, the implementation results of the DFS algorithm show in-depth exploration characteristics that result in faster execution times in several experiments. Execution times are in the range of 10 to 12 ms, with relatively low memory usage, namely between 50 and 55 memory units. Processor efficiency in the DFS algorithm is in the range of 0.0833 to 0.1, indicating a high processing rate but commensurate with more efficient memory usage. DFS tends to be suitable for applications that prioritize initial search speed and memory constraints, although the path search results are not always optimal.

Looking at the overall results in Table 2, DFS execution speed is higher because the deep path exploration approach reduces the time to find a solution, especially in graphs or mazes where the solution path is found faster early in the search. In contrast, BFS explores all paths at a certain level before proceeding to the next level, thus consuming more time. Furthermore, DFS has a superior memory usage because it only stores the currently active vertices on the path being explored. BFS requires more memory to store the vertex queue at each exploration level, which causes memory usage to increase as the graph or maze size increases. Finally, DFS's processor efficiency is better because it has a shorter execution time. However, BFS provides more consistent results, especially if the graph structure has a uniform distribution of solutions or many paths with the same weight.

4. Conclusion

Based on the evaluation results, the BFS and DFS algorithms have different performance characteristics in finding the shortest route in a campus maze. The DFS algorithm proved superior in terms of execution speed, with a shorter average time compared to BFS. This is due to its deep path exploration strategy that allows solutions to be found earlier, especially in graphs with solutions at a certain depth. In addition, DFS is also more efficient in memory usage because it only stores active nodes on the exploration path. In contrast, BFS requires more memory to store all nodes at each exploration level, although it provides more consistent results, especially in graphs with uniform weights. In terms of processor efficiency, DFS shows an advantage due to its shorter execution time, making it more optimal in systems with limited resources. Although BFS has the advantage of guaranteeing an optimal solution overall, DFS is more recommended in cases that require time and memory efficiency. Therefore, the selection of the appropriate algorithm is highly dependent on the specific needs and characteristics of the problem at hand, so that the chosen method can provide optimal results according to the application context. Further research is recommended to explore the use of other algorithms, such as A* or heuristic-based algorithms, to improve search performance in more complex scenarios.

Reference

[1] A. Muhardono, “Penerapan Algoritma Breadth First Search dan Depth First Search pada Game Angka,” *Jurnal Minfo Polgan* , vol. 12, no. 1, pp. 171–182, Mar 2023, doi: 10.33395/jmp.v12i1.12340.
[2] Muhammad Fatihul Irhab, “Optimalisasi Pencarian Jalur dalam Labirin Menggunakan Algoritma A*,” *Optimalisasi Pencarian Jalur dalam Labirin Menggunakan Algoritma A**, Jun 2024.
[3] Mochammad Darip, Sigit Auliana, AK Anam, Parimin, and Anugerah Agung, “Comparison of BFS and DFS Algorithm for Routes to Historical-Cultural Tourism Locations in Banten Province,” *Journal of Advances in Information and Industrial Technology*, vol. 6, no. 2, pp. 113–122, Oct 2024, doi: 10.52435/jaiit.v6i2.560.
[4] JT Santoso, M. Kom., “KECERDASAN BUATAN,” Semarang, Aug 2023.
[5] S. Khan, M. Sinku, and S. Mishra, “Hybridizing BFS and DFS for Enhanced Problem-Solving Efficiency in AI Applications,” 2025. [Online]. Available at: https://jsiar.com
[6] T. J. Rintala, A. Ghosh, and V. Fortino, “Network approaches for modeling the effects of drugs and diseases,” *Brief Bioinform* , vol. 23, no. 4, Jul 2022, doi: 10.1093/bib/bbac229.
[7] A. Mustaqim, DB Dinova, MS Fadhilah, R. Seivany, B. Prasetiyo, and MA Muslim, "Optimizing the Implementation of the BFS and DFS algorithms using the web crawler method on the kumparan site," *Journal of Soft Computing Exploration,* vol. 5, no. 2, p. 200–206, Jul 2024, doi: 10.52465/joscex.v5i2.309.
[8] R. Scheffler, “On the recognition of search trees generated by BFS and DFS,” *Theoretical Computer Science*, vol. 936, p. 116–128, Nov 2022, doi: 10.1016/j.tcs.2022.09.018.
[9] W. Wirgiawan, A. Amirul, A. Cirua, M. Akbar, and S. Cokrowibowo, “Perbandingan Kinerja Algoritma Greedy-Backtracking, BFS, DFS, dan Genetika pada Masalah Penukaran Koin,” Konferensi Nasional Ilmu Komputer (KONIK) 2021.
[10] MF Bernov, AD Rahajoe, and BM Mulyo, "Route Optimization of Waste Carrier Truck using Breadth First Search (BFS) Algorithm," *JEECS (Journal of Electrical Engineering and Computer Sciences)*, vol. 7, no. 2, p. 1293–1304, Jan 2023, doi: 10.54732/jeecs.v7i2.23.
[11] M. Qulub and I. Shanti Bhuana, “Implementasi Algoritma Depth-First Search dan Breadth-First Search pada Dokumen Akreditasi,” 2024. [Online]. Available at: http://jurnal.goretanpena.com/index.php/JSSR
[12] Y. Adiguna, D. Swanjaya, and M. Kom, “Perbandingan Algoritma Depth First Search, Backtracking dan A Star untuk Mencari Jalan Keluar Sebuah Labirin,” Kediri, Aug 2022.
[13] NM Diah, S. Riza, S. Ahmad, N. Musa, and S. Hashim, “Sudoku solutions: a comparative analysis of breadth-first search, depth-first search, and human approaches,” *Journal of Education and Learning*, vol. 19, no. 1, p. 561–569, Feb 2025, doi: 10.11591/edulearn.v19i1.21214.
[14] M. Čarapina, O. Staničić, I. Dodig, and D. Cafuta, “A Comparative Study of Maze Generation Algorithms in a Game-Based Mobile Learning Application for Learning Basic Programming Concepts,” *Algorithms,* vol. 17, no. 9, Sep 2024, doi: 10.3390/a17090404.

© 2025 Putra et al. Published by ITSK Soepraoen
This is an open-access article under the CC BY SA license. (https://creativecommons.org/licenses/by-sa/4.0/)

[15] B. Elkari *et al.*, “Exploring Maze Navigation: A Comparative Study of DFS, BFS, and A* Search Algorithms,” *Statistics, Optimization and Information Computing,* vol. 12, no. 3, p. 761–781, 2024, doi: 10.19139/SOIC-2310-5070-1939.

[16] A. GULLU and H. KUŞÇU, “Optimized Graph Search Algorithms for Exploration with Mobile Robot,” *EMITTER International Journal of Engineering Technology,* vol. 9, no. 2, p. 222–238, Dec 2021, doi: 10.24003/emitter.v9i2.614.

[17] N. Sugianti, A. Mardhiyah, and N. Romihim Fadlilah, “Komparasi Kinerja Algoritma BFS, Dijkstra, Greedy BFS, dan A* dalam Melakukan Pathfinding,” 2020.