

Effat University Repository

Exploring the Maze: A Comparative Study of Path Finding Algorithms for PAC-Man Game

Authors	Salem, Nema;Haneya, Hala;Balbaid, Hanin;Asrar, Manal
DOI	10.1109/LT60077.2024.10469459
Publisher	IEEE
Download date	2026-05-17 09:14:36
Link to Item	http://hdl.handle.net/20.500.14131/1557

Exploring the Maze: A Comparative Study of Path Finding Algorithms for PAC-Man Game

Nema Salem

*Electrical and Computer Engineering Department
Effat University
Jeddah, Saudi Arabia
nsalem@effatuniversity.edu.sa
orcid.org/0000-0002-8440-4111*

Hala Haneya

*Computer Science Department, ECoE
Effat University
Jeddah, Saudi Arabia
hahaneya@effat.edu.sa*

Hanin Balbaid

*Computer Science Department, ECoE
Effat University
Jeddah, Saudi Arabia
habalbaid@effat.edu.sa*

Manal Asrar

*Computer Science Department, ECoE
Effat University
Jeddah, Saudi Arabia
maasrar@effat.edu.sa*

Abstract—Artificial Intelligence (AI) has become an integral part of our lives, finding applications across various industries. Search algorithms play a crucial role in AI. This paper focuses on the comparison of different search algorithms within the context of path-planning in the UC Berkeley’s PAC-Man’s game. The algorithms under consideration include Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS), Iterative Deepening Depth First Search (IDDFS), and A^* Search. The objective is to identify the most effective algorithm in terms of path-finding performance. The study’s findings reveal that the A^* search algorithm outperforms the others in terms of score, cost, and node expansion, making it the most suitable choice for finding the shortest path in the PAC-Man’s game.

Index Terms—Artificial Intelligence, Search Algorithms, Path Finding, PAC-MAN Game

I. INTRODUCTION

Artificial Intelligence (AI) has revolutionized the gaming industry by enhancing game mechanics and creating more immersive and intelligent experiences for players. One area where AI has made significant strides is in the development and application of search algorithms in games [1], [2]. These algorithms play a crucial role in enabling game characters to navigate game worlds, find optimal paths, and make intelligent decisions in real-time.

Search algorithms in games involve the exploration of game states and the identification of the most favorable actions or paths to achieve specific objectives. By employing AI techniques, game developers can create intelligent agents capable of efficiently searching through complex environments, avoiding obstacles, and making informed decisions.

The application of search algorithms in games is particularly evident in path-finding, where game characters need to navigate through intricate game worlds to reach their goals. These algorithms enable characters to determine the shortest

or most optimal paths while considering various factors such as obstacles, terrain, and dynamic changes in the environment.

Different search algorithms have been developed and applied in games, each with its strengths and limitations. Algorithms such as Depth-First Search (DFS) and Breadth-First Search (BFS) offer simple yet effective approaches for path-finding. DFS explores paths depth-wise, while BFS explores paths in a breadth-first manner. These algorithms provide a foundation for understanding search techniques and serve as benchmarks for more advanced algorithms. The Uniform Cost Search (UCS) algorithm considers the cost associated with each path and selects the one with the lowest cumulative cost. It is particularly useful in games where movement costs vary or where characters need to reach specific locations efficiently. The A^* search algorithm is widely used in games due to its ability to find the shortest path while considering both the cost and heuristic estimation of reaching the goal. A^* combines the advantages of UCS and heuristic search, making it a powerful tool for path-finding in dynamic game environments. Moreover, Iterative Deepening Depth-First Search (IDDFS) combines depth-first search with iterative deepening, allowing for efficient exploration of large search spaces while also finding optimal paths. By applying these search algorithms, game developers can create intelligent game characters that can navigate complex environments, avoid obstacles, and make strategic decisions based on their objectives and surroundings. This enhances the realism and challenge of games, providing players with engaging and immersive experiences.

A search problem includes many items including the search space, start state, goal, search tree, actions, transition model, path cost, and optimal solution. Search algorithms have certain properties that are used to determine their efficiencies, such as completeness, optimality, time-complexity, and space-complexity. These algorithms are divided into two categories. The blind search that has not have any prior information about

the search space such as BFS, UCS, DFS and IDDFS. The heuristic search that has information about the search space and the goal state such as A^* . Fig. 1 illustrates the categories of search algorithms [3].

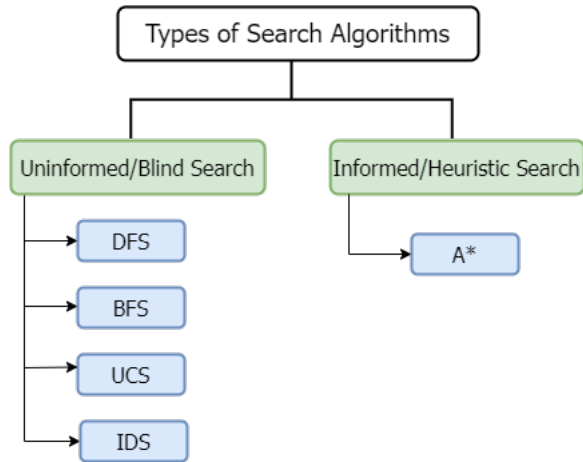


Fig. 1: Types of search algorithms.

This paper explores the application of search algorithms in the context of the widely popular and classic Japanese video game, PAC-Man [4] [5]. The game is to guide PAC-Man through a closed maze, consuming all the dots, while evading the ghosts that relentlessly pursue him. Beyond its entertainment value, PAC-Man provides an intriguing problem environment for testing and evaluating search algorithms. By leveraging the challenges presented in PAC-Man, this paper delves into the exploration of search algorithms, aiming to analyze their effectiveness and performance within the game's context. The structure of this article is as follows. Section II gives a description of five search algorithms. Section III gives a description about the used mazes and the implementation of the search algorithms. Section IV demonstrates the simulation results, with discussion. Lastly, section V concludes the work.

II. SEARCH ALGORITHMS

A. Depth-First Search

The DFS algorithm explores a tree by traversing as far as possible along each branch before backtracking. It starts at an initial node and explores deeper into the tree by visiting adjacent nodes until it reaches a dead end. It then backtracks to the previous node and continues exploring any unvisited adjacent nodes until all nodes have been visited. Fig. 2 illustrates the DFS algorithm. First, the searching starts at the root node S and then goes to the branch where node A is present. Then, it goes to node B, followed by node D. However, after node D there is no children, so it retraces the path in which it traversed and reach node B but now it goes through the un-traced path to E, and repeats the same process until reaching the goal node G.

While DFS has its advantages in terms of simplicity, memory efficiency, and exploration of deep paths, it also has limitations such as lack of completeness, suboptimal solutions,

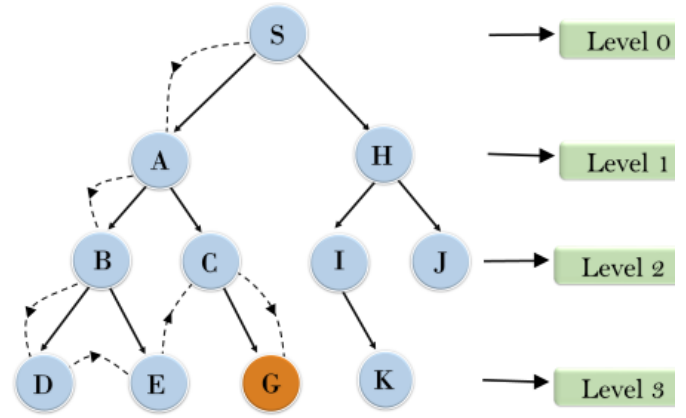


Fig. 2: Depth-first search [3].

and potential inefficiency on wide and balanced trees. The suitability of DFS depends on the specific problem domain and the characteristics of the graph being explored.

In [6], DFS was compared to other algorithms such as BFS and A^* to determine the shortest path between nodes for a mobile robot within various mazes. The authors concluded that DFS did perform well in finding the shortest path and it may provide longer paths. In [7], the authors presented the design and development of a line maze solver robot based on DFS. The results were acceptable and the robot was able to solve a looped maze in 63.40 seconds and a non-looped maze in 84.97 seconds. However, the robot fails to complete the looped maze under certain conditions.

B. Breadth-First Search

The BFS is a graph traversal algorithm used to explore and search for nodes in a tree. It starts at a given source node and explores all the neighboring nodes at the current depth level before moving to the nodes at the next depth level. It iterates layer by layer until the goal is reached, using a queue data structure that follows first-in first-out. Fig. 3 demonstrates the iterations and decisions taken by BFS. It starts with the source node S at layer 0 and visits all nodes of current layer then traversing through the remaining layers to perform the same operation until the goal node is reached [8].

BFS is complete as it guarantees finding a solution if one exists, given that the tree is finite and connected. Since it explores nodes in a breadth-first manner, it guarantees that the first occurrence of a node during traversal is the shortest path to reach that node. It only requires enough memory to store the nodes in the current level being explored. It does not store information about the entire graph or tree, which makes it memory-efficient compared to other graph traversal algorithms like DFS.

While BFS is memory-efficient during execution, it requires additional memory to keep track of visited nodes and the queue data structure. In graphs with a large number of nodes and complex connections, the memory requirements can be significant. In dense graphs, where the number of edges is close to the maximum possible, BFS can have a higher time

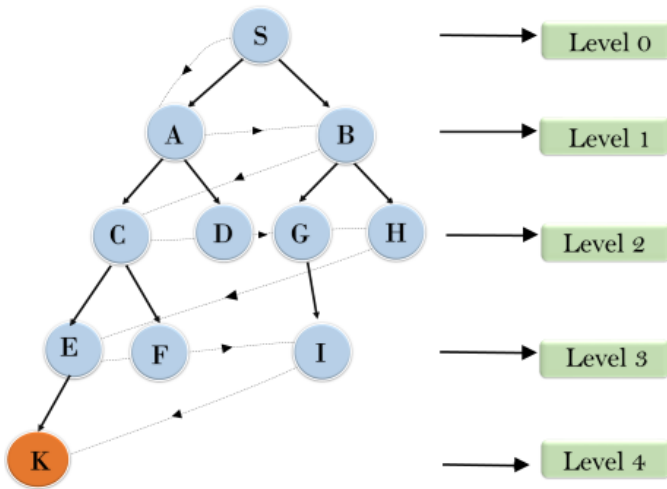


Fig. 3: Breadth-first Search [3].

complexity compared to other algorithms like DFS. This is because BFS visits all neighbors of a node before moving to the next level, resulting in a larger number of nodes in the queue. It explores nodes in a breadth-first manner, which means it can be inefficient for large graphs with a high branching factor. The number of nodes to be visited can grow exponentially with the depth of the graph, leading to slower execution times. It is not designed to find optimal solutions in graphs with weighted edges. It does not consider the edge weights and assumes all edges have the same cost.

Bidirectional BFS is an optimization of the standard BFS algorithm. It performs two separate BFS searches: one starting from the source node and the other starting from the target node. The searches proceed simultaneously, with each search exploring one level of nodes at a time. The algorithm terminates when a node is discovered that has been visited by both searches, indicating the shortest path has been found. It is a powerful optimization for finding the shortest path between two nodes in an unweighted graph. However, it may not be suitable for graphs with complex edge weights or when there are multiple target nodes [9].

C. Uniform Cost Search

The UCS traversal algorithm finds the path with the lowest cost between a source node and a goal node in a weighted graph. Unlike BFS or DFS, it takes into account the cost associated with each transition between nodes. It is commonly used in various applications, such as route planning, navigation systems, and resource allocation, where finding the lowest-cost path is crucial [10]. Fig. 4 shows the way that UCS calculates the total cost from the initial node, S, to any of the destination nodes [G1, G2, G3]. The directed vertices represent the direction and cost of the path adding up to the overall cost of the path which is a sum of all the paths.

The UCS is an optimal and complete algorithm when space is an issue and requires no heuristics. It is used in maze problems since it prioritizes the minimum cumulative cost

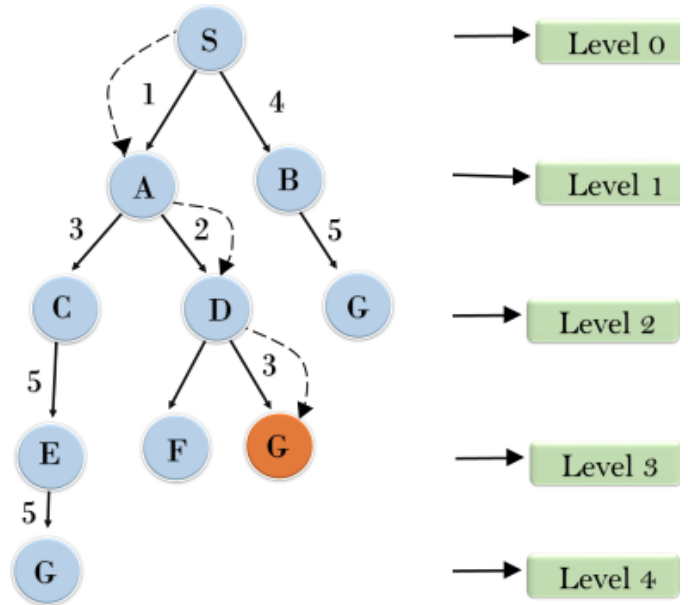


Fig. 4: Uniform-cost Search [3].

while the DFS algorithm gives maximum priority to maximum depth [11]. Its main drawback is that it is only concerned about the cost of the path and completely neglects the number of steps involved in searching which can cause it to be stuck in an infinite loop. Since UCS solely focuses on minimizing the cost, it may continue to explore paths with lower costs indefinitely, even if the number of steps required becomes unreasonably large. This can occur when there are cycles or loops in the graph, causing the algorithm to repeatedly revisit nodes and never terminate.

D. Iterative Deepening Depth-First Search

The IDDFS is an unformed search algorithm that combines the benefits of DFS and BFS. Fig. 5 illustrates the operation of the IDDFS among a tree. This algorithm performs DFS up to a certain specified depth limit [12]. If a goal is not found at a specified limit, then the search process will terminate. The limit will be incremented by one, and the process will start again until the goal is reached. In the case shown in Fig. 5, the algorithm will find the goal after the fourth iteration.

The IDDFS is complete, and memory-efficient algorithm that provide Optimal solution for unit-cost paths. Unfortunately, it performs redundant work by re-exploring nodes at each depth level, does not perform well in graphs with a high branching factor, does not take advantage of memory sharing between iterations, and does not consider edge weights or costs, which can lead to suboptimal solutions or inefficient exploration in such scenarios [13].

The authors in [14] compared IDDFS with Monte Carlo Tree Search (MCTS). The algorithms were compared on the premise of playing multi-agent visibility based pursuit-evasion games. Results showed that both perform well while being as algorithms for the pursuers, but IDDFS performs better as algorithm for the evader. The difference in performance comes

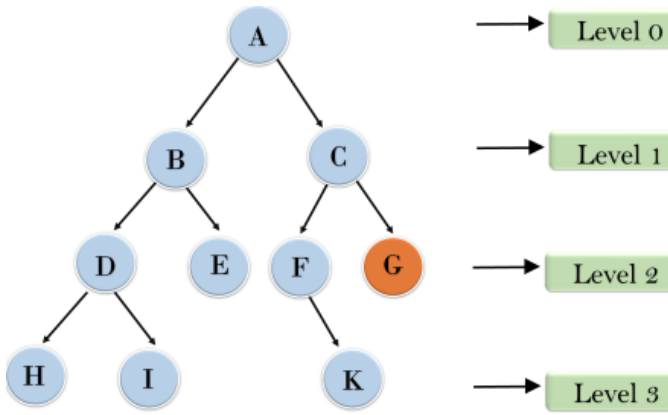


Fig. 5: Iterative Deepening Depth-First Search [3].

down to the branching factor, IDDFs is better with smaller branching factors, while MCTS is better with larger branching factors.

E. A* Search

The A* is widely used for finding the shortest path between two nodes in a graph, taking into account both the cost of the path and an estimated heuristic value to guide the search [15]. It is quick and accurate when estimating a path. The heuristic function value is represented by $f(n) = g(n) + h(n)$, where $g(n)$ is the cost that is required to reach a target node from the the root, and $h(n)$ is the heuristic value.

Closed list and open list refer to two distinct data structures used to keep track of the nodes during the A* search process. The open list, known as fringe or frontier, holds the nodes that have been discovered but not yet fully explored. These are the nodes that are potential candidates for expansion and further exploration. The open list is typically implemented as a priority queue, where the nodes are ordered based on their estimated total cost (the sum of the cost to reach the node from the start node and the estimated cost to reach the goal node). The node with the lowest estimated total cost is selected for expansion first. The closed list, explored set, stores the nodes that have been fully expanded or explored. Once a node is expanded, it is moved from the open list to the closed list to indicate that it has been processed. The closed list is used to avoid revisiting nodes that have already been explored, preventing redundant exploration and potential infinite loops. It helps ensure that each node is expanded only once during the search. During the A* search algorithm, the open list is initially populated with the start node, and the closed list is empty. The algorithm then proceeds by iterative selecting the node with the lowest estimated total cost from the open list, expanding it, and adding its neighboring nodes to the open list. As nodes are expanded, they are moved from the open list to the closed list. Both lists are illustrated in Fig. 7, and 6.

The A* search is complete and optimal, but it consumes a huge amount of memory as each explored node to be kept in memory, resulting in a space complexity. In [16], the authors

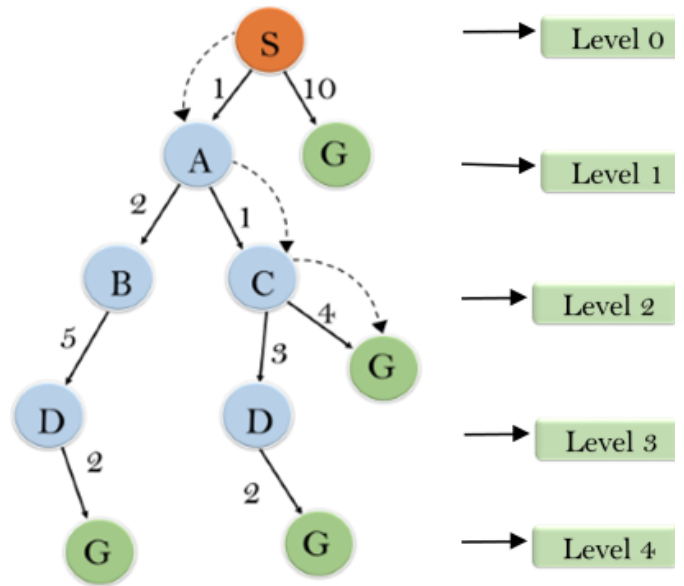


Fig. 6: A* Search Open List [3].

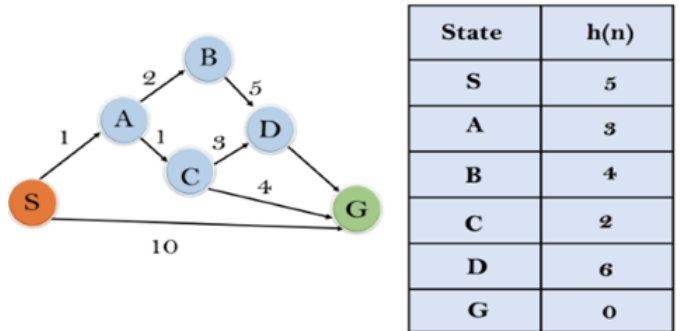


Fig. 7: A* Search Closed List [3].

used a combination of A* and Navigation mesh to optimize the shortest path-finding problem on enemies/ghosts of the PAC-Man game. The results showed that the movement of ghosts in catching PAC-Man using the A* algorithm was good in terms of the ghosts taking less steps to reach the goal.

III. PROPOSED ALGORITHM

This work examined the performance of the five search algorithms, mentioned in section II, with respect to the different types of mazes in the PAC-Man game. The best path is to be selected based on the cost in each algorithm.

The A* search algorithm is demonstrated by the code that follows. Prior to the main function, the code specifies a heuristic function since A* is an informed search. In the event that there is no route cost, this heuristic function is null and has only one line that returns a zero. A suite of utilities are included in the fringe package, which has been used to implement the primary function. The least-cost path from a given (initial - goal) node may be found with this package. Expanded nodes are then pushed to the visited queue after a queue and array are created to hold the visited nodes. When

one state remains after all nodes have been verified and popped from the isEmpty list, the goal state is returned. If not, the visited list iterates continuously.

```

def nullHeuristic(state, problem=None): """
    A heuristic function estimates the cost from the
    current state to the nearest
    goal in the provided SearchProblem. """
    return 0

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined
    cost and heuristic first."""

    # create fringe to store nodes
    fringe = util.PriorityQueue()
    # track visited nodes
    visited = []
    # push initial state to fringe
    fringe.push((problem.getStartState(), [], 0),
               heuristic(problem.getStartState(), problem))
    while not fringe.isEmpty():
        node = fringe.pop()
        state = node[0]
        actions = node[1]
        # goal check
        if problem.isGoalState(state):
            return actions

    if state not in visited:
        visited.append(state)
        # visit child nodes
        successors = problem.getSuccessors(state)
        for child in successors:
            # store state, action and cost = 1
            child_state = child[0]
            childaction = child[1]
            if child_state not in visited:
                # add child nodes
                childact = actions + [childact]
                cost =
                    problem.getCostOfActions(childact)
                fringe.push((child_state, childact,
                             0), cost +
                             heuristic(child_state, problem))

```

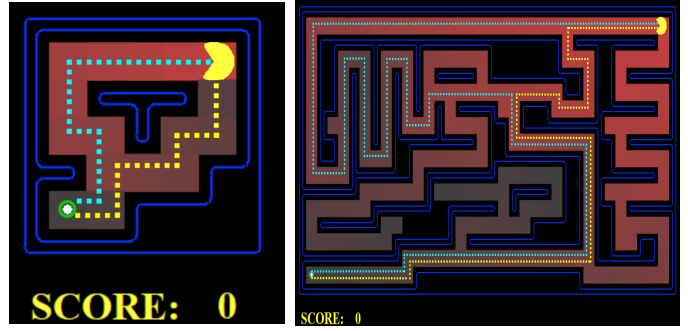
IV. SIMULATION RESULTS

The path finding algorithms: DFS, BFS, UCS, IDDFS, and A^* are implemented to navigate tiny, medium, and big PAC-Man mazes, shown in Fig. 8a, 8b, and 8c; respectively.

a) *Tiny Maze*: The algorithms BFS, UCS, and A^* followed the yellow path to traverse the maze and consume the dot marked in green, which is their objective, whereas DFS and IDDFS take the aqua path. Table I provides the outcomes of each algorithm.

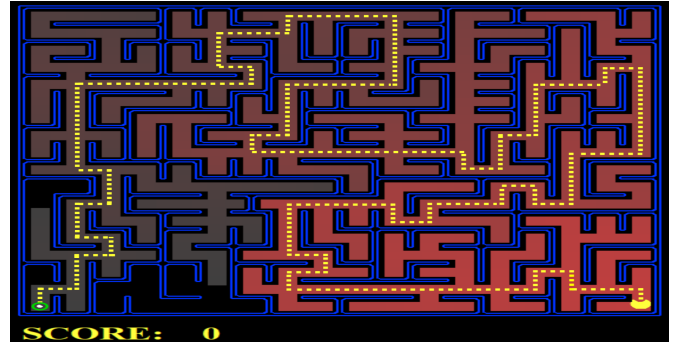
b) *Medium Maze*: In order to navigate the medium maze and attain the goal of eating the green-circled dot, the BFS, UCS, IDDFS, and A^* algorithms followed the yellow path, whereas DFS followed the aqua path. Table I provides the outcomes of each algorithm.

c) *Big Maze*: The five algorithms traversed the big maze using the identical yellow path in order to reach the objective and consume the green-circled dot. Table I provides the outcomes of each algorithm.



(a) Navigating tiny maze.

(b) Navigating medium maze.



(c) Navigating big maze.

Fig. 8: Navigating maze with different sizes.

The three top-performing algorithms in the tiny maze, A^* , BFS, and UCS, each had a cost of 8 and a score of 502, according to the results. However, A^* stood out due to its reduced number of expanded nodes, which was 14. The same three algorithms: BFS, UCS, and A^* had the highest score of 442 for the medium maze. However, the A^* search performed best, with a lower cost of 68 and the fewest expanded nodes (219). But in the big maze, all five algorithms—DFS, BFS, UCS, IDDFS, and A^* search—had the same score of 300 and cost of 210. DFS, however, fared the best because it had the fewest expanded nodes—390—while A^* search had the most expanded nodes—538, to be exact. This demonstrates how the DFS works better in expansive settings with solutions located far from the source.

V. CONCLUSION

AI-driven search algorithms are now a basic feature of game development, allowing avatars to traverse virtual environments and choose the best path. Search algorithms, ranging from basic techniques like DFS and BFS to more complex methods like UCS, A^* , and IDDFS, are essential to the development of intelligent and dynamic gaming experiences. PAC-Man is still a fun game for players to play, but it also provides an invaluable framework for assessing and contrasting search algorithms, which makes it a fascinating and difficult problem environment for AI researchers and game developers.

In this paper, we used AI search algorithms to present a path-finding computer game inspired by PAC-MAN. We described each algorithm in detail and compiled the simulation

TABLE I: Five search algorithms performance in the three mazes.

Search Algorithm	Tiny maze			Med maze			Big maze		
	Score	Cost	Extended Nodes	Score	Cost	Extended Nodes	Score	Cost	Extended Nodes
DFS	500	10	15	380	130	146	300	210	390
BFS	502	8	15	442	68	269	300	210	620
UCS	502	8	15	442	68	268	300	210	619
IDDFS	500	10	86	440	70	8635	300	210	60211
A*	502	8	14	442	68	219	300	210	538

results for all path-finding algorithms across various environments according to how well they performed in terms of score, cost, and node expansion, which showed how much memory was used. The A* algorithm proved to be the most effective in two of the environments and was deemed efficient in the third, according to the findings.

Beyond gaming, these algorithms are offering solutions to path-finding, optimization, and search problems in a wide range of fields, including web crawling, social network analysis, compiler design, routing, resource allocation, planning, robotics, and puzzle solving. Their efficient and effective exploration of search spaces makes them valuable tools in various real-world applications. It is important to consider the trade-off between time complexity and memory consumption when selecting an algorithm based on the specific problem requirements and constraints, which can be covered in a further work.

REFERENCES

- [1] S. Muggleton, "Alan turing and the development of artificial intelligence," *AI communications*, vol. 27, no. 1, pp. 3–10, 2014.
- [2] M. Taddeo and L. Floridi, "How ai can be a force for good," *Science*, vol. 361, no. 6404, pp. 751–752, 2018.
- [3] JavaTPoint, "Search algorithms in ai," 2023. [Online]. Available: <https://www.javatpoint.com/ai-informed-search-algorithms>
- [4] K. Collins, *From Pac-Man to pop music: interactive audio in games and new media*. Routledge, 2017.
- [5] U. Berkeley, "Berkeley ai materials," 2014. [Online]. Available: <http://ai.berkeley.edu/home.html>
- [6] T. Terzimehic, S. Silajdzic, V. Vajnberger, J. Velagic, and N. Osmic, "Path finding simulator for mobile robot navigation," in *2011 XXIII International Symposium on Information, Communication and Automation Technologies*. IEEE, 2011, pp. 1–6.
- [7] A. S. Hidayatullah, A. N. Jati, and C. Setianingsih, "Realization of depth first search algorithm on line maze solver robot," in *2017 International Conference on Control, Electronics, Renewable Energy and Communications (ICCREC)*. IEEE, 2017, pp. 247–251.
- [8] HackerEarth, "Breadth first search tutorials and algorithms," 2023. [Online]. Available: <https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/>
- [9] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–10.
- [10] D. Jungnickel and D. Jungnickel, "The greedy algorithm," *Graphs, Networks and Algorithms*, pp. 135–161, 2013.
- [11] M. J. Pathak, R. L. Patel, and S. P. Rami, "Comparative analysis of search algorithms," *International Journal of Computer Applications*, vol. 179, no. 50, pp. 40–43, 2018.
- [12] K. L. Lim, K. P. Seng, L. Yeong, S. Ch'ng, and K. A. Li-minn, "The boundary iterative-deepening depth-first search algorithm," in *Second International Conference on Advances in Computer and Information Technology: ACIT 2013*. Institute of Research Engineers and Doctors, LLC, 2013, pp. 119–124.
- [13] T. N. Lina and M. S. Rumetna, "Comparison analysis of breadth first search and depth limited search algorithms in sudoku game," *Bulletin of Computer Science and Electrical Engineering*, vol. 2, no. 2, pp. 74–83, 2021.
- [14] V. Lisỳ, B. Bořanskỳ, and M. Pěchouček, "Anytime algorithms for multi-agent visibility-based pursuit-evasion games," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, 2012, pp. 1301–1302.
- [15] A. Rafiq, T. A. A. Kadir, and S. N. Ihsan, "Pathfinding algorithms in game development," in *IOP Conference Series: Materials Science and Engineering*, vol. 769, no. 1. IOP Publishing, 2020, p. 012021.
- [16] M. Zikky, "Review of a*(a star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the pacman game," *EMITTER International journal of engineering technology*, vol. 4, no. 1, pp. 141–149, 2016.