applied sciences

MDPI

Systematic Review

A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games

Sharmad Rajnish Lawande [1], Graceline Jasmine [1,*], Jani Anbarasi [1] and Lila Iznita Izhar [2,*]

[1] School of Computer Science and Engineering, Vellore Institute of Technology, Chennai 600127, India; sharmad123.lawande@gmail.com (S.R.L.); janianbarasi.l@vit.ac.in (J.A.)
[2] Department of Electrical and Electronics Engineering, Universiti Teknologi Petronas, Seri Iskandar 32610, Malaysia
* Correspondence: graceline.jasmine@vit.ac.in (G.J.); lila.izhar@utp.edu.my (L.I.I.)

Abstract: This paper provides a performance comparison of different pathfinding Algorithms used in video games. The Algorithms have been classified into three categories: informed, uninformed, and metaheuristic. Both a practical and a theoretical approach have been adopted in this paper. The practical approach involved the implementation of specific Algorithms such as Dijkstra's, A-star, Breadth First Search, and Greedy Best First. The comparison of these Algorithms is based on different criteria including execution time, total number of iterations, shortest path length, and grid size. For the theoretical approach, information was collected from various papers to compare other Algorithms with the implemented ones. The Unity game engine was used in implementing the Algorithms. The environment used was a two-dimensional grid system.

Keywords: pathfinding; Breadth First Search; A-star; Dijkstra's; Greedy Best First; Unity

Citation: Lawande, S.R.; Jasmine, G.; Anbarasi, J.; Izhar, L.I. A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games. *Appl. Sci.* **2022**, *12*, 5499. https://doi.org/10.3390/app12115499

Academic Editor: Giancarlo Mauri

Received: 25 March 2022
Accepted: 24 May 2022
Published: 28 May 2022

1. Introduction

The gaming industry continues to grow at a rapid pace due to the advancements in technologies, mainly in the field of artificial intelligence (AI). Currently, AI is the latest technological trend that is being used in these games for various enemy characters as well as other characters and actions. The main objective of using an AI system is to provide a challenging experience to the player in terms of decision-making or devising different strategies by enemy characters as well as helping the player to overcome the different hurdles present in the game. Enemy game-objects may use AI for developing a mechanism to outsmart the player and win the game. Special characters may use AI for providing support to the player in order to win the game.

Pathfinding refers to the concept of finding the optimal path from source node to destination node in the least time. Several Algorithms have been devised to the shortest path from source to destination by trying to avoid all the obstacles on the way. They may also use AI for pathfinding. These special character components used in games are called NPCs or non-player characters [1]. Even though significant progress has been made in the past few years towards pathfinding, there are many problems which constantly attract attention from researchers. One such problem is the demand for the high performance that these Algorithms need to satisfy in games. Moreover, since often they need to compute paths for multiple components and the resources allocated to these Algorithms are limited, there is a demand for Algorithms with high performance in less execution time. This paper summarizes the pathfinding Algorithms based on the performance so that a suitable Algorithm can be chosen for better optimization.

Pathfinding may be categorized into two groups: static and dynamic. Static refers to discovering the route globally in a static environment. Dynamic pathfinding on the other hand refers to finding the route locally in dynamic surroundings [2]. NPCs may use pathfinding AI Algorithms to reach a particular target from a start node by finding the

most feasible solution. Some of these pathfinding Algorithms may take various obstacles into consideration and some may not. These pathfinding Algorithms require high memory and processing power to find the most optimal solution by avoiding all obstacles in order to reach the destination node [3]. Therefore, a benchmark comparison was performed to analyze the performance of the pathfinding Algorithms by implementing them in an AI environment and monitoring their performance in Unity Software.

2. Path Planning

In order to develop an efficient path from the start to the end node, a proper in-depth analysis has to be performed so that all the collisions or obstacles can be safely avoided while determining the least cost path in least time [4]. Considering factors such as collision avoidance, performance optimization, and static and dynamic constraints, several types of Algorithms have been developed to find an efficient path between two points [5]. Graph pathfinding Algorithms were created with the goal of discovering the shortest path between two vertices for a connect graph [6]. The main objective of these Algorithms was to only get the least distance within a graph without taking into consideration the performance optimization to reduce execution time and avoid obstacles [7]. These Algorithms were mostly used in networking the field particularly for developing routing protocols [8].

Search Algorithms are another class of Algorithms that are mostly used in video games as well as in robotics to reach a particular target from the beginning [9]. These Algorithms cover the whole map of a given area and try to search for all the possible paths from source to destination and out of those, they select the shortest path. There are two types of search Algorithms, uninformed and informed search Algorithms. In the uninformed search Algorithms, which are also known as blind search, there is no information regarding the number of steps or the costs of path from current state to the goal. Some examples include the Breadth First Search and Depth First Search Algorithms. Another example of uninformed Algorithms is the Dijkstra's Algorithm which can be also used in graphs to find the shortest path but only for positive costs [10]. An improved version of the Algorithm was developed later on called the Bellman–Ford Algorithm which was for both positive as well as negative costs and is an uninformed Algorithm [11]. Another uninformed Algorithm is the Floyd–Warshall Algorithm which was inspired from these two to find the minimum path for positive and negative costs but the number of nodes to be visited was reduced in comparison with Dijkstra's Algorithm [12].

Informed search Algorithms are better optimized as compared to uninformed search Algorithms. The unique feature of these Algorithms is that they make use of a heuristic function to calculate the efficient path [13]. A search method or a heuristic function is informed if it uses additional information about nodes that have not yet been explored to decide which nodes to examine yet [14]. Heuristic function tries to search for a solution that is most near to the best solution in the shortest time [15]. The entire area does not have to be checked by limiting the Algorithm to a specific area, thereby saving significant amount of time. Some examples include A*, D*, HPA*, etc. [16].

Deterministic, non-deterministic, and procedural generation are the various forms of game AI. This paper focuses on the various deterministic AI-based gaming Algorithms. This deterministic AI can be implemented using several tools like Behavior Tree, Decision Tree, etc. Based on stationary or in motion obstacles, it can be further divided into two categories, static and dynamic pathfinding. Static pathfinding can be categorized into uninformed search, informed search, and metaheuristic Algorithm. Various pathfinding Algorithms that have been analyzed and evaluated are shown in Figure 1. This work extensively makes use of the pathfinding-based Algorithms to analyze the efficiency of the model for the game play grid environment with and without obstacles.

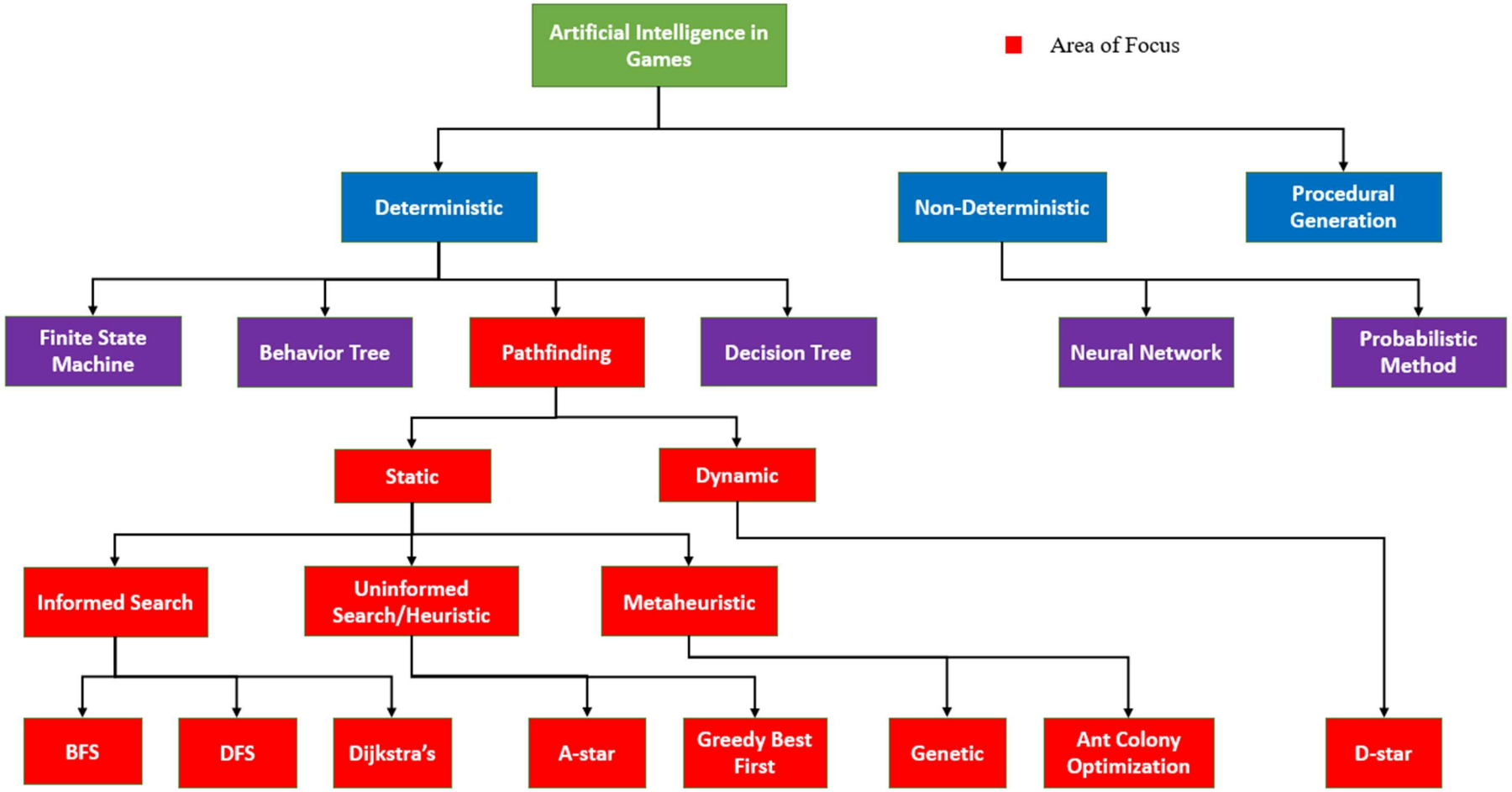


Figure 1. Classification of intelligence-based pathfinding Algorithms in gaming.

2.1. Pathfinding Using Grids

A grid is a connection or a network of vertices or points through edges in order to form a graph (Figure 2). The performance of the pathfinding Algorithms is determined by attributes of the graph that is formed by the grid. The grids can be regular or irregular in pattern. Regular patterns can be of triangular, hexagonal, square, or cubic falling under 2D or 3D categories. Waypoints, mesh navigation, and visibility graph come under irregular patterns of grid system. In this paper, pathfinding Algorithms in gaming have been experimented and analyzed on a 2D Square Grid (Octile) system.

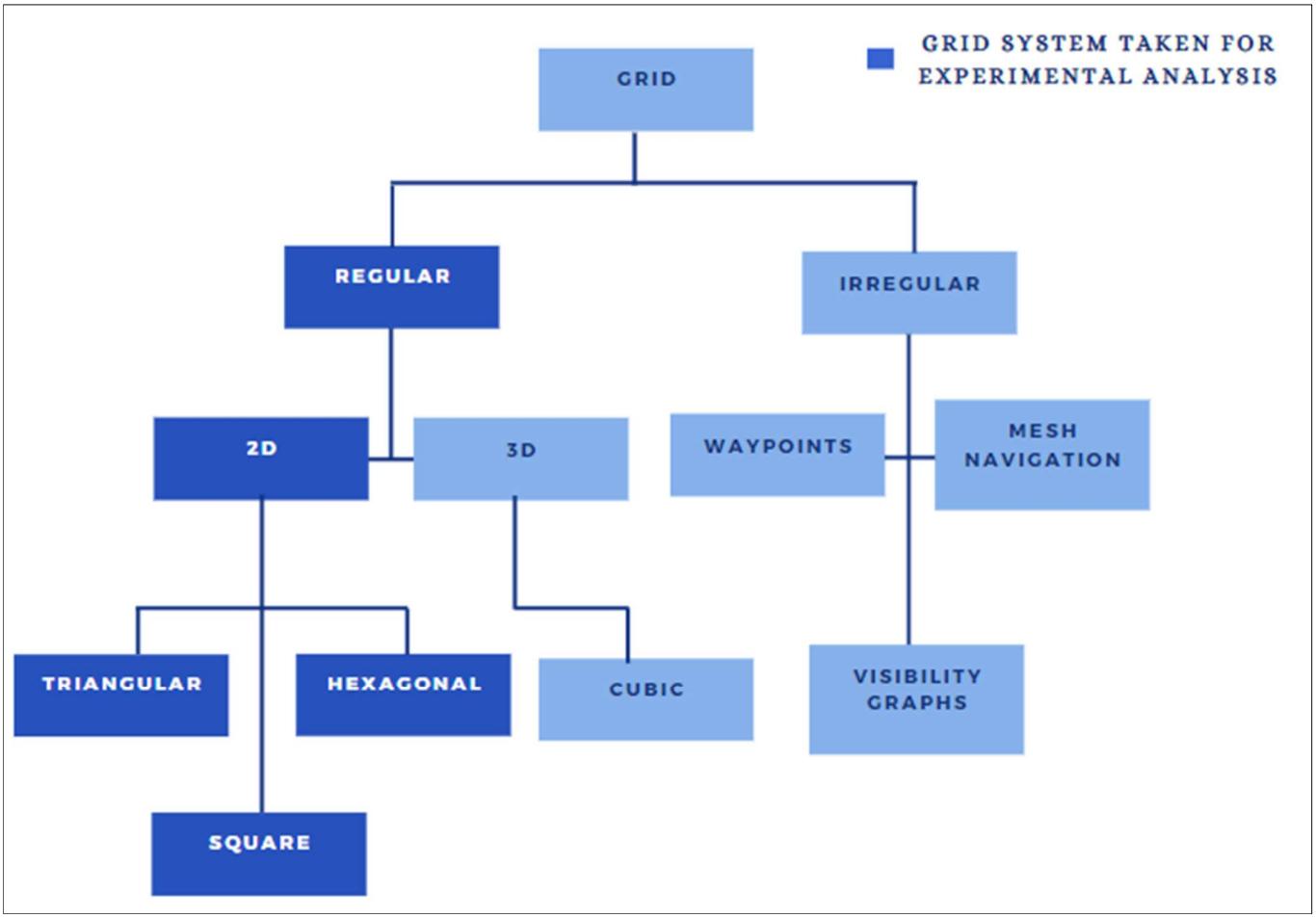


Figure 2. Hierarchical classification for pathfinding using grids.

2.1.1. Regular Grids

Regular grids are most extensively used in the field of pathfinding by developers since they provide uniformity to the environment. These are usually represented in 2-dimenional or 3-dimensional Euclidean space in the form of a tessellation. Every smallest unit of these grids have a definite uniform shape of equal sides such as a triangle, square, hexagon, or a cube. On the basis of these shapes, regular grids are further classified into triangular, octile, hexagonal, and cubic as shown in Figure 3.

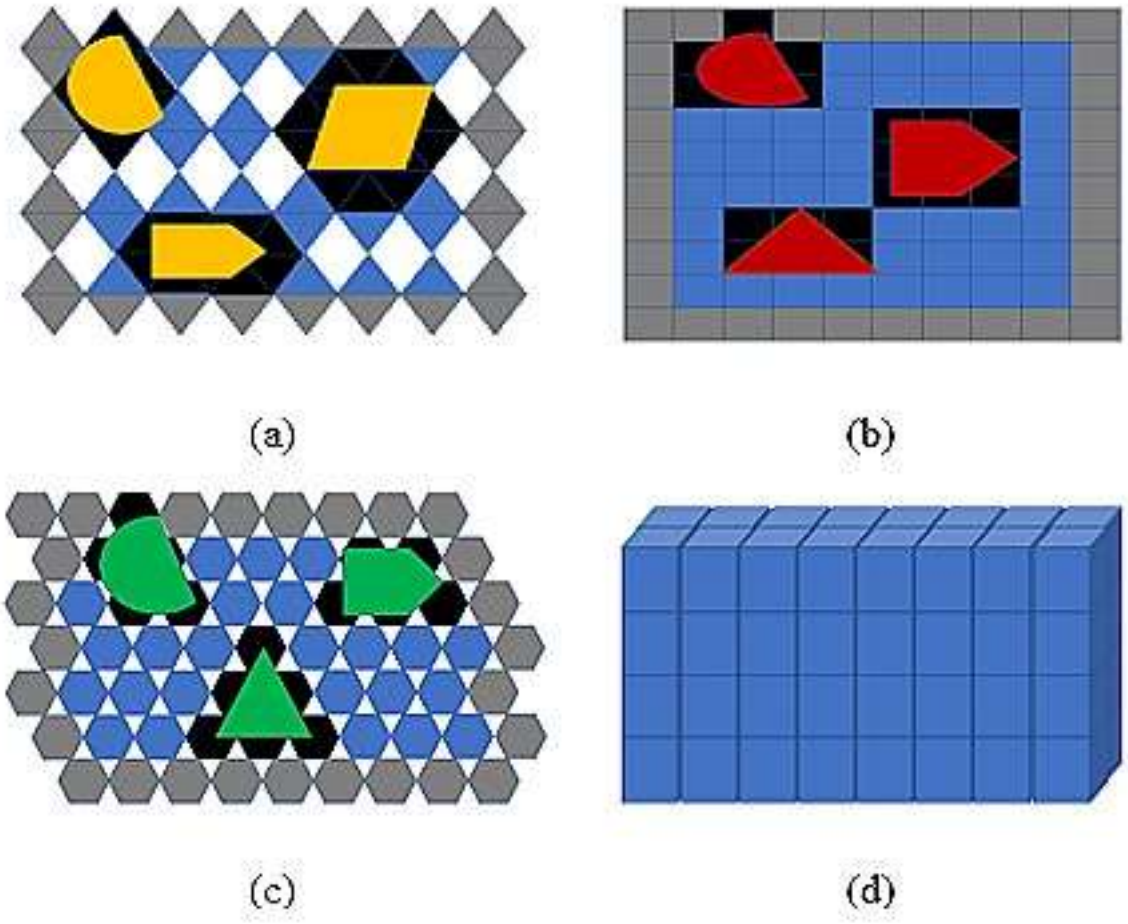


Figure 3. (**a**) Top left: 2D triangular grid with obstacles, (**b**) top right: 2D square (octile) grid with obstacles, (**c**) bottom left: 2D hexagonal grid with obstacles, (**d**) bottom right: 3D cubic grid without obstacles.

1. 2D triangular grid: These grids are the least popular when it comes to pathfinding in games. As compared to hexagonal and octile grids, their usage is less, but these grids have some desirable properties. Here, the smallest unit is an equilateral triangle connected using one side and vertices. Figure 3a shows a 2D triangular grid with three obstacles. The gray triangles represent the border area of the grid whereas the blue triangles are the ones with no obstacles present. The area covered with black triangles shows the region covered by the yellow obstacles.
2. 2D square (octile) grid: These are the most widely used grids to represent graphs in the gaming industry particularly for pathfinding as well as in the field of robotics. This grid system is the most researched system for pathfinding as several pathfinding Algorithms have been proposed and tested using this grid system. Figure 3b shows a 2D octile grid with three obstacles. The gray squares represent the border area of the grid whereas the blue squares are the ones with no obstacles present. The area covered with black squares shows the region covered by the red obstacles.
3. 2D hexagonal grid: Some of the properties of these grids are similar to the 2D square grids. The graphs formed using this grid system requires less search time and memory complexities as compared to those formed from the octile system. Figure 3c shows a 2D hexagonal grid with three obstacles. The gray hexagons represent the border area of the grid whereas the blue hexagons are the ones with no obstacles present. The area covered with black hexagons shows the region covered by the green obstacles [17].
4. 3D cubic grid: Apart from the other grid systems which used the 2D environment, the 3D Cubic grid shown in Figure 3d is based on a continuous 3-dimensional environ-

ment. Not much study has been done on this grid system for pathfinding, but it can be applied in other fields such as image processing and computer graphics [18].

2.1.2. Irregular Grids

Irregular grids are not formed from the smallest unit shape unlike the regular grids. On the basis of its pattern, it is classified into waypoints, visibility graphs, and mesh navigation as shown in Figure 4. They also have a number of applications in the fields of pathfinding and robotics. They are of three types, as follows.

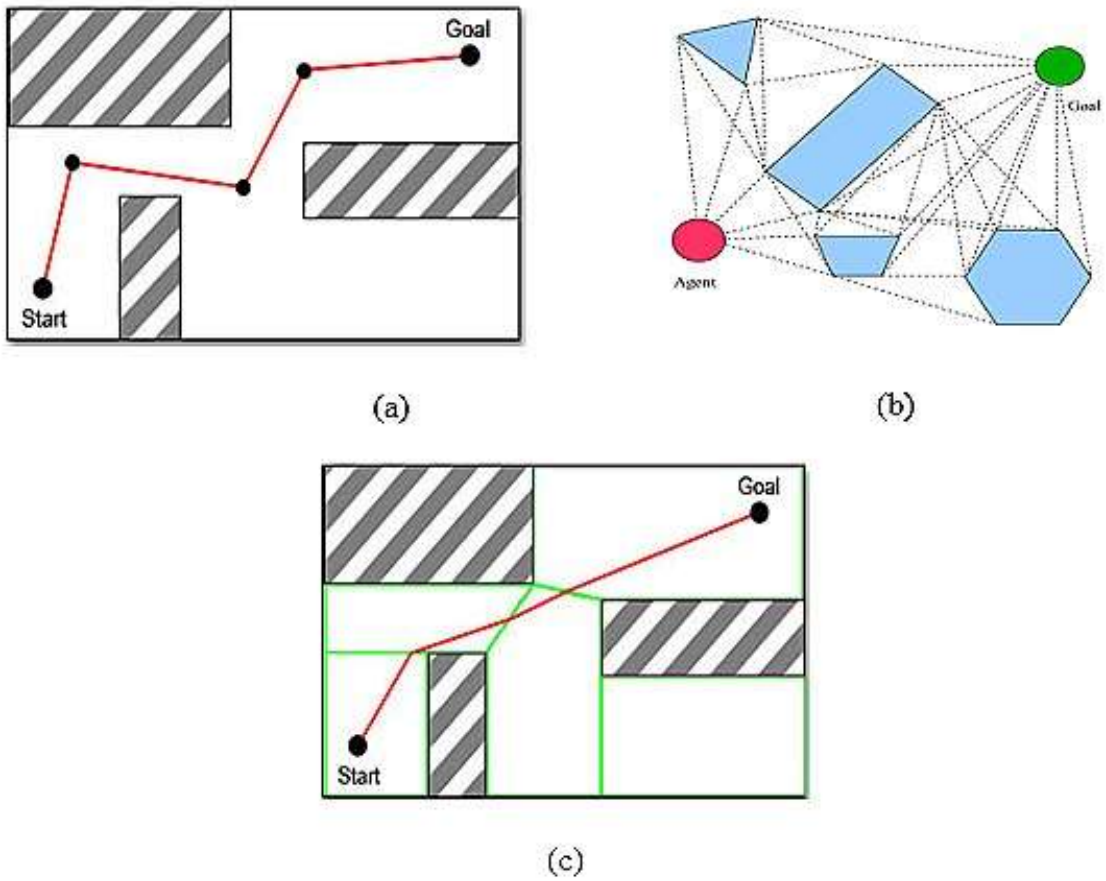


Figure 4. (**a**) Top left: waypoints, (**b**) top right: visibility graph, (**c**) bottom: navigation mesh.

1. Waypoints: Waypoints have a number of applications mainly in robotics as well as pathfinding. As the name suggests, waypoints are navigation guides or markers that direct the Algorithm to move in a particular direction of shortest path.
2. Visibility graphs: In the fields of computational geometry and graph theory, visibility graphs are fundamental structures. They have applications in several other fields and in recent times have been applied to Euclidean distance calculations for shortest paths when obstacles are present [17].
3. Mesh navigation: A mesh can be formed from different shapes, commonly triangles and other polygons. Mesh graphs are somewhat identical to the visibility graphs in terms of looks. However, the complexity of visibility graphs is greater than the mesh graphs. Most of the applications of this graph are present in games [18].

The different types and its categories of grid along with its smallest unit such as equilateral triangle, square, regular hexagon, cube, navigation marker, graph node, and edge and its applications are detailed in Table 1.

Table 1. Grid classification based on grid type, grid dimension, smallest unit, and applications.

Grid	Grid Type	Grid Dimension	Smallest Unit	Applications	Ref.	Year
Triangular	Regular	2D	Equilateral Triangle	Computer Graphics, Image Processing	[18]	2015
Square	Regular	2D	Square	Video Games, Robotics	[5,9]	2013, 2015
Hexagonal	Regular	2D	Regular Hexagon	Video Games, Robotics	[17,18]	2015, 2018
Cubic	Regular	3D	Cube	Computer Graphics, Image Processing, Robotics	[18]	2015
Waypoint	Irregular	-	Navigation Marker	Video Games, Robotics	[9,17]	2015, 2018
Visibility Graph	Irregular	-	Graph Node	Games, Computational Geometry	[5,18]	2013, 2015
Mesh Navigation	Irregular	-	Edge	Video Games	[18,19]	2016, 2015

3. Existing Pathfinding Techniques

Graph pathfinding Algorithms were created with the goal of discovering the shortest path between two vertices for a connected graph. The main objective of these Algorithms was to only get the least distance within a graph without taking into consideration the performance optimization to reduce execution time and avoid obstacles. These Algorithms were mostly used in networking fields, particularly for developing routing protocols.

3.1. Uninformed Search Algorithms

None of the Algorithms present in this category possess any extra information related to the destination node apart from the one provided though problem definition. All the Algorithms present in this category work on the concept of blind search, i.e., they try to reach the goal using brute force by not knowing anything about the search space.

3.1.1. Breadth First Search Algorithm (BFS)

BFS was first published in 1959 by Edward Moore and is one of the fundamental Algorithms used in games for pathfinding [20]. A popular use of this Algorithm is to find the shortest path in a grid or a maze [21]. Some of the real-world uses of this Algorithm include GPS tracking, analysis in networks and graphs, and search engines [22]. In BFS, the cells are visited one at a time. Therefore, the cells which are only a single step away from the start cell are visited first, then again, all cells which are two steps away from the start cell are visited, and this continues until all cells are visited [23]. Through BFS, we can find the shortest path from start cell to end cell with minimum number of steps traversed as shown in Figure 5. The main advantage this Algorithm has is that the solution will always be found, no matter the type of problem. If there are multiple solutions, all the solutions will be found by the BFS Algorithm and the minimum cost solution will be selected from all these solutions by the Algorithm. The disadvantage this Algorithm has is its relation to memory usage. All the nodes are stored in the tree by this Algorithm and every node will be tested on level 'n' to get a solution to level 'n + 1' [24]. The time T(n) and space S(n) complexity of the search Algorithm is given in Equations (1) and (2) where **s** refers to the depth of shallowest solution and $\mathbf{n^i}$ refers to the number of nodes in level 'i'.

$$T(n) = 1 + n^2 + n^3 + \ldots \ldots + n^s = O(n^s) \quad (1)$$

$$S(n) = O(n^s) \quad (2)$$

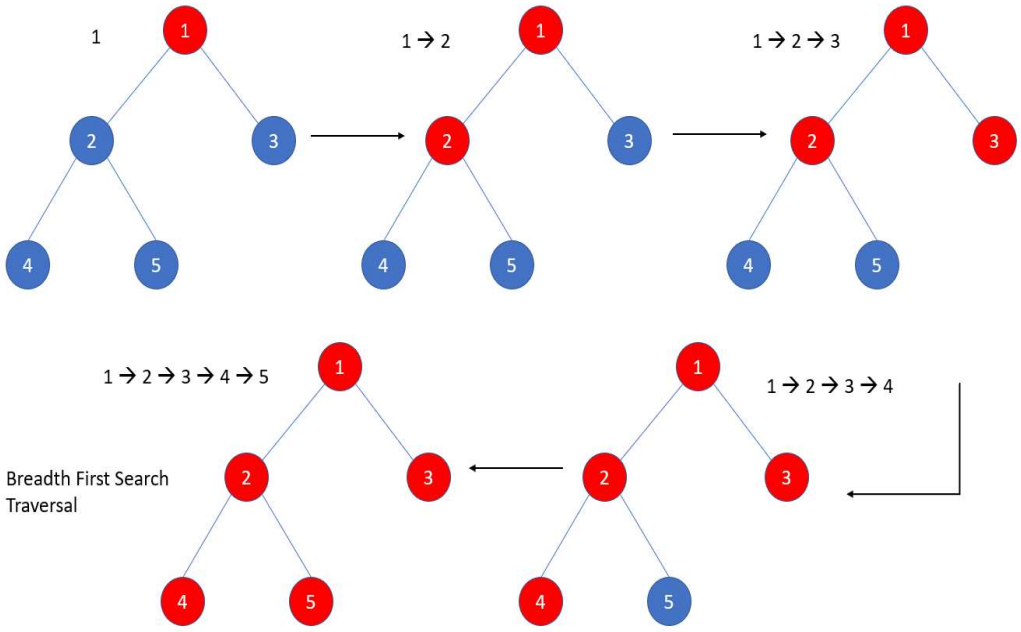


Figure 5. Breadth first search traversal.

In Figure 5, the BFS Algorithm will traverse the vertices in breadth-wise fashion from the top node in the graph. First, vertex 1 will be visited since it is at the top and put in queue. Next, the child nodes of vertex 1, which are vertices 2 and 3, will be visited as marked in red and put in the queue. The Algorithm 1 will then visit vertex 4 since there are no child nodes in vertex 3 and finally vertex 5.

Algorithm 1 Breadth First Search Algorithm.

Input: Set of all vertices.
Output: Search Path sequence vertices **S_BFS[]**
Step 1: Set **s** as the source vertex, Stack, **BF[]** = **Ø** and **S_BFS[]** = **Ø**
where **S_BFS[]** is an array having all the visited vertices and **BF[]** is a queue to store the vertices for processing
Step 2: Push the vertex **s** in Queue **BF[]**
Step 3: Until **BF[]** is empty, iterate the steps 4 to 7
Step 4: **q** = Pop **BF[]**
Step 5: Mark the vertex **q** as visited and place it in the array **S_BFS[]**
Step 6: If **q** is the goal, then finish the searching process
Step 7: Else push one adjacent vertex of **q** in **BF[]**
Step 8: Repeat steps 3 to 6 until all the vertices are in **S_BFS[]**

3.1.2. Depth First Search Algorithm (DFS)

This search strategy explores the deepest node first, then backtracks to explore other nodes. It uses LIFO (Last In First Out) order, which is based on the stack, in order to expand the unexpanded nodes in the search tree. The search proceeds to the deepest level of the tree where it has no successors. This search expands nodes till infinity, i.e., the depth of the tree [25]. One of the few advantages of this Algorithm is that much less memory is required since only the path from root to current node has to be kept inside the stack. As compared to BFS it requires less time in order to reach its destination as shown in Figure 6. One disadvantage is that not all solutions will be found in case of multiple solutions unlike the BFS Algorithm. In addition, as the Algorithm traverses in depth-wise fashion, there is a possibility of an infinite loop. If the nodes that are visited in DFS are not marked while traversing, the same nodes will be visited more than once, thereby ending up in an infinite

loop. The time T(n) and space S(n) complexity is given in Equations (3) and (4) where **d** refers to the depth of the search tree and $\mathbf{n^i}$ refers to number of nodes in level i.

$$T(n) = 1 + n^2 + n^3 + \ldots\ \ldots + n^d = O(n^d) \quad (3)$$

$$S(n) = O(n \times d) \quad (4)$$

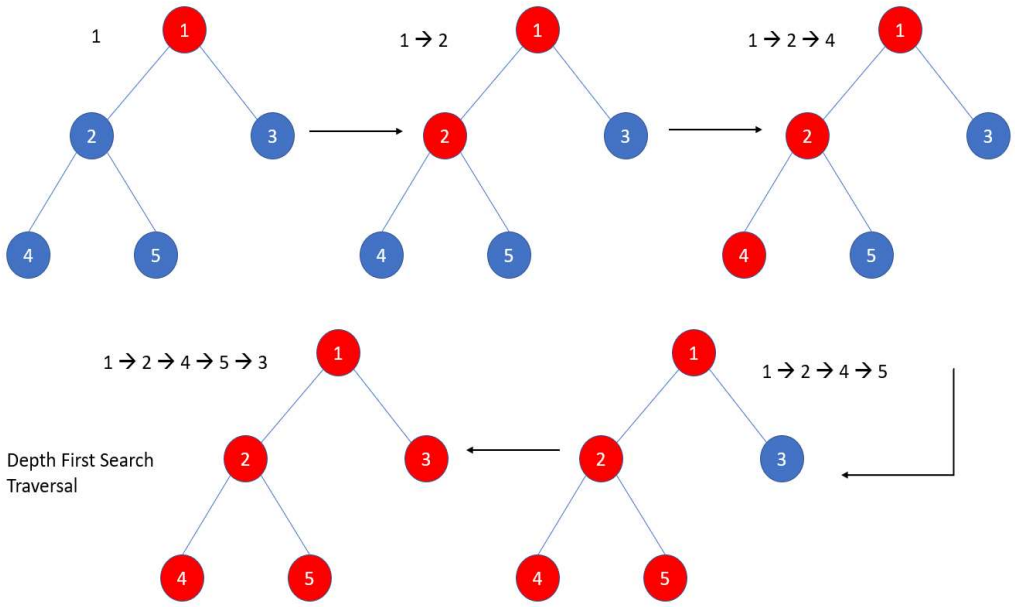


Figure 6. Depth first search traversal.

In Figure 6, the DFS Algorithm will traverse the vertices in depth-wise fashion from the top node in the graph. First, vertex 1 will be visited since it is at the top and put in stack. Next, the first child node of vertex 1, which is vertex 2 will be visited as marked in red and put in stack respectively. The Algorithm will then visit vertex 4 since the left most node is visited first in DFS. Then, vertex 5 will be visited which is the remaining node of vertex 2. Finally, vertex 3 will be visited, since it is the remaining node of vertex 1. The path is returned once all vertex elements are popped out of the stack.

3.1.3. Dijkstra's Algorithm

First introduced in 1956 by Edsger Dijkstra and published three years later, this conventional Algorithm is used to find the shortest distance between start node and end node in a graph [26]. It is used in a variety of applications such as digital mapping services such as Google Maps, telephone networks, IP routing for finding Open Shortest Path First, robotics, designate file server, etc. [27]. It was also used in traffic information systems in order to track source and destinations from another specific source and destinations [28]. This Algorithm was preferred earlier in video games for finding the optimum path until the A-star Algorithm which could find the solution more quickly [29]. There is no heuristic function used in this Algorithm, but it is extended to the A-star Algorithm by making use of the heuristic function [30]. This Algorithm requires a lot of memory usage as all nodes are expanded in order to find the destination node unlike the case of A-star which improves its functionality [31]. One of the drawbacks of this Algorithm is that it cannot be used for negative value costs, thereby leading to acyclic graphs and therefore the correct shortest path cannot be found [32,33]. For each node in a graph, a label is assigned which is used to determine the minimal length from the start point to all the other nodes of the graph [34]. After each step, the value of the label of the graph nodes decreases; that is, the Algorithm runs sequentially [35].

For Dijkstra's Algorithm, G(n) which is the cost required to move from start node to present node 'n', and heuristic value H(n), which is the acceptable cost to move from present node to target node, are assigned to 0 for Dijkstra's Algorithm 2. It cannot be overestimated and is given in Equation (5). The total cost required to reach the target node is:

$$F(n) = H(n) + G(n) \quad (5)$$

therefore, resulting in F(n) = G(n).

In Figure 7, the start vertex is taken as A. Vertices D and B are connected to vertex A but minimum distance is from vertex D which is 1. From vertex D, B and E are connected but total distance from A to D to E is 2 which is less than from A to D to B which is 3. Therefore, vertex E is visited. From E, only B or C can be chosen since D is already visited, but total distance from A to D to E to C is 7 which is less than one from A to D to E to B to C which is 9. Therefore, the shortest path will be A to D to E to C with minimum cost of 7.

Algorithm 2 Dijkstra's Algorithm.

Input: Set of all vertices.
Output: Search Path sequence vertices **S_DJK[]**
Step 1: Set **d[s] = 0, S_DJK[] = ϕØ,** where **s** is the source vertex and **S_DJK[]** is an array having all the visited vertices
Step 2: For all vertices v except **s,** set **d[v]= ∞**
Step 3: Find **q** not in **S_DJK[]** such that **d[q]** is minimum
Step 4: Add **q** to **S_DJK[]**, such that **S_DJK[]** has now been visited
Step 5: Update **d[r] = min(dist[r], dist[q] + cost[q][r])**, for all **r** adjacent to **q** such that **r** is not in **S_DJK[]**
Step 6: Repeat Steps **3** to **5** until all the nodes are in **S_DJK[]**
Step 7: Retrieve the array **S_DJK[]**, having shortest path from the source vertex **s** to all other vertices

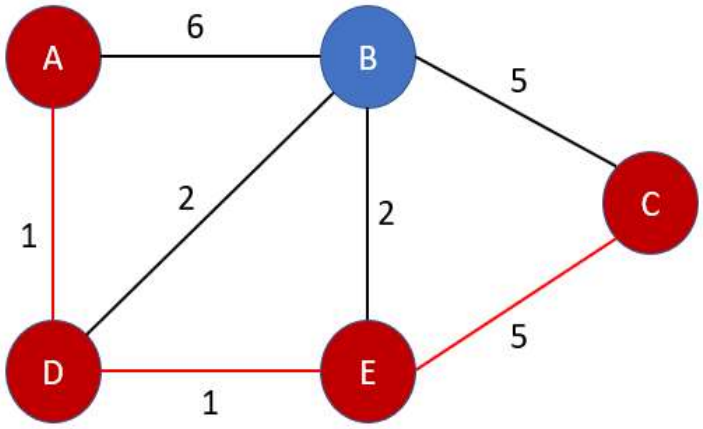


Figure 7. Shortest path using Dijkstra's Algorithm.

3.2. Informed Search or Heuristic Approach Algorithms

Heuristic Algorithms are basically optimized techniques that are used to find the solution to a particular problem more speedily and efficiently as compared to the basic conventional Algorithms [36]. Time is the main focus of these Algorithms. Parameters such as accuracy and cost may be compromised in order to find the solution in the least time [37]. These Algorithms basically make use of a heuristic function which is used to predict the closeness of the final destination with respect to the current position [38]. A heuristic function helps in reducing the memory usage by searching for only the promising node and leaving all other nodes that are not required [39]. The way a heuristic function works is that it computes the cost from a specific node to destination node and if this cost is close to the actual cost value, then that node is selected for further expansion while others are not expanded [40]. This heuristic functionality helps in improving the speed and efficiency of finding the path [41]. While using a heuristic approach, when the actual cost is

a bit overestimated, the most promising and optimal path is found by the Algorithm in less time [42].

3.2.1. A-Star Algorithm

This is one of the most famous Algorithms used in the gaming industry due to its simplicity [43]. Published in 1986 by Hart, Nilsson, and Raphael, the main objective of this Algorithm is to search for the most efficient solution from start node to the end node [44]. Video games belonging to various categories, such as racing, real-time strategy, RPG (role-playing game), etc., generally make use of this Algorithm due to its accuracy and high efficiency [45]. This Algorithm is an advanced version of Dijkstra's Algorithm since it makes use of the heuristic function to predict the shortest path [46]. In the case of Dijkstra's Algorithm, the value of the heuristic function is zero thereby providing an assurance of finding the shortest route [47]. The Algorithm basically makes use of three parameters to determine the best possible route to reach the destination where G(n) is the cost required to move from start node to present node 'n' and H(n) is the heuristic value which is the acceptable cost to move from present node to target node. It cannot be overestimated. The cost required to reach the target node is given in Equation (6).

$$F(n) = H(n) + G(n) \quad (6)$$

Figure 8 shows a sample grid in which A* Algorithm 3 avoids the obstacles and reached the destination (marked in red) in the shortest possible path from the source (marked in blue). The Algorithm first begins at (8,8) and reaches (4,2). Next, it chooses (3,3) instead of (4,3) since it is shorter. Similarly, at (3,3) it takes (2,2) instead of (2,3) and finally reaches the destination of (1,1).

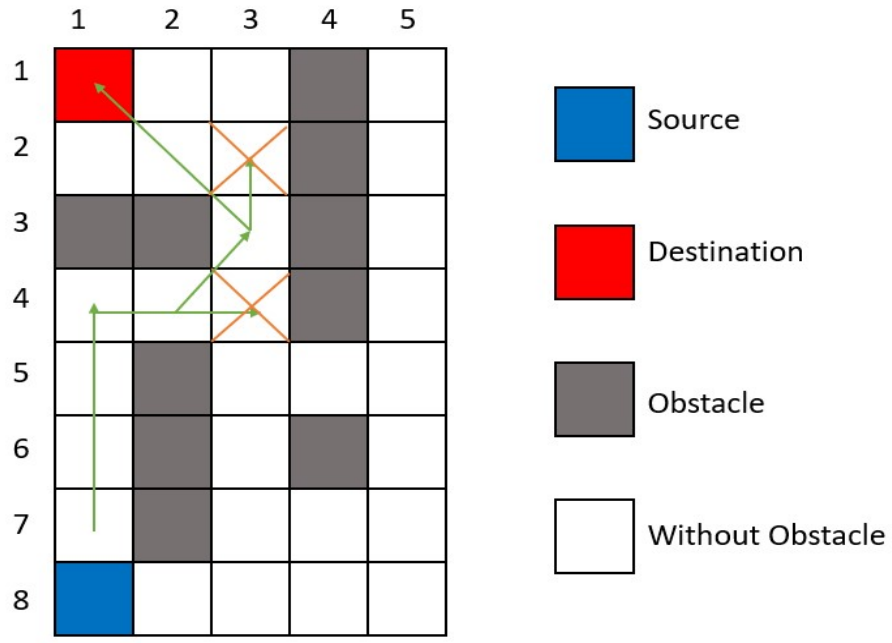


Figure 8. A* Algorithm goes from (4,2) to (3,3) and not (4,3) and similarly from (3,3) to (2,2) and not (2,3).

Algorithm 3 A* Algorithm.

Input: Set of all vertices.
Output: Search Path sequence vertices **S_ASR[]**
Step 1: Set **d[s] = 0, S_ASR[] = ϕ**, where **s** is the source vertex and **S_ASR[]** is an array having all the visited vertices
Step 2: For all vertices **v** except **s,** set **d[v] = ∞**
Step 3: Find **q** not in **S_ASR[]** such that **d[q]** is minimum; where **d[q]** is determined by cost function **f(q)** as follows:
f(q) = g(q) + h(q), for all **q** adjacent to **s**
where **g(q)** is the actual cost from node **q** to the initial node, and **h(q)** is the cost of the optimal path from the target node to **q**.
Step 4: If vertex **q** is a goal point, return success and exit.
Step 5: If any adjacent vertices of **q** is the goal point, then return success & retrieve the arrived shortest path
Step 6: Repeat Steps 3 to 5 until all the vertices are in **S_ASR**

3.2.2. Greedy Best First Algorithm

The Greedy Best First Search is another popular heuristic Algorithm that always has a track of a border in order to find the goal node. As the name suggests, the Algorithm 4 will always try to pick the best track at that particular moment. This Algorithm makes use of the concepts of both breadth first search and depth first search Algorithms in order to achieve more efficiency [48]. Using this concept, the most promising node can be chosen. The node which is nearest to the destination node can then be expanded and an estimate cost is made using the heuristic function [49]. For Greedy Best First Search: G(n) = 0 and the total cost required to reach the target node is: F(n) = H(n) + G(n) resulting in F(n) = H(n). Considering the Search Path Sequence vertices to be **S_GBF[]**, the Algorithm is same as above A* Algorithm with **g(q) = 0**.

Algorithm 4 Greedy Best First Search Algorithm.

Step 1: Place the starting node into the **OPEN** list.
Step 2: If the **OPEN** list is empty, Stop and return failure.
Step 3: Remove the node n, from the OPEN list which has the lowest value of **h(n)**, and places it in the **CLOSED** list
Step 4: Expand the node **n**, and generate the successors of node **n**.
Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
Step 6: For each successor node, Algorithm checks for evaluation function f(n), and then check if the node has Been in either **OPEN** or **CLOSED** list. If the node has not been in both list, then add it to the **OPEN** list.
Step 7: Return to Step 2

3.2.3. D-Star Lite Algorithm (Dynamic A*)

Found in 1994 by Anthony Stenz, the D* or "Dynamic A*" Algorithm is an incremental heuristic search Algorithm that can be used for finding the shortest path in situations where the environment maybe unknown or partially unknown. Earlier, most of the pathfinding Algorithms were developed in ways where the environment model was considered to be accurate and complete [50]. However, in the real world, there exists many situations where the environment would change suddenly without any warnings or there exists information which may be incomplete or even non-existent [51].

Machines which are automated that have the ability to operate in unknown or partially unknown environment, for example, planetary exploration robot or character in video games, require a method that has the capability of fast and efficient re-planning helping them to move more intelligently in a time of critical situations or difficult terrains [52]. Similar to the A* Algorithm, the D* Algorithm 5 creates a list of nodes for evaluation termed "OPEN list". There are several different states of nodes such as NEW, RAISE, LOWER, OPEN, and CLOSED [53]. The NEW state refers to the fact that the current node was never kept in the open list. OPEN state means that the node is in the open list [54]. The CLOSED

state means it is no longer kept in the open list [55]. The RAISE state refers to the fact that the cost of the node was higher than the previous time it was kept on the open list [56]. Similarly, LOWER state means that the cost of the node was lower than the previous time it was kept on the open list [57].

Algorithm 5 D* Algorithm.

Input: Set of all vertices.
Output: Search Path sequence vertices **S_DSR[]**
Step 1: Set **d[s] = 0, S_DSR[] = ϕ,** where **s** is the source vertex and **S_DSR** is an array having all the visited vertices
Step 2: For all vertices v except **s,** set **d[v] = ∞**
Step 3: Find **q** not in **S_DSR** such that **d[q]** is minimum; where **d[q]** is determined by function **f(q)** as follows:

$$f(q) = \begin{cases} 0 & if\ q = s \\ min_{q' \in \mathrm{Pred(q)}}(g(q') + c(q', q)) & otherwise \end{cases}$$

where **f(q)** determines the next move based on the g-values.
Step 4: Check if its g-value is equal to its **f(q)** value then it is locally consistent, otherwise it is locally inconsistent.
Step 5: The g-values of all vertices are equal to their start distances $\Longleftrightarrow$ all vertices are locally consistent
Step 6: To estimate the distance to the goal, heuristic function, **h(q,q_goal)** is computed.
The heuristic estimate **h(q,q_goal)** of the distance between vertices **q** and **q_goal** must satisfy

$$h\left(q, q_{goal}\right) \begin{cases} = 0 & if\ q = q_{goal} \\ \leq c(q, q') + h() & otherwise \end{cases}$$

for all vertices **q** $\in$ **Q** and **q˜′** $\in$ **Succ(q)**
Step 7: Repeat Steps 3 to 6 until all the vertices are in **S_DSR**

3.2.4. Hierarchical Pathfinding A* (HPA*)

HPA* was first designed in 2004 by Adi Botea and his colleagues [58]. The Algorithm makes use of the concept of divide and conquer approach in order to reduce the memory usage and complexity [59]. The search problem is divided into multiple smaller problems [60] using the principle of HPA* and the results are cached for every path segment [61]. When the Algorithm is executed, the game level is broken down into grids having the same size [62]. Each element of the grid is inspected in order to check whether the closest neighboring element can be fitted with the current element [63]. The second stage involves splitting the entire grid into grid elements of smaller size [64]. The Abstract Problem Graph is then formed and while declaring the two elements through which the edge is connected, the transition is made [65]. Different or same grid may contain these elements [66]. In the abstract graph, the location of source and the destination is added [3]. The A-star Algorithm is used to find the shortest path from source to destination element [67]. In this way, the HPA* Algorithm divides the main problem into many small problems [68] and solves them separately thereby reducing memory and increasing the speed [69].

3.2.5. Jump Point Search Algorithm (JPS)

The Algorithm expands by choosing (pruning) certain nodes in the grid map (called Jump Points) while the intermediate nodes are not expanded (skipping the exploration of a lot of nodes). The technique used by JPS is based on avoiding useless searches of symmetrical paths that A* searches, saving lots of time with the same memory load. Moving from a Jump Point to another is done by travelling in a concrete direction while recursively applying two neighbor pruning rules (one for horizontal and vertical steps and the other for diagonal ones) until reaching a dead-end, a non-walkable area, or the next Jump Point. To keep going with JPS, it is important to understand the concept of "pruning", which is the elimination of nodes to expand. That is to say the way in which the Algorithm decides which nodes to explore. After the pruning of nodes, the ones that remain in the tree are called "natural neighbors", which are the only ones that we want to consider. However,

sometimes we need to consider one or two nodes that are not natural. Those are called "forced neighbors" [70]. In Jump Point Search, our objective is to avoid symmetric paths by "jumping" all nodes that can be optimally reached by a path that does not visit the current node. This means that we chose a Jump Point if the optimal path must, obligatorily, pass through that node. Once it has reached an obstacle or another Jump Point, the recursion stops. So, a Jump Point y with a neighbor z will be a successor of a Jump Point x only if to reach z we need to visit x and y. To make it real, we need to consider two pruning rules, one for straight moves (horizontal and vertical) and another one for diagonal moves.

In Figure 9 (Left), from a Jump Point x (with parent p(x)), we recursively go straight until y and set it as a successor Jump Point of x because z cannot be reached (optimally) if we do not pass through x and y. The nodes in the middle are not evaluated or explored. In Figure 9 (Right), from a Jump Point x (with parent p(x)), we recursively go diagonally until y and set it as an x Jump Point Successor (the same than left). In this case, after each diagonal step, we do a straight recursion (marked with discontinue lines) and only if the two straight recursions fail, we keep going diagonally [71]. Also in the image, the node w is shown, which is a forced neighbor of x because of being in a place in which, to reach it optimally from the coming direction (p(x)), we need to pass through x [72].

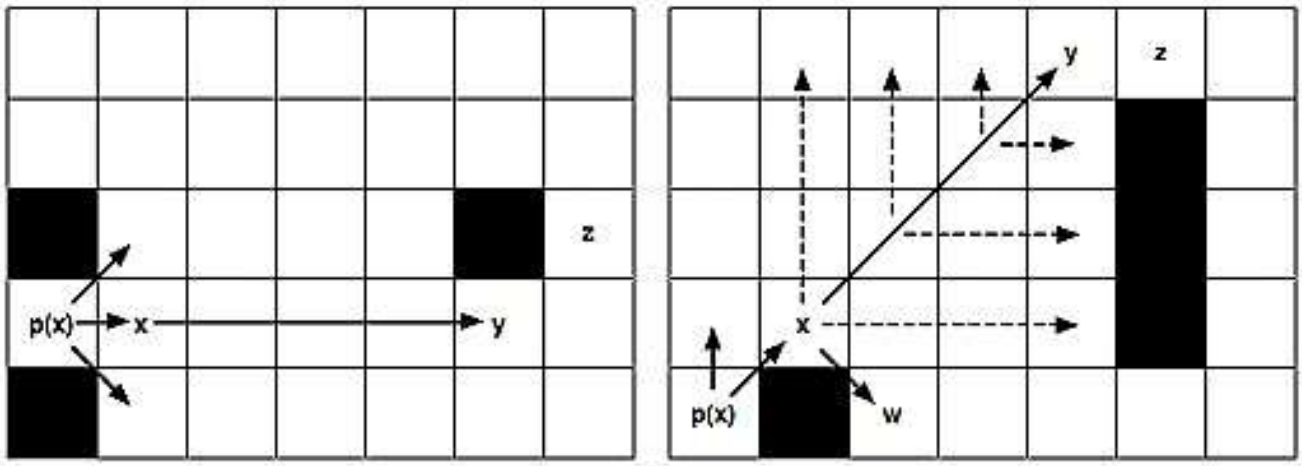


Figure 9. Left: from a jump point x (with parent p(x)), the Algorithm recursively goes straight until y and sets it as a successor jump point of x. **Right**: from a Jump Point x (with parent p(x)), the Algorithm recursively goes diagonally until y and set it as an x Jump Point Successor.

The heuristic function, time, and space complexities of pathfinding Algorithms are detailed in Table 2.

Table 2. Heuristic function, time and space complexities of pathfinding Algorithms.

Sr. No	Algorithm	Heuristic Function H(n)	Time Complexity T(n)	Space Complexity S(n)
1.	BFS	-	$T(n) = 1 + n^2 + n^3 + \ldots\ldots + n^s = O(n^s)$	$S(n) = O(n^s)$
2.	DFS	-	$T(n) = 1 + n^2 + n^3 + \ldots\ldots + n^d = O(n^d)$	S(n) = O(n*d)
3.	Dijkstra's	H(n) = 0 F(n) = G(n)	$T(n) = O(N^2)$	-
4.	A-star	F(n) = H(n) + G(n)	$T(n) = O(b^d)$	$S(n) = O(b^d)$
5.	Greedy Best First	G(n) = 0 F(n) = H(n)	$T(n) = O(b^m)$	$S(n) = O(b^m)$

3.3. Metaheuristic Approach Algorithms

The word metaheuristic come from two words, meta and heuristic. Meta means "beyond" and heuristic means "to discover". Meta-heuristics are problem-independent

techniques. Unlike the heuristic Algorithms, they do not take advantage of any specificity of the problem. In general, they are not greedy. In fact, they may even accept a temporary deterioration of the solution (example, the simulated-annealing technique), which allows them to explore more thoroughly the solution space and thus to hopefully get a better solution (that sometimes will coincide with the global optimum) [73]. Their main objective is to make use of the search space as much as possible to discover a nearly good solution to an optimization problem. Unlike the heuristic Algorithms, they are much better in terms of performance and do not focus on a particular problem [74].

3.3.1. Genetic Algorithm (GA)

This is a very popular Algorithm belonging to the metaheuristic category of Algorithms based on the concept of natural selection. Introduced first in 1960 by John Holland, the Algorithm has numerous applications in the fields of climatology for monitoring changes in global temperatures, neural networks, robotics, natural language processing, etc. In the world of video games, this Algorithm is used for finding the route with least cost. The search space for this Algorithm is termed as population [75]. In this population, there are several elements called chromosomes. From this population, the Algorithm generates randomly a set of chromosomes or a suitable set of solutions called the initial solution. Every element of this initial solution is inspected by a function that decides which acceptable set of chromosomes will remain until the end. This function is termed as the fitness function. The larger the value of this function, the greater the chances of survival of the move towards the next generation [76]. Figure 10 shows the six different steps involved in the Algorithm.

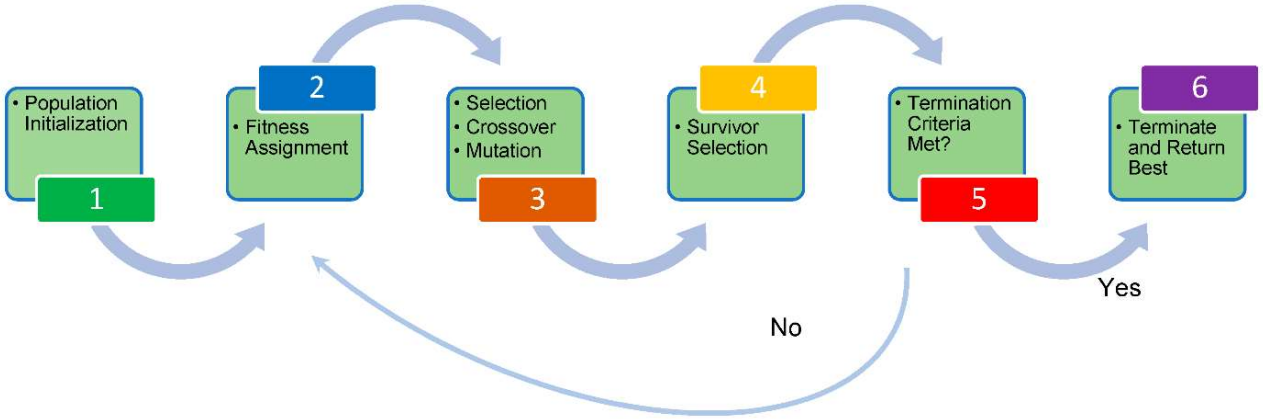


Figure 10. Process flow of genetic Algorithm.

1. **Initial Population:** The Algorithm begins with a set of individuals called a Population. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution). In a genetic Algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1 s and 0 s). We say that we encode the genes in a chromosome. The distinction between a gene, chromosome and population is represented in Figure 11 [77].

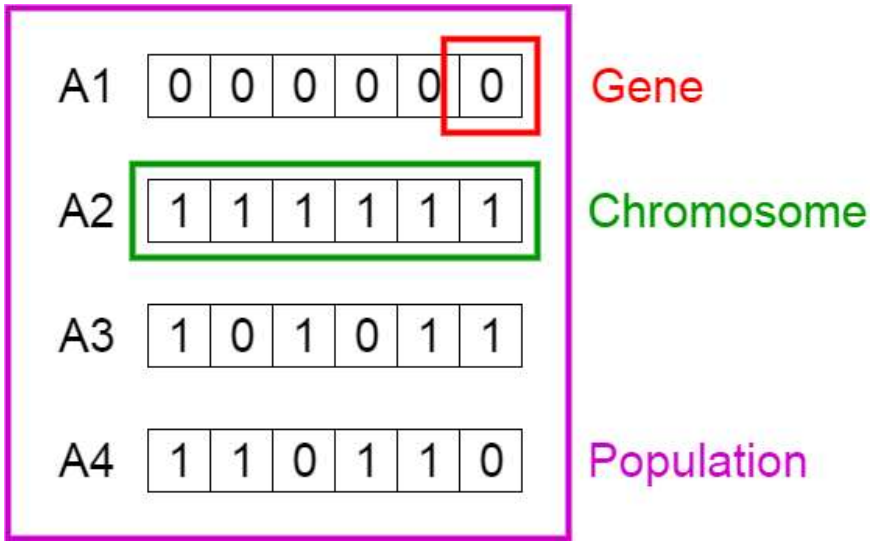


Figure 11. Distinction between population, chromosome, and gene.

2. **Fitness Function:** The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.
3. **Selection:** The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.
4. **Crossover:** Crossover is the most significant phase in a genetic Algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached. The new offspring are added to the population as shown in Figure 12 [78].
5. **Mutation:** In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped. Mutation occurs to maintain diversity within the population and prevent premature convergence. The Algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then, it is said that the genetic Algorithm has provided a set of solutions to our problem as shown in Figure 13 [79].

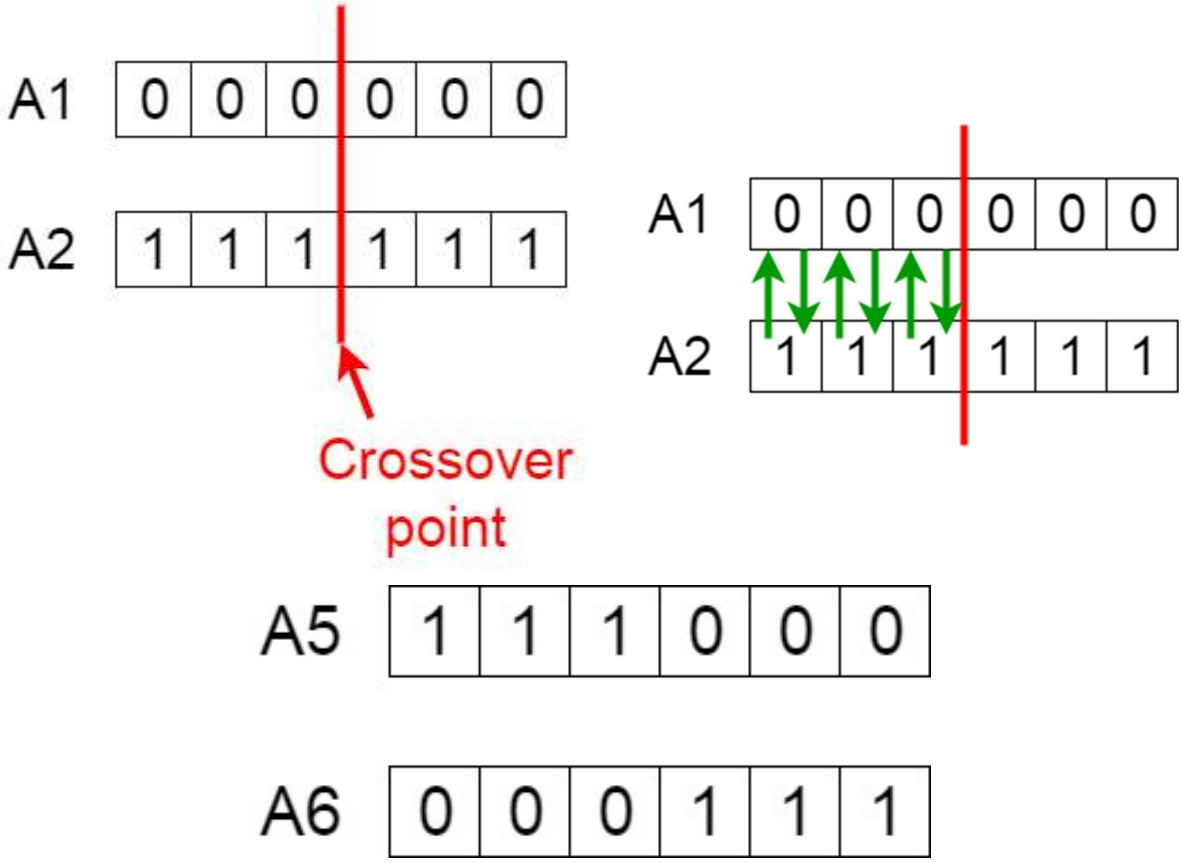


Figure 12. **Top left**: crossover point, **top right**: exchanging genes among parents, **bottom**: new offspring.

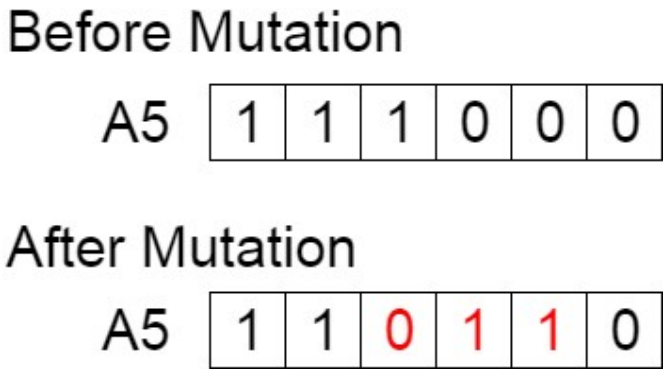


Figure 13. Mutation: before and after.

3.3.2. Ant Colony Optimization (ACO)

Ant Colony Optimization was first introduced in 1992 by Marco Dorigo [80]. As the name suggests, this Algorithm is based on the way of communication between ants in a colony [81]. These colony ants make use of chemical signals called pheromones, helping them to communicate trails from their nest to a food source [82].

Figure 14 represents the general flowchart for ACO. In the first stage, the trail of pheromone is initialized. The second stage involves iteration in which, through probabilistic state transition rule, a complete solution is constructed by every ant to the problem [83]. The pheromone state determines the state transition rule. Once a solution is generated by the ants, another rule of global pheromone is applied. It involves two phases:

1. Evaporation phase wherein a pheromone fraction is evaporated.

2. Reinforcement phase wherein an amount of pheromone is deposited by every ant which is proportional to the fitness of solution. The process is repeated until halting criteria is achieved.

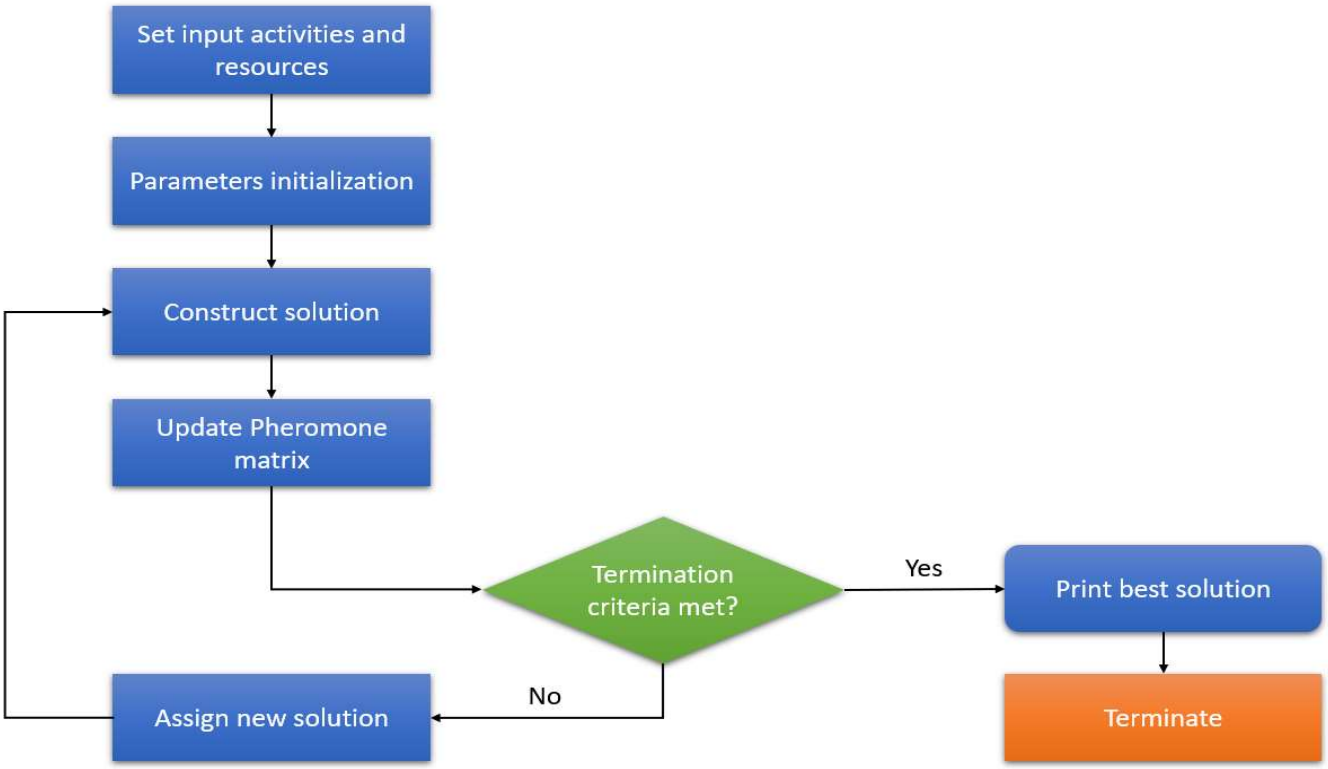


Figure 14. General flow chart for Ant Colony Optimization Algorithm.

Even if we place obstacles across their path, the pheromone will help them in discovering a minimum cost path which takes less time. Similarly, based on this concept of natural behavior, this popular Algorithm can be applied for pathfinding applications in video games [84].

4. Proposed Algorithm

This paper analyzes the performances of different pathfinding Algorithms so that game developers can make a suitable choice from these Algorithms for development of games involving pathfinding. The Algorithms have been appropriately classified based on different parameters and graphical representations so that a clear distinction is made among the Algorithms in terms of performance. Currently, in video games, various types of pathfinding Algorithms are used to reach from the source to destination. Some of these Algorithms are designed to avoid stationary obstacles or enemies along the path whereas some Algorithms are developed specifically for dynamic obstacles while some are for both.

However, we do not know how these Algorithms might perform in terms of hardware usage or speed or even finding the shortest path given different environmental scenarios. The proposed work involved the formation of grids of various sizes, 4×4, 8×8, 16×16, and 32×32. The grids were generated with and without obstacles identifying the performance of the Algorithms in different scenarios. The number of iterations to be executed is initialized as 'k'. Fixing the cell to start the computation is performed based on the player's preference. The next cell movement is computed for each Algorithm; Dijkstra's Algorithm 6 $\boldsymbol{D_{ij}}$, A-star $\boldsymbol{A^*}$, Breadth First Search $\boldsymbol{B_{FS}}$, and Greedy Best First $\boldsymbol{G_{BF}}$. Compute each and every visited cell for each Algorithm to compute the best move to determine the shortest path length for the game played. The shortest path length L and execution time t show that GRIN_PF outperforms the existing path finding approaches.

Algorithm 6 Proposed GRIN_PF.

Input: Grid G_i where $i \in 4 \times 4,\ 8 \times 8,\ 16 \times 16, 32 \times 32$. Number of iterations **k**.
Output: Shortest Path Length **L**, Visited Blocks **V**, Execution Time **t**.
Step 1: Formation of regular Grid of equal size cell with obstacles G_{io} and without obstacles G_{ino}
Step 2: Initialize the iteration count **c = 0**
Step 3: Based on the position of the player the starting cell can be fixed.
Step 4: Identify the next move using Algorithm
(a) Breadth First Search B_{FS} = **S_BFS**
(b) Dijkstra's Algorithm D_{ij} = **S_DJK**
(c) A-star A^* = **S_ASR**
(d) Greedy Best First G_{BF} = **S_GBF**
Step 5: Compute the visited block V_j *where* $j \in B_{FS},\ D_{ij},\ A^*, G_{BF}$
Step 6: **c = c + 1**
Step7: Repeat the steps for the given Iteration **k**, Go to Step 3.
Step 8: Compute the best move for each Algorithm.
Step 9: Determine the Shortest Path Length L_z *where* $z \in B_{FS},\ D_{ij},\ A^*, G_{BF}$
Step 10: Execution time t_z *where* $z \in D_{ij},\ A^*,\ B_{FS},\ G_{BF}$

5. Performance Comparison of Some Heuristic Pathfinding Algorithms

Tables 3 and 4 show the performance analysis of implemented heuristic pathfinding Algorithms without and with obstacles respectively. The parameters taken were grid size, total iteration blocks, total computed blocks without shortest path length, execution time in milliseconds, and shortest path length. Grid size is the size of the grid environment used for implementing the Algorithm. The Algorithms are implemented on 4×4, 8×8, 16×16, and 32×32 grid systems. Total iteration blocks refer to the total number of blocks the Algorithm has traversed in order to reach destination node. Total computed blocks without SPL refers to the total blocks traversed by Algorithm excluding the shortest path length blocks. Execution time is the time required by Algorithm to reach the goal node from start node. It is measured in milliseconds. Finally, the shortest path length is the minimum number of blocks taken by the Algorithm to reach the destination node.

Table 3. Performance analysis of pathfinding Algorithms without obstacles.

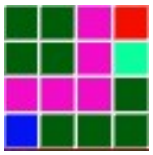
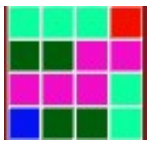
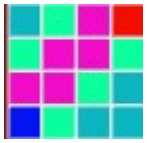
Grid Size	Grid Model	Algorithm	Total Iterations (Blocks)	Total Computed Blocks without SPL	Execution Time (ms)	Shortest Path Length (Blocks)
4×4		Dijkstra's	14	9	39.70	5
		A-star	10	5	27.95	5
		Breadth First Search	14	9	33.76	5
		Greedy Best First	6	1	14.38	5

Table 3. *Cont.*

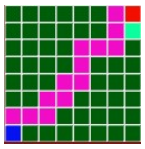
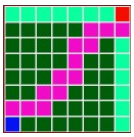
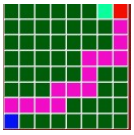
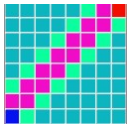
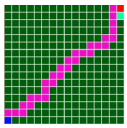
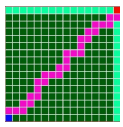
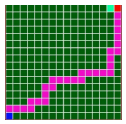
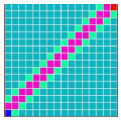
Grid Size	Grid Model	Algorithm	Total Iterations (Blocks)	Total Computed Blocks without SPL	Execution Time (ms)	Shortest Path Length (Blocks)
8 × 8		Dijkstra's	62	49	177.08	13
		A-star	50	37	140.5	13
		Breadth First Search	62	49	164.49	13
		Greedy Best First	14	1	36.01	13
16 × 16		Dijkstra's	254	225	899.27	29
		A-star	226	197	787.43	29
		Breadth First Search	254	225	868.56	29
		Greedy Best First	30	1	97.90	29

Table 3. *Cont.*

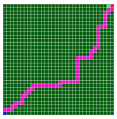
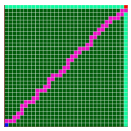
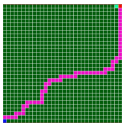
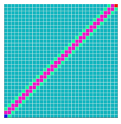
Grid Size	Grid Model	Algorithm	Total Iterations (Blocks)	Total Computed Blocks without SPL	Execution Time (ms)	Shortest Path Length (Blocks)
32 × 32		Dijkstra's	1022	961	7500.14	61
		A-star	962	901	6757.75	61
		Breadth First Search	1022	961	7449.17	61
		Greedy Best First	62	1	346.60	61

Table 4. Performance analysis of pathfinding Algorithms with obstacles.

Grid Size	Grid Model	Algorithm	Total Iterations (Blocks)	Total Computed Blocks without SPL	Execution Time (ms)	Shortest Path Length (Blocks)
4 × 4		Dijkstra's	6	1	15.17	5
		A-star	6	1	12.92	5
		Breadth First Search	6	1	13.90	5
		Greedy Best First	6	1	14.25	5

Table 4. *Cont.*

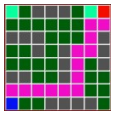
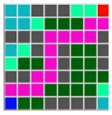
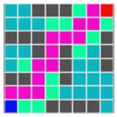
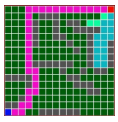
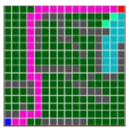
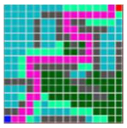
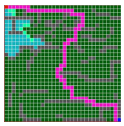
Grid Size	Grid Model	Algorithm	Total Iterations (Blocks)	Total Computed Blocks without SPL	Execution Time (ms)	Shortest Path Length (Blocks)
8 × 8		Dijkstra's	30	15	102.60	15
		A-star	32	17	91.35	15
		Breadth First Search	42	27	120.54	15
		Greedy Best First	14	1	40.19	13
16 × 16		Dijkstra's	177	146	743.47	31
		A-star	140	109	512.45	31
		Breadth First Search	178	147	663.76	31
		Greedy Best First	85	48	298.61	37

Table 4. *Cont.*

Grid Size	Grid Model	Algorithm	Total Iterations (Blocks)	Total Computed Blocks without SPL	Execution Time (ms)	Shortest Path Length (Blocks)
32 × 32		Dijkstra's	754	679	5678.25	75
		A-star	587	512	3975.34	75
		Breadth First Search	752	677	5316.65	75
		Greedy Best First	224	143	1396.73	81

6. Graphical Representation of Implemented Pathfinding Algorithms

Figure 15 shows a graph that compares the performances of the four implemented Algorithms without taking the obstacles into consideration. The execution time is taken as a parameter for comparing the performances of the Algorithms. The x-axis represents the Algorithm names and the y-axis depicts the execution time in milliseconds. The Algorithms compared are Dijkstra's, A-star, BFS, and Greedy Best First Search.

The 4 × 4 grid system without obstacles shows that the execution time for Dijkstra's is 39.7 ms, for A-star is 27.95 ms, for BFS is 33.76 ms, and Greedy BFS is 14.38 ms. From this, we can see that in the case of the 4 × 4 system, the Greedy BFS Algorithm performed the best among all Algorithms. The worst performing was Dijkstra's. A-star was the second best and BFS was third best in performance.

For the 8 × 8 grid system without obstacles, it is shown that the execution time for Dijkstra's is 177.08 ms, for A-star is 140.5 ms, for BFS is 164.49 ms, and Greedy BFS is 36.01 ms. From this, we can see that in case of the 8 × 8 system, the Greedy BFS Algorithm again performed the best among all Algorithms. Once again, Dijkstra's Algorithm performed the worst. A-star was the second best and BFS was third best in performance. In addition, the performance of BFS and Dijkstra's was almost similar for the 8 × 8 system as there was very little difference in execution times.

The 16 × 16 grid system without obstacles shows that the execution time for Dijkstra's is 899.27 ms, for A-star is 787.43 ms, for BFS is 868.56 ms, and Greedy BFS is 97.9 ms. From this, we can see that in the case of the 16 × 16 system, the Greedy BFS Algorithm again performed the best among all Algorithms. Once again, Dijkstra's Algorithm performed the worst. A-star was the second best and BFS was third best in performance. Moreover, the performance of BFS and Dijkstra's was again almost similar for the 16 × 16 system as there was very less difference in execution times. In this system, the A-star Algorithm's performance was also close to Dijkstra's and BFS.

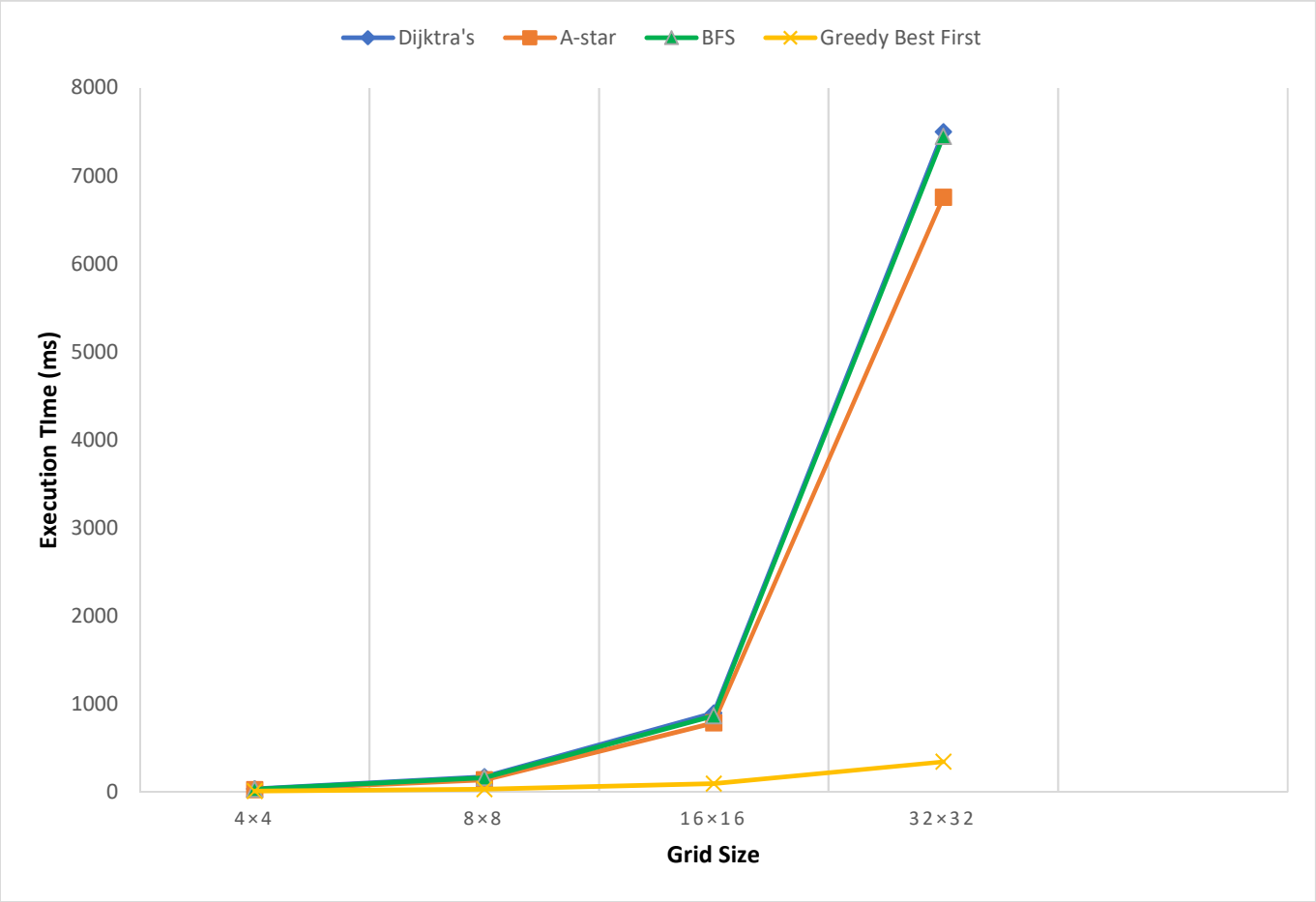


Figure 15. Graphical representation of pathfinding Algorithms without obstacles.

The 32 × 32 grid system without obstacles concludes that the execution time for Dijkstra's is 7500.14 ms, for A-star is 6757.75 ms, for BFS is 7449.17 ms, and Greedy BFS is 346.6 ms. From this, we can see that in the case of the 32 × 32 system, the Greedy BFS Algorithm again performed the best among all Algorithms. Once again, Dijkstra's Algorithm performed the worst. A-star was the second best and BFS was third best in performance again. Moreover, the performance of BFS and Dijkstra's was again almost similar for 32 × 32 system as there was very little difference in execution times. In this system, the A-star Algorithm's performance was also close to Dijkstra's and BFS again.

From the above graph, we can conclude that the best pathfinding Algorithm for without obstacles, which required the least memory and least execution time, was Greedy Best First Search. However, this Algorithm is not optimal. The worst Algorithm for without obstacles requiring most memory usage was Dijkstra's Algorithm. The second-best Algorithm was found to be A-star, followed by Breadth First Search. In addition, it can be seen that as the grid size increases, the difference in the execution times of Greedy BFS and others increases significantly.

Figure 16 shows a graph that compares the performances of the four implemented Algorithms by taking the obstacles into consideration. The 4 × 4 grid system with obstacles shows that the execution time for Dijkstra's is 15.17 ms, for A-star is 12.92 ms, for BFS is 13.9 ms, and Greedy BFS is 14.25 ms. From this, we can see that in the 4 × 4 system alone with obstacles, the A-star Algorithm performed the best among all Algorithms. The worst performing was Dijkstra's. BFS was the second best and Greedy BFS was third best in performance. Only for the 4 × 4 system, the performances of all Algorithms were very similar since execution time differences were small.

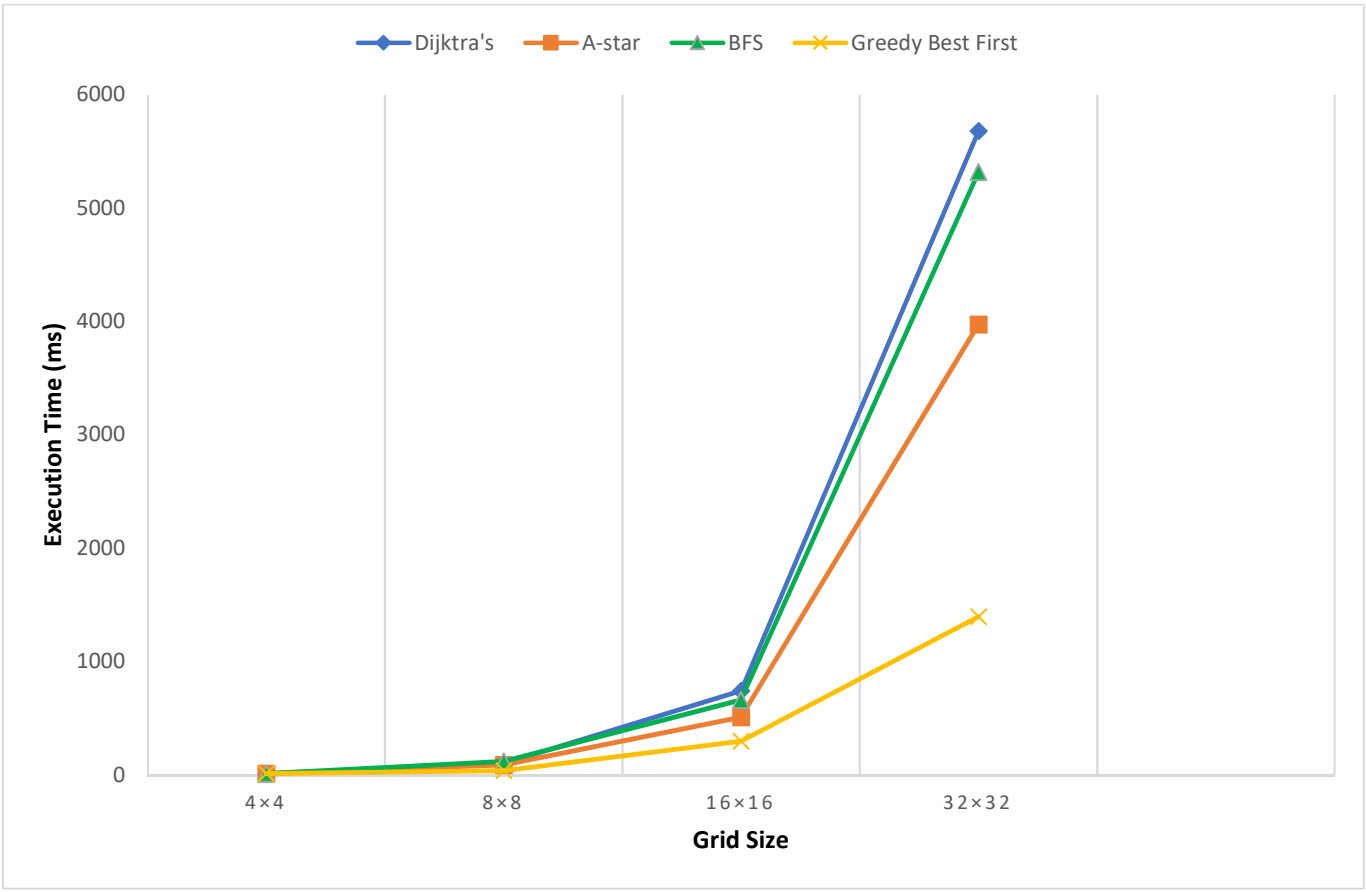


Figure 16. Graphical representation of pathfinding Algorithms with obstacles.

The 8 × 8 grid system with obstacles shows that the execution time for Dijkstra's is 102.6 ms, for A-star is 91.35 ms, for BFS is 120.54 ms, and Greedy BFS is 40.19 ms. From this, we can see that in the 8 × 8 system alone with obstacles, the BFS Algorithm was the worst in terms of performance. The Greedy BFS Algorithm performed the best among all Algorithms. A-star was the second best and Dijkstra's was third best in performance. In addition, the performance of A-star and Dijkstra's was almost similar for 8 × 8 system as there was very little difference in execution times.

The 16 × 16 grid system with obstacles shows that the execution time for Dijkstra's is 743.47 ms, for A-star is 512.45 ms, for BFS is 663.76 ms, and Greedy BFS is 298.61 ms. From this, we can see that for the 16 × 16 system, the Greedy BFS Algorithm again performed the best among all Algorithms. Dijkstra's Algorithm performed the worst. A-star was the second best and BFS was third best in performance.

The 32 × 32 grid system with obstacles shows that the execution time for Dijkstra's is 5678.25 ms, for A-star is 3975.34 ms, for BFS is 5316.65 ms, and Greedy BFS is 1396.73 ms. From this, we can see that in the 32 × 32 system, the Greedy BFS Algorithm again performed the best among all Algorithms. Once again, Dijkstra's Algorithm performed the worst. A-star was the second best and BFS was third best in performance again. It can be seen that as the grid size increases, the difference in the execution times of Greedy BFS and others increases significantly. Furthermore, the performance of BFS and Dijkstra's was again almost similar for the 32 × 32 system as there was very little difference in execution times. In this system, the A-star Algorithm's performance was also close to Dijkstra's and BFS again.

From Figure 16, we can conclude that on average, the overall fastest pathfinding Algorithm for with obstacles, which required the least average memory and least average execution time, was Greedy Best First Search. However, the Greedy Best First is not optimal.

The worst Algorithm on average for without obstacles requiring most memory usage was Dijkstra's Algorithm. The second-best Algorithm was found to be A-star, followed by Breadth First Search. Moreover, it can be seen that as the grid size increases, the difference in the execution times of Greedy BFS and others increases significantly. Pre-processing is one of the main problems in pathfinding, thereby making complex pathfinding in real-time difficult [85]. Most of the pathfinding engines are unable to handle worlds in dynamic environments and have difficulty in producing movements which look realistic [86]. The main reason this happens is because of the pre-processing stages where the nodes for the pathfinder are produced to travel based on the representation of maps in a static environment. However, if a node is covered along the predetermined path by a dynamic obstacle, the agent will think it can walk where the object is located [87,88]. The second problem encountered is when an agent goes along the nodes in a path along a straight line, unrealistic movement occurs [89]. This is mainly due to the fact that there is some trade-off between the speed and realistic movement. This problem is being improved in different games through the application of splines in between different nodes in order to smooth out the path [90,91].

7. Summary of Pathfinding Algorithms

Table 5 represents the final summary of the Algorithms. Two Algorithms are taken and compared in each row. The category of Algorithm, execution time, memory usage as well as the final statement about which Algorithm is better, has been mentioned.

Table 5. Final summary of pathfinding Algorithms.

Compared Algorithms	Algorithm Category	Execution Time	Memory Usage	Implemented	Author	Year	Better Algorithm
1. BFS 2. DFS	1. Uninformed 2. Uninformed	1 > 2 (if DFS traverses in right path) 1 < 2 (if DFS runs in infinite loop)	BFS requires more memory than DFS	1. Yes 2. No	[90]	2019	Guarantees optimal solution, more memory required—BFS Does not guarantee optimal solution, less memory required—DFS
1. Dijkstra's 2. BFS	1. Uninformed 2. Uninformed	1 < 2	BFS requires more memory than Dijkstra's	1. Yes 2. Yes	[92]	2015	Dijkstra's
1. Dijkstra's 2. A*	1. Uninformed 2. Informed	1 > 2	Dijkstra's requires more memory than A*	1. Yes 2. Yes	[4,6]	2015, 2016	A*
1. A* 2. Greedy Best First	1. Informed 2. Informed	1 > 2	A* requires more memory than Greedy Best First	1. Yes 2. Yes	[92]	2015	A* (since Greedy Best First is not optimal)
1. A* 2. D* Lite	1. Informed 2. Informed	1 < 2 (Restricted environment) 1 > 2 (Open environment)	D* Lite requires less memory in restricted environment and more memory in open environment than A*	1. Yes 2. Yes	[93]	2009	Restricted environment—D* Lite Open environment—A*
1. A* 2. HPA*	1. Informed 2. Informed	1 > 2	A* requires more memory than HPA*	1. Yes 2. No	[4,92]	2015, 2015	HPA*
1. JPS 2. A*	1. Informed 2. Informed	1 < 2	A* requires more memory than JPS	1. No 2. Yes	[92]	2015	JPS

Table 5. *Cont.*

Compared Algorithms	Algorithm Category	Execution Time	Memory Usage	Implemented	Author	Year	Better Algorithm
1. HPA* 2. JPS	1. Informed 2. Informed	1 < 2	JPS requires more memory than HPA*	1. No 2. No	[92]	2015	HPA*
1. Dijkstra's 2. GA	1. Informed 2. Metaheuristic	1 > 2	Dijkstra's requires more memory than GA	1. Yes 2. No	[91]	2008	GA
1. A* 2. GA	1. Informed 2. Metaheuristic	1 < 2	GA requires more memory than A*	1. Yes 2. No	[94,95]	2016, 2017	A*
1. GA 2. ACO	1. Metaheuristic 2. Metaheuristic	1 > 2	GA requires more memory than ACO	1. No 2. No	[88,89]	2010, 2016	ACO

8. Conclusions

In this work, we presented GRIN_PF, a pathfinding approach for computing the performance of different pathfinding Algorithms so that game developers can make a suitable choice from these Algorithms for the development of games involving pathfinding. The Algorithms have been appropriately classified based on different parameters and graphical representations so that a clear distinction can be made among the Algorithms in terms of performance.

Our experimental evaluation was performed on various grids of different sizes. Four different path finding Algorithms were analyzed for a same set of inputs and its results were examined. Based on the results, the game developer can choose the most suitable one.

Performance analysis was done using two different scenarios. The first one was examining the pathfinding Algorithms on various grids without obstacles and the second one was about examining the same set of Algorithms with obstacles. Finally, the shortest path length was calculated based on the minimum number of blocks taken by the Algorithm to reach the destination node. In case of dynamic objects introduced in the path, much effort might be required to improve the reactive abilities of the AI agent. A solution may be to make the agent to be aware of its surroundings. This can be done by fitting the agent with sensors so that its pathfinder can guide it but cannot control it. However, based on the results of our proposed Algorithm, the agent can use a better solution of utilizing the appropriate Algorithm during runtime.

Author Contributions: Conceptualization, S.R.L. and G.J.; methodology, S.R.L.; software, S.R.L.; validation, S.R.L., G.J. and J.A.; formal analysis, S.R.L.; investigation, S.R.L. and G.J.; resources, S.R.L.; data curation, S.R.L.; writing—original draft preparation, S.R.L.; writing—review and editing, S.R.L. and G.J.; visualization, S.R.L. and G.J.; supervision, G.J. and J.A.; project administration, S.R.L. and G.J.; funding acquisition, L.I.I. All authors have read and agreed to the published version of the manuscript.

Funding: This study was supported by the Yayasan Universiti Teknologi PETRONAS (YUTP), Malaysia, under Grant 015LC0-134.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rafiq, A.; Kadir, T.A.; Ihsan, S.N. Pathfinding Algorithms in game development. In *IOP Conference Series: Materials Science and Engineering*; IOP Publishing: Pahang, Malaysia, 2020; Volume 769, p. 012021. [CrossRef]
2. Sazaki, Y.; Primanita, A.; Syahroyni, M. Pathfinding car racing game using dynamic pathfinding Algorithm and Algorithm A*. In Proceedings of the 2017 3rd International Conference on Wireless and Telematics (ICWT), Palembang, Indonesia, 27–28 July 2017; pp. 164–169. [CrossRef]
3. Zarembo, I.; Kodors, S. Pathfinding Algorithm efficiency analysis in 2D grid. Environment. Technologies. Resources. In Proceedings of the International Scientific and Practical Conference, Rēzekne, Latvia, 20–22 June 2013; Volume 2, pp. 46–50. [CrossRef]

4. Foudil, C.; Noureddine, D.; Sanza, C.; Duthen, Y. Path finding and collision avoidance in crowd simulation. *J. Comput. Inf. Technol.* **2009**, *17*, 217–228. [CrossRef]
5. Anbuselvi, R.; Phil, M. Path finding solutions for grid based graph. *Adv. Comput. Int. J.* **2013**, *4*, 51–60. [CrossRef]
6. Panda, M.; Mishra, A. A survey of shortest-path Algorithms. *Int. J. Appl. Eng. Res.* **2018**, *13*, 6817–6820.
7. Graham, R.; McCabe, H.; Sheridan, S. Pathfinding in computer games. *ITB J.* **2003**, *8*, 57–81. [CrossRef]
8. Zafar, A.; Agrawal, K. Novel optimization using hierarchical Path finding A* (HPA*) Algorithm for strategic gaming setup. *Int. J. Eng. Technol.* **2018**, *7*, 54. [CrossRef]
9. Mathew, G.E. Direction based heuristic for pathfinding in video games. *Procedia Comput. Sci.* **2015**, *47*, 262–271. [CrossRef]
10. Gregory, J. Engine Support Systems. In *Game Engine Architecture*; AK Peters/CRC Press: Boca Raton, FL, USA, 2009; pp. 217–280. [CrossRef]
11. Lim, K.L.; Seng, K.P.; Yeong, L.S.; Ang, L.M.; Ch'ng, S.I. Uninformed pathfinding: A new approach. *Expert Syst. Appl.* **2015**, *42*, 2722–2730. [CrossRef]
12. Khantanapoka, K.; Chinnasarn, K. Pathfinding of 2D & 3D game real-time strategy with depth direction A* Algorithm for multi-layer. In Proceedings of the 2009 Eighth International Symposium on Natural Language Processing, Bangkok, Thailand, 20–22 October 2009; pp. 184–188. [CrossRef]
13. Amit's, T. Map Representations on Pathfinding. Available online: http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html (accessed on 24 February 2021).
14. Coppin, B. *Artificial Intelligence Illuminated*; Jones & Bartlett Learning: Burlington, MA, USA, 2004.
15. Available online: https://www.baeldung.com/cs/greedy-vs-heuristic-Algorithm (accessed on 11 January 2021).
16. Kapi, A.Y.; Sunar, M.S.; Zamri, M.N. A review on informed search Algorithms for video games pathfinding. *Int. J.* **2020**, *9*, 2756–2764. [CrossRef]
17. Available online: https://vgc.poly.edu/~{}csilva/papers/phd96.pdf (accessed on 17 January 2021).
18. Abd Algfoor, Z.; Sunar, M.S.; Kolivand, H. A comprehensive study on pathfinding techniques for robotics and video games. *Int. J. Comput. Games Technol.* **2015**, *2015*, 736138. [CrossRef]
19. Available online: https://www.gamedev.net/tutorials/programming/artificial-intelligence/navigation-meshes-and-pathfinding-r4880 (accessed on 3 February 2021).
20. García, L.L.; Arellano, A.G.; Cruz-Santos, W. A parallel path-following phase unwrapping Algorithm based on a top-down breadth-first search approach. *Opt. Lasers Eng.* **2020**, *124*, 105827. [CrossRef]
21. Zhou, R.; Hansen, E.A. Breadth-first heuristic search. *Artif. Intell.* **2006**, *170*, 385–408. [CrossRef]
22. Rahim, R.; Abdullah, D.; Nurarif, S.; Ramadhan, M.; Anwar, B.; Dahria, M.; Nasution, S.D.; Diansyah, T.M.; Khairani, M. Breadth first search approach for shortest path solution in Cartesian area. *J. Phys. Conf. Ser.* **2018**, *1019*, 012036. [CrossRef]
23. Zhou, Y.; Wang, W.; He, D.; Wang, Z. A fewest-turn-and-shortest path Algorithm based on breadth-first search. *Geo-Spat. Inf. Sci.* **2014**, *17*, 201–207. [CrossRef]
24. Ajwani, D.; Dementiev, R.; Meyer, U.; Osipov, V. Breadth first search on massive graphs. In *9th Implementation Challenge of DIMACS*; The Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University: Piscataway, NJ, USA, 22 March 2006.
25. Putri, S.; Tulus, T.; Napitupulu, N. Implementation and Analysis of Depth-First Search (DFS) Algorithm for Finding The Longest Path. In Proceedings of the International Seminar on Operational Research (InteriOR), Medan, Indonesia, 27–28 July 2011. [CrossRef]
26. Anita Neeraj, V.; Abhishek, B. A Review Paper On Examination Of Dijkstra's And A* Algorith To Find The Shortest Path. *Int. J. Creat. Res. Thoughts (IJCRT)* **2018**, *6*, 635–641.
27. Jadeel, J. Understanding Dijkstra Algorithm. *SSRN Electron. J.* **2013**, *10*, 1–27. [CrossRef]
28. Kadry, S.; Abdallah, A.; Joumaa, C. On the optimization of Dijkstra's Algorithm. In *Informatics in Control, Automation and Robotics*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 393–397.
29. Zhou, M.; Gao, N. Research on Optimal Path based on Dijkstra Algorithms. In Proceedings of the 3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019), Dalian, China, 29 March 2019; Atlantis Press: Dordrecht, The Netherlands, 2019. [CrossRef]
30. Nagamani, S. Survey—Application of A* Algorithm in Dynamic Ambulance Routing Problem and other Strategies & Methods. *Int. J. Adv. Sci. Technol.* **2020**, *29*, 406–413.
31. Van Den Berg, J.P.; Overmars, M.H. Roadmap-based motion planning in dynamic environments. *IEEE Trans. Robot.* **2005**, *21*, 885–897. [CrossRef]
32. Dijkstra, E.W. A note on two problems in connexion with graphs. *Numer. Math.* **1959**, *1*, 269–271. [CrossRef]
33. Pathak, M.J.; Patel, R.L.; Rami, S.P. Comparative analysis of search Algorithms. *Int. J. Comput. Appl.* **2018**, *179*, 40–43.
34. Harabor, D.; Grastien, A. Online graph pruning for pathfinding on grid maps. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4 August 2011; Volume 25, pp. 1114–1119.
35. Ahmed, D.T.; Shirmohammadi, S. Ïntelligent path finding for avatars in Massively Multiplayer Online Games. In Proceedings of the 2009 IEEE Workshop on Computational Intelligence in Virtual Environments, Nashville, TN, USA, 30 March–2 April 2009; pp. 61–65. [CrossRef]
36. Gregory, J. *Game Engine Architecture*; AK Peters/CRC Press: Boca Raton, FL, USA, 2018. [CrossRef]

37. Neukart, F.; Morar, S.A. Operations on quantum physical artificial neural structures. *Procedia Eng.* **2014**, *69*, 1509–1517. [CrossRef]
38. Hart, P.E.; Nilsson, N.J.; Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.* **1968**, *4*, 100–107. [CrossRef]
39. Sharma, C.K. Shortest Path Searching for Road Network using A* Algorithm. *Int. J. Comput. Sci. Mob. Comput.* **2015**, *4*, 513–522.
40. Latuconsina, R.; Purboyo, T.W. Shortest Path Algorithms: State of the Art. *Int. J. Appl. Eng. Res.* **2017**, *12*, 13610–13617.
41. Andrias, R.M.; Sunar, M.S. User/player type in gamification. *Int. J. Adv. Trends Comput. Sci. Eng.* **2019**, *8*, 89–94. [CrossRef]
42. Mueller, S.T.; Perelman, B.S.; Simpkins, B.G. Pathfinding in the cognitive map: Network models of mechanisms for search and planning. *Biol. Inspired Cogn. Archit.* **2013**, *5*, 94–111. [CrossRef]
43. Cui, X.; Shi, H. A*-based pathfinding in modern computer games. *Int. J. Comput. Sci. Netw. Secur.* **2011**, *11*, 125–130.
44. Permana, S.H.; Bintoro, K.Y.; Arifitama, B.; Syahputra, A. Comparative analysis of pathfinding Algorithms a*, dijkstra, and bfs on maze runner game. *IJISTECH Int. J. Inf. Syst. Technol.* **2018**, *1*, 1. [CrossRef]
45. Zikky, M. Review of A*(A star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the Pacman game. *EMITTER Int. J. Eng. Technol.* **2016**, *4*, 141–149. [CrossRef]
46. Smołka, J.; Miszta, K.; Skublewska-Paszkowska, M.; Łukasik, E. A* pathfinding Algorithm modification for a 3D engine. In Proceedings of the MATEC Web of Conferences, Lublin, Poland, 1 January 2019; EDP Sciences, Les Ulis, France, 2019; Volume 252, p. 03007. [CrossRef]
47. Wang, X.Z. The comparison of three Algorithms in shortest path issue. *J. Phys. Conf. Ser.* **2018**, *1087*, 022011. [CrossRef]
48. Samal, A.; Saxena, A.; Ray, D. Comparative Study of Algorithms in Artificial Intelligence: Best First Search, Greedy Best First Search and Iterative Deepening. *Int. J. Softw. Hardw. Res. Eng.* **2018**, *6*, 6–11.
49. Samara, G.; Aljaidi, M. Aware-routing protocol using best first search Algorithm in wireless sensor. *Int. Arab J. Inf. Technol.* **2018**, *15*, 592–598.
50. Stentz, A. Optimal and efficient path planning for partially known environments. In *Intelligent Unmanned Ground Vehicles*; Springer: Boston, MA, USA, 1997; pp. 203–220.
51. Sazaki, Y.; Satria, H.; Syahroyni, M. Comparison of A∗ and dynamic pathfinding Algorithm with dynamic pathfinding Algorithm for NPC on car racing game. In Proceedings of the 2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA), Lombok, Indonesia, 26–27 October 2017; pp. 1–6. [CrossRef]
52. Available online: https://informatika.stei.itb.ac.id/~{}rinaldi.munir/Stmik/2017-2018/Makalah/Makalah-IF2211-2018-134 (accessed on 23 March 2021).
53. Stentz, A. The focussed dˆ* Algorithm for real-time replanning. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, QC, Canada, 20–25 August 1995; pp. 1652–1659.
54. Available online: https://bth.diva-portal.org/smash/get/diva2:1474900/FULLTEXT02.pdf (accessed on 4 April 2021).
55. Ramalingam, G.; Reps, T. An incremental Algorithm for a generalization of the shortest-path problem. *J. Algorithms* **1996**, *21*, 267–305. [CrossRef]
56. Mathew, K.; Tabassum, M.; Ramakrishnan, M. Experimental comparison of uninformed and heuristic AI Algorithms for N puzzle solution. In Proceedings of the International Journal of Digital Information and Wireless Communications, Hongkong, China, 12 November 2013; pp. 12–14.
57. Koenig, S.; Likhachev, M. Fast replanning for navigation in unknown terrain. *IEEE Trans. Robot.* **2005**, *21*, 354–363. [CrossRef]
58. Uwe, K. *Applying Graph Partitioning to Hierarchical Pathfinding in Computer Games*; Institut f¨ur Mathematik und Informatik, Universit: Leipzig, Germany, 2011.
59. Jansen, M.; Buro, M. HPA* enhancements. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, Palo Alto, CA, USA, 6–8 June 2007; Volume 3, pp. 84–87.
60. Sturtevant, N.; Buro, M. Partial pathfinding using map abstraction and refinement. In Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, Pittsburgh, PA, USA, 9–13 July 2005; Volume 5, pp. 1392–1397.
61. Yiu, Y.F.; Du, J.; Mahapatra, R. Evolutionary heuristic a* search: Pathfinding Algorithm with self-designed and optimized heuristic function. *Int. J. Semant. Comput.* **2019**, *13*, 5–23. [CrossRef]
62. Chattopadhyay, I.; Mallapragada, G.; Ray, A. ν☆: A robot path planning Algorithm based on renormalised measure of probabilistic regular languages. *Int. J. Control* **2009**, *82*, 849–867. [CrossRef]
63. Bagheri, S.M.; Taghaddos, H.; Mousaei, A.; Shahnavaz, F.; Hermann, U. An A-Star Algorithm for semi-optimization of crane location and configuration in modular construction. *Autom. Constr.* **2021**, *121*, 103447. [CrossRef]
64. Yiu, Y.F.; Du, J.; Mahapatra, R. Evolutionary heuristic a* search: Heuristic function optimization via genetic Algorithm. In Proceedings of the 2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), Laguna Hills, CA, USA, 26–28 September 2018; pp. 25–32. [CrossRef]
65. Holte, R.C.; Perez, M.B.; Zimmer, R.M.; MacDonald, A.J. Hierarchical A*: Searching abstraction hierarchies efficiently. In Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, Portland, OR, USA, 4–8 August 1996; Volume 1, pp. 530–535.
66. Yao, J.; Zhang, B.; Zhou, Q. The optimization of A* Algorithm in the practical path finding application. In Proceedings of the 2009 WRI World Congress on Software Engineering, Xiamen, China, 19–21 May 2009; Volume 2, pp. 514–518. [CrossRef]

67. Foead, D.; Ghifari, A.; Kusuma, M.B.; Hanafiah, N.; Gunawan, E. A systematic literature review of A* pathfinding. *Procedia Comput. Sci.* **2021**, *179*, 507–514. [CrossRef]
68. Zhou, R.; Hansen, E. Multiple sequence alignment using Anytime A*. In Proceedings of the 18th National Conference on Artificial Intelligence (AAAI), Edmonton, AB, Canada, 28 July–1 August 2002; Volume 2002, pp. 975–976.
69. Botea, A.; Müller, M.; Schaeffer, J. Near optimal hierarchical path-finding. *J. Game Dev.* **2004**, *1*, 1–30.
70. Min, J.G.; Ruy, W.S.; Park, C.S. Faster pipe auto-routing using improved jump point search. *Int. J. Nav. Archit. Ocean. Eng.* **2020**, *12*, 596–604. [CrossRef]
71. Tanner, B. *Jump Point Search Analysis*; Florida State University. fsu. edu.: Tallahassee, FL, USA, 2014.
72. Available online: https://harablog.wordpress.com/2011/09/07/jump-point-search (accessed on 21 April 2022).
73. Gunantara, N.; Nurweda Putra, I. The characteristics of metaheuristic method in selection of path pairs on multicriteria ad hoc networks. *J. Comput. Netw. Commun.* **2019**, *2019*, 7983583. [CrossRef]
74. Available online: https://analyticsindiamag.com/understanding-metaheuristics-Algorithm-in-800-words (accessed on 9 July 2021).
75. Kumar, R.; Kumar, M. Exploring genetic Algorithm for shortest path optimization in data networks. *Glob. J. Comput. Sci. Technol.* **2010**, *10*, 8–12.
76. Leigh, R.; Louis, S.J.; Miles, C. Using a genetic Algorithm to explore A*-like pathfinding Algorithms. In Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, Honolulu, HI, USA, 1–5 April 2007; pp. 72–79. [CrossRef]
77. Hasan, B.S.; Khamees, M.A.; Mahmoud, A.S. A heuristic genetic Algorithm for the single source shortest path problem. In Proceedings of the 2007 IEEE/ACS International Conference on Computer Systems and Applications, Amman, Jordan, 13–16 May 2007; pp. 187–194. [CrossRef]
78. Ito, T. A genetic Algorithm approach to piping route path planning. *J. Intell. Manuf.* **1999**, *10*, 103–114. [CrossRef]
79. Available online: https://towardsdatascience.com/introduction-to-genetic-Algorithms-including-example-code-e396e98d8bf3 (accessed on 21 April 2021).
80. Mugal, N.G.; Dhadse, R.; Solanke, P.A.; Chandekar, R.; Jaiswal, P. An Overview of Minimum Shortest Path Finding System Using Ant Colony Algorithm. *Int. J. Eng. Res. Technol.* **2014**, *3*, 564–568.
81. Gómez, O.; Barán, B. Omicron ACO.A New Ant Colony Optimization Algorithm. *CLEI Electron. J.* **2005**, *8*, 1–8. [CrossRef]
82. Blum, C.; Dorigo, M. Deception in ant colony optimization. In *International Workshop on Ant Colony Optimization and Swarm Intelligence*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 118–129. [CrossRef]
83. Ma, G.; Duan, H.; Liu, S. Improved ant colony Algorithm for global optimal trajectory planning of UAV under complex environment. *Int. J. Comput. Sci. Appl.* **2007**, *4*, 57–68.
84. Duan, H.B.; Ma, G.J.; Wang, D.B.; Yu, X.F. An improved ant colony Algorithm for solving continuous space optimization problems. *J. Syst. Simul.* **2007**, *19*, 974–977.
85. Li, T.; Qi, L.; Ruan, D. An efficient Algorithm for the single-source shortest path problem in graph theory. In Proceedings of the 2008 3rd International Conference on Intelligent System and Knowledge Engineering, Xiamen, China, 17–19 November 2008; Volume 1, pp. 152–157. [CrossRef]
86. Panahi, S.; Delavar, M.R. A GIS-based dynamic shortest path determination in emergency vehicles. *World Appl. Sci. J.* **2008**, *3*, 88–94.
87. Cordeau, J.F.; Gendreau, M.; Hertz, A.; Laporte, G.; Sormany, J.S. New heuristics for the vehicle routing problem. *Logist. Syst. Des. Optim.* **2005**, *9*, 279–297. [CrossRef]
88. Ryan, M.R. Exploiting subgraph structure in multi-robot path planning. *J. Artif. Intell. Res.* **2008**, *31*, 497–542. [CrossRef]
89. Yu, H.; Chi, C.J.; Su, T.; Bi, Q. Hybrid evolutionary motion planning using follow boundary repair for mobile robots. *J. Syst. Archit.* **2001**, *47*, 635–647. [CrossRef]
90. Hassan, A.; Shahid, M.; Hayat, F.; Arshad, J.; Jaffery, M.H.; Rehman, A.U.; Ullah, K.; Hussen, S.; Hamam, H. Improving the Survival Time of Multiagents in Social Dilemmas through Neurotransmitter-Based Deep Q-Learning Model of Emotions. *J. Healthc. Eng.* **2022**, *2022*, 3449433. [CrossRef]
91. Khalid, A.; Jaffery, M.H.; Javed, M.Y.; Yousaf, A.; Arshad, J.; Ur Rehman, A.; Haider, A.; Althobaiti, M.M.; Shafiq, M.; Hamam, H. Performance Analysis of Mars-Powered Descent-based Landing in a Constrained Optimization Control Framework. *Energies* **2021**, *14*, 8493. [CrossRef]
92. Haider, S.K.; Jiang, A.; Almogren, A.; Rehman, A.U.; Ahmed, A.; Khan, W.U.; Hamam, H. Energy Efficient UAV Flight Path Model for Cluster Head Selection in Next-Generation Wireless Sensor Networks. *Sensors* **2021**, *21*, 8445. [CrossRef]
93. Tlili, T.; Harzi, M.; Krichen, S. Swarm-based approach for solving the ambulance routing problem. *Procedia Comput. Sci.* **2017**, *112*, 350–357. [CrossRef]
94. Kim, S.M.; Peña, M.I.; Moll, M.; Bennett, G.M.; Kavraki, L.E. A review of parameters and heuristics for guiding metabolic pathfinding. *J. Cheminform.* **2017**, *9*, 51. [CrossRef]
95. Ballesteros, S.; Pedro, P.; Escobar, Z.; Antonio, H. Description of the classification of publications and the models used in solving of the vehicle routing problem with pickup and delivery. *Rev. Ing. Univ. Medellín* **2016**, *15*, 287–306. [CrossRef]