An Extension of Pathfinding Algorithms for Randomly Determined Speeds

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

Chase Q. Wu
Department of Data 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 in 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* can 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, 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) [10]. Previous studies have attempted to address various aspects of pathfinding such as environmental representation, computational optimization, and heuristic approaches. When evaluating the efficacy of algorithms, three factors are commonly considered, i.e., path cost, memory consumption, and execution time [13].

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

In this study, we investigate a pathfinding problem concerning traversal time, where the speed across nodes 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 speed. Due to the nondeterministic nature of this problem, there does not exist a polynomial-time optimal solution to solve such a problem. Thus, a heuristic approach is required to create an effective solution in polynomial time.

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

The contributions of our work 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
- Using 3D voxel space, similar to real-world topology, to evaluate algorithms.

II. Related Work

We conduct a brief survey of work in various environments.

A. Advancement of Pathfinding

The origin of most modern pathfinding algorithms comes from Dijkstra's algorithm where all nodes in a graph are checked to calculate the distance between them to find the path of minimum distance [6]. One famous use of Dijkstra's algorithm is in NASA's Perseverance rover, which 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 that it cannot handle negative weights on edges. This led to the A* ("A-Star") algorithm, one of the most popular algorithms that handles this weakness, while improving the actual pathfinding performance. It accomplishes this by using a heuristic to estimate the cost from the start to the end of a path [23]. A* has been one of the more popular adaptations of Dijkstra's algorithm, used by researchers worldwide with the advantage of analyzing its surroundings before committing to a path. It accomplishes this by using a heuristic function to calculate each node's cost, allowing the agent to rank the nodes around it [10]:

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

where function g denotes the total cost between the current node n and the starting node, function h denotes the total cost

between the current node n and the ending node, and function f denotes the total cost of node n. $f(n)$ is calculated for each possible node around n and then used to rank them to find the lowest cost path [10]. This allows for much quicker and more accurate pathfinding as there is less backtracking when calculating the distance between two nodes [8].

The A* algorithm can also be adapted for multiple environments. For example, in [11], A* 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 A* and its derivatives is in modern computer games. In the video game Age of Empires, where military units move on a 256×256 grid, A* can be used to determine the movement of military units. There are various avenues by which A* can be expanded upon: the environment representation, the heuristic function, the use of memory, and the data structure by which the information about the nodes is stored.

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 [16]. Each voxel is a cube of uniform size. An advantage of the voxel model is its simplicity as 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 [9]. 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.

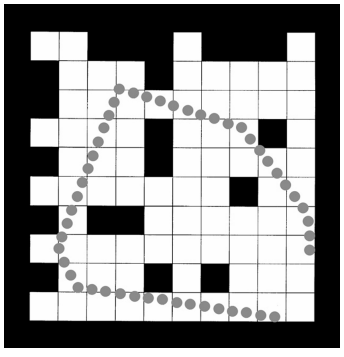

Fig. 1. "Movement of the animat through its environment after training." as illustrated in [3].

A major use of pathfinding in the real world is in Unmanned Autonomous Vehicles (UAVs). "Crusher", an unmanned ground vehicle in [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 in an environment. For example, [22] used the voxel model to identify vegetation in urban environments and record the state of said vegetation. With extensive research on this model, various avenues surface that future studies can expand upon: map representations, path techniques, and grid techniques [5].

C. Problem and Gap

There is an empirical gap in the prior research due to a lack of rigorous research in navigation in 3D voxel environments. Previous research has addressed several aspects of navigation, such as spherical environments [11], 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 [15], and 3D voxel-based games [17]. 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 [9].

To adapt to the constraints of the real world, it is important to consider agent speed and traversal time. The study done by [17] 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 in the environments used to simulate various speed restrictions in the real world. Under these conditions, we raise the question: how can graph-based pathfinding algorithms perform in a 3D voxel-based virtual environment where speed is randomly determined? This study develops a heuristic to find the path that minimizes time and distance, examining node-based algorithms, A* and Dijkstra's, in voxel-based environments, determining the navigational capabilities of such models in 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. 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. The range $[0.5, 3.0]$ voxels per millisecond is proposed as a starting point, representing an estimated range of speed limits under different conditions. However, these values can be tuned and validated through experimental testing to ensure they reflect realistic speed distributions. 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, gravity is considered by adjusting the speed. The gravity function $s(n) \times \frac{\Delta z}{\Gamma}$ is added to $s(n)$. Here,

Δz represents the height difference and Γ was an arbitrarily chosen parameter representing the influence of gravity, which was 10 for this study. This parameter can be tuned to simulate different gravitational impacts due to 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, minimizing 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:

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

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
$v_n = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$	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 adjacent nodes, i and $i-1$
Γ	Constant representing the effect of gravity on traversal.
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 in a computer simulation. In experimental research design, we aim to investigate cause-and-effect relationships and must consider many factors that may influence a particular phenomenon. An example would be comparing a computer vision-based pathfinding algorithm against an A* benchmark.

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 each A* and Dijkstra to consider time. These algorithms are tested and evaluated via computer simulations, which can quickly run multiple pathfinding scenarios in short periods. During these, the environment is randomly varied across each trial to draw statistically meaningful conclusions about the performance.

B. Time*

In our solution, referred to as Time* ("Time-Star"), we expand upon the A* heuristic function by including speed estimation 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 in 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:

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

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.

1) Theoretical Analysis

The time complexity of our Time* algorithm is primarily governed by the characteristics of A*. A* has a time complexity of $O(b^d)$, where b is the branching factor and d is the depth of the search space. While the integration of

the heuristic function that accounts for time-varying speed and elevation introduces additional computational overhead, it does not fundamentally alter the worst-case time complexity. The heuristic's role is to guide the search more efficiently, potentially reducing the number of nodes expanded, but the exponential nature of the complexity remains.

Looking at the space complexity of Time*, it is consistent with A*. Typically this is $O(|V|)$, where $|V|$ is the number of nodes, accounting for the storage of node information such as costs and parent nodes. The heuristic function, which includes additional parameters like speed and elevation, necessitates storing extra data for each node. However, this increase is minimal compared to the overall space requirements, maintaining a space complexity of $O(|V|)$.

Overall, the heuristic design in Time* is crafted to minimize traversal time by considering both speed and elevation changes. This adaptation optimizes the algorithm's performance by steering the search towards paths that, while not necessarily the shortest in distance, are likely to result in faster traversal times. By incorporating terrain elevation into the speed calculations, Time* dynamically adjusts to varying environments, enhancing its ability to find time-efficient paths in complex, uneven landscapes.

2) Procedures

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.

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 Dijkstra's algorithm, which uses edge weights, Dijkstra-Time focuses on edge traversal time, calculating distance iteratively and factoring 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.

1) Theoretical Analysis

Dijkstra's algorithm, known for its exhaustive search, has a time complexity of $O(|E| + |V| \log |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices. In the Dijkstra-Time variant, the time-varying speed and elevation adjustments add computational steps, but these additions do not alter the overall time complexity. The algorithm's nature of exploring all possible paths to find the shortest path ensures that this complexity remains consistent, even with additional considerations.

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 position 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
```

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
```

In terms of space complexity, both Dijkstra and Dijkstra-Time maintain $O(|V|)$, which accounts for storing all node and edge information. Similar to Time*, the inclusion of speed and elevation data in Dijkstra-Time necessitates storing additional information for each node. However, this increase is minimal relative to the overall memory usage, keeping the space complexity in line with the original Dijkstra's algorithm.

Overall, Dijkstra-Time adapts Dijkstra's exhaustive search approach by integrating time-varying speeds and elevation changes into cost calculation. This enhancement allows it to evaluate the true cost of traversal in environments with varying terrain. While Dijkstra is primarily focused on finding the shortest path, Dijkstra-Time is optimized to consider both distance and traversal time, leading to more practical pathfinding solutions in scenarios where time efficiency is critical.

V. Implementation and Performance Evaluation

A. Unreal Engine 5 Environment

For the voxel-based environment, we create a grid of nodes. 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 [21]. 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 [7]. The environment is generated randomly using the *FastNoiseGenerator* plug-in, and a Perlin noise map. The Perlin noise map is a noise function that generates natural gradients, making it useful for realistic terrain generation [19]. This means that the generated terrain is not truly random, instead simulating realistic terrain.

This study uses quintic interpolation to smoothen out noise values to maintain realistic terrain. Both Euclidean and Manhattan distance functions are considered in cellular noise calculations to create curved cell boundaries, but the Euclidean distance function is selected as the distance is calculated via two distinct points, finding the shortest and most direct path [12]. The noise generator uses the following parameters for all terrain generation instances: 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, we generate each environment as a $16 \times 16 \times 16$ voxel space. In 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 are repeated for $32 \times 32 \times 16$, $36 \times 36 \times 16$, $64 \times 64 \times 16$, $250 \times 250 \times 16$, $512 \times 512 \times 16$, and $1024 \times 1024 \times 16$ 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, and each algorithm computes a path across the terrain. The execution time, traversal time, path length, and total path cost are all measured and recorded in an Unreal Engine 5 DataTable, and then exported to a Comma-Separated-Value (CSV) file.

For data analysis, Python was used with Pandas, Matplotlib, and Seaborn. Pandas allows for CSV files to be read 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. Using Pandas, we take each metric's mean and standard deviation for each algorithm and compare them. Such values are calculated for each grid 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 in 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 grid size. All algorithms tested are plotted in the same graph for ease of comparison. The horizontal axis represents the grid 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 is data beyond the 95th percentile and data below the 5th percentile. From the data, it is evident that while the Time* 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 grid size.

1) Execution Time

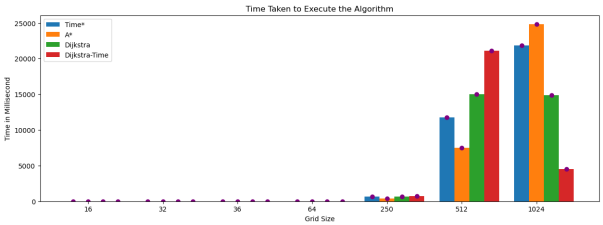

Fig. 2. Time taken by each algorithm

Fig. 2 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 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

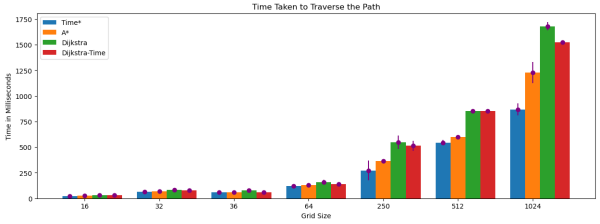

Fig. 3. Time to traverse taken by each algorithm on each environment scale.

Fig. 3 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 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, while optimizing the traversal time of the path much better. This could be attributed to the additional computational cost of considering the randomly determined speed in the environment, resulting in longer computation times and longer paths.

3) Path Size

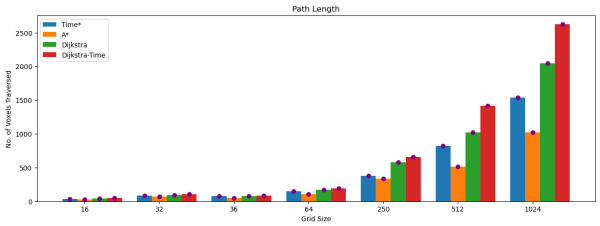

Fig. 4. Number of nodes traversed by each environment and algorithm.

Fig. 4 shows the mean path size by each algorithm across different problem sizes. Regarding path size, Time* does not have the shortest path, which is accomplished by A*, but this is likely due to the additional speed consideration, causing 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

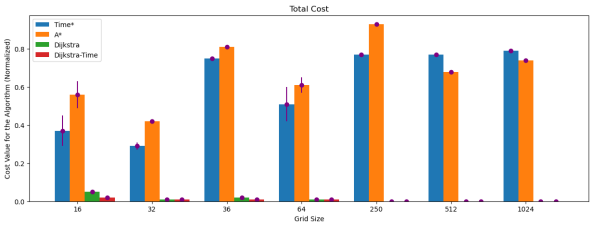

Fig. 5. Costs of each algorithm on each environment scale.

Fig. 5 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. These environments will 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 grid sizes. Our trial data allows 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 700 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. Execution 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 times. 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* or Time*. 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 in 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* can 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 grid 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 [17], speed plays an important role in determining the

feasibility of a given path. This study has also shown the effectiveness of heuristic algorithms, specifically the A* algorithm, in 3D voxel spaces, indicating that further research into such algorithms would prove beneficial. 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 [20]. 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, in 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 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. We have also been able to show the effectiveness of heuristic-based algorithms, such as A* and Time*, in 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 in pathfinding should be explored in combination with newer technologies such as deep learning and neural networks. Research in this specific area can amplify pathfinding success by using deep learning to analyze various possible paths [18] or by using imitation learning to simplify computations [14]. Further research can also explore the importance of speed and traversal time as metrics for evaluation in the overall field of pathfinding and also explore other representations of real-world environments beyond just voxel-based ones. Other aspects for future investigation include the speed range and gravity function selected in the study, which should be tuned and analyzed in comparison with the real world to find optimal values.

Acknowledgment

I would 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. Thank you to Visra Solutions, LLC for sponsoring this research.

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.

[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.

[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.

[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.

[6] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[7] 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).

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] Daniil Kirilenko, A. Andreychuk, A. I. Panov, and K. Yakovlev, "TransPath: Learning Heuristics For Grid-Based Pathfinding via Transformers," arXiv (Cornell University), Dec. 2022.

[15] 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.

[16] L. Meng and A. Forberg, "3D Building Generalisation," Generalisation of Geographic Information, pp. 211–231, Jan. 2007.

[17] 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.

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

[19] K. Perlin, "An image synthesizer," ACM SIGGRAPH Computer Graphics, vol. 19, no. 3, pp. 287–296, Jul. 1985.

[20] 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.

[21] 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.

[22] 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.

[23] 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.