An Extension of Pathfinding Algorithms for Randomly Determined Speeds

Visvam K. Rajesh
Student
Hunterdon Central Regional High School
Flemington, NJ USA
vrajesh@hcrhs.org

Chase Q. Wu
Department of Computer Science
New Jersey Institute of Technology
Newark, NJ USA
chase.wu@njit.edu

***Abstract*—Pathfinding is the search of an optimal path between two points on a graph. This paper investigates the performance of pathfinding algorithms within 3D voxel environments, focusing on optimizing paths for both time and distance. Utilizing computer simulations in Unreal Engine 5, four algorithms – A*, Dijkstra's algorithm, Dijkstra's algorithm with speed consideration, and a novel adaptation referred to as Time* – are tested across various environment sizes. Results indicate that while Time* exhibits a longer execution time than A*, it significantly outperforms all other algorithms in traversal time optimization. Despite slightly longer path lengths, Time* is able to compute more efficient paths. Statistical analysis of the results suggests consistent performance of Time* across trials. Implications highlight the significance of speed-based pathfinding algorithms in practical applications and suggest further research into optimizing algorithms for variable speed environments.**

***Index Terms*—Pathfinding, Dijkstra's Algorithm, A*, Voxel**

I. Introduction

In pathfinding scenarios, the world is often represented as a 2-dimensional graph, in which an agent traverses across nodes connected by edges. This concept of representing environments as graphs is known as graph theory [1]. Common pathfinding algorithms find the best path between two nodes by optimizing the distances between nodes (represented by weights on each edge between the nodes) [13]. Previous studies have attempted to address various areas in regards to pathfinding such as environmental representation, computational optimization, and heuristic approaches. When it comes to evaluating the efficacy of various algorithms, three factors are commonly considered, i.e., path cost, memory consumption (the required amount of memory to find a path), and execution time (total time for an algorithm to find a path) [16].

Many studies have used the graph theory model to represent the world but have also taken into consideration the voxel model. This model discretizes 3D space into cubic "voxels", of which each one's center acts as the location of the node in the graph. This model allows for more dynamic representation of 3D space and acts as a reduction from a more complex environment into a much simpler one, allowing for expanded processing capabilities.

In this study, we investigate a pathfinding problem with regard to traversal time, in which the speed across nodes in the environment is subject to dynamic changes over time. We aim to optimize traversal time due to its application in real-world scenarios and its relevance to the factor of speed. Due to the nondeterministic nature of this problem, there does not exist a polynomial-time optimal solution to solve such a problem. As such, a heuristic approach is required to create an effective solution in polynomial time.

We propose a heuristic approach to solve this problem by taking into consideration the Euclidean distance from the goal and the mean randomly determined speed from the goal. In this study, we explore the efficacy of traversal time as a metric and a heuristic cost function in 3D voxel space.

The contributions of our research are summarized as follows:

- a rigorous formulation of a constrained pathfinding problem with time-varying speeds;
- design of a pathfinding algorithm adapted from A* in 3D voxel space; and
- superior performance over existing approaches through extensive experiments.

II. Related Work

We conduct a brief survey of work related to pathfinding in various environments.

A. Advancement of Pathfinding

The origin of most modern pathfinding algorithms comes from Dijkstra's algorithm. In this algorithm, all nodes in a graph are checked to calculate the distance between them to find the path of minimum distance [8]. One famous use of Dijkstra's algorithm is in NASA's Perseverance rover, where it used the Enhanced Navigation (ENav) library. Their variation of Dijkstra's algorithm is called the Approximate Clearance Evaluation (ACE) algorithm. The algorithm develops a costmap by analyzing the terrain, where each cell in the costmap has a cost of the weighted sum of tilt, roughness, and minimum time needed to traverse a cell [2]. A common weakness of Dijkstra's algorithm is the fact that it cannot handle negative weights on edges. This led to the development of the A* ("A-Star") algorithm, one of the most popular pathfinding algorithms that handles this weakness, while improving upon the actual pathfinding performance. It

accomplishes this by using a heuristic to estimate the cost from the start to the end of a path [28]. A* has been one of the more popular adaptations of Dijkstra's algorithm, used by researchers worldwide. It has the advantage of analyzing its surroundings before committing to a path. It accomplishes this by using a heuristic function to calculate the cost of each node, allowing the agent to rank the nodes around it [13]:

$$f(n) = g(n) + h(n), \tag{1}$$

where function g represents the total cost between the current node n and the starting node, function h represents the total cost between the current node n and the ending node, and function f represents the total cost of node n. This value $f(n)$ is calculated for each possible node around n and then used to rank all of them to find the most effective path [13]. This allows for much quicker and more accurate pathfinding as there is less backtracking necessary when calculating the distance between two nodes [10]. The A* algorithm can be adapted for multiple environments. For example, in [14], the A* algorithm is modified for a sphere-shaped environment by creating sphere-shaped borders around obstacles to standardize their pathfinding environment. They also modified the algorithm to handle dynamic changes in environments, such as new obstacles. The most common use of the A* algorithm and its derivatives is in modern computer games. One example is the video game Age of Empires, where military units move on a 256×256 grid. The A* algorithm can be used to determine the movement of military units. There are various avenues by which A* can be expanded upon: the representation of the environment, the heuristic function of the A* algorithm, the use of memory by the A* algorithm, and the data structure by which the information about the nodes is stored [6].

B. Voxel-Based Environments

Voxel-based environments operate on the voxel model, one of the five fundamental ways of describing 3D environments as described by [20]. Each voxel is a small cube of uniform size. An advantage of the voxel model is its simplicity such that the environment can be treated as an image with an extra dimension. In images, each unit square is called a pixel, whereas in these environments, each unit cube is called a voxel. As the number of voxels increases, so does the quality of the environment [11]. In [3], a virtual environment of a 12×12 array is considered with the perimeter being "solid" and the inner portion having a random assortment of "solid" cells, with all other cells being considered "empty". This array would be translated into a 3D maze, in which the agent would use vision input to create a path through the maze, with each of these cells being considered a voxel, as shown in Fig. 1.

Another example of voxel-based environments can be seen in the video game Warframe, which has 44 publicly available voxel-grid maps. These maps can be traversed using the A* algorithm by considering each voxel as a node in the graph used by A* [21]. A challenge brought by this is the ability to move between height layers. The work in [26] overcame this challenge by selecting all of the ground voxels, setting them up as the nodes in the graph, and setting the differences in height as the weights of the edges. By viewing the voxel-based environment in this manner, the A* heuristic can be easily adapted to work in a 3D voxel space.

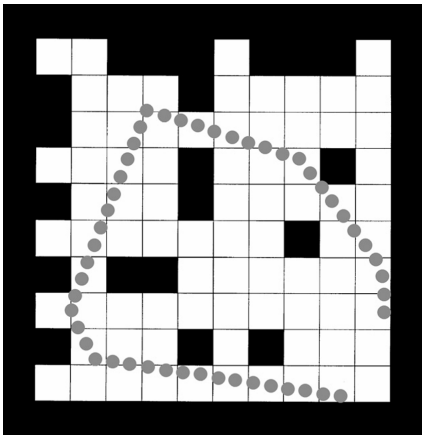

Fig. 1. Movement of the animat through its environment after training as illustrated in [3].

From a navigation perspective, a dilation is often performed on a voxel environment to remove any obstacles to make pathfinding easier. For example, this gets the navigable floor space that is adapted into a graph for a pathfinding algorithm to use [12]. This primary voxel model of a real-world space can be accomplished using laser scans to form a point cloud [12], allowing an agent trained for a virtual voxel environment to work in real-world spaces as well. This method of using voxel-based environments to accomplish real-time navigation in real-world environments is used by [19] to navigate a humanoid robot through a 3D environment. They used depth cameras to form their voxel-based environment and the A* algorithm to pathfind throughout it.

A major use of pathfinding in the real world is in Unmanned Autonomous Vehicles (UAVs). "Crusher", an unmanned ground vehicle developed by [4], uses laser detection and ranging (LADAR) analysis to analyze its surroundings and build a voxel map. Another application of the voxel model is quickly recording objects within an environment. For example, [27] used the voxel model to identify vegetation within urban environments and record the state of said vegetation. With the variety of research on the voxel model, various avenues surface that future studies should expand upon: map representations, path techniques, grid techniques, etc. [5].

C. Problem and Gap

There is an empirical gap in the prior research due to a lack of rigorous research within navigation in 3D voxel environments. Previous research has addressed several aspects of navigation, such as spherical environments [14], small 3D mazes [3], and real-world terrain [4] [2]. Further research has been conducted on video games, such as object recognition in arcade games [18], and 3D voxel-based games like Warframe [21]. Extending research into 3D voxel-based environments

would prove useful as the variability and simplicity of such an environment would allow for more complex model building for the real world [11].

In order to adapt to the constraints of the real world, it is important to consider agent speed and traversal time. However, the study done by [21] used the standard A* algorithm to pathfind through voxel-based environments, failing to consider the factor of agent speed. Thus, we include a random speed component on every voxel within the environments used for our study to simulate the various speed restrictions in the real world. Under these conditions we raise the question, how can graph-based pathfinding algorithms perform within a 3D voxel-based virtual environment in which speed is randomly determined? This study develops a heuristic to find the path that minimizes time and distance, examining multiple node-based algorithms–A* and Dijkstra–within voxel-based environments, determining the navigational capabilities of such models within a closed testing environment, and comparing them against two adaptations considering speed and traversal time.

III. Problem Formulation

The environment for this research uses 3D voxel space to represent a 3D environment on which an agent will traverse upon. The agent may only traverse this space given certain constraints, such as speed and terrain height. In this space, each surface voxel is provided an x, y, and z coordinate where the x, y coordinates represent horizontal movement and the z coordinate represents terrain height and vertical movement. Traversing upwards will slow the speed of the agent and traversing downwards will increase the speed of the agent.

A. Environment

Given a graph $G = (V, E)$ where there are n nodes and m edges, and $V = \{v_0, \ldots, v_n \mid v \in \mathbb{R}^3\}$ is the nodes representing the center of each voxel, where $v_n = \left[\begin{smallmatrix} x \\ y \\ z \end{smallmatrix}\right] \forall n$ and $E = \{e_0, \ldots, e_n\}$. Each node v_n has an associated randomly determined speed returned by the function $s(n)$ where n is a given node, which randomly varies from 0.5 to 3.0 voxels per millisecond. This range represents a realistic estimation of speed limits where some routes may have lower speed restrictions than others. Each node will be represented by a voxel in 3D space, where each voxel maintains the same size. As such the edges connecting these nodes, connecting the center of two adjacent voxels that share the same face, represent the distance to traverse between nodes. When traversing across nodes of different heights, $s(n) \times \frac{\Delta z}{10}$ is added to $s(n)$ to handle the factor of gravity when climbing up or down. The constant of 10 was chosen arbitrarily to handle the variation in real-world environments, forcing our solution to adapt to varying conditions, and making it more robust.

B. Objectives

Given a starting node v_s and a goal node v_g in a 3D voxel environment, we wish to compute the optimal path P from v_s to v_g, which minimizes both the path length $|P|$ and the overall traversal time represented by the cost function $C(P)$, i.e., $\min |P|$ and $\min C(P)$. The cost function is calculated as:

$$C(P) = \sum_{i=1}^{n} \left(s(i) + \left(s(i) * \frac{\Delta z}{10} \right) \right) * d(i), \qquad (2)$$

where $d(i)$ represents the Euclidean distance function, taking the distance between i and $i-1$:

C. Constraints

We aim to achieve the above objectives under the following constraints: traversal may occur in 3 dimensions octally via adjacent nodes, but the agent cannot traverse between nodes if the difference in their z coordinates (Δz) is greater than 2. This represents a difference in height too steep for any real-world agent to traverse. A path $P \subset V$ from v_s to v_g must represent an open walk in the graph, meaning that $v_s \neq v_g$. Also, $|P| \leq |V|$.

TABLE I
Mathematical notations used in the problem formulation.

Notation	Description
$G = (V, E)$	Graph with node set V and edge set E
n	Number of nodes
m	Number of edges
$v_n = \left[\begin{smallmatrix} x \\ y \\ z \end{smallmatrix}\right]$	Coordinates of node n
$s(n)$	Speed associated with node n, varying from 0.5 to 3.0 voxels per millisecond
$d(n)$	Euclidean distance between two nodes, n and $n-1$, represented by the equation, $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$
Δz	Difference in z coordinates between two nodes, n and $n-1$
P	Path from v_s (starting node) to v_g (goal node)
$C(P)$	Cost function representing the total traversal time

IV. Methodology

To gauge the effectiveness of the algorithms in our study, their performance must be measured by the time taken to reach the goal node. This can be accomplished through experimentation in computer simulation. Experimental research aims to investigate cause-and-effect relationships. Within experimental research design, we must consider many factors that may influence a particular phenomenon [24]. An example would be comparing a computer vision-based pathfinding algorithm against an A* benchmark [7].

A. Our Approach

We propose to develop a heuristic function to optimize traversal time, in which the time-varying speed is estimated by taking the average speed between an adjacent node and the current node (the "edge traversal time") and the average speed between the current node and the goal node (the "heuristic traversal time"). We also consider the change in elevation between two nodes in each speed calculation. We consider four algorithms in this study: Dijkstra's algorithm, due to its

primary influence in the pathfinding field, A*, due to its use of heuristics when finding a path, along with two algorithms that adapt A* and Dijkstra to consider time. These algorithms are tested and evaluated via computer simulations, which are able to quickly run multiple pathfinding scenarios within short periods. During these simulations, the environment is randomly varied across each trial to draw statistically meaning conclusions about the performance.

*B. Time**

In our solution, referred to as Time* ("Time-Star"), we expand upon the A* heuristic function by including our estimation of speed in cost calculations. Our new cost function takes into consideration the edge traversal time and the heuristic traversal time to the goal node. For a given node n, the cost function $c(n)$ is represented as:

$$c(n) = c(n_{\text{-}1}) + t(n_{\text{-}1}, n) + h(n_{\text{-}1}, n), \tag{3}$$

where $n_{\text{-}1}$ is the previous node within the path, $t(n_{\text{-}1}, n)$ is the edge traversal time, and $h(n_{\text{-}1}, n)$ is the heuristic traversal time to the goal node. These functions are represented respectively as:

$$t(n_1, n_2) = \frac{d(n_1, n_2)}{e(n_1, n_2)}, \tag{4}$$

$$h(n_1, n_2) = \frac{d(n_2, n_{\text{goal}})}{e(n_1, n_2)}, \tag{5}$$

where d is the Euclidean distance between two nodes and e is the speed of traversal between two given nodes. These functions are represented respectively as follows:

$$d(n_1, n_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}, \tag{6}$$

$$e(n_1, n_2) = \frac{u(n_1) + u(n_2)}{2}, \tag{7}$$

where $u(n)$ is a function to estimate the randomly determined speed, represented as:

$$u(n) = s(n) + \left(s(n) * \frac{\Delta z}{10} \right), \tag{8}$$

where $s(n)$ is the randomly determined speed of the given node n, and Δz is that node's elevation change, calculated by taking the difference between each node's z coordinate. By modifying the A* heuristic function to accommodate the random speeds, the standard A* procedures are used to find a path from start to finish.

The algorithm consists of two procedures, "Find Path" and "Get Best Neighbor". "Find Path" consists of an overall search loop, which relies on "Get Best Neighbor" to calculate the optimal node based on the constraints and cost function of the algorithm. "Get Best Neighbor" searches all adjacent nodes and uses the cost function as described to find the best node. "Find Path" compiles these nodes into an array, which is returned as the computed path.

Algorithm 1 Find Path Time*

```
procedure FindPath(Start, End, size, graph)
    Path ← []                                   ▷ Initialize an empty path
    OpenList ← graph                            ▷ Initialize OpenList with graph nodes
    CurrNode ← Start                            ▷ Start with the starting node
    Path.append(Start.Pos)                      ▷ Add starting position to path
    lastPoint ← Path[Path.Num − 1]
    PrevNode ← OpenList[lastPoint]
    while OpenList is not empty do
        BestNode ← GetBestNeighbor(CurrNode, size, OpenList)
        if BestNode ≠ nullptr then
            OpenList.remove(BestNode.Pos)
            if BestNode.Pos == End.Pos then
                Path.append(End.Pos)            ▷ Add end position to path
                break                           ▷ Path found, exit loop
            PrevNode ← CurrNode
            Path.append(BestNode.Pos)           ▷ Add best node position to path
            CurrNode ← BestNode                 ▷ Update selected node
        else
            if CurrNode is not an edge node then
                currPoint ← CurrNode.Pos
                OpenList.add(PrevNode.Pos)
                Path.remove(PrevNode.Pos)
                PossiblePaths ← All possible nodes around currPoint
                for i ← 0 to PossiblePaths.Num do
                    point ← PossiblePaths[i]
                    if (point ≠ currPoint) and point is not in Path then
                        if OpenList does not contain point then
                            OpenList.add(point, graph[point])
                        OpenList[point].hasVisited ← false
                CurrNode ← PrevNode
                PrevNode ← graph[Path[Path.Num − 1]]
            break                               ▷ Exit loop on failure
    return Path                                 ▷ Return the path
```

Algorithm 2 Get Best Neighbor Time*

```
procedure GetBestNeighbor(node, size, nodes)
    MinX ← node.Pos.X
    MaxX ← node.Pos.X + 1
    MinY ← node.Pos.Y − 1
    MaxY ← node.Pos.Y + 1
    MaxZ ← node.Pos.Z + 2
    BestNode ← nullptr
    if node.Pos.X < 0 or node.Pos.X ≥ size or
       node.Pos.Y < 0 or node.Pos.Y ≥ size or
       node.Pos.Z < 0 or node.Pos.Z ≥ size then
        return NULL
    for x ← MinX to MaxX do
        if x ≥ 0 then
            for y ← MinY to MaxY do
                if y ≥ 0 then
                    point ← FVector2D(x, y)
                    if point ≠ FVector2D{node.Pos}
                    then
                        if nodes.Contains(point) then
                            if nodes[point].Pos.Z ≤
                            MaxZ and
                             not nodes[point].hasVisited
                            then
                                nodes[point].cost ←
                                Cost(node, nodes[point])
                                if BestNode == nullptr
                                then
                                    nodes[point].hasVisited ←
                                    true
                                    BestNode ←
                                    nodes[point]
                                else if nodes[point].cost ≤
                                BestNode.cost then
                                    nodes[point].hasVisited ←
                                    true
                                    BestNode ←
                                    nodes[point]
    return BestNode                ▷ Return the best node
```

Algorithm 3 Find Path Dijkstra-Time

```
procedure FindPath(Start, End, size, graph)
    Path ← []                      ▷ Initialize an empty path
    OpenList ← graph ▷ Initialize OpenList with graph
    nodes
    CurrNode ← Start    ▷ Start with the starting node
    Path.append(Start.Pos) ▷ Add starting position to
    path
    while OpenList is not empty do
        BestNode ←
        GetBestNeighbor(CurrNode, size, OpenList)
        if BestNode ≠ nullptr then
            OpenList.remove(BestNode.Pos)
            if BestNode.Pos == End.Pos then
                Path.append(End.Pos) ▷ Add end posi-
                tion to path
                break              ▷ Path found, exit loop
            Path.append(BestNode.Pos) ▷ Add best
            node position to path
            CurrNode ← BestNode ▷ Update selected
            node
        else
            if CurrNode is not an edge node then
                currPoint ← CurrNode.Pos
                OpenList.add(PrevNode.Pos)
                Path.remove(PrevNode.Pos)
                PossiblePaths                ←
                All possible nodes around currPoint
                for i ← 0 to PossiblePaths.Num do
                    point ← PossiblePaths[i]
                    if (point ≠ currPoint)  and  point
                    is  not  in  Path then
                        if OpenList does not contain point
                        then
                            OpenList.add(point, graph[point])
                        OpenList[point].hasVisited ←
                        false
                CurrNode ← PrevNode
                PrevNode ← graph[Path[Path.Num −
                1]]
            break                  ▷ Exit loop on failure
    return Path                    ▷ Return the path
```

C. Dijkstra-Time

A more iterative approach is used for the Dijkstra-based algorithm, called Dijkstra-Time, where there is no heuristic function, instead identifying the best path via edge traversal time $t(n_{-1}, n)$. Unlike the original Dijkstra's algorithm, which uses edge weights, Dijkstra-Time focuses on edge traversal time. It then calculates distance iteratively and factors it in place of a heuristic function. As it explores the graph, it updates the tentative distance for each node based on the traversal time. Since this algorithm factors in the traversal time when determining the shortest path, it ensures that the chosen path minimizes the traversal time. The following is the pseudocode used for this adaptation.

Algorithm 4 Get Best Neighbor Dijkstra-Time

```
procedure GetBestNeighbor(node, size, nodes)
    MinX ← node.Pos.X
    MaxX ← node.Pos.X + 1
    MinY ← node.Pos.Y − 1
    MaxY ← node.Pos.Y + 1
    MaxZ ← node.Pos.Z + 2
    BestNode ← nullptr
    if node.Pos.X < 0 or node.Pos.X ≥ size or
       node.Pos.Y < 0 or node.Pos.Y ≥ size or
       node.Pos.Z < 0 or node.Pos.Z ≥ size then
        return NULL
    for x ← MinX to MaxX do
        if x ≥ 0 then
            for y ← MinY to MaxY do
                if y ≥ 0 then
                    point ← FVector2D(x, y)
                    if point ≠ FVector2D{node.Pos}
                    then
                        if nodes.Contains(point) then
                            if nodes[point].Pos.Z ≤
                            MaxZ and
                             not nodes[point].hasVisited
                            then
                                nodes[point].cost ←
                                Cost(node, nodes[point])
                                if BestNode == nullptr
                                then
                                    nodes[point].hasVisited ←
                                    true
                                    BestNode ←
                                    nodes[point]
                                else if nodes[point].cost ≤
                                BestNode.cost then
                                    nodes[point].hasVisited ←
                                    true
                                    BestNode ←
                                    nodes[point]
    return BestNode                ▷ Return the best node
```

V. Implementation and Performance Evaluation

A. Unreal Engine 5 Environment

For the voxel-based environment, we create a grid of nodes, with each node containing, x, y, and z coordinate information along with a randomly determined speed that ranges from 0.5 voxels per millisecond to 3.0 voxels per millisecond. For consistency, traversal time is calculated by adding up the individual speeds of the nodes involved in a given path, along with the height difference factor, multiplied by the individual edge distances for all algorithms. This grid is represented in code as a map, where $\begin{bmatrix} x \\ y \end{bmatrix}$ is the key and each node is a value. This map is used for analyzing the environment and finding the most optimal path. Each node object contains a vector containing position data, $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$, and a function to return the randomly determined speed, $s(n)$.

The environment is developed using Unreal Engine 5, one of the most commonly used 3D game engines featuring procedural mesh generation, allowing for randomly generated environments [26]. Unreal Engine 5, being a game engine, is commonly used for creating video games, but its technologies have been applied to other purposes, such as cinema and, in the case of this study, simulation [9]. The actual environment is generated randomly using the *FastNoiseGenerator* plug-in, using a Perlin noise map. The Perlin noise map is a noise function that can generate more natural gradients, making it useful for terrain generation [23]. The purpose of Perlin noise within this study is to make sure that the generated terrain is realistic, allowing for the most effective algorithms to be used within the real world. This of course means that the generated terrain is not truly random, instead simulating realistic terrain. An example of this can be seen in Fig. 2.

Furthermore, this study uses quintic interpolation to smoothen out noise values in an attempt to maintain realistic terrain. Within these calculations, both Euclidean and Manhattan distance functions are considered within cellular noise calculations to create curved cell boundaries, but the Euclidean distance function is selected as the distance is being calculated via two distinct points, to find the shortest and most direct path [15]. The noise generator uses the following parameters for all instances of terrain generation: 0.15f for frequency, 5 octaves, 5.0f lacunarity, 0.5 gain, and 0.45f cellular jitter. All tests are performed on an ASUS ROG Strix G15 2022 Gaming Laptop, using an AMD Ryzen 7 6800H processor, NVIDIA GeForce RTX 3050 graphics card, and 16 gigabytes of DDR5 RAM. Due to the extensive computational power required to generate environments, the 512 and 1024 grid-size environments are tested on an Amazon EC2 G4dn Extra Large Windows instance, using an NVIDIA T4 GPU, 16 GB RAM, and 4 vCPUs.

Due to the limited access to computational resources, in this study, we generate each environment as a 16×16 $times$ 16 voxel space. Within each trial, the agent is required to find a path from the bottom-left corner to the top-right corner of this space, represented by the coordinates $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 16 \\ 16 \end{bmatrix}$ respectively. Such trials would be repeated for 32 x 32 x 16, 36 x 36 x 16, 64 x 64 x 16, 250 x 250 x 16, 252 x 252 x 16, 512 x 512 x 16, and 1024 x 1024 x 16 spaces. In total, 100 environments are generated per grid size, and each algorithm makes one attempt to pathfind through each environment. The standard environment height of 16 is selected to complement the selection of the constant 2 as the maximum height difference between nodes.

First, using Unreal Engine 5 and the *FastNoiseGenerator* plug-in, the environment is generated at runtime, where each algorithm computes a path from a random point A to a random point B on the terrain. This environment is represented as a map, where a 2-dimensional vector is the key value, used to locate the node at a specific location in the environment. This map would be accessed by the algorithm to pathfind through the environment. The time taken to compute, time taken to traverse, path size, and total path cost are all measured and

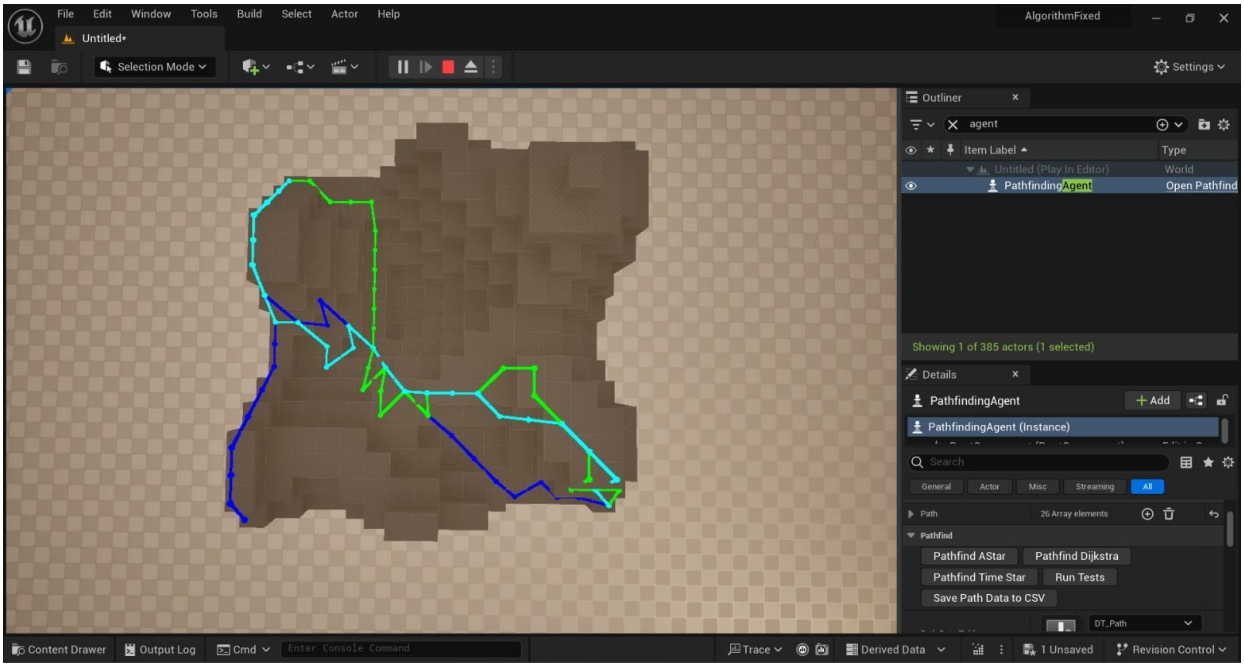

Fig. 2. Environment development in Unreal Engine 5. This figure illustrates an example of the environment the algorithms would be pathfinding on. The environment shown specifically is a 16 x 16 x 16 voxel space, with terrain generated by the Perlin noise algorithm [23]. On the sidebar are buttons used to automate the data collection process.

recorded in an Unreal Engine 5 DataTable, and then exported to a Comma-Separated-Value (CSV) file.

Python is a popular programming language used often with tools such as Pandas, Matplotlib, and Seaborn for data analysis. Pandas allows for CSV files to be read for data analysis and Matplotlib and Seaborn are used to generate graphs. The CSV file–containing computational time, traversal time, path length, and total path cost–exported from the Unreal Engine 5 DataTable is read into a Pandas Dataframe for data analysis. Using Pandas, we then take each metric's mean and standard deviation for each algorithm and compare them. Such values are calculated for each environment size and then plotted using Matplotlib and Seaborn for ease of comparison. The algorithm that takes the least amount of time on average to traverse is considered the most effective algorithm. We also look into possible outliers within the data to consider in our conclusions.

B. Results

The trial data are used to generate graphs that plot the mean and standard deviation of the recorded metrics against environment size. All algorithms tested are plotted in the same graph for ease of comparison. The horizontal axis represents the environment size, which is the number of voxels on each side of the environment. The vertical axis represents the mean and standard deviation of the recorded metrics: Time taken, time calculated, path size, and total cost. For data analysis, outliers are eliminated, which for our study represent data beyond the 95th percentile and data below the 5th percentile. From the data, it is evident that while our adaptation of the A* algorithm takes more time to compute a path, the individual paths generated optimize for time performance better than any other algorithm tested. It is also clear that all algorithms struggle to perform on a 1024 x 1024 environment size.

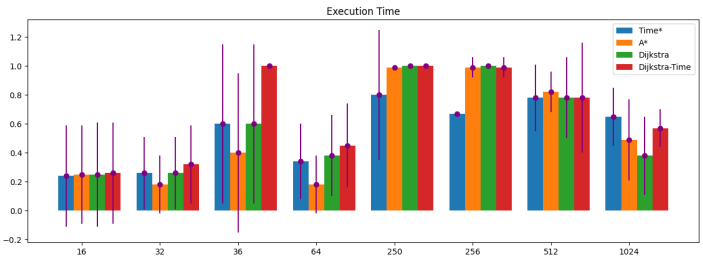

Fig. 3. Time taken by each algorithm

1) Execution Time: Fig. 3 shows the mean execution time taken by each algorithm across different problem sizes. Regarding execution time, Time* takes longer to compute a path than A* but performs better than Dijkstra and Dijkstra-Time. On the other hand, the standard deviation for Time* is generally lower than both Dijkstra algorithms. This could have occurred due to Time*'s additional speed considerations and the fact that A* uses an optimized heuristic to compute a path. A* has the shortest execution time, due to its optimized heuristic function.

2) Traversal Time: Fig. 4 shows the mean traversal time by each algorithm across different problem sizes. In terms of traversal time, Time* proves to be more effective than any of the other algorithms tested. Time* consistently achieves the lowest mean traversal time through the different environment

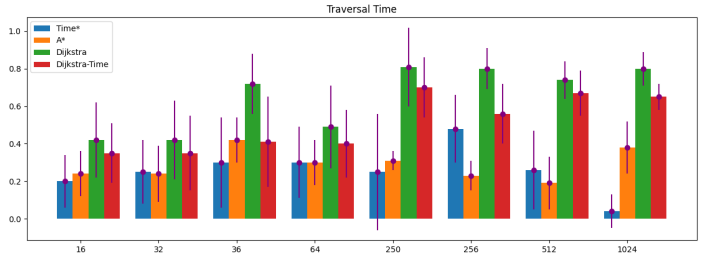

Fig. 4. Time to traverse taken by each algorithm on each environment scale.

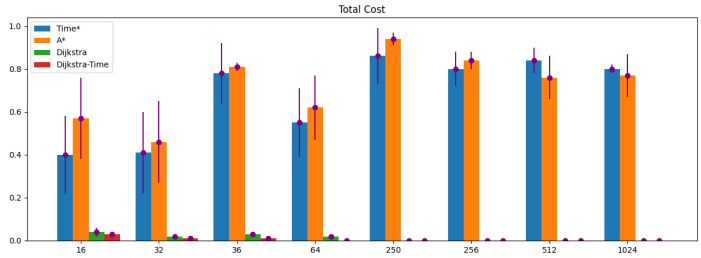

Fig. 6. Costs of each algorithm on each environment scale.

scales, proving that our heuristic function with speed consideration effectively optimizes paths for traversal time. A* is close behind Time*, with the Dijkstra-based algorithms following suit. Interestingly, the Dijkstra-Time algorithm performs worse than the original Dijkstra's algorithm in all of the other metrics tested, whereas it accomplishes optimizing the traversal time of the path much better. This could likely be attributed to the additional computational cost of considering the randomly determined speed within the environment, resulting in longer computation times and longer paths.

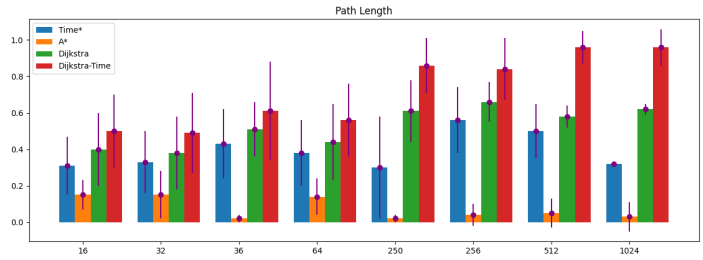

Fig. 5. Number of nodes traversed by each environment and algorithm.

3) Path Size: Fig. 5 shows the mean path size by each algorithm across different problem sizes. Regarding path size, Time* is not able to have the shortest path, which is accomplished by A*, but this is likely due to the additional speed consideration. This consideration causes a path that is not necessarily the shortest in distance, but the quickest to traverse. The Dijkstra-based algorithms both consistently end up in the last place across all of the metrics measured, likely due to their non-optimized cost function. Dijkstra-Time also has the greatest standard deviation, showing the lack of scalability throughout the various environments and different scales of environments. This further exemplifies the effectiveness of A*-based solutions compared to Dijkstra-based solutions.

4) Traversal Cost: Fig. 6 shows the mean traversal cost by each algorithm across different problem sizes. When looking at path cost, it is difficult to fairly compare the Dijkstra-based and A*-based algorithms against each other as both types of algorithms calculate cost differently. Understanding this limitation, we can conclude based on the data that the speed considerations added onto these algorithms allow for decreased path cost, indicating that the resulting paths successfully optimize all of their goal metrics, which for this study are distance and time, with a greater focus on time.

C. Analysis

From the data collected, it is clear that Time* can accomplish its goal of finding the quickest path between two points. There needs further analysis specifically for grid sizes greater than 500 as from the data all algorithms seem to perform worse on those environment scales. Furthermore, these environments need to be tested on much larger scales, such as 1500 x 1500.

VI. Discussion

In the experiments, we investigate the performance metrics of various algorithms on different environment sizes. Our trial data allow for the generation of graphs illustrating the mean and standard deviation of recorded metrics including computational time, traversal time, total path cost, and path size. Conveniently, all algorithms are plotted on the same graph for ease of comparison. From the 800 simulations conducted, it is clear that Time* performs much better than its parent algorithm, A*. It also performs much better than Dijkstra's algorithm and better than Dijkstra-Time. It is also clear that our heuristic function effectively optimizes for traversal time, compared with other algorithms, allowing it to perform far more effectively.

A. Computational Time

The Time* algorithm exhibits longer execution times compared with the actual A* algorithm. This is indicative of the additional time complexity required to handle the random speed component of the environment at hand. Both Dijkstra-Time and Dijkstra's algorithm have much longer execution time. This is likely due to Dijkstra's lack of a heuristic function to optimize paths, leading to additional computation to find a path. The Time* algorithm does, however, have a lower traversal time than all of the other algorithms.

B. Traversal Time

The area that Time* is the strongest in is traversal time. This metric indicates that sacrificing some computational speed can result in more effective outcomes. A*, which has a faster execution time, could not compute an efficient path and Dijkstra's algorithm could not come close to either A* algorithm. This suggests that heuristic functions can provide significant advantages in execution speed while staying efficient. Interestingly, Dijkstra-Time comes in second, with traversal time better than the original A* algorithm, indicating

that while the algorithm may not be computationally efficient, it can compute efficient paths.

C. Path Size

Looking at the path size, A* is consistently able to optimize for the shortest path, which is consistent with its execution time performance. Comparatively, Time* has a longer path size but, consistent with other metrics, can build a more efficient path than all other algorithms, as indicated by its ability to optimize traversal time. Understandably, the shortest path by distance is not the quickest to traverse. Interestingly, the time-based Dijkstra algorithm has the longest sizes even though it could not develop the most efficient path, likely due to the lack of a heuristic function. This infers that within 3D voxel spaces, A* and its derivatives may be the most computationally effective, but if we consider the random speed component of our environment, both Time* and Dijkstra-Time can compute efficient paths. Again, likely due to the heuristic function, Time* can compute paths in less time than Dijkstra-Time.

D. Standard Deviation

Across all metrics, Time* is able to effectively minimize the standard deviation better than A* and both Dijkstra-based algorithms. This indicates that Time* can maintain consistent performance across trials as opposed to the other algorithms tested. This may be because the calculations involving the Time* cost function went far more in-depth, considering the height differences and average speeds between nodes, resulting in numerous factors holding the calculations into place. Given so many factors influencing the algorithmic performance, small changes to any given factor would not result in significant variance from the mean, indicating a low standard deviation. Another interesting note about the data is that the standard deviation seems to increase as environment size increases, indicating that the types of effective paths vary more significantly at greater sizes.

E. Implications

From the results of this study, it is clear that there are applications for speed-based pathfinding algorithms that optimize for time and distance. Previous studies have focused primarily on distance-based path optimization without significant consideration for time. In practical applications, such as video games [21], speed plays an important role in determining the feasibility of a given path. The shortest path is not necessarily the quickest path to traverse. This study has also shown the effectiveness of heuristic algorithms, specifically the A* algorithm, within 3D voxel spaces, indicating that further research into such algorithms would prove beneficial in finding the fastest path. Currently, further research is expanding such algorithms into the field of autonomous vehicles, where the A* algorithm acts as a base for more complex algorithms [25]. Such applications would still benefit from speed compensation algorithms, like the one presented in this study. Ideally, future studies should emphasize optimizing other algorithms with speed compensation in situations where speed may vary significantly, requiring an additional consideration of speed to find a path that can efficiently traverse a given environment.

VII. Conclusion

We have shown the pathfinding capabilities of four algorithms, A*, Dijkstra, Time*, and Dijkstra-Time. These algorithms were tested in a 3D voxel-based environment, on which speed was randomly determined. Each node in this environment would have a random speed associated with it, which was assigned during terrain generation. This information was used by the four algorithms to calculate the most efficient path as determined by their individual cost functions. We primarily evaluated the efficiency of their resulting paths by measuring the time it took to traverse them.

From our trials, it became apparent that Time* proved most effective in terms of optimizing traversal time, while sacrificing computational speed and path length. As a by-product of this study, we have also been able to show the effectiveness of heuristic-based algorithms, such as A* and Time*, within 3D voxel-based environments. As such, we hope that future research will expand the use of time-based heuristics, as seen in Time*, into other research areas, such as autonomous vehicles. Beyond this, the importance of heuristics within pathfinding should be explored in combination with newer technologies such as deep learning and neural networks. Research within this specific area is able to amplify pathfinding success by using deep learning to analyze various possible paths [22] or by using imitation learning to simplify computations [17]. Further research can also explore the importance of speed and traversal time as metrics for evaluation within the overall field of pathfinding and also explore other representations of real-world environments beyond just voxel-based ones. Researchers planning on expanding upon this study in-specific should strive to adapt the algorithms presented to other environments and test on much larger scales than those presented in this study.

In summary, this study has shown the effectiveness of heuristic-based pathfinding algorithms along with the application of time-based heuristics within this field. As such, we have also proven the effectiveness of Time*, for environments in which speed is randomly determined. This study has also illustrated the utility of 3D voxel-based environments and their applications within the field of pathfinding. It is clear that opportunity for further research exists via various avenues, as mentioned above, and such opportunities would prove beneficial for the field of pathfinding as a whole.

Acknowledgment

I'd like to thank my parents, who have supported me throughout this journey along with Ms. Donhauser, my wonderful AP Research teacher for teaching me the essentials of the research process. I would also like to thank my amazing mentor Dr. Wu of NJIT for assisting me in formulating the mathematics behind this research project.

REFERENCES

[1] L. Euler, "Solutio problematis ad geometriam situs pertinentis," Euler Archive - All Works, vol. 53, 1741, Available: https://scholarlycommons.pacific.edu/euler-works/53

[2] N. Abcouwer et al., "Machine Learning Based Path Planning for Improved Rover Navigation," 2021 IEEE Aerospace Conference (50100), Mar. 2021, doi: https://doi.org/10.1109/aero50100.2021.9438337.

[3] M. J. Aitkenhead and A. J. S. McDonald, "A neural network based obstacle-navigation animat in a virtual environment," Engineering Applications of Artificial Intelligence, vol. 15, no. 3–4, pp. 229–239, Jun. 2002, doi: https://doi.org/10.1016/s0952-1976(02)00043-x.

[4] J. A. Bagnell, D. Bradley, D. Silver, B. Sofman, and A. Stentz, "Learning for Autonomous Navigation," IEEE Robotics Automation Magazine, vol. 17, no. 2, pp. 74–84, Jun. 2010, doi: https://doi.org/10.1109/MRA.2010.936946.

[5] D. Brewer and N. Sturtevant, "Benchmarks for Pathfinding in 3D Voxel Space," Proceedings of the International Symposium on Combinatorial Search, vol. 9, no. 1, pp. 143–147, Sep. 2021, doi: https://doi.org/10.1609/socs.v9i1.18464.

[6] X. Cui and H. Shi, "A*-based Pathfinding in Modern Computer Games," International Journal of Computer Science and Network Security, vol. 11, no. 1, pp. 125–130, 2011, Available: https://vuir.vu.edu.au/id/eprint/8868

[7] M. Dann, Fabio Zambetta, and J. Thangarajah, "Real-Time Navigation in Classical Platform Games via Skill Reuse," Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, pp. 1582–1588, Aug. 2017, doi: https://doi.org/10.24963/ijcai.2017/219.

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, vol. 1, no. 1, pp. 269–271, Dec. 1959, doi: https://doi.org/10.1007/bf01386390.

[9] T. de Goussencourt, J. Dellac, and P. Bertolino, "A Game Engine as a Generic Platform for Real-Time Previz-on-Set in Cinema Visual Effects," HAL Archives Ouvertes, Oct. 01, 2015. https://hal.science/hal-01226184 (accessed Apr. 22, 2024).

[10] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah, and E. Gunawan, "A Systematic Literature Review of A* Pathfinding," Procedia Computer Science, vol. 179, pp. 507–514, 2021, doi: https://doi.org/10.1016/j.procs.2021.01.034.

[11] R. Goldstein, S. Breslav, and A. Khan, "Towards voxel-based algorithms for building performance simulation," in Proceedings of the IBPSA-Canada eSim Conference, 2014.

[12] B. Gorte, S. Zlatanova, and F. Fadli, "Navigation in Indoor Voxel Models," ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, vol. IV-2/W5, pp. 279–283, May 2019, doi: https://doi.org/10.5194/isprs-annals-iv-2-w5-279-2019.

[13] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100–107, 1968, doi: https://doi.org/10.1109/tssc.1968.300136.

[14] A. G. Hussein Qahtan and A.-B. Ali Emad, "Path-Planning Dynamic 3D Space Using Modified A* Algorithm," IOP Conference Series: Materials Science and Engineering, vol. 928, p. 032016, Nov. 2020, doi: https://doi.org/10.1088/1757-899x/928/3/032016.

[15] Hari Sagar Ranjitkar and S. Karki, "Comparison of A*, Euclidean and Manhattan distance using Influence map in MS. Pac-Man," Jan. 2016.

[16] Aya Kherrour, M. Robol, M. Roveri, and P. Giorgini, "Evaluating Heuristic Search Algorithms in Pathfinding: A Comprehensive Study on Performance Metrics and Domain Parameters," Electronic proceedings in theoretical computer science, vol. 391, pp. 102–112, Sep. 2023, doi: https://doi.org/10.4204/eptcs.391.12.

[17] Daniil Kirilenko, A. Andreychuk, A. I. Panov, and K. Yakovlev, "TransPath: Learning Heuristics For Grid-Based Pathfinding via Transformers," arXiv (Cornell University), Dec. 2022, doi: https://doi.org/10.48550/arxiv.2212.11730.

[18] Y. Liang, M. C. Machado, E. Talvitie, and M. Bowling, "State of the Art Control of Atari Games Using Shallow Reinforcement Learning," arXiv (Cornell University), Jan. 2015, doi: https://doi.org/10.48550/arxiv.1512.01563.

[19] D. Maier, A. Hornung, and M. Bennewitz, "Real-time navigation in 3D environments based on depth camera data," In 2012 12th IEEE-RAS International Conference on Humanoid Robots, pp. 692–697, Nov. 2012, doi: https://doi.org/10.1109/humanoids.2012.6651595.

[20] L. Meng and A. Forberg, "3D Building Generalisation," Generalisation of Geographic Information, pp. 211–231, Jan. 2007, doi: https://doi.org/10.1016/b978-008045374-3/50013-2.

[21] T. K. Nobes, D. Harabor, M. Wybrow, and S. Walsh, "Voxel Benchmarks for 3D Pathfinding: Sandstone, Descent, and Industrial Plants," Proceedings of the International Symposium on Combinatorial Search, vol. 16, no. 1, pp. 56–64, Jul. 2023, doi: https://doi.org/10.1609/socs.v16i1.27283.

[22] M. Pándy et al., "Learning Graph Search Heuristics," arXiv (Cornell University), Dec. 2022, doi: https://doi.org/10.48550/arxiv.2212.03978.

[23] K. Perlin, "An image synthesizer," ACM SIGGRAPH Computer Graphics, vol. 19, no. 3, pp. 287–296, Jul. 1985, doi: https://doi.org/10.1145/325165.325247.

[24] J. Rahman, "Experimental, Quasi-Experimental, and Ex Post Facto Designs," 2013. Available: http://www.cs.utsa.edu/ korkmaz/teaching/research-seminar/books-slides-updated-fall13/Research_seminar_Ch09.pdf

[25] W. Sheng, B. Li, and X. Zhong, "Autonomous Parking Trajectory Planning With Tiny Passages: A Combination of Multistage Hybrid A-Star Algorithm and Numerical Optimal Control," IEEE Access, vol. 9, pp. 102801–102810, 2021, doi: https://doi.org/10.1109/access.2021.3098676.

[26] G. Silva, G. Reis, and C. Grilo, "Voxel Based Pathfinding with Jumping for Games," Lecture notes in computer science, pp. 61–72, Jan. 2019, doi: https://doi.org/10.1007/978-3-030-30241-2_6.

[27] B. Wu et al., "A Voxel-Based Method for Automated Identification and Morphological Parameters Estimation of Individual Street Trees from Mobile Laser Scanning Data," Remote Sensing, vol. 5, no. 2, pp. 584–611, Feb. 2013, doi: https://doi.org/10.3390/rs5020584.

[28] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia, "Survey of Robot 3D Path Planning Algorithms," Journal of Control Science and Engineering, vol. 2016, pp. 1–22, 2016, doi: https://doi.org/10.1155/2016/7426913.