Evaluating Pathfinding Algorithms in a Weighted Square Grid

Kesley Raimundo
Londrina, Brazil
kesleyraimundo@gmail.com
ORCID: 0000-0001-7376-750X

***Abstract*—This paper evaluates the efficiency and adaptability of several pathfinding algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Best-First Search (GBFS), and A* (A-star)—within a square grid environment. Performance metrics such as execution time, number of steps, and path cost were analyzed across some grid sizes from 10x10 to 100x100. Our findings indicate that while A* and Dijkstra's algorithm consistently found the most cost-effective paths, DFS and GBFS were faster, albeit at a higher cost. This study highlights the importance of choosing the right algorithm based on specific environmental criteria and suggests avenues for future research in dynamic scenarios and real-world applications. The experimental data and source code are available in a public GitHub repository for further exploration and verification.**

***Index Terms*—Pathfinding algorithms, square grid, Breadth-First Search, Depth-First Search, Dijkstra's algorithm, Greedy Best-First Search, A* algorithm, algorithm evaluation, computational navigation.**

I. Introduction

Pathfinding is a fundamental computational problem with wide-ranging applications in fields such as robotics, game development, logistics, and navigation systems [1]–[3]. It involves determining an efficient route between two points within a specified environment. Typically, this problem is represented through a graph, where an algorithm is tasked to find most optimal path from a starting vertex to a destination vertex, while minimizing predefined cost functions like distance, time, or resources [4], [5].

Despite its straightforward premise, the complexity of pathfinding significantly increases with the expansion of the environment. For instance, in a $n \times n$ grid employing Manhattan distance, and without obstacles, the total number of optimal paths N , those with length $2n$, can be expressed as:

$$N = \binom{2n}{n} = \frac{(2n)!}{n! \cdot n!}$$

For large values of n, this equation asymptotically approaches an exponential behavior $N \sim 4^n$ [6], [7]. This becomes particularly significant in real-world scenarios, which commonly involve large n values. Therefore, if we aim to solve real-world pathfinding problems, it is crucial that we have efficient algorithms that can adeptly navigate this vast search space.

Pathfinding algorithms can broadly be categorized into two classes: deterministic and heuristic. Deterministic algorithms follow a fixed rule or set of instructions to determine the path from the start vertex to the destination, ensuring predictable behavior across multiple runs. Examples include Depth-First Search (DFS) and Breadth-First Search (BFS), which systematically explore all possible paths [4], [7]. On the other hand, heuristic algorithms, such as A* and Greedy Best-First Search, employ informed guesses to prioritize paths that appear to lead more directly towards the goal [5], [8].

While the effectiveness of these algorithms is deeply influenced by the intricacies of specific problems, employing a grid as a simplified model offers a foundational framework for systematic analysis. This approach not only facilitates the comparative evaluation of different algorithms but also highlights their adaptability and performance when faced with variations in the problem's parameters. The grid, therefore, stands as a pragmatic simplification for initial studies, providing a clear and manageable platform from which to launch a comprehensive investigation into pathfinding strategies. It acts as a conceptual bridge, connecting the theoretical foundations of algorithm design with their practical deployment in real-world scenarios [9], [10]. This dual role underscores the grid's significance not just as an academic tool, but also as a stepping stone towards developing robust solutions for complex, real-life navigation problems.

Informed by these principles, this study sets out to evaluate a range of pathfinding algorithms within a square grid environment. Our examination encompasses deterministic strategies to heuristic-based techniques. The goal is to dissect the comparative effectiveness, adaptability, and performance subtleties of these algorithms as they navigate the structured yet dynamic challenges presented by the grid environment [11], [12].

This paper is organized as follows: Section II establishes the experimental framework, outlining the grid environments and scenarios under which the pathfinding algorithms are assessed. Section III presents the pathfinding algorithms explored in this study, divided into deterministic and heuristic categories, and details the nature of each algorithm. In Section IV, we discuss the experimental results, providing data from a series of tests conducted on various grid sizes and analyzing the performance metrics of each algorithm. Section V offers conclusions drawn from the comparative performance analysis of the algorithms, highlighting their respective strengths and limitations in different scenarios. Finally, Section VI outlines future perspectives and the potential avenues for further research, underscoring the need for expanded studies and innovation in pathfinding algorithm applications. The paper concludes with an attachment

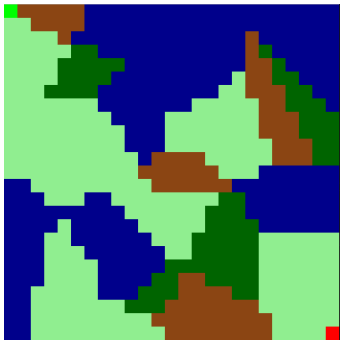

Fig. 1. Example grid setup for $n = 25$.

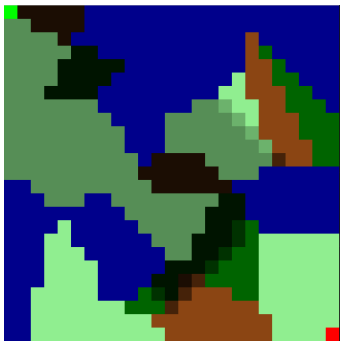

Fig. 2. BFS exploration after 200 steps, showcasing explored and frontier vertices in the grid defined by figure 1

section providing access to the source code and supplementary materials used throughout the experiments, facilitating further research and verification of the results presented.

II. Problem Setting

This study utilizes a square grid environment of dimensions $n \times n$, leveraging the Manhattan distance metric. Each grid cell represents a unique terrain type, each with an associated traversal cost. The grid coordinates are defined such that x increases from left to right and y increases from top to bottom. It is guaranteed that there will always be a viable path. The starting point is located at coordinates $(0, 0)$ and is marked with a bright green, while the goal is situated at $(n-1, n-1)$ and is colored red. The terrain types are visually distinguished as follows: plains (light green) with a cost of 1, swamp (dark green) with a cost of 2, mountain (brown) with a cost of 3, and ocean (deep blue) which is impassable. An example board configuration is depicted in Figure 1.

The primary objective for the algorithms is to find a feasible and cost-effective path from the starting point to the goal.

For visual clarity during the algorithm's execution, the current frontier vertices are darkened by 50%, and vertices that have already been visited are darkened by 80%. This enhancement clearly demonstrates the algorithm's progression through the grid. Figure 2 illustrates the exploration process of the BFS algorithm. Upon successfully identifying a path, the solution path is highlighted in yellow, as shown in Figure 3.

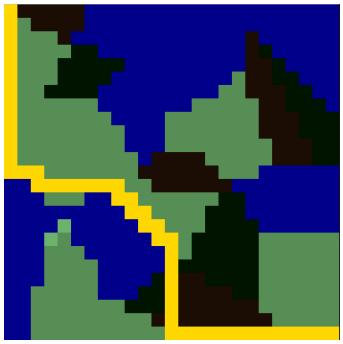

Fig. 3. Visual representation of a successful path found by BFS.

The Pygame library was selected for developing visual representations in Python due to its flexibility in handling various board configurations. The architecture of the code is structured around a 'Board' class that encapsulates all essential attributes and methods for managing the grid. This includes setting weights and drawing on the screen. Additionally, each algorithm is represented by its own class, containing a wrapper method to facilitate the execution of one step of the algorithm. An 'Experiment' class is also defined, which holds information about an instance of the 'Board' class and an algorithm. This class includes a 'run' method that allows the algorithm to interact with the board once it is configured. Furthermore, a 'finalize' method is used to generate results among other functionalities. The entire process is orchestrated from a main file, which passes each algorithm to an instance of the 'Experiment' class.

For comparative analysis, the algorithms will be tested a hundred times each on a grid with $n = 10$, $n = 25$, $n = 50$, and $n = 100$. Results will be compiled into tables and charts to visually compare the performance of each algorithm. These findings will be discussed in detail, noting key insights and patterns observed in relation to the theoretical expectations. The research culminates with conclusions and future perspectives.

III. Algorithms

This section provides a comprehensive overview of the pathfinding algorithms examined in this study. These algorithms are grouped into two main categories: deterministic and heuristic. Deterministic algorithms, which are known for their predictable behavior and consistent results, are foundational in the field of pathfinding. They include well-established methods such as BFS and DFS, which operate under fixed rules. On the other hand, heuristic algorithms, such as Greedy Best-First Search (GBFS) and A* (A-star), introduce a heuristic component to guide the search process. While GBFS seeks efficiency potentially at the cost of optimality, A* maintains optimality when using an admissible heuristic. This categorization not only aids in structured analysis but also highlights the distinct approaches each type of algorithm utilizes in navigating complex environments.

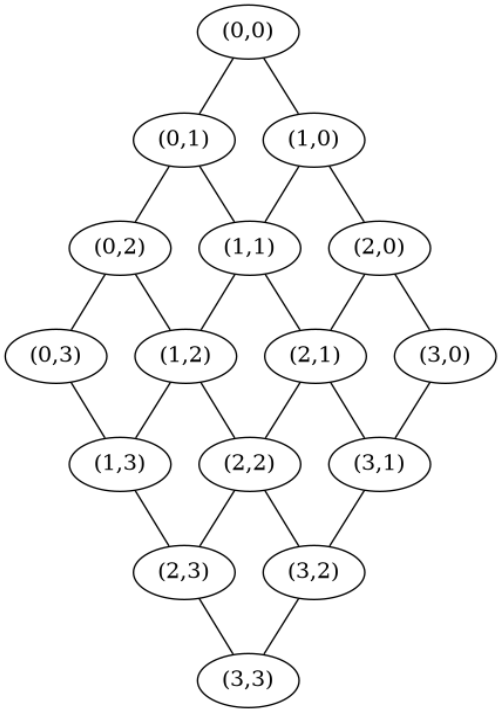

Fig. 4. Graph representation for a non-weighted grid with $n = 4$

Derministic Algorithms

Breadth-First Search: The BFS algorithm is an essential tool for traversing or searching tree and graph structures. It begins at a designated source vertex and methodically explores all neighboring vertices at the current level before progressing to vertices at the next level. This breadth-first traversal strategy ensures that vertices are explored in layers, which is particularly beneficial for applications that require processing on a level-by-level basis, such as network routing protocols or spreading processes in social networks.

The graph in figure 4 provides a clear visual representation of how BFS operates on a grid with $n = 4$. Each vertex in the graph represents a vertex labeled by a coordinate pair (x, y), where x and y denote the row and column positions on the grid, respectively. The root of the tree, $(0, 0)$, is the starting point for the BFS. From there, the algorithm explores its immediate neighbors—$(0, 1)$ and $(1, 0)$—before moving onto the next set of neighbors. As shown in the diagram, each subsequent level of the tree represents a new 'wave' of exploration extending outward from the source. This layer-by-layer expansion is typical of BFS, illustrating its utility in ensuring all vertices are visited in the shortest path order from the starting point.

BFS starts by marking the source vertex $(0, 0)$ as visited and adding it to a queue. The algorithm continues until the queue is empty, dequeuing a vertex, checking its neighbors, and enqueuing any unvisited ones. This approach ensures vertices are explored in order of increasing distance from the source.

The FIFO (First-In-First-Out) nature of the queue is key to BFS. It guarantees vertices are visited in their discovery order, preserving this sequence. Importantly, this order ensures that when a vertex v is first reached, the discovered path is the shortest one from the source. However, its performance may degrade in densely connected graphs due to the rapid growth of the queue, impacting both runtime and memory usage.

Depth-First Search: The DFS algorithm is another one of the fundamental algorithm used for exploring graphs. It starts at a chosen vertex marking it as visited. From there, DFS dives deeper into the graph, moving from one vertex to its adjacent unvisited vertex, marking each as visited along the way.

The graph in figure 4 also serves to illustrate DFS's path through a grid with $n = 4$. Beginning at the root $(0, 0)$, DFS would explore deeper, possibly moving to $(0, 1)$ and then further to $(0, 2)$, before backtracking when no further moves are possible, and then proceeding to explore other branches like $(1, 0)$ to $(2, 0)$, and so forth. This method continues, exploring as deeply as possible along each branch before backtracking to previously visited vertices to explore unvisited paths.

DFS typically uses a stack to keep track of the vertices. This can be implemented either iteratively with an explicit stack or recursively, utilizing the call stack to backtrack when necessary. The exploration completes when all vertices are visited, and all paths explored.

Although DFS is thorough in exploring all paths, it does not guarantee the shortest path between points. Its strength lies in its ability to perform a complete traversal, useful for applications such as cycle detection in graphs, topological sorting in directed acyclic graphs, or finding connected components in undirected graphs.

Dijkstra: Dijkstra's algorithm enhances BFS for weighted graphs by efficiently finding the shortest paths from a starting vertex to all other vertices. The algorithm initializes by setting the distance of the starting vertex to zero and all other vertices to infinity. It uses a priority queue to organize the vertices according to their shortest known distances, ensuring that vertices with the shortest distances are processed first.

The algorithm begins by dequeuing the vertex with the smallest distance from the priority queue and examining each of its neighbors. For each adjacent vertex v, if the path through u offers a shorter distance than currently recorded, the algorithm updates v's distance. This process, known as "relaxation," repeats until the priority queue is empty, indicating that the minimum distances to all reachable vertices are determined.

The strategic use of a priority queue is crucial for efficiency, optimizing the selection of vertices and focusing on the most promising paths from the outset. Implementations can vary from binary heaps to more complex structures like Fibonacci heaps, affecting the algorithm's performance and speed.

Differing from BFS, which does not differentiate between edge weights, Dijkstra's algorithm takes into account the diversity in edge costs, making it adept at handling graphs where these weights represent various metrics like distance or time.

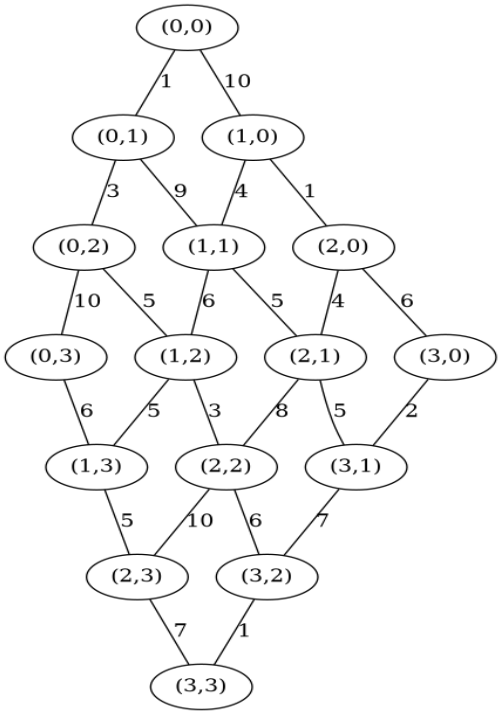

Fig. 5. Illustrative graph of a weighted grid with $n = 4$, showing edge weights and shortest path calculations.

Dijkstra's algorithm is particularly valuable in numerous real-world applications, from route planning in transportation networks to optimizing network data flow and logistical pathfinding, showcasing its versatility and critical role in operational strategies.

Heuristic Algorithms

Greedy Best-First Search: The GBFS is an informed search method that uses problem-specific concepts to guide its exploration towards the goal more directly and efficiently than uninformed methods like BFS. This search method employs a function that estimates the lowest cost from the current vertex to the goal, using prior knowledge to optimize the search by focusing on the most promising path locally.

GBFS starts at the initial vertex and expands the seemingly nearest vertex to the goal, based on its heuristic estimate. This chosen vertex is from an open list of vertices that have been discovered but not yet fully explored. As vertices are processed, their successors are evaluated with the same heuristic and added to the open list if unexplored.

Unlike BFS, which uniformly explores all adjacent vertices before moving deeper, GBFS makes a strategic choice to follow a more direct path towards the goal. This approach can lead to faster pathfinding, although it does not always guarantee the shortest or most optimal path. The effectiveness of GBFS largely depends on the heuristic. A well-formulated heuristic can significantly decrease the search time by avoiding less promising routes, whereas a less effective one may lead to increased search times or even failure to find a path.

A Algorithm:* The A* algorithm represents a sophisticated enhancement of the core principles found in both GBFS and Dijkstra's algorithm. It is specifically engineered to efficiently determine the shortest path in a weighted graph. A* combines the heuristic-driven efficiency of GBFS with the thoroughness and reliability of Dijkstra, employing heuristics to strategically guide the search and optimize path selection.

In A*, each vertex v in the graph is assessed using the evaluation function $f(v) = g(v) + h(v)$, where $g(V)$ is the known cost from the start vertex to v, and $h(v)$ is a heuristic that estimates the minimum cost from v to the goal. Crucially, this heuristic must be admissible, meaning it should not overestimate the actual cost to the goal. The heuristic's role is pivotal in enhancing A*'s efficiency by enabling the prioritization of more promising vertices, thus minimizing the number of vertices explored.

Like its predecessors, Dijkstra and GBFS, A* utilizes a priority queue to manage the exploration of vertices, assigning priorities based on $f(v)$. The process begins by placing the initial vertex in the priority queue with its corresponding $f(v)$ value. At each iteration, the vertex with the lowest $f(v)$ is removed from the queue for examination. For every successor w of vertex v, $g(w)$ is recalculated as $g(v) + \text{cost}(v, w)$, and $f(w)$ is updated accordingly. Should w already exist in the queue with a higher $f(w)$, it is updated to reflect the new, lower cost. This procedure is repeated until the goal vertex is reached or the queue empties, indicating that no viable path exists.

The effectiveness of A* hinges on the balanced interplay between the precise estimations of $g(n)$ and the heuristic efficiency of $h(n)$. This synergy allows A* to avoid less promising paths that might be pursued in a non-heuristic approach like Dijkstra's while also sidestepping the pitfalls of being overly reliant on the heuristic, as seen in GBFS.

IV. Experimental results

The initial series of experiments were conducted on a 10x10 grid. The corresponding results for elapsed time, number of steps, and total path cost for each tested algorithm are displayed respectively in figures 6, 7 and 8.

In terms of speed, DFS and GBFS emerged as the fastest algorithms, as depicted in Figure 6. This finding is consistent with expectations, as DFS systematically explores the deepest vertices of the graph, which, in this scenario, coincide with the goal location. Similarly, GBFS operates with a comparable strategy but incorporating the Manhattan heuristic.

On the other hand, BFS and Dijkstra proved to be the most time-consuming. This is due to their more horizontal approach to graph traversal, as they extensively explore the graph before progressing to deeper vertices. Consequently, in scenarios like this, they take longer to reach the goal. In contrast, the A* algorithm displayed slightly faster performance compared to the previous two.

The rapid completion by DFS and GBFS did not translate, however, to cost efficiency. As detailed in figure 8, the paths they discovered were generally more expensive than those

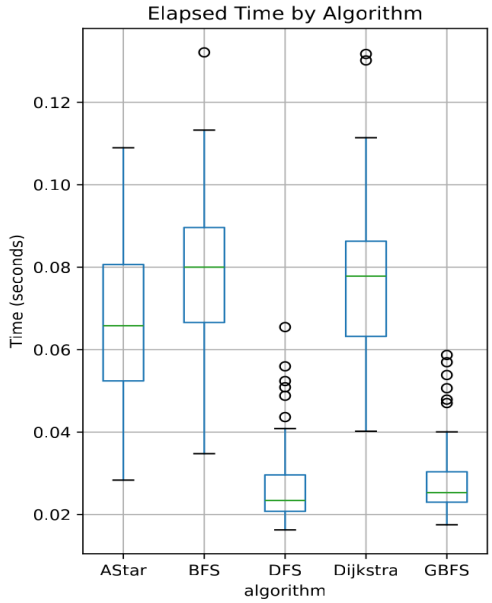

Fig. 6. Box plot for the time elapsed in seconds for each algorithm in a 10×10 grid.

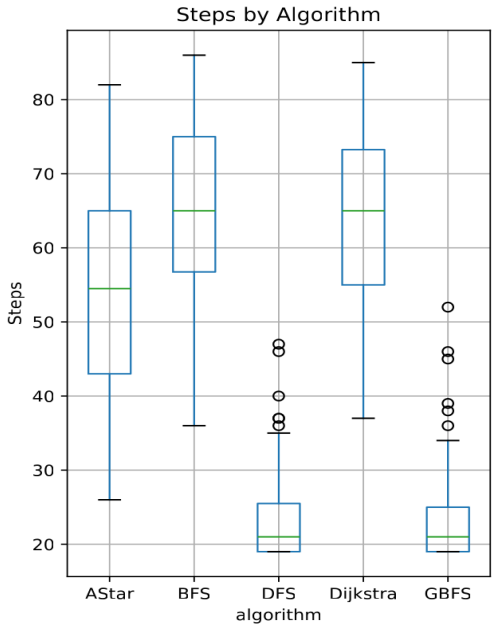

Fig. 7. Box plot for the number of steps taken by each algorithm in a 10×10 grid.

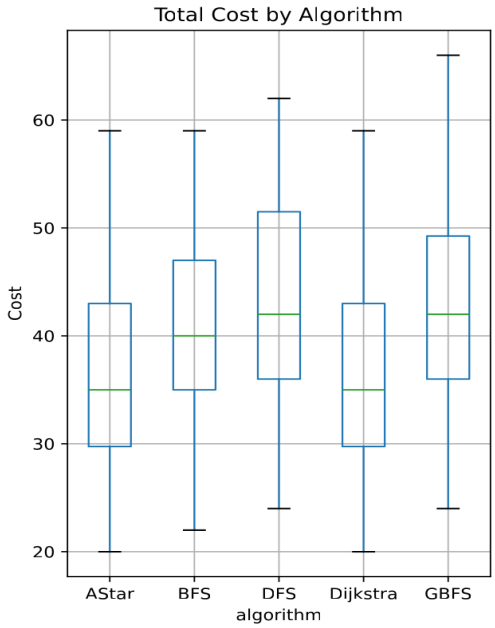

Fig. 8. Box plot for the total cost of the found path by each algorithm in a 10×10 grid.

identified by A*, which consistently found less costly routes. Also, we clearly see that the cost metrics for Dijkstra's and A* algorithms were equal, as evidenced in table I, since both guarantee the shortest possible path, unlike BFS which only achieves this under uniform weight conditions.

TABLE I
DESCRIPTIVE STATISTICS FOR THE TOTAL COST BY ALGORITHM FOR THE $n = 10$ GRID

Algorithm	Mean	Standard Deviation
BFS	37.926	6.804
DFS	40.086	8.011
Dijkstra	32.827	6.782
A*	32.827	6.782
GBFS	40.049	7.589

It is evident by comparing figures 6 and 7 that there are a correlation between the time and the number of steps. This happens because both are inherently linked by the algorithmic operations per step, only differing in value by a factor that depends on hardware efficiency.

In a larger grid experiments(25×25, 50×50 and 100×100), we observed a consistent pattern in algorithm performance. Therefore, for conciseness, it suffices discuss only the 100×100 grid results, illustrated in figures 9, 10, and 11.

In the larger grid experiments, as just stated, we observe a continuation of the patterns observed in the 10×10 case, underscoring the inherent behavior of algorithms within square grids. Once again, DFS and GBFS exhibit efficiency by swiftly navigating to the solution with minimal steps and time. In contrast, BFS and Dijkstra's algorithm adopt a more horizontally expansive approach across the graph, despite the goal often lying along the initial straight path traversed by

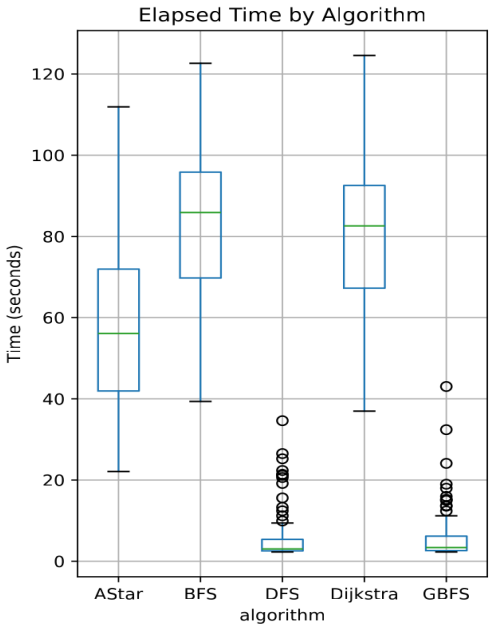

Fig. 9. Box plot for the time elapsed in seconds for each algorithm in a 100×100 grid.

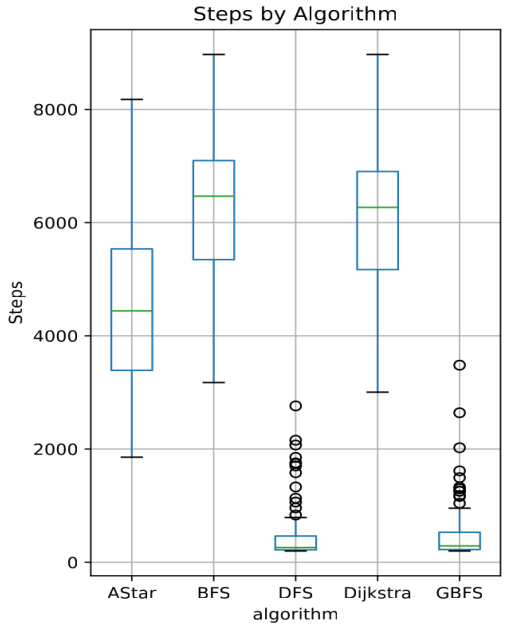

Fig. 10. Box plot for the number of steps taken by each algorithm in a 100×100 grid.

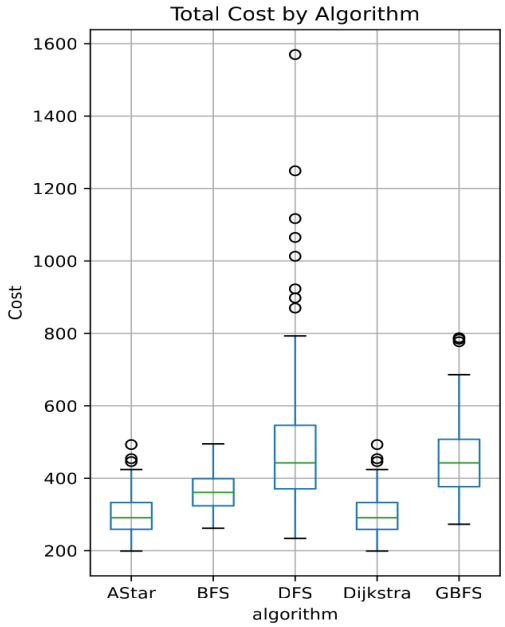

Fig. 11. Box plot for the total cost of the found path by each algorithm in a 100×100 grid.

DFS and BFS. The incorporation of a heuristic in A* further accelerates its performance compared to BFS and Dijkstra, ensuring the discovery of the shortest path with heightened certainty.

Additionally, on the larger grids, the utilization of heuristics in GBFS occasionally yields marginally superior outcomes compared to DFS. This observation is particularly pronounced in figure11 by noting the edge cases situated as outliers in the box plot, as well as (equivalently) the higher standard deviation in table II. In such cases, DFS may exhibit suboptimal performance compared to heuristic-driven strategies, emphasizing the nuanced impact of heuristics on algorithmic efficiency.

TABLE II
DESCRIPTIVE STATISTICS FOR THE TOTAL COST BY ALGORITHM FOR THE 100×100 GRID

Algorithm	Mean	Standard Deviation
BFS	362.940	53.509
DFS	503.510	219.910
Dijkstra	298.900	58.569
A*	299.230	58.392
GBFS	452.860	110.152

Figures 12, 13, and 14 offer a comprehensive comparison of each algorithm across various grid sizes[1].

In Figure 12, we observe that for $n = 10$, as indicated by the box plot, there is minimal variance in the time taken by each algorithm. This is expected since the grid size is relatively small, and all algorithms quickly reach the goal.

[1]Note that the plot contains only 10, 25, 50, and 100 points on the x-axis. However, the primary analysis does not hinge on having additional points for interpolation and smoothing of the curve. While a more detailed analysis could be pursued, it would demand considerable computational resources and might not alter any of the conclusions presented herein.

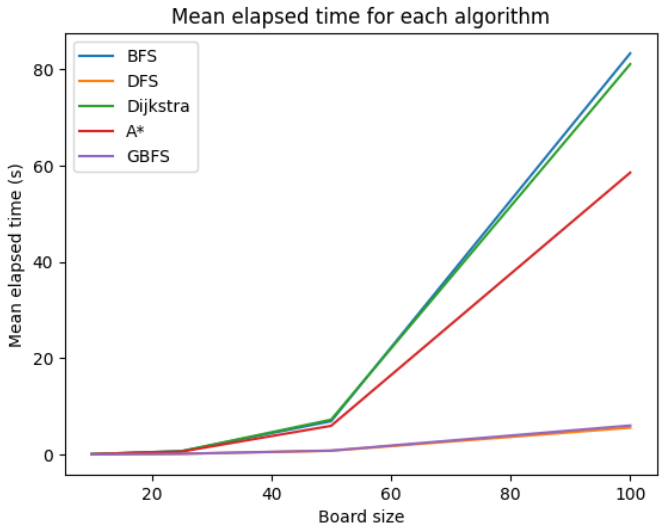

Fig. 12. Enter Caption

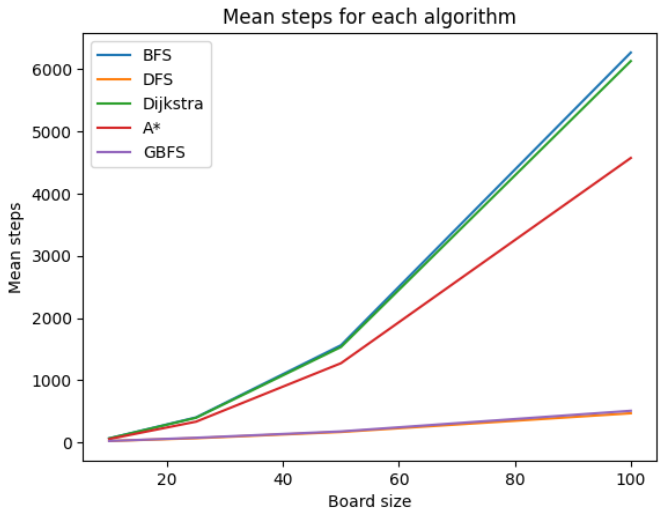

Fig. 13. Enter Caption

However, as the grid size increases, a distinct trend emerges. We observe a gradual increase in time for GBFS and DFS, with DFS slightly outperforming GBFS due to its disregard for weights. Conversely, BFS and Dijkstra's time requirements escalate rapidly with grid size. A* algorithm exhibits a moderate increase in time, trailing slightly behind GBFS and DFS in growth rate.

As previously mentioned, the pattern persists in the number of steps taken by each algorithm, as illustrated in Figure 13. This reaffirms the correlation between the number of steps and the time elapsed, underscoring the consistency of algorithmic behavior across metrics.

In Figure 14, as anticipated, we observe the precise alignment of Dijkstra's and A* algorithms' cost metrics, reflecting their shared ability to find the optimal path. Conversely, the cost incurred by the DFS algorithm escalates rapidly in

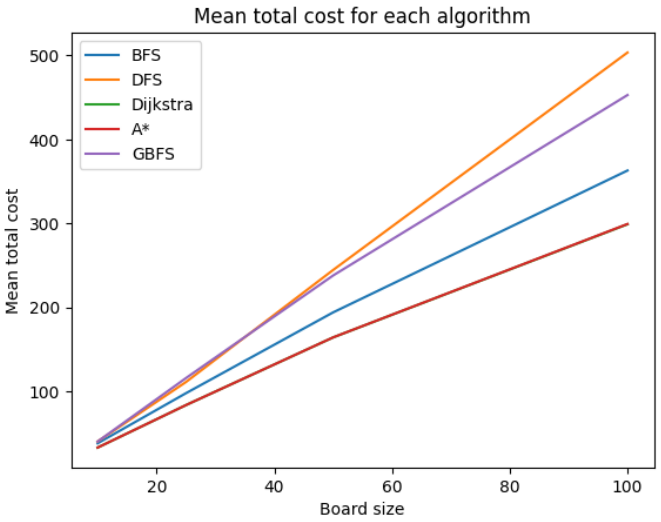

Fig. 14. Enter Caption

comparison to the other algorithms. This disparity arises from DFS's lack of heuristic awareness, leading to suboptimal path choices.

V. Conclusions

This study has conducted an analysis on the performance of five principal pathfinding algorithms: BFS, DFS, Dijkstra's, GBFS, and A*. Each algorithm was tasked to find the always feasible path between a fixed starting and ending cell across grids of sizes 10×10, 25×25, 50×50, and 100×100. Each grid was weighted randomly across 100 runs per algorithm for each grid size.

A* and Dijkstra's algorithms reliably produce optimal paths in varied environments, making them ideal for scenarios where minimizing path costs is critical. In general, A* provides a good balance since it finds an optimal path faster than BFS and Dijkstra. Conversely, DFS and GBFS excel in conditions where speed is essential, swiftly finding paths, though not necessarily the most cost-effective ones. This distinction underscores their potential in urgent situations where finding any feasible path quickly is more critical than ensuring its optimality.

For instance, during this work experiments, GBFS effectively identified 100 viable paths quickly to facilitate the experimental setup and subsequent evaluation of all algorithms. It is important to note that if no path is found, GBFS and other algorithms must still explore the entire graph to confirm the absence of a path. However, GBFS does this more rapidly when feasible paths are likely.

Conversely, in cases where starting and ending points are variable, GBFS might not always be the optimal choice, as it can end up scanning the entire graph in situations where BFS could find a path in just a few steps. This variability highlights that the choice of the most effective algorithm often depends heavily on the specific parameters of the experiment.

This can be clearly seen in Figure 4. Specifically, a starting point at (0,0) and an endpoint at (3,3) tend to favor DFS-like algorithms since they can proceed directly to the solution. However, if the endpoint were, for instance, (0,2), DFS would require many more steps compared to BFS-like algorithms.

Additionally, it appears that using the Manhattan distance heuristic generally improves outcomes slightly compared to not using a heuristic at all. However, this improvement is context-dependent, and there are scenarios where an inappropriate heuristic could exacerbate the difficulty of the problem.

These observations confirm that the performance and cost-effectiveness of each algorithm are significantly influenced by the specific requirements of the pathfinding scenario, emphasizing the importance of environment and experimental conditions in selecting the appropriate algorithm.

VI. Final Remarks and Future Perspectives

Although our experimental setup is straightforward, it yields significant insights into pathfinding algorithms. The conclusions drawn highlight potential avenues for further research, though these ideas certainly require empirical testing to substantiate their validity. The primary objective of this discussion is to illustrate the breadth of possibilities for future exploration in the field of pathfinding.

Expanding the scope of the experiments could involve randomizing the starting and ending points to simulate more dynamic scenarios. Additionally, introducing and testing a wider array of algorithms could uncover new efficiencies or reveal limitations in the current methodologies. It might also be worthwhile to explore variations of the existing algorithms. For instance, an approach where the algorithm initiates from both the start and endpoint, attempting to meet at a connecting vertex, could be tested for efficiency.

Further investigation could include testing state-of-the-art algorithms and comparing their performance against these standard ones. Moreover, integrating more complex heuristics or adapting the algorithms to handle different terrain types or obstacles within the grid could offer deeper insights.

Ultimately, continuous innovation and experimentation are crucial. Future studies could also look at the impact of algorithmic improvements in real-world applications such as robotics navigation, emergency evacuation simulations, and route planning in dynamic environments. Each of these possibilities not only extends the knowledge base but also enhances the practical utility of pathfinding algorithms in various sectors.

Attachment

The source code and supplementary materials used for the experiments described in this paper are available in a public GitHub repository. This ensures transparency, reproducibility, and facilitates further research based on this work. The repository includes all scripts, data, and detailed instructions necessary to replicate the results presented herein.

For access to the repository, please visit the following URL:

https://github.com/Kefsner/utfpr-ai-project

This repository is maintained by the author and is hosted under the username `Kefsner`. Interested parties are encouraged to fork, utilize, and contribute to the repository in accordance with the terms specified within the project's README and license files.

References

[1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Pearson, 2016.

[2] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[3] I. MILLINGTON and J. FUNGE, "Artificial intelligence for games. 2ª edição," *Burlington (EUA): Taylor & Francis Inc*, 2009.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[6] D. E. Knuth, *The art of computer programming*. Pearson Education, 2005.

[7] R. Sedgewick and K. Wayne, *Algorithms*. Addison-wesley professional, 2011.

[8] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[9] N. R. Sturtevant, "Benchmarks for grid-based pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144–148, 2012.

[10] D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid maps," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 25, no. 1, 2011, pp. 1114–1119.

[11] S. Rabin and N. R. Sturtevant, "Pathfinding architecture optimizations," in *Game AI Pro 360: Guide to Movement and Pathfinding*. CRC Press, 2019, pp. 1–12.

[12] D. Silver, "Cooperative pathfinding," in *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, vol. 1, no. 1, 2005, pp. 117–122.

[13] S. S. Skiena, *The algorithm design manual*. Springer, 1998, vol. 2.

[14] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms*. McGraw-Hill Higher Education New York, 2008.

[15] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education India, 2006.

[16] Z. B. Zabinsky *et al.*, "Random search algorithms," *Department of Industrial and Systems Engineering, University of Washington, USA*, 2009.

[17] K. L. Lim, K. P. Seng, L. S. Yeong, L.-M. Ang, and S. I. Ch'ng, "Uninformed pathfinding: A new approach," *Expert systems with applications*, vol. 42, no. 5, pp. 2722–2730, 2015.

[18] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical pathfinding." *J. Game Dev.*, vol. 1, no. 1, pp. 1–30, 2004.

[19] A. Kherrour, M. Robol, M. Roveri, and P. Giorgini, "Evaluating heuristic search algorithms in pathfinding: A comprehensive study on performance metrics and domain parameters," *arXiv preprint arXiv:2310.02346*, 2023.

[20] A. Chen and Z. Ji, "Path finding under uncertainty," *Journal of advanced transportation*, vol. 39, no. 1, pp. 19–37, 2005.

[21] R. Dube, A. Joshi, S. Bhagwat, P. N. Mahalle, and J. Barot, "Pathfinding visualizer: A survey of the state-of-art," in *International Conference on Information and Communication Technology for Intelligent Systems*. Springer, 2023, pp. 139–149.

[22] N. Rivera, C. Hernández, and J. Baier, "Grid pathfinding on the 2k neighborhoods," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[23] D. Harabor and A. Grastien, "An optimal any-angle pathfinding algorithm," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 23, 2013, pp. 308–311.

[24] A. Botea, "Ultra-fast optimal pathfinding without runtime search," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 7, no. 1, 2011, pp. 122–127.

[25] D. S. Ashish, S. Munjal, M. Mani, and S. Srivastava, "Path finding algorithms," in *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 1*. Springer, 2021, pp. 331–338.

[26] D. Demyen and M. Buro, “Efficient triangulation-based pathfinding,” in *Aaai*, vol. 6, 2006, pp. 942–947.