Enhancing Pathfinding Efficiency in Unity: An Evaluation of Artificial Intelligence Pathfinding Algorithms

Priyanka Datta[1], Amanpreet Kaur[2,*], and Archana Mantri[3]

ARTICLE INFO

Article history:
Received: 15 August 2024
Revised: 18 October 2024
Accepted: 21 December 2024
Online: 15 May 2026

Keywords:
Pathfinding algorithms
BFS
DFS
Dijkstra's
GBFS
A*

ABSTRACT

The mechanisms behind intelligent navigation in robots, games, and other Artificial Intelligence (AI) systems are called pathfinding algorithms. This article presents a performance comparison of different pathfinding algorithms used in video games, robotics, and other applications. The algorithms are classified into two categories: uninformed search and informed search. Dijkstra's, Breadth First Search (BFS), and Depth First Search (DFS) algorithms are classified as uninformed pathfinding algorithms, whereas A* and Greedy Best First Search (GBFS) are informed pathfinding algorithms. The implementation of BFS, DFS, Dijkstra's, A*, and GBFS was done on the Unity game engine, and their elapse times were compared to find the optimal pathfinding algorithm. In comparison to A* and other algorithms, the result shows that GBFS is the fastest algorithm. However, the GBFS algorithm falls short in terms of optimal as it relies solely on heuristic values. However, the A* algorithm reliably discovers the most optimal path. The algorithm determines the most optimal path by considering both the actual cost and the heuristic cost.

1. INTRODUCTION

Digital games have become a fantastic source of entertainment for adults, younger people, and even kids in recent times. During the COrona VIrus Disease (COVID) pandemic situation, when people were unable to go outside of their homes, digital games became part of individual lives, and the trend is continuing after the pandemic as well. Technologies are advancing all over the world, and the quality of digital games is improving at the same time [1]. Initially, in the 1950s, chess games were developed by writing computer programs. In 1970, the start of digital games ensued with the launch of "Pong and Space War". These games had two players and worked on discrete logic. This led to the development of single-player digital games, where the developer must employ Non-Player Characters (NPC) in order to add realism to the game [2]. The player or any user of the game cannot control these NPC characters in the games. The concept of NPC is to connect the gaming industry with Artificial Intelligence (AI). AI can be defined as "man-made thinking power" because the phrase "artificial" implies "man-made," and the word "intelligence" denotes "thinking power" [3] [4]. It is a branch of computer science that enables the development of an intelligent machine or character that can think like a human, work like a human, and also take decisions like a human [5].

Pathfinding algorithms are one of the most important AI components. Common searching techniques for resolving AI problems are pathfinding algorithms [6] [7]. These pathfinding techniques or algorithms are typically utilized by NPC in AI to solve certain problems and deliver the best outcomes [8]. Pathfinding algorithms are broadly classified into two categories based on the type of information they use during the search process: uninformed (or blind) search algorithms and informed (or heuristic-based) search algorithms. In contrast to uninformed search, which just allows the algorithm to determine whether the current state is a goal state or not, informed search implies that the algorithm is aware of the problem. Informed search further categorized into A* and Greedy Best First Search (GBFS) [9]. Both A* and GBFS use heuristics approach to guide their search. The A* algorithm incorporates a more comprehensive evaluation of both the actual cost incurred and the estimated cost to the goal. GBFS, on the other hand, is a straightforward heuristic-based algorithm that ranks nodes according to their estimated cost (heuristic cost) to the goal without considering the entire cost incurred so far. The uninformed search is also classified into Breadth First Search (BFS), Depth First Search (DFS) and Dijkstra's algorithm. The BFS and DFS are used for traversing or searching graphs, while the Dijkstra algorithm is mainly

[1]*Laxmi Institute of Technology, Sarigam, Gujarat, India.*
[2]*Chitkara University Institute of Engineering and Technology, Chitkara University, Punjab, India.*
[3]*Anurag University, Venkatapur, Hyderabad, Telangana, India.*
Corresponding author: Amanpreet Kaur; Email: Amanpreet.kaur@chitkara.edu.in.

designed for searching the shortest paths in weighted graphs. BFs and DFS don't consider the weights of edges, making them suitable for unweighted graphs, whereas Dijkstra's algorithm takes edge weights into account for finding the optimal paths in weighted graphs. The pathfinding algorithms' classification is displayed in Figure 1. This work extensively uses the searching algorithm and evaluates its efficacy on a grid environment used in video games, both with and without obstacles.

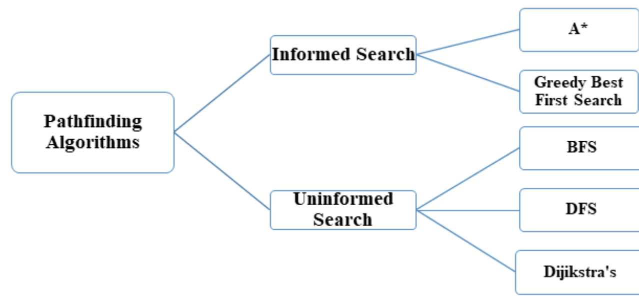


Fig. 1. A diagram that shows how pathfinding methods are classified.

1.1 Various Types of Grids

A grid is a collection of points or nodes through edges to form a graph [7]. Grids are broadly classified into two categories namely regular and irregular. Regular Grids are classified into triangular, square, hexagonal and voxel. Irregular grids are also classified into waypoints, mesh navigation and Sparse as shown in the Figure 2.

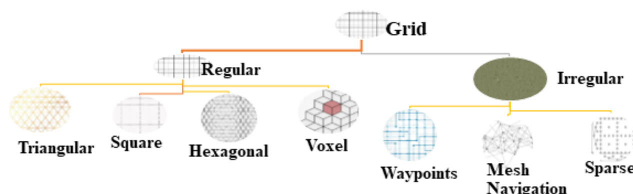


Fig. 2. The categorization of grid system.

1.1.1. Regular Grids

One of the most popular graph forms is the regular grid, which is extensively utilized by roboticists and computer game developers. Various video game developers have contributed to this field, creating titles like Company of Heroes, Civilization V, and Dawn of War 1 and 2 [10]. Various types of regular grids are briefly described in the following section.

a. *Triangular*: As seen in Figure 3, triangular grids do have some benefits, despite not being as common as square and hexagonal grids. Demyen and Buro introduced a technique based on constrained Delaunay triangulation that minimizes the search effort [11].

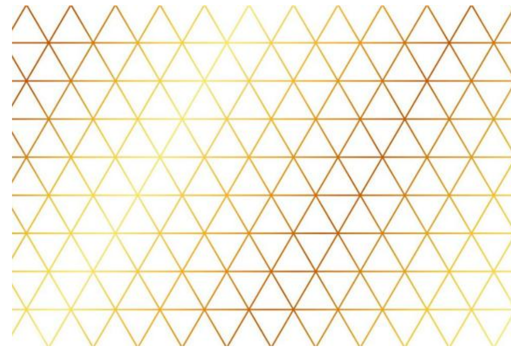


Fig. 3. The Hexagonal Grid.

b. *Square*: The most prevalent kind of regular grids are these. A 2D or 3D square grid is designed by dividing the environment into a consistent pattern of square cells. Pathfinding algorithms typically navigate via cells, each of which represents a unit of space, as depicted in the Figure 4.

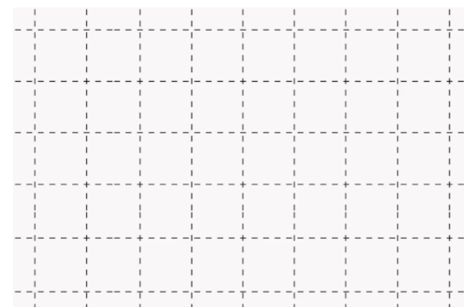


Fig. 4. The Hexagonal Grid.

c. *Hexagonal*: These grids are regular because every hexagon is equally spaced from the grids of its neighbors. They are typically utilized in games and simulations where movement can occur in multiple directions. It depicts the surroundings in a more realistic way, the Figure 5 below represent hexagonal grid.

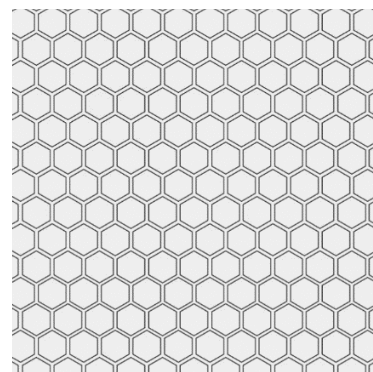


Fig. 5. The Hexagonal Grid

d. *Voxel:* Voxel grids are used in 3D environments to depict space as a regular arrangement of cubic

volumetric pixels called voxels, as shown in the Figure 6 below.

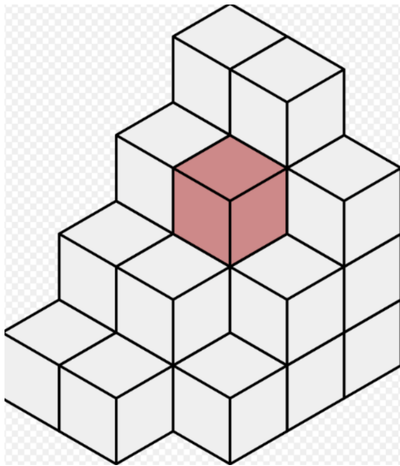


Fig. 6. The Voxel Grid.

1.1.2. Irregular Grids

In various sectors and applications, irregular grids are utilized. Different types of irregular grids are briefly illustrated in the following section.

a. *Waypoints*: In general, waypoints are particular nodes inside the grid that are identified for a variety of uses, including simulation, navigation and data analysis, as shown in Figure 7. A waypoint has irregular spaces between its nodes than a regular grid, which has uniform spacing between nodes [11].

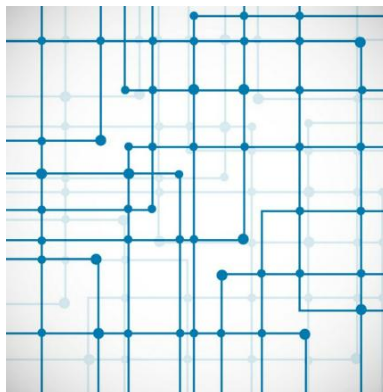


Fig. 7. The Voxel Grid.

b. *Mesh Navigation*: It represents navigable spaces with irregular polygons rather than standard cell arrangements. The geometry of the environment is frequently used to construct these polygons as shown in the Figure 8. Video games contribute to the majority of mesh graph applications [12].

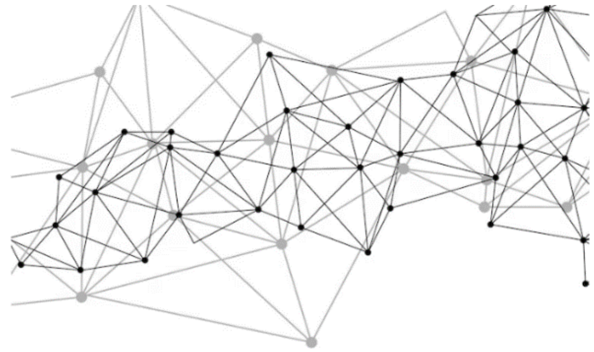


Fig. 8. The NavMesh Grid.

c. *Sparse*: These are asymmetrical in the sense that the grid does not include the entire environment, as depicted in the Figure 9. The overall size of the grid is decreased by including just pertinent sections or areas of interest [13].

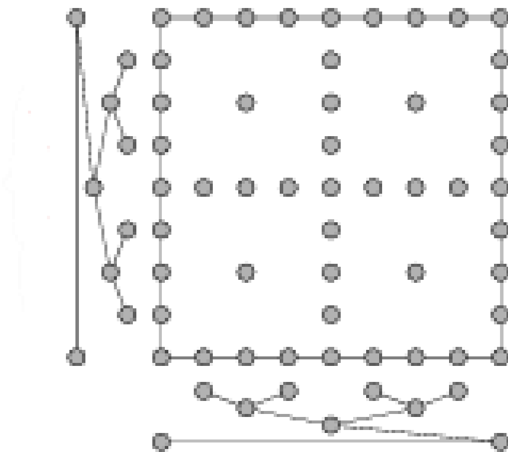


Fig. 9. The NavMesh Grid.

4. PATHFINDING ALGORITHMS

In this section, uninformed and informed pathfinding algorithms are illustrated with example.

*Breadth First Search (BFS)***:** It's an algorithm for exploring and searching graph data structures [14]. The search starts at the initial node (a), continues to level 1 from left to right, and then moves to the next level until all nodes are traversed or the desired node is obtained, as shown by the dotted arrow in Figure 10. This algorithm's primary advantage is that it always finds a solution, regardless of the nature of the problem. The BFS algorithm will find all solutions if there are more than one solution, and the minimum-cost solution can be chosen among the solutions [7]. The algorithm's drawback is that it takes up more space. In certain situations, when the target node is far away, it is slower.

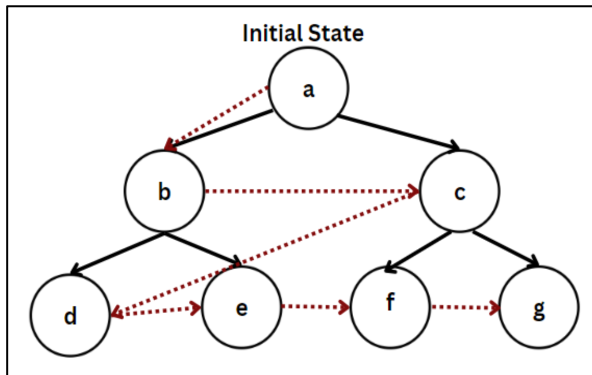


Fig. 10. The technique of BFS

Algorithm 1: The pseudocode of BFS

Input: Set of all nodes.
Step 1: Initialize the queue to track visited nodes.
Step 2: Add the initial node to the queue.
Step 3: While queue != empty:
  Step 3.1: Dequeue the initial node from the queue.
  Step 3.2: Mark the node as visited.
  Step 3.3: For each (unvisited neighbor of the node):
    Step 3.3.1: Add the neighbor node to the queue.
    Step 3.3.2: Mark the neighbor node as visited.
Step 4: The pseudocode terminates when the queue = = empty, indicating that all reachable nodes have been visited.
Output: Sequence of search path nodes.

Equations [1] and [2] below, where "b" is the branching factor and "d" is the depth of the search tree, illustrate the algorithm's space and time complexity, respectively.

$$S(n) = O(b^d) \quad [1]$$

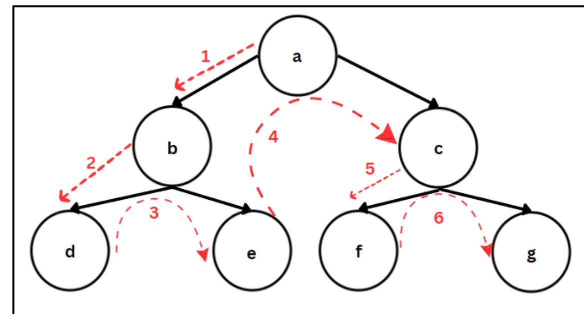
$$T(n) = O(bd) \quad [2]$$

Depth First Search (DFS): This algorithm is used to search and traverse tree or graph data structures. The algorithm starts at the initial node and travels down every branch as far as it can before backtrack, as shown by the numbered dotted arrow in Figure 11. It uses a stack data structure to store the nodes and use Last In First Out (LIFO) technique to traverse the nodes. It is a simple approach to understand and implement. In general, DFS has lower space complexity than BFS, especially for large graphs. It's possible that the DFS solution isn't optimal in terms of the shortest path. DFS may cause a stack overflow for large graphs if it is implemented using recursion.

Equations [3] and [4] illustrate the space and time complexity of the DFS algorithm respectively, where m stands for the maximum depth and b is the branching factor [7].

$$S(n) = O(b^m) \quad [3]$$

$$T(n) = O(b^m) \quad [4]$$



Figu. 11. Depth first search process

Algorithm 2: The pseudocode of DFS

Input: Set of all nodes.
Step 1: Initialize the stack to track visited nodes.
Step 2: Add the initial node to the stack.
Step 3: While stack != empty:
  Step 3.1: Pop the top node from the stack.
  Step 3.2: For each (unvisited neighbor of the popped node):
    Step 3.2.1: Mark the neighbor node as visited.
    Step 3.3.2: Push the neighbor node onto the stack.
Step 4: The pseudocode terminates when the stack = = empty, indicating that all reachable nodes have been visited.
Output: Sequence of search path nodes.

4.1 Dijkstra's Algorithm

Dijkstra's algorithm was created in 1956 by computer scientist Edsger Dijkstra. It is a pathfinding algorithm that find the shortest path between two nodes in a weighted graph [15]. The node with the least cost is chosen iteratively, and after investigating its neighbors, it modifies their cost in accordance with the overall distance from the starting node. This procedure keeps going until the algorithm visits every node and finds the shortest path, the pseudocode of Dijkstra'a algorithm is illustrated in Algorithm 3. Although Dijkstra's algorithm is a strong tool for finding the shortest paths in non-negative weighted graph, there are several applications where it should not be used due to its complexity and restrictions, especially when dealing with negative weights and cycles. The time complexity of Dijkstra's algorithm is shown in Equation [5].

$$T(V) = O((V+E)\log V) \quad [5]$$

In this approach, the cost to move from the starting node to the current node, "n," is denoted by g(n). The heuristic cost h(n) is the acceptable cost to reach the goal node from the current node. Equation [6] provides the formula for calculating the total cost, f(n).

$$f(n) = g(n) + h(n) \quad [6]$$

For the Dijkstra's algorithm, H(n) = 0. Hence the resulted equation is shown in equation (7):

$$f(n) = g(n) \quad [7]$$

Algorithm 3: The pseudocode of Dijkstra's Algorithm

Input: G = (V, E): the input graph; // V = vertices and E = edges
Step 1: Initialize cost of all nodes as Infinite;
Step 2: Create an empty priority queue Q; /* every item of Q is a pair (weight, node). Weight or cost is used as first item of pair to perform comparison between two pairs. */
Step 3: Insert start node to Q;
Step 4: cost[start node] = 0;
Step 5: while Q != empty:
  Step 5.1: u = minimum cost v from Q;
  Step 5.2: Loop through all adjacent of u and for every vertex of Q
  Step 5.3: if cost[v] > cost[u] + weight (u,v):
    Step 5.3.1: cost [v] = cost[u] + weight(u,v); // update the cost of v
    Step 5.3.1: Insert v into q; // even if v is already there
Step 6: Print distance array cost[]; // print all shortest path
Output: Shortest path

*4.2 A**

In a weighted graph, the A-star pathfinding algorithm effectively determines the shortest path between two nodes [16]. In 1986, Hart, Nilsson and Raphael published the algorithm. The aim of the algorithm is to find the optimum shortest path from the source node to the destination node [17]. It is an advanced version of Dijkstra's algorithm, as it uses the heuristic function h(n) to calculate the final cost f(n) required to reach the target node from the source node, as illustrated in Equation [8]. The heuristic function is computed either by the Manhattan distance, or Euclidean distance, or chebyshev distance.

$$f(n) = h(n) + g(n) \quad [8]$$

The algorithm is extensively utilized in many different industries, including network routing, robotics, video games, and navigation systems. The pseudocode of the algorithm is discussed in Algorithm 4.

The time complexity of A* algorithm depends on heuristic. Equations [9] and [10] illustrate the algorithm's space and time complexity, with b standing for the branch factor and d for the depth.

$$S(n) = O(b^d) \quad [9]$$

$$T(n) = O(b^d) \quad [10]$$

4.3 Greedy Best First Search (GBFS)

It is also a heuristic pathfinding algorithm that prioritizes traversing nodes that appear to be nearest to the goal node, depending on the heuristic function. Although it doesn't ensure that the optimal path will be found, in some applications, it can find a solution rapidly and effectively. The pseudocode of the algorithm is shown in the Algorithm 5, section of the article. The heuristic function of the algorithm is computed by considering only the cost require to reach goal node from the current node as shown by equation [11]. The space and time complexities of the algorithm is shown by Equation [12] and [13].

$$f(n) = h(n) \quad [11]$$

$$S(n) = O(b^d) \quad [12]$$

$$T(n) = O(b^d) \quad [13]$$

Algorithm 4: A* Algorithm

Input: Graph, Start node, Goal node and Heuristic function
Step 1: Create the following array:
  Step 1.1: openSet = {start};
  Step 1.2: cameFrom = { };
  Step 1.3: gScore = { }; // actual cost from start node to the present node
  Step 1.4: fScore = { }; // estimated total cost
Step 2: gScore[start] = 0;
Step 3: fScore[start] = heuristic (start, goal) + gScore(start);
Step 4: while openSet != empty:
  Step 4.1: current = the node with minimum fScore of openSet;
  Step 4.2: if current = = goal:
    Step 4.2.1: return Path(cameFrom, current);
  Step 4.3: remove current from openSet;
  Step 4.4: for each neighbour in neighbour(current, graph):
    Step 4.4.1: tentative_gScore = gScore[current] + cost(current, neighbour);
    Step 4.4.2: if ((neighbour !in openSet) || (tentative_gScore < gScore[neighbour])):
      Step 4.4.2.1: cameFrom[neighbour] = current;
      Step 4.4.2.2: gScore[neighbour] = tentative_gScore;
      Step4.4.2.3: fScore[neighbour] = gScore[neighbour] + heuristic(neighbour, goal);
      Step 4.4.2.4: **openSet** = neighbour;
Step 5: return "No path found";
Output: The shortest path between the goal node and the start node.

Algorithm 5: The pseudocode of GBFS

```
Input: Graph, Start node, Goal node, Heuristic function
Step 1: Start = Start node;
Step2: Goal = Target/ Goal node;
Step 3: Q = create a priority queue;
Step 4: enqueue(queue, start,0);          // Initial cost is 0
Step 5: visited ={};                      // keep the track of
        visited nodes
Step 6: while Q != empty:
    Step 6.1: current = dequeue(Q);
    Step 6.2: if (current = = Goal):
        Step 6.2.1: return reconstruct_path(visited, current);
    Step 6.3: mark_visited(visited, current);
    Step 6.4: for neighbour in neighbours(current):
        Step 6.4.1: if neighbour !in visited:
            Step 6.4.1.1: heuristic_cost =
                          heuristic(neighbour, Goal);  //
                          Calculate the cost to reach the
                          goal
            Step 6.4.1.2: enqueue(Q, neighbour,
                          heuristic_cost);
Step 7: return failure;   // Goal not found
Output: The shortest path, beginning at the start node and
ending at the goal node.
```

4.4 Space and Time complexities and Heuristic Function of pathfinding algorithms

Space complexity is the measure of memory or space needed for an algorithm to function. Similarly, "Time complexity" describes the length of time it takes for an algorithm to run. A heuristic function is a function that estimates the cost from the current node to the goal node. The space and time complexities and heuristic functions of BFS, DFS, Dijkstra's, A*, and GBFS algorithms are shown in Table 1. We can see that DFS and BFS have the same time complexity, but BFS requires more space than DFS. Dijkstra's algorithm has less time complexity compared to DFS and BFS. On the other hand, Dijkstra's algorithm requires more space than A*. The A* and GBFS algorithms are efficient with respect to time and space complexities. Heuristic functions are required by Dijkstra's algorithms, A* and GBFS. The Dijkstra algorithm computes the heuristic function by considering only the cost required to reach the current node from the start node; it doesn't consider the cost to reach the target node from the current node. GBFS, on the other hand, simply considers the cost of traveling from the current node to the goal node when computing the heuristic function. The cost to travel from the start node to the current node and the cost to travel from the current node to the goal node are both considered by the A* algorithm. In terms of heuristic functions and space and time complexity, the A* algorithm outperforms other pathfinding algorithms in terms of efficiency.

Table 1: Space and Time complexities and Heuristic function of pathfinding algorithms

Algorithms	Space omplexity	Time Complexity	Heuristic Function
BFS	$O(b^d)$	$O(b^d)$	-
DFS	$O(b^m)$	$O(b^m)$	-
Dijkstra's	O(V)	O((V+ E)log V)	-
A*	$O(b^d)$	$O(b^d)$	g(n) + h(n)
GBFS	$O(b^d)$	$O(b^d)$	h(n)

where:

- b is the branching factor.
- m is the maximum depth of the search tree.
- d is the depth of the shallowest solution.
- V is the number of nodes in the graph
- E is the number of edges in the graph.
- h(n) is the heuristic estimate of the cost to reach the goal from node n.
- g(n) is the cost to reach node n from the start node.

5. IMPLEMENTATION AND COMPARISON OF PATHFINDING ALGORITHMS

This section of the article, illustrate the implementation of BFS, DFS, Dijkstra, A* and GBFS algorithms on Unity 3D game engine. Regular 2D square grid was designed on Unity game engine. The elapse time is considered for comparing the algorithms. Elapse time is the amount of time needed by each algorithm to get from the source node to the goal node. The elapse time in seconds for four different scenarios of five algorithms is illustrated in Table 2. The columns in Table 2 depict the four scenarios based on the position of the goal and source nodes, such as vertical, horizontal, diagonal left and diagonal right in the graph. Across all node placements, it is observed that GBFS and A* have the fastest elapsed time, as depicted in Figure 12. With varying times based on node placement, Dijkstra's, DFS, and BFS have respectively slower elapsed time. Informed pathfinding algorithms like GBFS and A* employ heuristics to guide their search process, which frequently results in quicker pathfinding in different scenarios.

Table 2. Performance analysis of pathfinding algorithms without obstacles

Algo-rithms	Elapse Time when Source and Goal nodes are placed vertically (Seconds) **Scenario 1**	Elapse Time when Source and Goal nodes are placed horizontally (Seconds) **Scenario 2**	Elapse Time when Source and Goal nodes are placed diagonally (Left to Right) (Seconds) **Scenario 3**	Elapse Time when Source and Goal nodes are placed diagonally (Right to Left) (Seconds) **Scenario 4**
BFS	0.87	3.37	4.96	4.92
DFS	0.90	3.52	5.32	5.36
Dijkstra's	0.78	3.26	4.94	4.94
A*	**0.06**	**0.13**	4.68	4.61
GBFS	**0.06**	0.15	**0.21**	**0.20**

Table 3: Analysis of pathfinding algorithms' performance in the presence of obstacles.

Algorithms	Elapse Time when Source and Goal nodes are placed vertically (Seconds) **Scenario 5**	Elapse Time when Source and Goal nodes are placed horizontally (Seconds) **Scenario 6**	Elapse Time when Source and Goal nodes are placed diagonally (Left to Right) (Seconds) **Scenario 7**	Elapse Time when Source and Goal nodes are placed diagonally (Right to Left) (Seconds) **Scenario 8**
BFS	2.22	3.79	2.50	3.33
DFS	2.19	3.41	2.50	5.33
Dijkstra's	2.20	3.62	2.52	5.56
A*	0.77	2.83	0.84	2.46
GBFS	**0.25**	**0.24**	**0.27**	**0.37**

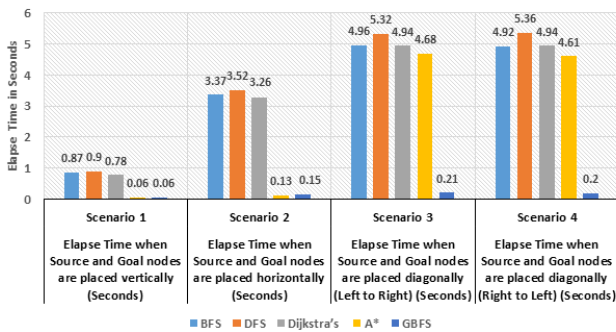


Fig. 12. Performance analysis of pathfinding algorithms without obstacles.

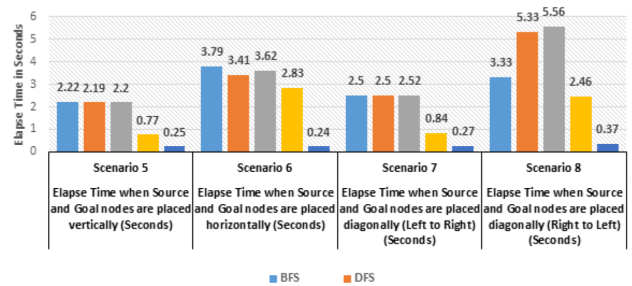


Fig. 13. Analysis of pathfinding algorithms' performance in the presence of obstacles.

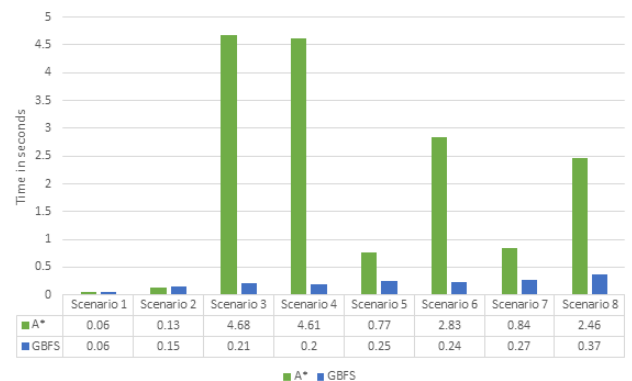


Fig. 14. Comparison of A* and GBFS pathfinding algorithms.

Table 3 represents the analysis of pathfinding algorithms when multiple obstacles are placed in different scenarios on the graph. Even in the face of obstacles, GBFS and A* retain their overall elapsed time, indicating that they can overcome difficulties. The algorithms show less sensitivity to node location and obstacles, perhaps due to their use of heuristic functions to traverse. Furthermore, the performance of Dijkstra's, DFS and BFS varies significantly, possibly because of variations in their handling and exploration of obstacles. Extended elapse times for Dijkstra's, DFS, and BFS can occur when obstacles are placed diagonally; this suggests that navigation through such obstacles may be challenging. Hence, when there are obstacles, the algorithm of choice should consider how well it can handle difficult scenarios with obstacles as shown in Figure 13. When obstacles are present, the effectiveness of heuristic algorithms like GBFS and A* is even more vital since they direct exploration around the hurdles.

Figure 14, present a bar graph for comparing A* and GBFS pathfinding algorithms. The elapse time in seconds, represented by the Y-axis of the graph. Different scenarios from 1 to 8 are depicted on the X-axis. Scenario 1 to 4 represent graphs without obstacles, as mentioned in Table 2, and scenarios 5 to 8 represent graphs with obstacles, as shown in Table 3. For easier visual comparison, the bars are categorized by color, with GBFS in orange and A* in blue. The elapsed time of GBFS, represented by the blue bar, is

nearly always less than the elapsed time of A*, represented by the green bar. This indicates that in these studied cases, GBFS typically identifies pathways faster than A*. The A* and GBFS algorithms have substantially different elapsed times in a few scenarios (e.g., Scenarios 5 and 8), suggesting that GBFS may have a greater advantage in those particular scenarios.

We may infer from Tables 2, 3, and Figure 14 that, on average, GBFS followed by A* was the fastest pathfinding algorithms—both with and without obstacles—that used the least amount of space and execution time. But GBFS isn't the optimal [7][18]. There is some condition when GBFS is not optimal as illustrates in the subsequent scenarios.

5.1 Case scenario 1:

Consider the graph shown in Figure 15 and the heuristic values of the node as shown in the Table 4 to compute the shortest path using GBFS algorithm as illustrated in Algorithm 5.

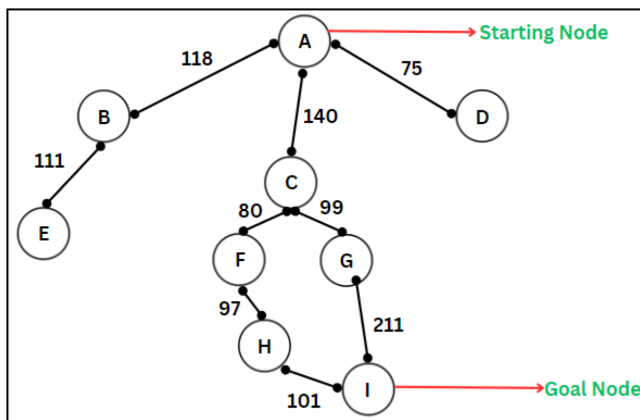


Fig. 15. Case Scenario 1.

Table 4: Heuristic values of all the nodes

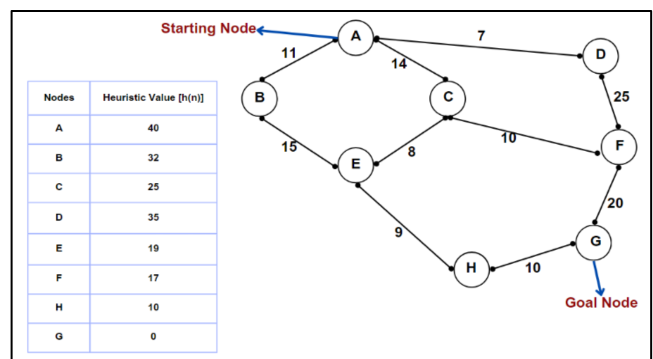
Node	H(n)
A	366
B	250
C	253
D	374
E	244
F	193
G	178
H	98
I	0

The starting node is A; to find the next node, we have to consider the adjacent node (B, C, or D) with the minimum heuristic cost. The heuristic cost of B is 250 less than the heuristic costs of C (253) and D (374); consider Table 4. So, node B is selected, and now the path is A->B. The adjacent nodes B are nodes A and E. The heuristic cost of node E is 244 less than the heuristic cost of node A (366). Hence, node E is selected, and the path is A->B->E. The only adjacent node in E is B, so it selects node B. Now the path will be A->B->E->B. Again, node E will be selected from node B, and the path will be A->B->E->B->E. Hence, the algorithm will run an infinite loop since node E is a dead node. In this condition, the GBFS algorithm will not be able to provide an optimal result.

5.2 Case Scenario 2:

The graph in Figure 16 consists of 9 nodes labeled A through H, with A as the starting node and H as the goal node. As seen in the Figure 14, each node has a heuristic value, which is the amount of cost needed to move from the current node to the goal node. The edges between the nodes represent the path, and the number next to the edges represents the actual cost of traversing those paths. The heuristic value is used by the GBFS algorithm to find an optimal path to reached the goal node from the start node. Considering algorithm 5, we start traversing from node A. The adjacent nodes of A are B, C, and D. The heuristic values of nodes B, C, and D are 32, 25, and 35, respectively. Since node C has a minimum heuristic value, node C is selected, and so the path is A->C. Then consider the adjacent nodes of C, that are A, E, and F. The heuristic values of A, E, and F are 40, 19, and 17, respectively. Node F has the minimum heuristic value, which is 17, so we choose F, A->C->F. The adjacent nodes of F are C, D, and G. The heuristic value of G is 0 since it is the goal node. Therefore, the G node will be selected. Hence, the path is A->C->F->G. The actual cost of the path is:

$$14 + 10 + 20 = 44 \quad [14]$$



Nodes	Heuristic Value [h(n)]
A	40
B	32
C	25
D	35
E	19
F	17
H	10
G	0

Fig. 16. Case Scenario 2.

Alternatively, if we choose the path A->C->E->H->G then the actual cost is:

$$14 + 8 + 9 + 10 = 41 \quad [15]$$

The result of equation [14] is greater than the result of equation [15]. Hence, in this scenario, the GBFS algorithm doesn't provide an optimal result.

5.3 Case Scenario 3:

The source and goal nodes are placed horizontally without any obstacles and analyze the elapse time of the A* and GBFS algorithms. The experiment on the Unity game engine was done repeatedly, 30 times. The Figure 17 depict that the A* algorithm has a lower elapsed time than GBFS for 19 times out of the 30 times. In this case, the probability of A* being faster than GBFS would be:

$$Probability = Favorable\ outcomes\ /\ Total\ outcomes \quad [16]$$

$$Probability = 19\ successes\ /\ 30\ trials \quad [17]$$

$$Probability = 0.63\ (approximately) \quad [18]$$

This means that in 63% of similar trials, A* is likely to be faster than GBFS. Hence, A* is a more optimal pathfinding algorithm than GBFS. Therefore, we can conclude that the A* algorithm is an optimal pathfinding algorithm.

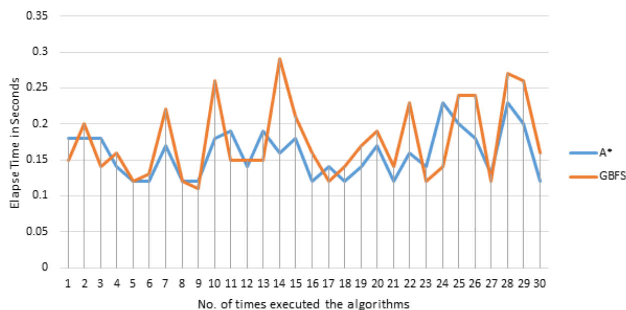


Fig. 17. Case Scenario 3.

6. DISCUSSION

In this work, various a pathfinding algorithm are analyzed to helps game developers reached the goal node in optimal time. This enables them to make informed decisions when selecting algorithms for game development projects that involve pathfinding. The algorithms have been classified based on various parameters and graphical representations to clearly distinguish their performance. In this paper, five different path finding algorithms are analyzed using the same set of inputs and examined their results.

Performance analysis was conducted using two distinct scenarios. The first one involved analyzing pathfinding algorithms on different grids without any obstacles. The second project focused on studying the same set of algorithms, but this time with obstacles included. Based on the results, it is clear that GBFS outperforms A* and other algorithms in terms of speed. However, the GBFS algorithm falls short of being optimal as it relies solely on heuristic values. As the paper explains, there are specific scenarios where the GBFS algorithm cannot accurately identify the most efficient shortest path. In contrast, the A* algorithm reliably discovers the most efficient shortest path. The algorithm determines the most efficient route by considering both the current cost and the estimated cost. In order to enhance the gameplay experience, researchers or developers can utilize A* algorithm to create intelligent movement for NPCs (Non-Player Characters) within the game environment.

7. CONCLUSION

In recent times, digital games have become a popular source of entertainment for people of all ages. Amidst the COVID pandemic, when individuals were confined to their homes, digital games seamlessly integrated into their lives, and this trend persists even after the pandemic. The era of digital games began in 1970 with the introduction of "Pong and Space War". These games were designed for two players and operated using discrete logic. It became necessary to create single-player digital games that incorporated NPC to enhance the game's realism. Players or users of the game do not have control over these NPC characters. Connecting the gaming industry with AI is the main idea behind NPC.

Pathfinding algorithms in AI can be defined as a set of instructions to obtain an efficient or optimal path starting from the start node to the goal node. Mainly, the algorithm is of two types: informed search and uninformed search. In this article, informed search algorithms A* and GBFS and uninformed search algorithms BFS, DFS, and Dijkstra's algorithms are implemented on the Unity game engine. In Unity, a regular square grid is designed in order to evaluate the pathfinding algorithms' performance. Elapse time is the parameter used to analyze the algorithms. The algorithms are implemented on the designed grid in Unity with and without obstacles to analyze their elapse time.

The outcome demonstrates that GBFS is the fastest algorithm when compared to A* and other algorithms. But the GBFS algorithm is not optimal since it works on heuristic values only. As the paper discusses, there are certain scenario in which the GBFS algorithm is unable to determine the optimal shortest path. For example, in case scenario 1, GBFS algorithm enter into infinite loop while searching for optimal path from start node to goal node. Similarly, in case scenario 2, the cost of the path from start node to goal node is more in case of GBFS algorithm compared to A* algorithm.

On the other hand, the A* algorithm consistently finds the optimal shortest path. The algorithm finds the shortest path depending on both the actual cost and the heuristic cost. So, researchers or developers can design NPCs using the A* algorithm to move the character intelligently throughout the game environment. The A* algorithm can also be used to design a robot's navigation, avoiding obstacles.

REFERENCES

[1] G. Singh, A. Mantri, O. Sharma, R. Dutta, and R. Kaur, "Evaluating the impact of the augmented reality learning environment on electronics laboratory skills of engineering students," *Computer Applications in Engineering Education*, vol. 27, no. 6, pp. 1361–1375, Nov. 2019, doi: 10.1002/CAE.22156.

[2] A. Gupta and S. Sirpal, "AI in Gaming and Entertainment," in *Applying AI-Based IoT Systems to Simulation-Based Information Retrieval*, IGI Global, 2023, pp. 63–76. doi: 10.4018/978-1-6684-5255-4.ch004.

[3] Java Tutorial Point, "Artificial Intelligence Tutorial," *javaTPoint*, 2011. https://www.javatpoint.com/artificial-intelligence-tutorial (accessed Aug. 08, 2022).

[4] S. Hotel, G. Sponsors, and S. Sponsors, "Green growth in GMS: Energy, environment and social issues," *The 8th International Conference 2013*, no. December, pp. 18–20, 2013.

[5] N. Q. Uoc, N. T. Duong, L. A. Son, and B. D. Thanh, "A Novel Automatic Detecting System for Cucumber Disease Based on the Convolution Neural Network Algorithm," *GMSARN International Journal*, vol. 16, no. 3, pp. 295–301, 2022.

[6] Q. H. Giap, T. T. Q. Nguyen, T. H. Trinh, and K. T. Nguyen, "Toward Enhancing Mid-Term Load Forecasting: RNN-Based Models vs Transformer-Based Models," *GMSARN International Journal*, vol. 19, no. 2, pp. 397–404, 2024.

[7] S. R. Lawande, G. Jasmine, J. Anbarasi, and L. I. Izhar, "A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games," *Applied Sciences (Switzerland)*, vol. 12, no. 11, 2022, doi: 10.3390/app12115499.

[8] G. Singh, A. Mantri, O. Sharma, and R. Kaur, "Virtual reality learning environment for enhancing electronics engineering laboratory experience," *Computer Applications in Engineering Education*, vol. 29, no. 1, pp. 229–243, Jan. 2021, doi: 10.1002/CAE.22333.

[9] P. Datta, A. Kaur, and A. Mantri, "Virtual Reality Based Training Simulator: A Bibliometric Analysis," *2023 International Conference on Disruptive Technologies, ICDT 2023*, pp. 666–671, 2023, doi: 10.1109/ICDT57929. 2023.10150887.

[10] J. Carsten, A. Rankin, D. Ferguson, and A. Stentz, "Global planning on the Mars Exploration Rovers: Software integration and surface testing," *Journal of Field Robotics*, vol. 26, no. 4, pp. 337–357, Apr. 2009, doi: 10.1002/rob.20287.

[11] Z. Abd Algfoor, M. S. Sunar, and H. Kolivand, "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games," *International Journal of Computer Games Technology*, vol. 2015, pp. 1–11, 2015, doi: 10.1155/2015/736138.

[12] M. Kallmann, "Navigation Queries from Triangular Meshes," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, Berlin, Heidelberg, 2010, pp. 230–241. doi: 10.1007/978-3-642-16958-8_22.

[13] N. Sturtevant, "A Sparse Grid Representation for Dynamic Three-Dimensional Worlds," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 7, no. 1, pp. 73–78, Oct. 2011, doi: 10.1609/aiide.v7i1.12438.

[14] R. Rahim, R. Dijaya, M. T. Multazam, A. D. Gs, and D. Sudrajat, "Puzzle game solving with breadth first search algorithm," *Journal of Physics: Conference Series*, vol. 1402, no. 6, pp. 1–5, 2019, doi: 10.1088/1742-6596/1402/6/066040.

[15] B. Anita Neeraj, V.; Abhishek, "A Review Paper On Examination Of Dijkstra's And A* Algorith To Find The Shortest Path," *Int. J. Creat. Res. Thoughts (IJCRT)*, vol. 6, pp. 635–641, 2018.

[16] R. N. Sarbini, I. Ahmad, R. O. Bura, and L. Simbolon, "Development of Pathfinding Using a-Star and D-Star Lite Algorithms in Video Game," *Journal of Theoretical and Applied Information Technology*, vol. 102, no. 3, pp. 832–841, 2024.

[17] A. S. Silvester Dian Handy Permana, Ketut Bayu Yogha Bintoro, Budi Arifitama, "Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game," *International Journal Of Information System & Technology*, vol. 1, no. 2, pp. 1–8, 2018.

[18] Y. Xie, Y. Chen, Z. Wang, and Y. Ou, "AI intelligent wayfinding based on Unreal Engine 4 static map," *Journal of Physics: Conference Series*, vol. 2253, no. 1, pp. 1–9, 2022, doi: 10.1088/1742-6596/2253/1/ 012016.