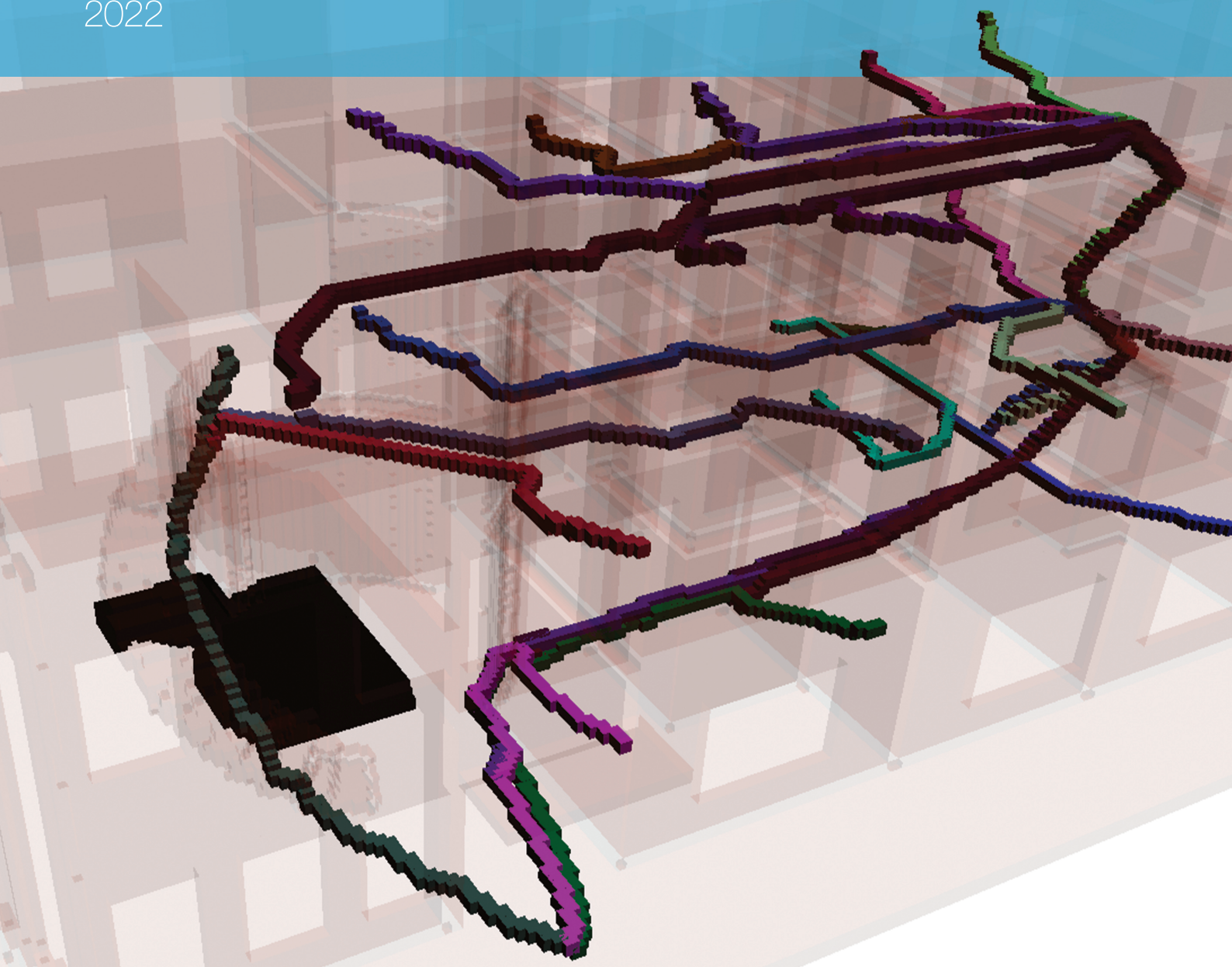
MSc Thesis in Geomatics for the Built Environment

Using voxelised spaces for the generation and visualisation of dynamic evacuation routes

Michiel de Jong

2022



TU Delft CGI

USING VOXELISED SPACES FOR THE GENERATION AND VISUALISATION OF DYNAMIC EVACUATION ROUTES

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of

Master of Science in Geomatics for the Built Environment

by

Michiel de Jong

June 2022

The work in this thesis was made in the:

GIS Technology group
Architectural Engineering + Technology
Faculty of Architecture & the Built Environment
Delft University of Technology

In cooperation with:

CGI

Supervisors:	ir. Robert Voûte
	Prof.dr.ir. Peter van Oosterom
Co-reader:	Dr.ir. Martijn Meijers

ABSTRACT

In the modern world, evacuating a building in a safe and orderly manner remains a challenge. Fire-based emergencies in a building are dynamic environments that can be simulated to better understand, analyse, and contribute to safer and smarter buildings. While different spatial representations exist, *voxels* provide a structured, flexible and efficient 3-dimensional grid for applications like analysis, classification, surface reconstruction and simulation. Furthermore, voxels implicitly contain topological and spatial relations that are relevant for 3-dimensional events such as evacuations in a rapidly changing building, with a fire spanning multiple floors.

Voxels can suffer from the problem of scale and resolution, where high resolution voxel scenes take up a lot of memory space. For this, there are solutions that make more efficient data storage for voxels possible. These include but are not limited to: the regular voxel grid, the sparse voxel octree, directed acyclic graphs and the use of space filling curves. Finding the shortest safe path for the evacuees is a challenge, especially if the area is dynamic, and there are other actors that have to share the space. Many different pathfinding algorithms exist, each with their own speciality, such as A*, any-angle pathfinding algorithms like Theta* and incremental algorithms like D*-Lite.

In this thesis, we look at whether voxelised indoor spaces can form the basis for evacuation simulations with multiple actors in a dynamic situation. We do this by comparing both the voxel data structure and pathfinding algorithm combinations in a dynamic evacuation simulation application. The comparison is done by looking at the quality of the paths, if the algorithms are able to adapt to a dynamic situation and the performance of the paths, both in computation times and memory load.

These experiments reveal that a time-aware variant of A* is able to outperform the other algorithms, when applied on a sparse Morton grid. Additionally, it shows that the use of a sparse Morton grid is preferable to implementing a full octree or the use of a non-sparse regular voxel grid for dynamic multi-actor voxel scenes. Finally, the experiments show that dynamic events can be added into pathfinding algorithms by separating walking the path from finding the path, and using a data structure that is time-aware.

ACKNOWLEDGEMENTS

I would like to thank my first mentor, Robert Voûte, for his support, and above all his input at every step of the way of this graduation project. He has helped me from the inception of this project, while also giving me a lot of personal freedom in shaping this project for myself. Next, I would like to thank professor Peter van Oosterom, who has found time to read through everything I wrote, and share his vast knowledge to help me improve my thesis. Thirdly, I would like to thank Martijn Meijers for fruitful discussions along the way and his comments as co-reader. From CGI I would like to thank Bart Staats for helping me think of this project, and the many talks we had, especially at the start of the project. I would also like to thank the other CGI interns that were always supportive and patient when listening to me ramble on about my struggles.

I would like to thank my family for supporting me during this time, and also for many of them who have had to read this thesis many times over, especially Philip. Finally, I would like to thank my closest friends for always being there for me and also allowing me to sometimes give them a lecture about voxels.

CONTENTS

ACRONYMS

1 INTRODUCTION

In this chapter, the motivation, relevance and relation to the field of Geomatics of the research conducted in this thesis will be briefly discussed. Consequently the research questions will be presented and a general overview of the thesis will be given.

1.1 MOTIVATION

In modern society, proper evacuation management in complex buildings represent a serious safety challenge. With fires being prevalent in an urban environment, and people finding it difficult to identify safe paths to exit the building [Khan et al., 2018] in a chaotic, changing and unsafe or unfamiliar environment. Evacuation modelling can help designers make decisions on how to properly manage these types of situations in the buildings they design, but also help building managers simulate how an emergency would play out in an existing building and can give insight in to how to guide the users of that building through the emergency. 3D models of indoor spaces are used to generate input for either the simulation or guidance. [Gorte et al., 2019a].

Voxels provide a structured and flexible 3D solution for the storage and analysis of spatial data. Voxelised spaces have been used extensively for the past years to preprocess, perform segmentation, classification, and reconstruction of both indoor and outdoor 3D representations of the built environment [Xu et al., 2021]. Research has shown that using voxels to automatically reconstruct and segment an indoor environment [Hübner et al., 2021] that has been generated from a point cloud and to detect the navigable space in the indoor environment [Flikweert et al., 2019] and [Xiong et al., 2017] has proven to be effective.

Additionally, because voxels provide a highly structured way to encode true 3D environments, they allow for more true-to-reality analyses in situations that are 3-dimensional, such as evacuations, where a possible fire on the ground floor will invariably affect the floors above, as well as the fact that voxels are often used in computational fluid dynamics simulations, which are a natural part of evacuation simulations due to the flow of smoke. In a digital representation of space that is not 3-dimensional, this is a spatial connection that could be missed. For instance, many pathfinding applications make use of 2-dimensional grids, and it would be logical to assume that this would work well in a building, where for instance, a floor could be converted to a 2-dimensional grid. Pathfinding could then be done on the floors and links between floors could be modelled as required. But this approach misses the point of the fundamentally 3D nature of the real world, and would need additional attention to adapt this 2D version to the 3D world. This is not impossible of course, but using voxels for this eliminates the need to perform these adaptions.

1.1.1 Scientific Relevance

While research has been done on 2D/2.5D/3D pathfinding in a static environment using voxels, such as in Koopman [2016] and Gorte et al. [2019b], Brewer and Sturtevant [2018], and Muratov and Zagarskikh [2019], this has not yet been researched in a dynamic setting. Dynamic in this sense means that the pathfinding area can

change while the paths are being calculated. As well as the fact that in Koopman [2016], there is still a pre-processing stage that extracts a search graph from the voxels, and the pathfinding is not done on the voxels itself. The method that Koopman [2016] uses, could however, be adapted to be used in a dynamic voxelised environment, by allowing the search graph to change while the paths are being computed. Like Koopman [2016], many other approaches, such as Gorte et al. [2019b] use a networked graph based on semantic information to perform hierarchical pathfinding. Hierarchical pathfinding is a pathfinding paradigm where pathfinding is performed on multiple levels, where first a general path is found between clusters, and once this coarse path has been found, the detailed path is found within the clusters and the full path is assembled.

Voxels are also a part of the field of computer graphics, where they serve as the basis of volumetric rendering (voxel rendering). Research has been done to devise ways to improve the performance of very high resolution scenes, such as Dado et al. [2016], van der Laan et al. [2020] for static voxel scenes, and Careil et al. [2020] for dynamic voxel scenes. The use of these data structures outside of computer graphics is limited. While data structures such as the sparse voxel octree (SVO) have become more mainstream, these have not yet become the standard when dealing with voxels [Aleksandrov et al., 2021]. The use of (parts) of this research could improve the performance of voxels in the field of Geomatics.

Lastly, many methods exist for pathfinding Noori and Moradi [2015]. With each algorithm having their own niche of specialised use, such as D* or HPA*, and other more general algorithms, like A* or Dijkstra's algorithm that just focus on finding the shortest path (with "shortest", being the least cost, which could be distance, time, turns etc.). Finding *the right algorithm for the right job* can be tricky. Though Noori and Moradi [2015] provides an overview of pathfinding algorithms used in gaming, these often focus on performance in terms of memory limitation, and less on the more semantically enriched representation of the digital world used in the field of Geomatics.

Thus, most of the established tools and resources in Geomatics, such as pgRouting, make use of a limited number of algorithms, mostly A* or Dijkstra. While these algorithms focus on finding the shortest path, the shortest path might not always be the safest path, depending on the cost function. It should be noted however that algorithms that are more specialised are often more complex, and that most grid-based pathfinding algorithms are usually only tested in 2D grid space, such as Koenig and Likhachev [2002] and Nash et al. [2009].

1.1.2 Societal Relevance

When a fire based emergency in a building does occur, things may not always go as the architects, or building managers intended, with many of these unpredictabilities being due to human behaviour. In Kobes et al. [2010], a psychonomic approach to fire safety is recommended, owing to the discrepancy that exists to between the assumption of fire safety and the knowledge gained from incident evaluations and experiments. The differences are visible in Table 1.1 from Kobes et al. [2010].

(Real time) indoor navigation and real time guidance could therefore be a valuable tool in the management of the built environment. Indoor navigation requires an indoor model, an indoor position, a path from origin to destination and a human interpretation of the path [Flikweert et al., 2019]. Research has shown that wayfinding and correctly interpreting direction that guide evacuees towards a safe exit is not always easy for people unfamiliar with the building, or due to a complex layout or large

size. [Kobes et al., 2010].

Point of departure or assumption in (Dutch) policy	Knowledge from incident evaluations and experiments
People who are mobile can escape without assistance	All people in a fire situation may be confronted with some degree of limitation, and are, therefore, potentially less or not at all self-reliant
People use escape route signage to find the closest exit	Incident evaluations show that in 400 cases of escape from fire, 92% of the survivors were not aware of the presence of escape route signage.
People escape via the nearest (emergency) exit	People usually escape via familiar exit routes and rarely via emergency exits. The objective walking distance does not determine the choice of route. Familiar routes are experienced as being shorter than unfamiliar routes.
People escape immediately after hearing a fire alarm bell	People (in groups) customarily ignore ambiguous cues like a fire alarm bell. People are more likely to respond to verbal cues. Social rules have a strong influence on people's (non-) reaction to cues of danger.
People's walking speed is constant, regardless of whether or not they are walking through smoke	People who are exposed to the effects of fire walk more slowly than the pace concluded from walking experiments in normal environmental conditions.

Table 1.1: Difference between policy and actual fire safety. Source: Kobes et al. [2010].

The current standard for evacuation management in large buildings consists of a paper map at key points showing the evacuation route(s), combined with emergency lighting showing where the emergency exits are, with many rules and regulations determining visibility and capacity of each route [Ministerie van Binnenlandse Zaken en Koninkrijksrelaties, 2011]. However, evacuations in real emergencies are rarely static, orderly and simple, as evidenced by the discrepancies shown in Table 1.1. Thus, simulation of such dynamic evacuations can help building users, managers and designers understand the possible chaos that could ensue in an emergency better, especially concerning the issue of crowds, which could block safe paths due to the volume of people needing to evacuate the building.

To set a static evacuation scenario apart from a dynamic evacuation is the inclusion of simulated sensor input from the fire. In modern buildings adhering to local fire codes, there exist both input and output systems that could be included in a fire safety management system. Input could be: temperature sensors, fire notification alarms, smoke sensors, automatic fire doors. Output could be: evacuation alarms, signage, emergency lighting, fire extinguishers, automatic fire doors, spoken fire alarm [Kobes et al., 2010].

In Wang et al. [2015] a 3D escape directional map is presented in a Building Information Model (BIM) evacuation simulation, together with a walk-through video. With a 3D directional map, the route is visualised for the user, thus enabling the user to plan ahead, and giving them a 3D overview of what is to come. An example of a directional map is shown in Figure 1.1, made with Esri ArcGIS software. A more immersive strategy is using a augmented reality to display the path to follow. However, this does require the position of the user to be known by using an indoor positioning system.

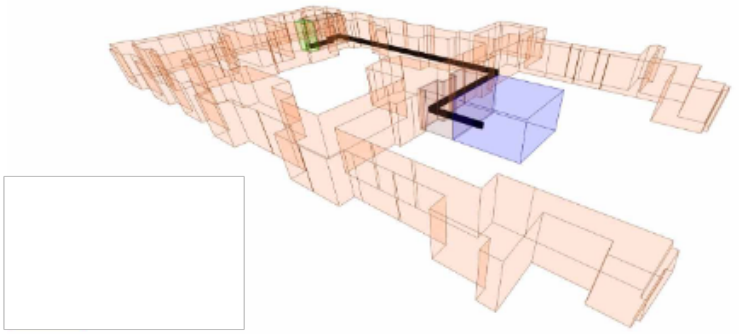


Figure 1.1: A 3D directional map showing a route through a building. Figure from Alattas [2022]

1.2 RESEARCH QUESTIONS

Considering the motivation behind this project, the main research question of this thesis is:

Which algorithm is best suited for multi-actor real-time pathfinding in a dynamic 3D voxelised indoor space?

As previously mentioned in Section 1.1, most implementations that use voxels and perform pathfinding do so by either pre-processing the voxel grid into a navigational graph or by casting the 3D space in 2 dimensions, and perform pathfinding on the 2D grid. Conversely, many of these implementations do not consider dynamic environments. Therefore, the goal of this research is to find out which pathfinding algorithms are best suited for real time pathfinding in a dynamic voxelised space. To achieve this, the following sub-questions have been defined:

- Which data structure for the voxelised space is best suited for dynamic pathfinding algorithms?
- How to include dynamic events into the pathfinding algorithms?
- Is it possible to combine evacuation simulation with pathfinding?

1.2.1 Scope of research

This thesis will focus on pathfinding directly in the voxel space, so the generation of a navigational mesh or other higher level abstractions will not be included in this research. Automatically segmenting the voxelised space or otherwise assigning semantics such as deriving the navigable space is considered pre-processing, but not part of the research itself. Furthermore, the graph is virtual and computed on the fly. Also, only indoor pathfinding inside the voxelised space will be considered.

1.2.2 Assumptions

In order to limit myself to the research at hand, several assumptions have to be made.

- A voxel can only hold one person at a time, and has a size of 0.08 m.
- A person has a walking speed of 1.42 m/s, and 0.9 m/s in stairwells Shi et al. [2009].
- The study area will be void of obstacles, doors, and other clutter.

- The best path is the shortest path in terms of distance.

1.3 READING GUIDE

This thesis consists of 5 further chapters. In Chapter 2, the theoretical foundation for the research proposed in this thesis will be laid. In particular, attention will be given to the various spatial data structures and pathfinding algorithms.

In Chapter 3, the methodology to solve the problem stated in the research question will be presented in a theoretical manner. The implementation of this methodology as well as the datasets and tools used is documented in Chapter 4.

In Chapter 5, the results of the experiments done with the implementation of the methodology are presented and analysed. Lastly, in Chapter 6, these a conclusion will be drawn from these results, and the limitations and recommendations of this research will be discussed.

Additionally, there are two appendices, Appendix A and Appendix B, which contain extra information about the research done in this thesis.

2 THEORETICAL BACKGROUND

In this chapter, I will review the theoretical background of the subjects that are at the basis of the research carried out in this thesis. First, a general introduction to digital representations of the our 3 dimensional world. Next, the voxelised space will be discussed, and lastly, pathfinding algorithms will be discussed.

2.1 DIGITAL REPRESENTATIONS OF THE WORLD

Digitally representing the world in a way that is both useful for humans to understand and possible for computers to handle is at the core of Geomatics. In the field of Geomatics there are, broadly speaking, three main ways to represent spatial data in 3 dimensions: point clouds, triangle or polyhedral meshes (often in the form of boundary representations) and voxel data. A point cloud, in its simplest form, can be considered to be a direct representation of the studied area, where measured points are assigned a 3D coordinate [Xu et al., 2021]. A mesh is a boundary representation of geometry, where a collection of points, lines and triangles are used to digitally represent geometry. A mesh stores features such as topology, curvature and edges, to support accurate geometric analysis [Lv et al., 2021], where not only the points that make up the studied area are stored, but these points are the boundary of a surface. Lastly, voxels (a contraction of volumetric pixels) are a discrete representation of the 3D space, usually through a 3 dimensional grid.

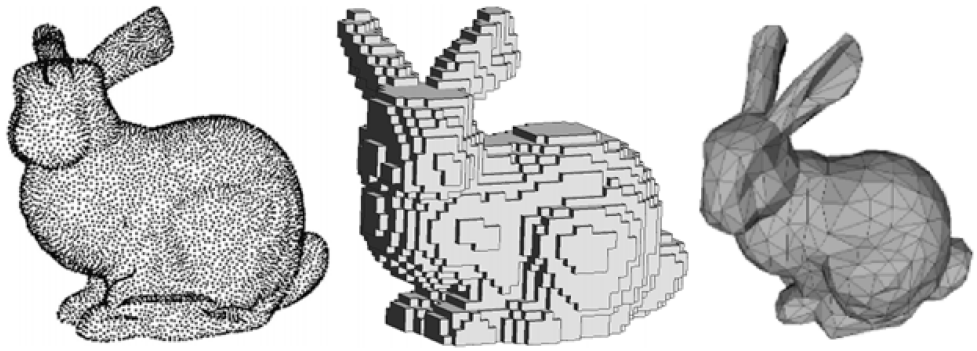


Figure 2.1: The Stanford bunny model represented in the 3 spatial data representations: point cloud (left), voxel (centre), and 3D mesh (right). Figure from Hoang et al. [2019]

2.2 THE VOXELISED SPACE

A voxel model of an indoor space such as a building can thus be considered as a structured 3-dimensional grid in a topologically implicit way [Xu et al., 2021], which makes it a suitable representation for purposes such as, but not limited to, analysis, simulation and visualisation. A voxel space can be manually constructed, or can be the result of a process known as voxelisation, where a dataset represented by a different data structure is converted to the 3D grid. In this process, a point cloud, or even a Computer Aided Design (CAD) drawing can be voxelised, with the resolution of

the resulting voxelised space being determined according the the needs of the user. The simplest way to represent voxel data is by using a 3D grid, where every 3D index in the voxel grid represents a voxel; however, this method of representing the voxel space is very expensive at higher resolutions, resulting in very large datasets that make using voxels less practical. Thus, different ways exits to represent voxels more efficiently. [Jones, 1989], see next section for more efficient voxel data structures.

2.3 VOXEL DATA STRUCTURES

Multiple topological data structures exist for storing and performing operations on voxels, such as searching, transforming, updating or rendering. A few examples of these data structures are: the voxel grid, the voxel octree and the directed a-cyclic graph, as well as some hybrid forms such as a Morton code based sparse grid. In the following sections, these data structures will be discussed in more detail, but first some general terminology that is applicable to all data structures that deal with voxels. In a *uniform*, *dense* voxelised space, all voxels have the same size and all voxels are known: for all voxels, their position is known, as well as their value (e.g. air, wall, with multiple attributes possible). This representation describes the complete 3D space of the study area, which has some downsides, such as storing information for many empty voxels. In contrast to the dense voxel space, there is the *sparse* voxel space which only encodes those voxels which are not empty; if a voxel is not stored it is considered empty [Gorte et al., 2019b]. Lastly, we can distinguish between *static* and *dynamic* voxelised spaces. In a static voxelised space, the space does not change, and in a dynamic voxelised space it does.

2.3.1 Voxel Grid

The regular voxel grid is the simplest way to store voxel data. In this section the different aspects of the voxel grid are discussed.

Construction

Constructing a regular voxel grid is often the direct result (or part of the) voxelisation process. While many different approaches exist for voxelisation [Aleksandrov et al., 2021], with differing parameters regarding rasterisation (e.g. Bresenham, supercover, etc.), the general idea is that for every voxel in the grid, it is checked if the voxel in question intersects the input geometry. If the voxel in question does intersect the input geometry, this is recorded at the index of that voxel, which should be the current index in the grid traversal, by setting a Boolean to true, an integer value to 1 or and red, green, blue (RGB) triplet to a certain colour for example. This is highly dependent on the implementation and the information the voxel grid is required to store.

Size and complexity

The standard way to represent voxelised spaces are regular grids, where voxels are stored as 3D arrays, allowing for fast neighbour access and fast lookup [Aleksandrov et al., 2021]. Regular grids (often just called voxels grids) store all the voxels, and all voxels are of the same size. This regularity and simplicity is one of the advantages of this data structure. Due to the fast access and lookup of voxels, this data structure can be used to store both static and dynamic voxel data. A typical implementation of a regular voxel grid is a one dimensional array which stores voxel elements and its attributes, which can be accessed by an $[x, y, z]$ index [Jones, 1989], or a 3D array with elements accessed by index $[x][y][z]$. Both methods store the same amount of voxels, and both methods require looping over all voxels (namely $x * y * z$) voxels to traverse

the entire grid. The order in which the voxels are stored has an influence of the performance. The standard method is to store the voxels in increasing order, per dimension, so $[0,0,0]$, $[0,0,1]$, $[0,0,2]$ ... until the entire grid has been processed. Using space filling curves (SFCs) can greatly influence both traversal performance as well as neighbour finding by providing a higher level of spatial clustering [Holzmüller, 2017]. Because of the high number of voxels stored ($x * y * z = 16777216$ a 256^3 voxel grid), storing voxels in this way is quite expensive on storage (i.e. $O(n^3)$ space complexity), which can lead to unwieldy datasets when a high resolution is required. In Table 2.1, an example of the memory footprint or a simple voxel grid using regular a simple example: if every voxel uses 32 bits of memory (this could be a simple integer value representing some attribute data), it shows the size of the resulting voxel grid.

	Number of voxels	Size in memory [GB]
256^3	16.5 million	0.07
512^3	134 million	0.54
1024^3	1 billion	4.29
2048^3	8.5 billion	34.36
4096^3	68.5 billion	274.88

Table 2.1: Table showing the number of voxels and their memory footprint for different resolutions of voxel grids.

Neighbour access

Accessing neighbours in a voxel grid is simple: for the required connectivity (i.e. typically 6, 18 or 26 connectivity) the appropriate indices have to be generated and then used to access the neighbouring voxels. This can be done directly without any preprocessing. For 6-connectivity, the neighbours are only the voxels that share a face with voxel (x,y,z).

$$(x,y,z)_{6neighbours} = (x \pm 1, y, z), (x, y \pm 1, z), (x, y, z \pm 1)$$

For 18-connectivity, the voxels that share faces or edges with voxel (x,y,z) are considered neighbours.

$$\begin{aligned}(x,y,z)_{18neighbours} = {} & (x \pm 1, y, z), (x, y \pm 1, z), (x, y, z \pm 1), \\ & (x \pm 1, y \pm 1, z), (x \pm 1, y, z \pm 1), (x, y \pm 1, z \pm 1)\end{aligned}$$

For 26 connectivity, the voxels that share either a face, an edge or a vertex are considered neighbours of voxel (x,y,z).

$$\begin{aligned}(x,y,z)_{26neighbours} = {} & (x \pm 1, y, z), (x, y \pm 1, z), (x, y, z \pm 1), \\ & (x \pm 1, y \pm 1, z), (x \pm 1, y, z \pm 1), (x, y \pm 1, z \pm 1), \\ & (x \pm 1, y \pm 1, z \pm 1)\end{aligned}$$

These indices can then be used in the 1D array directly to access the information stored at the index. For Morton code based grids, a different method of accessing neighbours in a 1D array has to be used, which is explained in Section 3.3.3.

Typical uses and optimisations

Multiple ways exist to compress the size of the voxel grid, to make it more manageable in size. Many of these methods come from the world of computer graphics, where voxel based rendering is seen as an alternative to traditional triangle based rendering. It is however good to note that most of these representations are not useful in the context of Geomatics, as often within that field more data is stored in the voxel than an

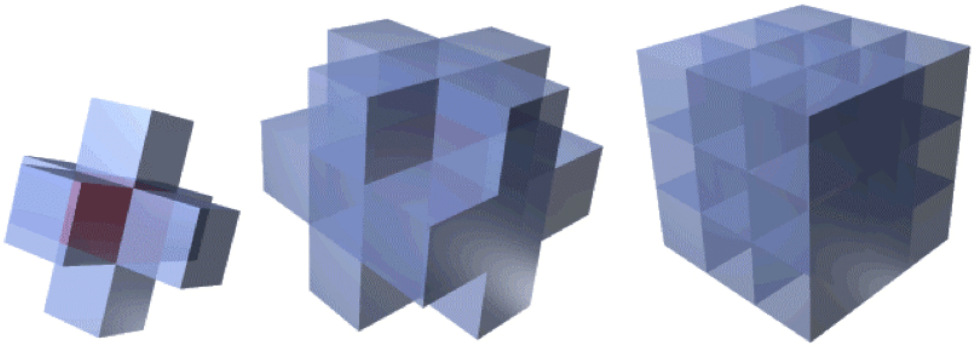


Figure 2.2: The three most used 3D voxel connectivity types: 6 connectivity(left), 18 connectivity(centre) and 26 connectivity(right). Figure from Tankyevych [2010].

RGB value such as is often the case with render scenes. Concurrently, using voxels to render a scene requires a much higher resolution than most Geomatics related applications require. This will be discussed in more detail in section 2.3.3. Compression methods such as using run-length-encoding (RLE), can be used to perform some relatively simple compression [Houston et al., 2004]. RLE is a very simple compression method where, if there are elements that are repeated in a part of a certain sequence, instead of storing all the elements, only the first element is stored, and the number of times it repeats in this part of the sequence.

Because of the nature of the dense voxel grid, i.e. using $[x, y, z]$ coordinates to index all the voxels, there is little extra possibility to improve on space complexity. Because of this, and also because many of spatial datasets have a high level of sparseness [Dado et al., 2016], alternative data structures have to be considered when very large or high resolution datasets are to be used, such as sparse or hierarchical data structures.

2.3.2 Sparse Voxel Octree

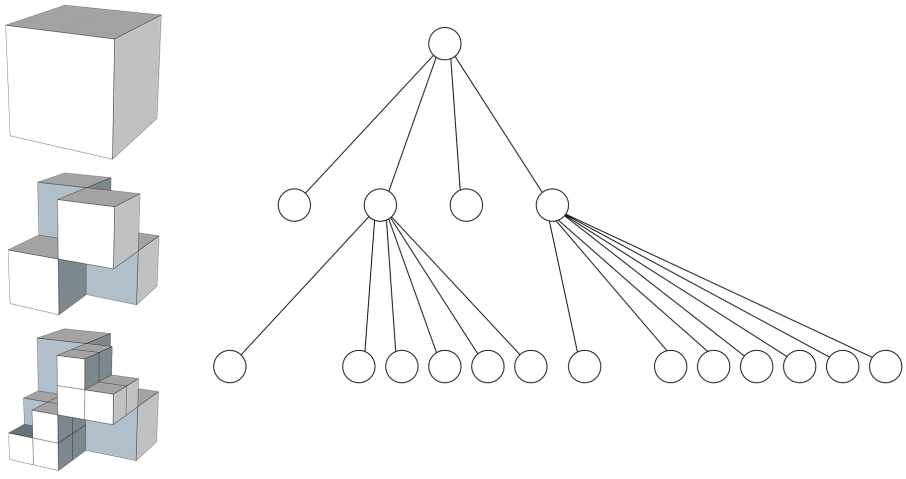


Figure 2.3: A schematic illustration of a sparse voxel octree.

The SVO is a hierarchical data structure for storing voxels. An octree is a recursive subdivision of space. Starting out at the root, the space is subdivided into 8 children, and depending on whether something is present in these subdivisions determines whether the children themselves need to be subdivided as well, until a sufficient resolution has been achieved. In Figure 2.3, the SVO can be seen in a conceptual sense.

While dense octrees, where empty nodes would also be stored, could also technically exist, they are not used.

Construction

Numerous methods exist for the construction of SVOs [Baert et al., 2013], both top-down and bottoms-up. Additionally, the construction is also dependent on the type of octree that is required. Generally speaking, there are two approaches: pointer-based octrees and linear octrees [Cormen et al., 2022]. In pointer-based octrees, leaf nodes are stored, as well as all their parents are stored, with relationships recorded through pointers. This makes it easy to traverse the tree by following either the child or parent pointer from or to a node. Linear octrees however store only the leaf nodes, and use a locational code to identify the octants. This code contains information about the position of the octant, and its level in the tree. [Sundar et al., 2008].
Both approaches have their advantages, but broadly speaking: linear octrees perform well in terms of space, and often require less overhead, whereas pointer based octrees are easier to traverse due to their parent/child relationships being explicitly stored. Both approaches often make use of SFCs to partition the space in an efficient manner and to cluster the data in a spatial manner.
Of the SFCs, used in methods like Baert et al. [2013] and Sundar et al. [2008], the Morton curve (or Z-order curve) is the most common. The Morton curve is a linearisation of a an n-dimensional grid, in which n dimensions are mapped to one dimension [Morton, 1966], while maintaining their locational integrity in the grid, through interleaving the bits of the coordinates. In Figure 2.4 we can see how this works in two dimensions.

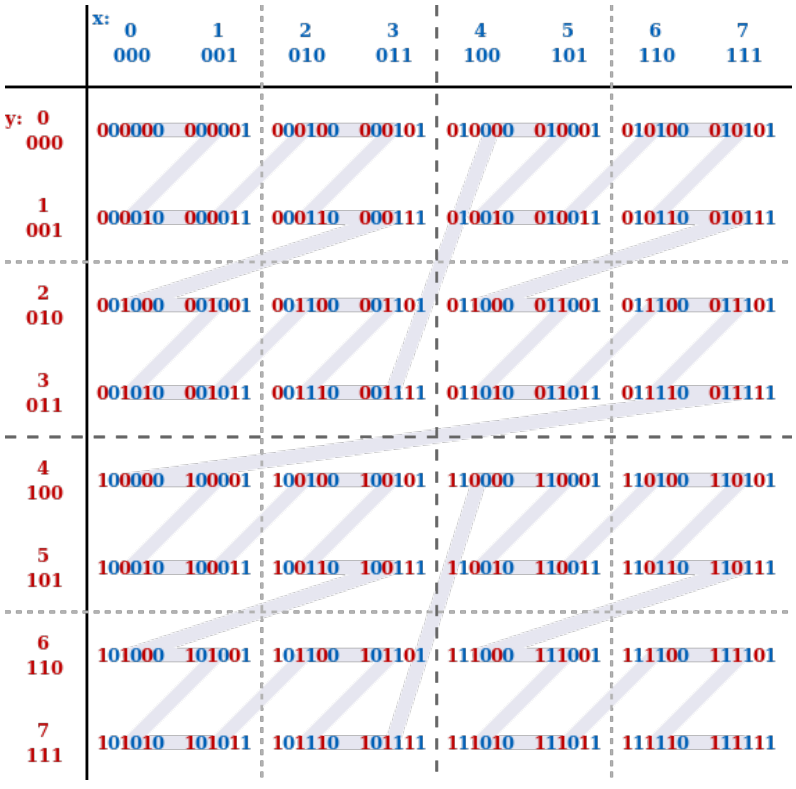


Figure 2.4: Morton encoding in two dimensions. Figure from Wikipedia - Z-order curve

Thus, using this, when constructing an SVO, every node will be assigned the Morton code of their position, which will function as their index in the storage array of voxels. This strategy can be used in both a pointer-based and linear SVO, especially because Morton curves are hierarchical, as can be seen in Figure 2.5.

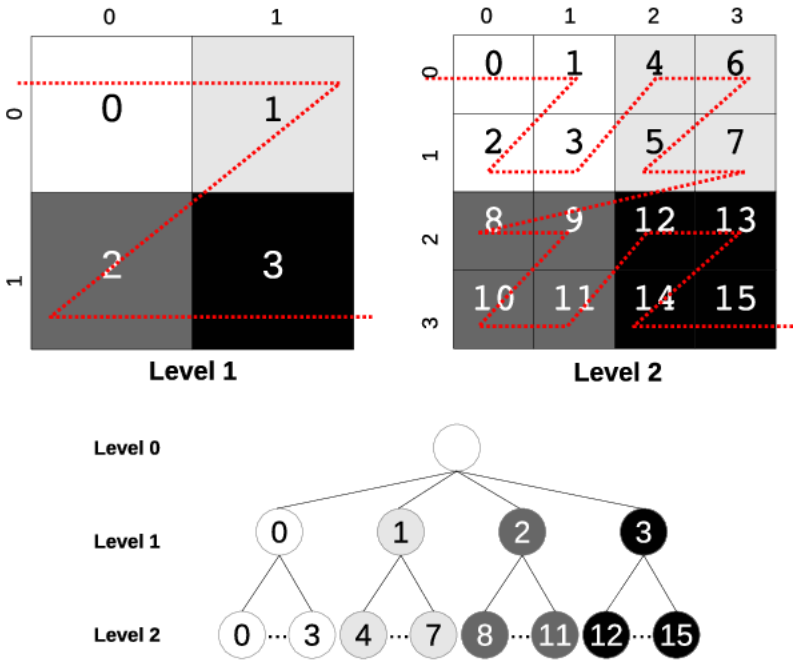


Figure 2.5: An illustration of the hierarchical nature of Morton codes. Figure from Baert et al. [2013]

Size and complexity

When comparing the size and complexity of an SVO compared to say, a regular voxel grid, it is important to note the type of SVO (pointer vs. linear) as discussed above. However, generally speaking a regular voxel grid will usually be less complex compared to a run-of-the-mill SVO, due to the simple fact that SVOs require more information and information links than a regular n-dimensional grid (parent–child relationships, etc.). In terms of space, an SVO can save space by not encoding for space that is not filled [Dado et al., 2016]. For very larges volumes of course, this will still amount to very large voxel numbers as evidenced by Table 2.1.

Neighbour access

Accessing neighbours in an SVO is not as straightforward as in a regular voxel grid, due to the fact that not all neighbours might be of the same size as the starting node. This presents multiple challenges. Different methods exist, but one of the most widely known is the method by Samet [1989]. This method deals with finding equal sized neighbours, as well as smaller and larger sized neighbours, based on their common ancestors. Yet again however, the exact implementation of this method is highly dependent on the type of SVO. The method by Samet [1989] can be used on either pointer octrees or linear octrees. The method starts by computing the face neighbours of equal size (6-connectivity) of a node by finding the nearest common ancestor. Then the edge neighbours of equal size are found, and after that the vertex neighbours of equal size. These procedures also account for the finding of neighbours of non-equal sizes by, when traversing up and down the octree, also checking if the ancestors of the start node share a face, edge or vertex.

Neighbour access using this method however is significantly more complex than in a regular grid, because numerous operation have to be done to record the neighbour relations for every node. While other methods do exist, like Vörös [2000], all of them require some form of computation. This is acceptable, of course, in a static SVO where this can be done in a pre-processing stage, while doing this in real time in a dynamic

SVO can present performance costs. It should be noted that storing all the neighbour relationships (i.e.pointers to all neighbours for all voxels) is very expensive too.

Typical uses and optimisations

While SVOs do present significant decreases in space complexity, the data structure is significantly more advanced and complex than a regular grid and thus requires more overhead and a lot of *bookkeeping* when using them for dynamic data. This does not make it the catch-all definite improvement to a regular voxel grid, it really depends on the use and requirements of the dataset.
Furthermore, while SVOs efficiently deal with the high level of sparsity in most datasets [Dado et al., 2016], it is also possible to compress spatial datasets even further by making use of the fact that most spatial datasets have a high level of geometric redundancy.

2.3.3 Sparse Voxel Directed Acyclic Graph

A sparse voxel directed acyclic graph (SVDAG) is a generalisation of an SVO where geometrically identical subtrees are merged, this way, even higher levels of compression can be achieved that allow large static scenes to be fully present in memory [Dado et al., 2016]. This data structure is being used in the field of computer graphics when using rendering very high resolution voxel scenes while maintaining traversal efficiency. It should be noted that this data structure is highly compressed, usually aiming for 1 bit of information for every node [Dado et al., 2016]. In this section some aspects of this data structure will be discussed.

Construction

The construction of an SVDAG starts with an SVO. The paper by Kämpe et al. [2013] provides a bottom-up method for transforming an SVO to an SVDAG. Starting at the leaf nodes, it looks at the arrangement of the geometry by checking the *childmask*, which is a bitmask that refers to the geometrical arrangement of the voxels that make up this node (in this paper, the leaf nodes are not the voxels themselves, but rather, the parents of the smallest voxels). Because of this, there exists a finite arrangement of that these parent voxels can hold ($2^8 = 256$ in fact, due to every leaf having a combination of 8 possible children). This fact is leveraged, and thus leaf nodes can be compared, and if leaves are identical, they can be merged and their parent nodes can now point to the same child node.

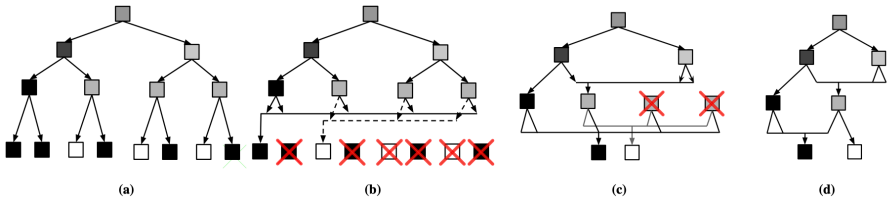


Figure 2.6: Reducing a sparse voxel tree, illustrated using a binary tree, instead of an octree, for clarity. a) The original tree. b) Non-unique leaves are reduced. c) Now, there are non-unique nodes in the level above the leaves. These are reduced, creating non-unique nodes in the level above this. d) The process proceeds until we obtain our final directed acyclic graph. From Kämpe et al. [2013].

Moving to the next level, the same procedure is performed, and this time identicalness is measured not only by checking the identical childmasks but also the identical pointers, which, if identical, can be merged again. This is repeated until it is not

possible to merge any more nodes, at which point the construction of the SVDAG is complete. The procedure is schematically laid out in Figure 2.6.

Size and complexity

The entire purpose of the SVDAG is to compress an SVO, so naturally, the resulting data structure is smaller than an SVO of the same dataset would be, and also smaller than a voxel grid of the same scene.
This is also due to the fact that most implementations of SVDAGs are specifically for rendering scenes, and thus can often only hold positional and colour data.

Neighbour access

Neighbour access in the SVDAG is done by traversing up and town the graph, and has mostly been implemented for ray tracing. Because this data structure has a much higher level of abstraction and is not used much outside of computer graphics.

Typical uses and optimisations

Currently in the literature, the sole use for SVDAGs is for rendering, and because of this, the implementations only concern themselves with performance in that sector. High compression is achieved by storing only what is necessary for encoding and thus rendering geometry: childmasks, pointers and contours Kämpe et al. [2013]. This makes the data structure highly specialist, while also hampering its use outside this domain: when more data is needed per node, as is often the case outside (as well as inside) of computer graphics, then the compression factor of this data structure lowers, and the upsides of less memory consumption do not outweigh the downsides of having such a complex data structure.

2.4 PATHFINDING ALGORITHMS

Finding the shortest path between two points on a graph or a grid, and pathfinding in general is at the heart of many technologies, from logistics to gaming. Generally, pathfinding, works by starting at a vertex, and visiting adjacent vertices and keeping track of the cost of visiting these vertices repeatedly until the destination vertex is found. Many different pathfinding algorithms exist, with different optimisations for different types of data and goals. Some algorithms have lower memory consumption, whereas in other applications memory is not an issue, but computation time is a tighter constraint. In this section, various pathfinding algorithms relevant for this research will be explained, as well as some of the core principles of pathfinding.

2.4.1 Grid vs graph

In pathfinding in general, there are two general approaches: using graphs or using a grid, which is a fixed type of graph in this context, with one node per cell. When using a graph, the 3D space has to be subdivided into spaces, the centres of which will form the nodes of the graph, and from these nodes a navigational network can be constructed with paths between the nodes which are adjacent, which can be used to perform searches. Many pathfinding algorithm are designed to work optimally on a graph. When using a grid, the space is subdivided into tiles (or cubes) of a certain size, and algorithms determine when an agent can move from tile to tile, and keeps track of which tiles have already been visited, such as in Wirth and Szabó [2018]. This approach is considered computationally expensive when the grid resolution is high [Gorte et al., 2019a], and especially expensive when pathfinding on a 3-dimensional grid, due to the fact that the pathfinding in a 3D cubic space is NP-hard [Canny and

Reif, 1987]. Graphed (or vector) based pathfinding can also have performance problems, as described in Van Bemmelen et al. [1993], where a vector based solution for cross country pathfinding turns out to be a very heavy resource user. Another factor to consider is the choices of movement. In a graph, a node only has a limited number of neighbours, and thus can only move in a limited number of directions. This is also true for a grid, where often one can move only towards that neighbours of a cell. Van Bemmelen et al. [1993] provide a solution by using a higher connected grid, where new neighbours are added between the lower connectivity neighbours, as illustrated in Figure 2.7.

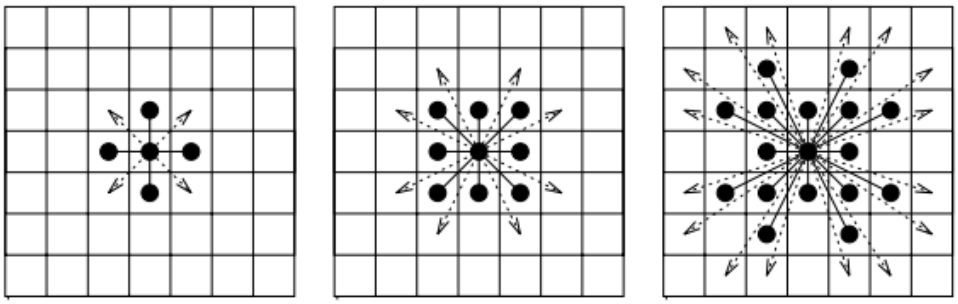


Fig. 4: 8-connected. Fig. 5: 16-connected. Fig. 6: 32-connected.

Figure 2.7: Higher levels of connectivity in a 2D grid. Figure from Van Bemmelen et al. [1993].

2.4.2 Dijkstra's Algorithm

Dijkstra's algorithm (complete algorithm in Algorithm 2.1, also known as the uniform cost search algorithm, is a shortest path algorithm for searching a weighted graph. To initialise the algorithm, all the node distances are set to unknown, except for the starting node, which is set to 0. All nodes are then added to a queue. The algorithm starts at the starting node, and then moves outward by visiting the starting node's neighbouring nodes. For each visited nodes neighbours in the graph, the distance from the current node to the source is calculated, and added to the cost of travelling to the neighbour from the current node. If that result is smaller than the cost of travelling from the neighbour to the source, the distance is updated and the relationship between the two nodes is recorded: every node gets a pointer to a previous node. When all the nodes have been visited, it is possible to reconstruct the back by starting from the goal, and travelling along the previous pointers and adding the nodes to a path. One can also choose to use the result of Dijkstra's algorithm as a shortest path tree, because it holds all distances from all the nodes in the graph to the goal [Dijkstra, 1959].

Dijkstra's algorithm performs well when used on a weighted graph, but less so when a large number of nodes has to be visited. It does however guarantee the shortest path, and works well when you have no knowledge of the graph (it is an *uninformed* algorithm). It is not efficient to use Dijkstra's algorithm on grids, (which are of course essentially very large graphs to a search algorithm), and therefore it is not suitable for 3D pathfinding with voxels. It is however a very fundamental in simple algorithm, and it lays the basis for many other pathfinding algorithms in this thesis.

Algorithm 2.1: Dijkstra's algorithm

```
1  Function Dijkstra's algorithm(graph, source):
2      for vertex v in Graph.Vertices do
3          distance[v] ← ∞
4          previous[v] ← ∅
5          add v to PQ
6      distance[source] ← 0
7      while PQ is not empty do
8          u ←vertex in Q with min(distance[u])
9          remove u from PQ
10         for neighbours of v of u still in PQ do
11             alt ← distance[u] + cost(u, v)
12             if alt < dist[v] then
13                 dist[v] ← alt
14                 prev[v] ← u
15     return dist[], prev[]
16 Function reconstructPath(dist[], prev[]):
17     path ← ∅ u ← goal
18     if prev[u] is defined ∨u ← source then
19         while u is defined do
20             insert u at the beginning of S
21             u ← prev[u]
```

Note: *PQ* denotes a priority queue, which is a queue container that sorts elements according to a certain priority. Often there is a *top*() function that accesses the element with the highest priority and a *pop*() function which removes the highest priority element from the queue.

2.4.3 A* Algorithm

A* (*a-star*), is a generalised, *informed* version of Dijkstra's algorithm, which uses a heuristic for optimised searches, the full procedure is displayed in Algorithm 2.2. This heuristic estimates the cost from the current node to the destination cost. In A*, it is essential not to overestimate the heuristic, which could significantly hamper performance. The heuristic is used to choose which node to visit, meaning less nodes to visit than with Dijkstra's algorithm. The heuristic is problem dependent, and in spatial pathfinding, metric such as Euclidean or Manhattan distance between the current node and the destination node can be used to estimate the distance, and thus to choose which node to visit next.[1] A* is a best first algorithm, but it is still also a greedy algorithm, needing to store many visited nodes in a large search area, and has a space and time complexity of $O(b^d)$ Hart et al. [1968].

The goal of A* is to minimise the number of nodes to visit while still being able to find the most optimal path. The algorithm starts at the starting node, and maintains two array-like data structures. A priority queue called the *PQ* and a map called *cameFrom*. The *PQ* can be thought of as the *frontier* of the search, i.e. all nodes currently under consideration. The *cameFrom* map is a map which links nodes with their preceding nodes (this is also often done with pointers). Additionally, A* maintains two values: $g(n)$, which is the cost so far of the starting node to the current node and $h(n)$, which is the heuristic function, which estimates the cost from the current node to the goal node. To initialise the search, the starting node's $g(n)$ value is set to 0, and the starting node is added to the queue. Then the algorithm visits all the neighbours of the cur-

1 In Algorithm 2.2 the heuristic returns a distance value, and this does make the most sense for spatial pathfinding algorithms in a grid, but the heuristic could be any value even unrelated to the physical distance between nodes.

Algorithm 2.2: A* Algorithm

```
1  Function main(s_start, s_goal):
2      PQ ← ∅
3      cameFrom ← ∅
4      g(s_start) ← 0
5      insert s_start into PQ with priority g(s_start) + h(s_start, s_goal)
6      while PQ ≠ ∅ do
7          s_current ← PQ.top()
8          if s_current = s_goal then
9              return reconstructPath(cameFrom)
10         PQ.pop(s_current)
11         foreach s_neighbour ∈ s_current do
12             g_tentative ← g(s_current) + cost(s_current, s_neighbour)
13             if g_tentative < g(s_neighbour) then
14                 cameFrom[s_neighbour ← s_current]
15                 g(s_neighbour) ← g_tentative
16                 if s_neighbour ∉ PQ then
17                     insert s_neighbour into PQ with priority
                       g(s_start) + h(s_start, s_goal)
18     return;
19 Function reconstructPath(cameFrom):
20     s_current ← s_goal
21     while s_current ≠ s_start do
22         add s_current to path
23         s_current ← cameFrom[s_current]
24     return
25 Function heuristic(s, s'):
26     return distance(s, s')
```

rent node, which is the node with the lowest $g(n) + h(n)$ value, and then computes a new tentative $g(n)$ value for the neighbour nodes by taking the $g(n)$ score of the current node, plus the cost of travelling to the neighbour. If this tentative score is lower than the previous $g(n)$ score of the neighbour, the new lower cost is set as the new $g(n)$ score of the neighbour, and the current-previous relationship is recorded in the *cameFrom* map. The neighbour is then added to PQ with the new $g(n)+h(n)$ value for this node as its priority in the queue. This process repeats until PQ or the goal node is reached. Then the path can be reconstructed in a similar fashion to Dijkstra's algorithm: backwards and following the *cameFrom* map (or pointers) to previous nodes.

A* performs well, is generally well liked and is used ubiquitously, however, with increasing grid sizes, the traditional A* algorithm is not adequate and not able to keep up with modern day demands [Foead et al., 2021]. Small modifications or improved heuristic functions can greatly improve the performance of A*, and make it a even more efficient than it already is. Furthermore, A* is a *shortest path algorithm*, and the shortest path is not always the best path.

2.4.4 Theta* and Phi* Algorithm

Theta* (*theta-star*) is an any-angle pathfinding algorithm for grids [Nash and Koenig, 2013], which is a version of the A* algorithm which uses line of sight (LoS) checks

to update and visit nodes. The full Theta* algorithm can be found in Algorithm A.1, with the discerning method of the algorithm displayed below in Algorithm 2.3. Theta* has been developed for any-angle path planning on a grid, meaning it is not limited to movement on the primary axes and diagonals on a grid. However, the Theta* algorithm is not particularly well suited for dynamic environments, which is where Phi* comes into play. Phi* is able to recompute lines of sight when nodes change (i.e. from blocked to free), with similar performance to Theta* [Nash et al., 2009], making it a variant of Theta* which is suited for dynamic path planning at any angle. Additionally, Phi* is an *incremental* algorithm, meaning that it assumes that the agent (and thus the starting node) is moving. However, both algorithms are more performance heavy than, for instance, A*, due to their computationally expensive LoS checks. These LoS checks however, should produce smoother paths which are not dependent on obstacle shapes or the size of the agent Algfoor et al. [2015].

Theta* is a more complex algorithm than A*, but in principle it is quite similar. In fact, this algorithm becomes A* when lines 2–6 are removed from Algorithm 2.3 [Nash et al., 2009]. Therefore, the procedure is quite similar as described in Section 2.4.3 above, and only this function will be explained here. In this function, the Theta* and Phi* algorithms check whether vertex s' is in the line of sight of the parent of vertex s. If that is the case, the algorithm checks whether the vertex is also closer to the goal than the previous $g(n)$ value for s', if that is the case, the parent of s' will become the parent of s. This method should thus lower the amount of vertices to be visited because all the vertices that are between $parent(s)$ and s' can be skipped because the algorithms knows it can pass, and generate smoother and more natural paths simultaneously. However, due to the fact that Theta* and Phi* needs to perform these LoS checks to be able to skip nodes, it is not always more efficient than A* [Algfoor et al., 2015].

Algorithm 2.3: Compute Cost method from Basic Theta*

```
1  Function computeCost(s, s'):
2      if lineOfSight(parent(s), s') then
           // Path 2
3          if g(parent(s)) + cost(parent(s), s') < g(s') then
4              parent(s') ← parent(s)
5              g(s') ← g(parent(s) + cost(parent(s), s')
6              local(s') ← s
7      else
           // Path 1: identical to A*
8          if g(s) + cost(s, s') < g(s') then
9              parent(s) ← s
10             g(s') ← +cost(s, s')
11             local(s') ← s
```

2.4.5 D*-Lite and LPA*

D*-Lite (*d-star-light*) is a dynamic pathfinding algorithm by Koenig and Likhachev [2002], which is based on Lifelong Planning A* (LPA*), which in turn can be thought of as an dynamic version of A*, meaning that it can efficiently handle when costs in the search graph change, which was first published by Koenig and Likhachev [2001]. It is called D*-Lite because it fulfils the same goal as the original D* algorithm by Stentz [1993] which is autonomous robot navigation in unknown terrain, but it is unrelated to the D* algorithm aside from the name. D*-Lite performs well when navigating unknown terrain, and is therefore popular within the field of robotics [Noori

and Moradi, 2015]. The main difference between LPA* and D*-Lite is the that LPA* assumes a static start and end node, and D*-Lite assumes a moving start position. The full LPA* algorithm is presented in Algorithm 2.4, with the differences between the two being mainly in the *main()* function in the algorithm.

Algorithm 2.4: Lifelong Planning A* algorithms

```
1  Function main():
2      foreach s ∈ S do
3          rhs(s) = g(s) = ∞
4      rhs(s_start) ← 0
5      Q.insert(s_start, [h(s_start), 0])
6      while true do
7          computeShortestPath()
8          while no changes in search area do
9              sleep
10         foreach s_changed do
11             updateVertex(s_changed);
12 Function updateVertex(s):
13     if s ≠ s_start then
14         rhs(s) = min_{s'∈Pred(s)}(g(s') + c(s', s))
15     if s ∈ Q then
16         Q.remove(s)
17     if g(s) ≠ rhs(s) then
18         Q.insert(s, calculcateKey(s))
19 Function computeShortestPath():
20     while Q.topKey() <calculcateKey(s_goal) ∨ rhs(s_goal) ≠ g(s_goal) do
21         s_current ← Q.pop()
22         if g(s_current) > rhs(s_current) then
23             g(s_current) ← rhs(s_current)
24             foreach s' ∈ Succ(s_current) do
25                 updateVertex(s')
26         else
27             g(s_current) ← ∞
28             foreach s' ∈ Succ(s_current) ∪ {s_current} do
29                 updateVertex(s_current)
30 Function calculcateKey(s):
31     return [min(g(s), rhs(s)) + h(s); min(g(s), rhs(s))]
```

They key with both D*-Lite and LPA* is that it maintains not one, but two estimates per node: $g(n)$ and $rhs(n)$, where $g(n)$ is the same as in A*, an estimate of the distance to the goal, and $rhs(n)$ are one-step lookahead values for even more informed searching runs. The value of this $rhs(n)$ value is based on the $g(n)$ values of the predecessors of node n. This $rhs(n)$ value always maintains the following principle:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ min_{s' \in Pred(s)}(g(s') + cost(s', s)) & \text{otherwise} \end{cases} \tag{2.1}$$

When a node has the following state where $g(n) = rhs(n)$, then a node is considered to be *locally consisted* and a shortest path can be found with A*, but when the search area changes, the values can change, and only need to be calculated for nodes that are local

for the route that was found. This way, the algorithms do not have to recalculate all the nodes in the graph when the graph changes, only the nodes relevant to the found path have to be calculated. This fact makes these algorithms suitable for dynamic environments because they *learn* from their previous iterations.
Like A* and Theta*, both algorithms use a priority queue [2] to sort nodes for (re)evaluation. And just like having two values to estimate the distances to the goal node, the key to sort the priority queue is also two dimensional. The key value is defined as follows:

$$k(n) = \begin{bmatrix} k_1(n) \\ k_2(n) \end{bmatrix} = \begin{bmatrix} min(g(n), rhs(n)) + h(n, goal) \\ min(g(n), rhs(n)) \end{bmatrix} \tag{2.2}$$

With the keys in the queue being compared by lexicographic ordering, i.e. first $k_1(n)$ is evaluated, and then $k_2(n)$. Both LPA* and D*-Lite have been developed for 2 dimensional grids, but are theoretically feasible in 3D space, though not many implementations exist.

2.4.6 Hierarchical pathfinding

Many pathfinding algorithms, such as the ones described above are very efficient at medium scale datasets, but are not particularly efficient when dealing with large (gridded) datasets. Thus, to compensate for the high computational load of a fully gridded approach to use for pathfinding on a large area in real time, hierarchical pathfinding can be used. In this approach, the study area is subdivided into (often semantic) clusters (e.g. room, hallway, stairs), and a path is found between the clusters, and then the shortest path is found within the clusters. As many levels as required are possible, with a minimum of two. Because the clusters are of limited size, path finding algorithms such as A* can be used inside the cluster, such as in the Hierarchical Pathfinding A* (HPA*) approach developed by Botea et al. [2004] and used in a 3D voxelised space by Koopman [2016]. However, clustering approach used in HPA* is a 2D method that cannot be extended to 3D, thus when using HPA* in 3D, other methods have to be used to cluster the space.

To execute hierarchical pathfinding, the 3D space should be abstracted by means of clustering to be able to cover large areas in a more efficient way [Botea et al., 2004]. In Muratov and Zagarskikh [2019], they use an approach combining an SVO and HPA*, using the intrinsic properties of the octree to define the clusters, promising a significant reduction in memory costs and time for the pathfinding. By using the depth level of the octree for clustering, instead of a semantic clustering based on size, this process can be automated in 3D.

However, hierarchical pathfinding does still have some caveats, especially when dealing with a dynamic space, primarily due to the fact that the hierarchical data structure it has to run on is not always particularly well suited for dynamic data, as described in Figure 2.3.2. Furthermore, hierarchical pathfinding does not always find the optimal path Foead et al. [2021].

2.4.7 Which algorithms to use

When looking at the aforementioned pathfinding algorithms, the decision was made to focus on three algorithms: A*, Theta* and D*-Lite. A* was chosen because it is described as being versatile, fast and reliable, and reliable on grid [Foead et al., 2021]. Theta* was chosen because of its any-angle pathfinding qualities, and concerning it

2 $Q.topKey()$ returns the node with the smallest key from priority queue Q. $Q.pop()$ removes the node with the smallest key from priority queue Q and returns this node. $Q.insert(s, k)$ inserts node s into queue Q with priority k. $Q.remove(s)$ removes node s from the queue.

was devised specifically for (albeit 2D) grids. Lastly, D*-Lite was chosen because it was conceived for navigation in unknown dynamic terrain.
Dijkstra is widely considered, while guaranteeing to find the shortest path, to be too slow for dynamic situations [Koenig and Likhachev, 2001]. Phi* could be considered once Theta* has proven to be effective, and LPA* is inferior to D*-Lite regarding incremental pathfinding.

2.5 THE NAVIGABLE SPACE

The indoor navigable space is the free surfaces inside a building that actors in the indoor environment can use to navigate without obstacles blocking their way [Staats et al., 2017]. In Koopman [2016], there exist three walking actors and one flying actor who use the indoor space for movement: the walking adult, a person in a wheelchair, a drone, a vacuum cleaner robot. Each of the actors has a size, a mode of locomotion (e.g. driving, flying, walking), and a notion of best path (e.g. shortest, Manhattan, Morton). For evacuation management however only the walking adult and the person in wheelchair in wheelchair are relevant. Each of these actors however have a different notion of the navigable space.
Several approaches exist for determining the navigable space in an indoor environment. In Staats et al. [2017] and Flikweert et al. [2019] the trajectory of a Mobile Laser Scanner (MLS), together with voxel operations are used to generate the navigable space from an indoor point cloud, according to the IndoorGML definition. Whereas in Koopman [2016] and Gorte et al. [2019b] a distance transform, dilation and watershed method is used to determine the navigable space inside a voxelised space.

3 METHODOLOGY

In this chapter, the theoretical process of how this thesis will answer the research question will be described in detail. First, the general design of the research will be presented. Next, the steps taken to setup pathfinding on a grid and on an octree. Thirdly, adapting the algorithms to voxel spaces. Lastly, the testing environment will be presented.

3.1 RESEARCH DESIGN

In this section the general approach to answering the research questions of this thesis will be discussed. The main research question is as follows:

> *Which algorithm is best suited for multi-actor real-time pathfinding in a dynamic 3D voxelised indoor space?*

What is best *suited* can mean many things, thus, first clear testing parameters to measure need to be defined. While many parameters exist for measuring algorithms' performance, especially from the field of computer science, like time and space complexity, this is not the only factor to consider. Thus, suitability will not only be measured in terms of performance. Accuracy of paths is also important to consider, but for this the paths have to be visible, and thus the paths need to be simulated to check accuracy. Lastly, the length of the paths is a factor to consider when determining the efficiency of a pathfinding algorithm. Lastly, as mentioned in Section 1.2.2, an algorithm is only successful in the first place, when it is able to compute paths before the fire has invalidated the path that was calculated.

From the literature, the three pathfinding algorithms present themselves as being suited for this type of pathfinding: A*, Theta* and D*-Lite. A* is efficient and fast, albeit not great in dynamic environments. Theta* facilitates any angle movement, and could thus lead to better paths, and can be extended to Phi* for dynamic maps. D*-Lite is a planning algorithm for unknown 2D grids, thus making it potentially suitable for 3D grids.

Considering all the above, the following data will be compared across the algorithms, in order of importance:

1. Success: is an algorithm able to recalculate a path fast enough (i.e. in real time) in the dynamic scenario, and is it thus able to find a path at all or will it constantly be behind on reality.

2. Path length: the length of the paths produced is important when comparing algorithms.

3. Time: the number of seconds it takes to find a path or recalculate a path. This is a straightforward measure of efficiency of an algorithm. Conversely, the theoretical time complexity of an algorithm can also be determined, and presented in *Big O*-notation.

4. Size of data structure: the size of the data structure required to facilitate this pathfinding method. The data structure is key to the performance of the algorithm, and also part of this research. The size of the data structure is relevant for the performance of the algorithm.
5. Nodes visited: the number of nodes the algorithm has to find a path. This is a straightforward measure of efficiency of an algorithm. Conversely, the theoretical space complexity of an algorithm can also be determined, and presented in *Big O*-notation.

This means, a simulation environment will have to be created to compare algorithms across multiple scenarios. This simulation environment needs to be visible and variable, meaning that it has to be possible to dynamically change the parameters of the simulations in terms of people to evacuate and number of fires. What these parameters will entail is of course dependent on the dataset, but for every distinct space in the dataset, there should be the possibility of starting a path, so as to test a fictitious *maximum capacity* of the test building.

Lastly, a test dataset needs to be constructed. This dataset should have some key features to bring the simulation closer to reality than the simple pathfinding grids presented in papers like Koenig and Likhachev [2001], Koenig and Likhachev [2002], and Nash et al. [2009]. However, it should not be as complex or cluttered as the real world can be, as this is only exploratory research into the use of pathfinding algorithms on voxels.

3.2 A VOXELISED SPACE FROM A MESH

The first step is the creation of the voxelised space on which to test. While the actual voxelisation of meshes or point clouds is also the subject of research [Aleksandrov et al., 2021], it is not the focus of this thesis, so for this research a simple voxelised space is assumed to be the starting point of pre-processing the data for pathfinding.

3.2.1 Properties of the voxel grid

The voxel grid can be thought of as a 1D array that holds information about the voxels:

$$voxels = \begin{bmatrix} (0,0,0) \\ \vdots \\ (x_{max}, y_{max}, z_{max}) \end{bmatrix}$$

Where the z runs fastest, then y, then x. Meaning to loop over all the voxels, a triple nested loop according to Algorithm 3.1. Consequently, this means that internally, the voxels are ordered according to scan-line ordering, as such:

$$coord(x,y,z) \mapsto index\langle x + y \cdot x_{max} + z \cdot x_{max} \cdot y_{max} \rangle$$

This fact ensures fast access of voxels by coordinates, and is also the basis for fast neighbour access. However, what the array holds, is simple values, that denote the status of the voxel itself, not a *voxel* entity or object. In this simple voxel grid, simple integer values encode meaning, with Table 3.1 showing what the different values encode for. Lastly, the voxel grid also holds the voxel size, which is used to render the voxels.

By using only integer values to encode 3D information about the voxels, a lot of space is saved, because an integer [3] holds only 2 bytes, whereas even a simple data package

3 While it is possible to use 1 byte sized `char`, a choice was made at the start of the research to use `int` because of eventual scaling

encoding semantic, coordinates, perhaps neighbour pointers, would be in the order of 100s of bytes, as shall be shown in Section 3.3. Using this, the size of the voxel grid is $x_{max} \cdot y_{max} \cdot z_{max} \cdot 2$ bytes.

Algorithm 3.1: Looping over the voxel grid

```
1 for x ← 0 ... x ← x_max do
2     for y ← 0 ... y ← y_max do
3         for z ← 0 ... z ← z_max do
4             access voxel at (x, y, z)
```

Integer value	Meaning
0	air
1	filled
2	walkable
3	fire
4	fire
5	fire
> 10	path

Table 3.1: The different integer values with their corresponding meaning.

3.2.2 Filling the grid

When using 26-connectivity for voxels in combination with hollow walls that are positioned diagonally on the grid, tunnelling (see Figure 3.1) can occur [Aleksandrov et al., 2021]. While this can be remedied with a *thicker* voxelisation algorithm that uses a supercover line rasterisation algorithm, it can also help to fill walls and other cavities that ought to be actual *solids* to ensure tunnelling does not happen.

To fill the walls, floors and other cavities that a building might have, a simple flood fill algorithm was used, with a seed point being selected randomly that is *known* to be in inside of the cavities. When using a clean dataset of a real building, one can assume that all hollow spaces are connected, as floating hollow spaces do not occur in real life, therefore, only one seed point needs to be chosen. The flood fill algorithm is a simple algorithm often used in 2D for filling pixel spaces, but is easily adapted to 3D as shown in Algorithm 3.2.

Algorithm 3.2: Flood fill algorithm

```
   Input: s_seed(x, y, z), stack S
1  ; S ← ∅ S.add(seed)
2  while S ≠ ∅ do
3      s_current ← S.top()
4      S.pop()
5      s_current ← fill
6      foreach s_neighbour ∈ s_current do
7          if s_neighbour ≠ filled then
8              S.add(s_neighbour
```

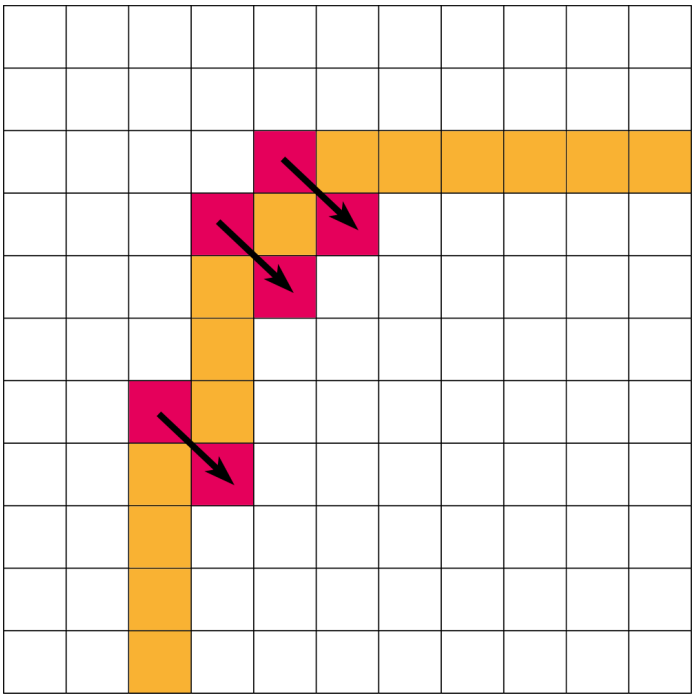


Figure 3.1: Tunnelling between 8-connectivity in a 2D pixel grid

3.2.3 Extracting navigable space

To ensure paths are found in spaces that actual users can also use for walking around in a building, the navigable space needs to be defined. What is used, is the navigable space as defined by Staats et al. [2017]: "free surfaces that are used to navigate inside a building without bumping into any obstacles". To determine this navigable space, first, walkable surfaces have to be determined. In the case of voxels, this is simple for all voxels where the following is true:

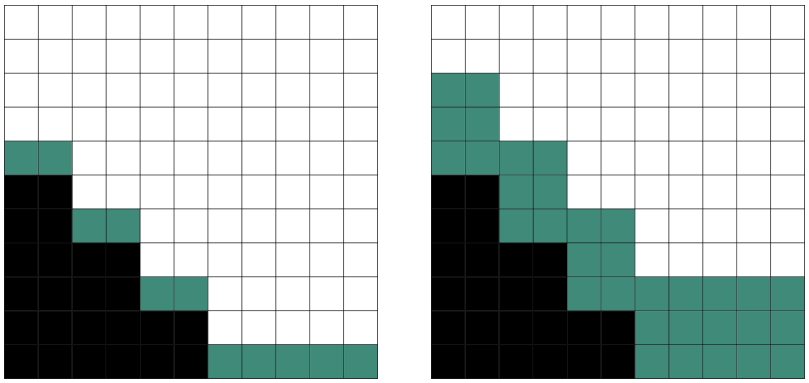
Theorem 1. *Given voxel* $s(x, y, z)$*, if* $s_{neighbour}(x, y-1, z) =$ *filled, then voxel* s *is a surface voxel*

Thus, using this given, the surface voxels can be defined. There is however, more to it, because not all floor surfaces are navigable for humans for locomotion. There needs to be enough space for a human actor to safely stand. To do this, one simply has to check that for all the floor voxels, a set number of voxels needs to be empty, depending on the voxel size and the size of the dataset. If these voxels do not have the aforementioned required space above them, they should be marked as non-walkable.

Once the voxels with too little vertical space above them have been unmarked as being walkable, layers of walkable voxels should be placed on the current layer to ensure connectivity between voxels that are part of vertical or inclined walkable spaces, such as stairs. If only the layer directly above the floor is demarcated as walkable, and the step height is larger than the voxel size, vertical movement along these stairs would never be possible, as shown in Figure 3.2.

3.2.4 Generating starting points

Currently, starting points are generated by selecting them manually in the dataset, however, they could also be automatically created by using a watershed algorithm on the dataset Koopman [2016], Gorte et al. [2019a]. Every starting point corresponds to a person for which a path has to be computed.



(a) No connectivity with one layer of walkable voxels

(b) Full connectivity with three layers of walkable voxels

Figure 3.2: The thickness of the walkable layer

3.3 CREATING THE SPARSE VOXEL OCTREE

The SVO was created according to the method of Baert et al. [2013], which is a bottom up approach. While this method was created specifically for out-of-core construction of SVOs, the method is perfectly acceptable and adaptable for other purposes. In this section, the steps taken to construct the SVO from the voxel grid described in Section 3.2 are documented.

3.3.1 Morton Ordering

The first step is to, for every voxel in the voxel grid which is walkable, generate Morton codes, and add these to a list. After all these Morton codes have been generated and added to an array, this array should be sorted in ascending order. This means the voxels itself are sorted in ascending order, because $n_{morton}(0,3,0) = 18 < n_{morton}(1,3,0) = 19$. Doing this ensures spatial locality and coherence when having to create octants for the octree.

3.3.2 Creating levels

The next step is to create the levels (full procedure explained in Algorithm 3.3, starting with the lowest level (i.e. the leaf level). This is done using the aforementioned sorted array of Morton codes. When using this array as a sort of stack, where the *top*() operation refers to the entry in the stack as the smallest Morton code, and this entry can be removed using *pop*() operation, one can create octants by counting up from 0, and by checking if the Morton code for a certain number exists in the stack. If it does, an object must created for this octant. This object should have a few properties: a pointer to the parent node, eventual pointers to the children nodes, coordinates, attributes, a Boolean to signify if it has changed and a level number.

This is done for all octants, and if the octants are not all empty, a parent node is also created. The children and their parent are added to a map that maps index (which is the Morton code) to the node objects for easy access. If the octants are empty, no nodes and no parent nodes are created, thus only creating and storing parts of the space where there is data. This procedure is done until the entire array (or stack) of Morton codes has been processed. If this is the case, all the input voxels have been converted to a node, with the accompanying semantic information (index, parents, children in case of non-leaf nodes) also being recorded.

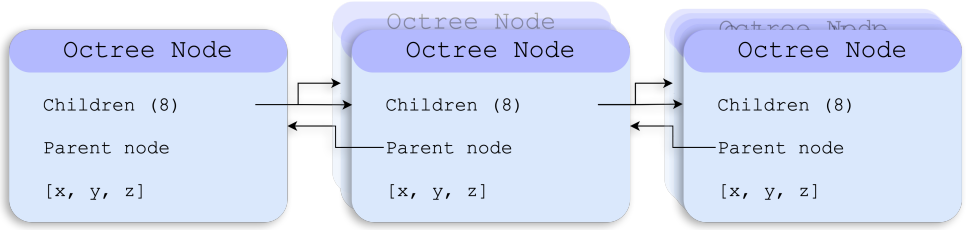


Figure 3.3: A schematic overview of the essential properties of the nodes and their relationships

Algorithm 3.3: Creating a level of the sparse voxel octree

Input: stack S_{morton}
Output: map of node objects M_{level} and M_{parent}

```
1  n ← 0
2  index ← 0
3  while S_morton ≠ ∅ do
4      empty ← true
5      create new node n_parent
6      for i ← 0 ... i ← 8 do
7          c ← S_morton.top()
8          if c = n then
9              create node s
10             S_morton.pop()
11             M_level[c] ← s
12             empty ← false
13         n++
14     if empty = false then
15         M_parent[index] ← n_parent
16     index++
```

3.3.3 Neighbour Access

As discussed in Figure 2.3.2, neighbour access and consequently generation is not as straightforward due to the non-uniformity of the octree. Additionally, due to how the nodes are stored in a hash map, they can be accessed in constant time by using their Morton code index, thus the fastest way to access the neighbours of a current node in the octree is if you have the Morton codes of the neighbour.

To achieve this, a method is used to take advantage of the properties of Morton codes bit interleaving. This method takes advantage of the fact that for a given node $v(x, y, z)$ with Morton code i, if one wants to know $(x+1, y, z)$, the difference in i is dependent on x, and independent from y and z. This same holds true when moving in other axis directions [Arndt, 2010]. Due to the way Morton ordering works, the offset Δi in each direction can be pre-computed for every voxel coordinate. This relationship is shown for an example voxel in Figure 3.4. Notable is that the relationship between the offset Δi in each direction is a multiple of the previous direction.

It is possible to pre-compute the offset value for each voxel and use a lookup-table to access these values, the full look-up table can be found in Appendix B. This way, one can access a voxels' neighbours by coordinate and index and check if these exist in the SVO. Of course, these operations can then be combined, to create 6-connectivity, 18-connectivity and 26-connectivity by using earlier computed neighbours. Both these

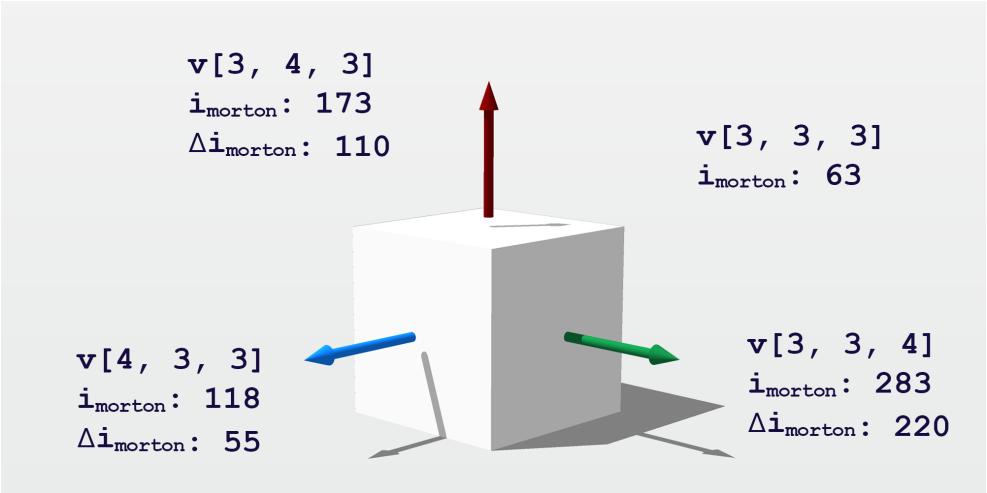


Figure 3.4: The relationship between Morton codes and the Morton codes of neighbours in an axis direction.

operations are done in constant time due to the fact that the SVO is built using a (hash) map and checking a key in a (hash) map is, on average, done in constant time.

While implementing the method described in Figure 2.3.2 has some advantages, that implementation is fully based on a static SVO, where there is always time to pre-compute the neighbour relations, and then use them at run-time. This is not possible in a dynamic SVO, where not only does the SVO need to recompute some relationships already, but running the neighbour algorithms costs even more time and resources.

This method does however only consider nodes on the same level as the node for which the neighbours are computed, so no parent or child nodes are considered as neighbours. This has both downsides and upsides. An obvious downside is that no use is made of the hierarchical nature of the data structure, but this is not truly the case: the fact that nodes are organised with this much spatial coherence is due to the data structure.

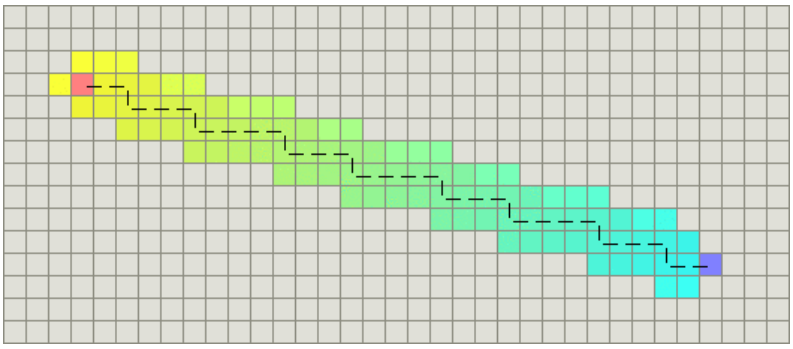
3.4 ADAPTING A*

The first algorithm that is tested is A*, which is arguably the simplest algorithm. While the algorithm is very similar to the original as described in Section 2.4.3, some adjustments had to be made to make the algorithms work for the different data structures.

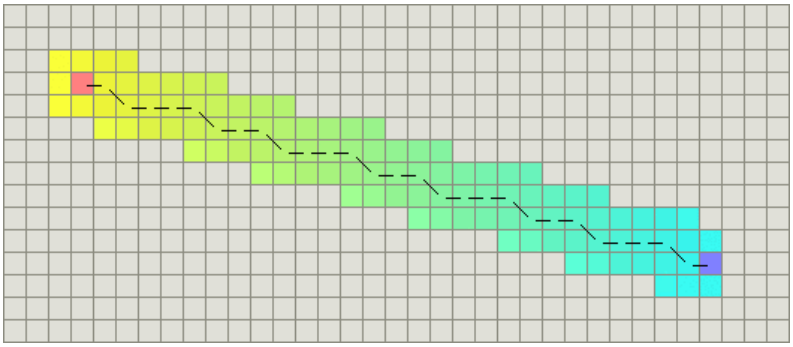
3.4.1 Heuristics

The first thing to decide is the heuristic to use for the algorithm. It is important to consider what we want to achieve with the algorithm: A* can find the shortest path accurately and fast with the perfect heuristic [Hart et al., 1968]. However, finding the shortest path often takes longer than finding a very good path, even faster. This depends on either over or underestimating the heuristic. When underestimating the heuristic, the algorithm will explore more possibilities before settling on a path, and when over estimating the heuristic it will just choose the path it has already found because all other options seem worse. Often, it is not necessary to find the *true* shortest path, because there could be many shortest paths, an a good path is also sufficient

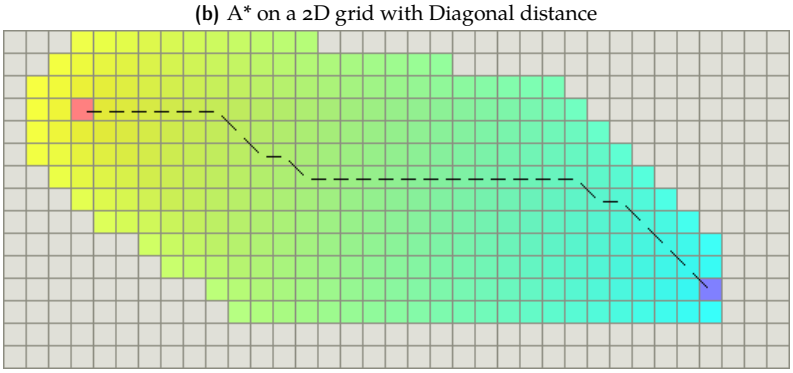
for the task at hand. What a "good" path is, is a testing and balancing act: a perfect heuristic leads to the true shortest paths, while a cheap overestimated heuristic will lead to faster computation. Therefore, there are a few options to use.



(a) A* on a 2D grid with Manhattan distance



(b) A* on a 2D grid with Diagonal distance



(c) A* on a 2D grid with Euclidean distance

Figure 3.5: Comparison of different heuristics on a 2D grid. Figures from https://www.redblobgames.com/

While A* can move in all 26 directions provided by 26-connectivity, it is not an any-angle pathfinding algorithm, therefore it does not make sense to use Euclidean (L_2) distance, because this would often lead to underestimating the distance between the current node and the goal node, because Euclidean distance cuts corners that A* cannot make. Additionally, calculating Euclidean distances with their quadratics and square roots, is rather expensive, and does not pay off if you want to find paths quickly.

$$d_{Euclidean}(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2}$$

Another option is to use Manhattan distance, which is also known as the taxicab or L_1, which more accurately predicts the distance when using 6-connectivity. Manhattan distances are less expensive than Euclidean distances, only additions and subtractions. With Manhattan distances, moving in all directions is considered as being 1

grid unit. Therefore, when using Manhattan distance, the cost function of the algorithm should also be accordingly adapted.

$$d_{Manhattan}(a,b) = |a_1 - b_1| + |a_2 - b_2| + |a_3 - b_3|$$

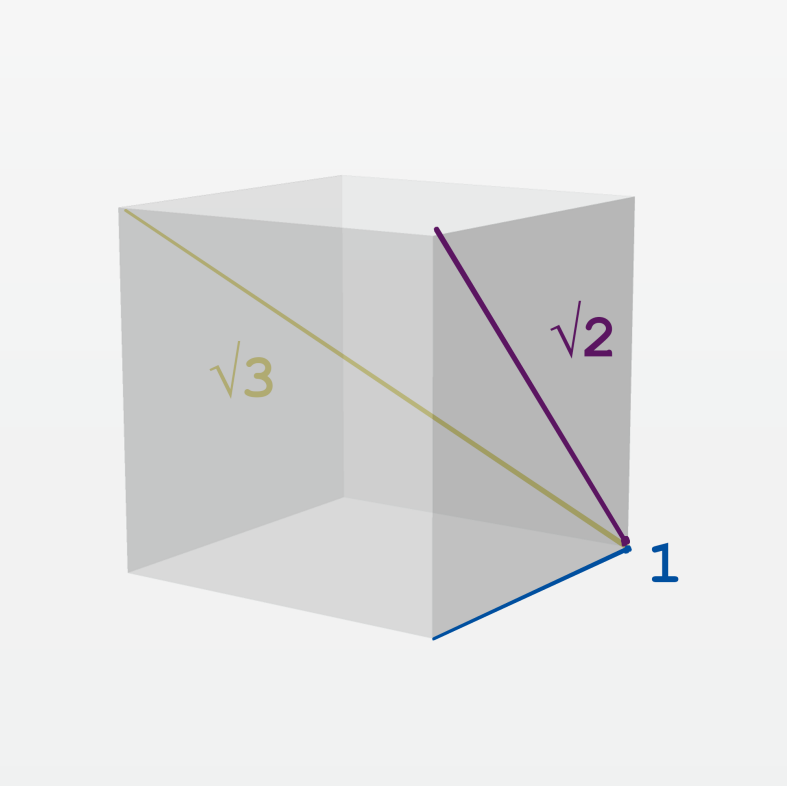


Figure 3.6: Distances on a cube

Thirdly, there is the Diagonal distance (L_∞), which attempts to take into account moving diagonally without resorting to the true, Euclidean distance. It can be thought of as a generalisation of Euclidean distance, adapted for the grid. The distance takes into account that for face neighbours, the distance needs to be scaled by 1, for edge neighbours, this distance needs to be scaled by $\sqrt{2}$, and for vertex neighbours, this distance needs to be scaled by $\sqrt{3}$, as per the Pythagorean theorem, made visual in Figure 3.6.

$$\begin{aligned} d_{Diagonal}(a,b) = &(\sqrt{3} - \sqrt{2}) \cdot min(\Delta x, \Delta y, \Delta z) + (\sqrt{2} - 1) \cdot (\Delta x + \Delta y + \Delta z) \\ &-min(\Delta x, \Delta y, \Delta z) - max(\Delta x, \Delta y, \Delta z) + 1 * max(\Delta x, \Delta y, \Delta z) \end{aligned}$$

For the cost function (the cost from moving from one node to the next), the metrics described above are also relevant, because this determines how *optimistic* or *pessimistic* A* will be about how much distance remains. If for instance, one uses a relatively accurate cost based on the Diagonal distance, but uses a heuristic that underestimates the distance remaining like Euclidean, the algorithm will spend more time trying to find the best path, with the opposite of course being true if it is the other way around.

For this research, Euclidean distance will be used, to facilitate as much freedom in movement for the evacuees and to find the shortest paths. In Chapter 4, I will show the different heuristics and their performance in action.

3.4.2 Regular Voxel Grid

The algorithm is suited for a voxel grid with good neighbour access, which is the case as described in Section 3.2. While the algorithm does call for some features like keeping track of things per node, and the voxel grid does not store nodes, only integers that convey meaning, this can be solved by slightly changing how A* keeps track track of its $g(n)$ values. Instead of storing this information in the grid itself, one can maintain an extra associative container (map, dictionary) to store the nodes that A* has already expanded, and record the $g(n)$ of the nodes there. This way, you only store what you use, and you do not have to check the entire grid if you want to look up a voxels $g(n)$ value.

Algorithm 3.4: On the fly neighbour generation for A*

```
1 ...main A* loop
2 foreach coordinates [x, y, z] of 18-connectivity of s_current do
3     check status in grid for [x, y, z]
4     if [x, y, z] = walkable then
5         create s_neighbour
6         continue with algorithm
7         ...
8     else
9         delete s_neighbour
```

Another thing to consider is how to deal with changes in the voxel grid. A* should only consider nodes that are walkable, all other nodes should not be accessed. However, this grid does not hold any nodes, only integer values that convey meaning. Thus, the simulation will change the integer values of in the grid, and nodes will only be created if it is created. This way of dynamically creating nodes is the key to adapting A* for this grid. The downside of not storing information for longer than you need is that you cannot deal with dynamic changes in a clever way, so when the grid changes, you have to recalculate the entire path. However, even when doing these repeated *one shot* A* runs, the speed and efficiency of the algorithm will be high.

3.4.3 Sparse Voxel Octree

Because the SVO stores a lot more information than the regular grid, it is easier to deal with dynamic changes in the scene. Other than being able to *know* when a node has changed, making it easier to change only those paths that have changed nodes in them, the algorithm runs the same as with the voxel grid. However, because the SVO stores only voxels that are walkable, and voxels are removed from the tree if they are no longer walkable, the algorithm need only check if the tree contains a neighbouring node before proceeding.

3.5 ADAPTING THETA*

These algorithms are much more complex than A* and require more information per node to function. They also need more auxiliary functions to operate, like a line of sight function. While the algorithm does use a heuristic, and it's function is rather similar to that of A*, it is less influential on the entire algorithm as a whole because other factors also influence the algorithm, making it more informed and less dependent on the whims of a single heuristic. However, because Theta* is an any-angle

pathfinding algorithm, the heuristic to use is Euclidean distance, because this comes closest to the path Theta* will probably find.

As stated in Section 2.4.4, the only essential difference between Theta*/Phi* and A* is the compute cost method (see Algorithm 2.3), which uses line of sight checks to cut corners before regressing back into A* if there is no line of sight between two nodes. But because Theta* maintains extra values to keep track of this, a node that is able to store more information is required. Where in A* the node was only used to keep track of the $g(n)$ values, Theta* also needs to store a parent relationship so the algorithm can skip nodes between which there is a line of sight.

The line of sight function is a well known algorithm that is used for ray tracing in the field of computer graphics. This algorithm can be thought of as a rasterisation algorithm in 3 dimensions, and was first described by Cohen-Or and Kaufman in 1997. The algorithm is described in Algorithm 3.5.

Algorithm 3.5: Line of Sight

```
1  Function sign(x):
2      return (x > 0) − (x < 0)
3  Function line(x, y, x, Δx, Δy, Δz):
4      line ← ∅
5      sx ←sign(Δx), sy ←sign(Δy), sz ←sign(Δz)
6      ax ← |Δx|, ay ← |Δy|, az ← |Δz|
7      bx ← 2 · ax, by ← 2 · ay, bz ← 2 · az
8      exy ← ay − ax, exz ← az − ax, ezy ← ay − az
9      n ← ax + ay + az
10     while n− do
11         check voxel v[x, y, z] is walkable
12         add v to line
13         if exy < 0 then
14             if exz < 0 then
15                 x ← x + sx
16                 exy ← exy + by, exz ← exz + bz
17             else
18                 z ← z + sz
19                 exz ← exz − bx, ezy ← ezy + by
20         else
21             if exz < 0 then
22                 z ← z + sz
23                 exz ← exz − bx, ezy ← ezy + by
24             else
25                 y ← y + sy
26                 exy ← exy − bx, ezy ← ezy − bz
```

If the line of sight algorithm returns a line with no obstructions, then the algorithm knows that there is a path between the two voxels, meaning it can go to the node on the other end of the line, and continue pathfinding from there.

3.6 ADAPTING D*-LITE

D*-lite and LPA* both, could theoretically be applied somewhat directly on the regular grid, and with some adaptions to the SVO. However, in practice, the algorithm

runs into problems when adapting it to 3 dimensions, especially when dealing with obstacles. The D*-Lite algorithm especially does not perform well in 3D with obstacles with some initial testing. The problem lying in the manner in which ties must be broken. Often, there might be multiple nodes with the same $g(n)$ and/or $rhs(n)$ values, and the algorithm can get stuck. This is where tie breaking functions come in. One popular way is to prefer paths that lie closest to the true distance (as the Nazgûl flies) between the start and goal node [Patel, 2022].

Algorithm 3.6: D*-Lite/LPA* with tie breaking

```
1  while s_current ≠ s_goal do
2      ...main planning loop
3      c_min ← ∞
4      t_min ← ∅
5      foreach s_neighbours ∈ s_current do
6          f(s) ← cost(s_current, s_neighbour) + g(s_neighbour)
7          straightLine ← d_euclidean(s_neighbour, s_goal) + d_euclidean(s_start, s_neighbour)
8          if f(s) ≈ c_min then
9              if t_min > straightLine then
10                 t_min ← straightLine
11                 c_min ← f(s)
12                 s_min ← s_neighbour
13         else if f(s) < c_min then
14             t_min ← straightLine
15             c_min ← f(s)
16             s_min ← s_neighbour
17     s_current ← s_min
```

Another hurdle is the fact that every instance (i.e. every start-goal iteration) of D* will have to hold their own version of the entire dataset, because the $g(n)$ and $rhs(n)$ values of the entire grid are dependent on the start and the goal, making it less efficient than on-the-fly node generation like in A* or even Theta*.

3.7 FIRE SIMULATION

To simulate the dynamic emergencies spoken of in the research questions, fire has to be simulated in the voxel grid. This will be done by abstracting a fire to a growing area of inaccessible voxels. The growth rate of this area should be able to vary. Two algorithms are used to achieve this: one for the regular grid, and one for the SVO. For the regular grid, the well-known midpoint circle algorithm (see Algorithm 3.7), optimised for integers, is used, and for the SVO a modified flood fill algorithm is used (as described in Algorithm 3.2).

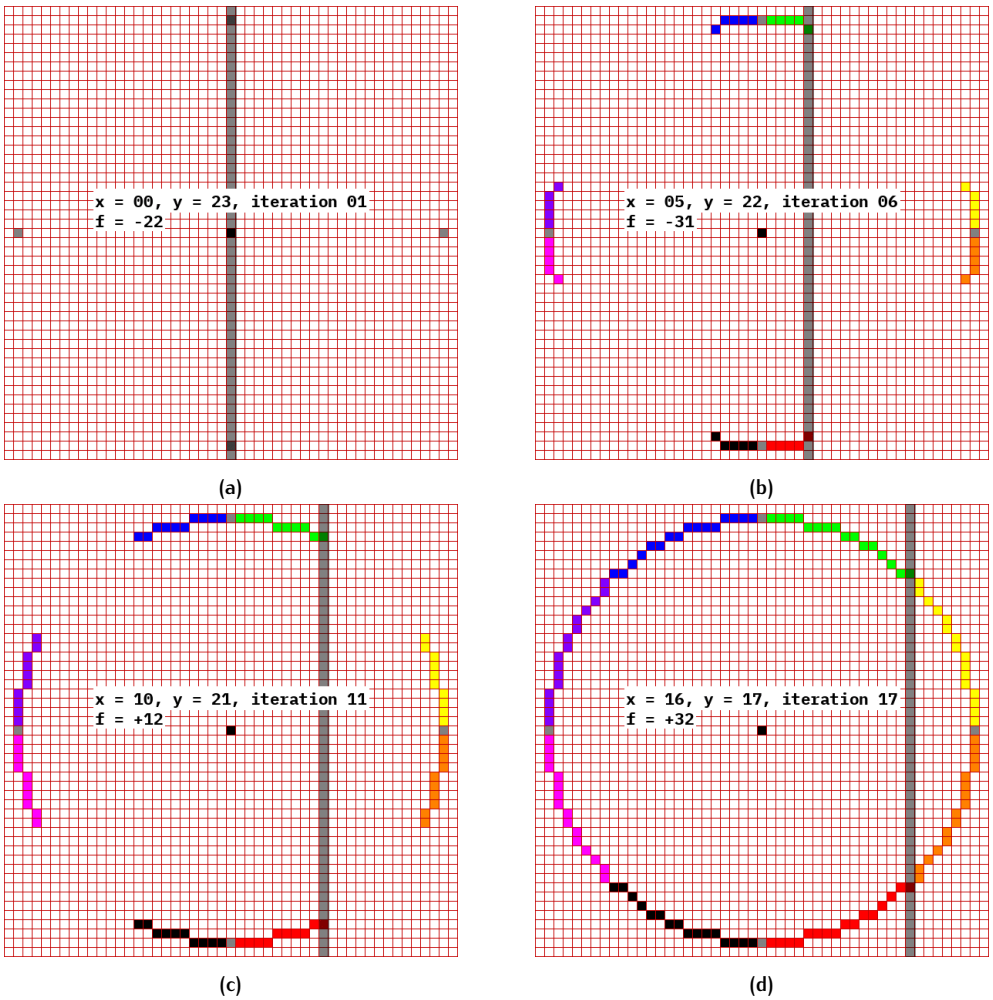


Figure 3.7: The midpoint circle algorithm in 2D. Figure from Wikipedia

The midpoint circle algorithm is performed by calculating one octant of the circle, and then mirroring this in all other octants. The algorithm works by trying to approximate the circle formula on the $y - plane$ with $x^2 + z^2 = r^2$ on a 3D grid where y is up. Thus every voxel on the circle should be about the same distance from the centre of the circle. The algorithm does this by starting out at the top (i.e. 0°, north) of the circle and choosing voxels that satisfy $x^2 + z^2 \leq r^2$ while maximising $x^2 + z^2$. The algorithm is run inside the simulation for an increasing radius value to simulate a growing fire. Once the horizontal limit of the fire is reached, the algorithm will be run on the voxels above the current circle, thus producing a growing cylinder of "fire".

For the SVO, a limited 2-dimensional version of the flood fill algorithm is used to simulate the fire. The algorithm is limited in both its speed and it's spread, ensuring

a comparable spread to the grid based fire algorithm. The algorithm will produce a sheet of "fire" voxels, then also travel upwards, giving the impression that a fire is rising. However, since the SVO only holds the walkable voxels, it will actually remove voxels from the tree.

Algorithm 3.7: Midpoint circle algorithm optimised for integer only maths

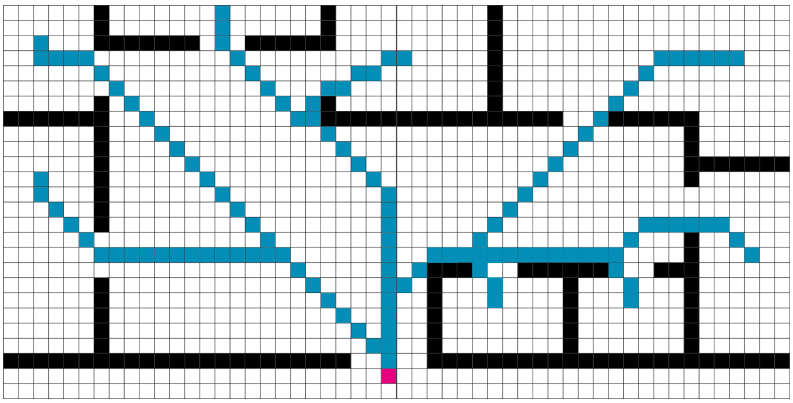
Input: $(x_0, y_0, z_0), radius$
$f \leftarrow 1 - radius$
$d(F_x) \leftarrow 0$
$d(F_z) \leftarrow -2 \cdot radius$
$x \leftarrow 0$
$z \leftarrow radius$
$plot(x_0,\ y_0,\ z_0 \pm radius)$
$plot(x_0 \pm radius,\ y_0,\ z_0)$
while $x < z$ **do**
  if $f \geq 0$ **then**
    $z \leftarrow z - 1$
    $d(F_z) \leftarrow d(F_z) + 2$
    $f \leftarrow f + d(F_z)$
  $x \leftarrow x + 1$
  $d(F_x) \leftarrow d(F_x) + 2$
  $f \leftarrow f + d(F_x) + 1$
  $plot(x_0 \pm x,\ y_0,\ z_0 + z)$
  $plot(x_0 \pm x,\ y_0,\ z_0 - z)$
  $plot(x_0 \pm z,\ y_0,\ z_0 + x)$
  $plot(x_0 \pm z,\ y_0,\ z_0 - x)$

3.8 SMARTER PATHS: TIME-AWARE A*

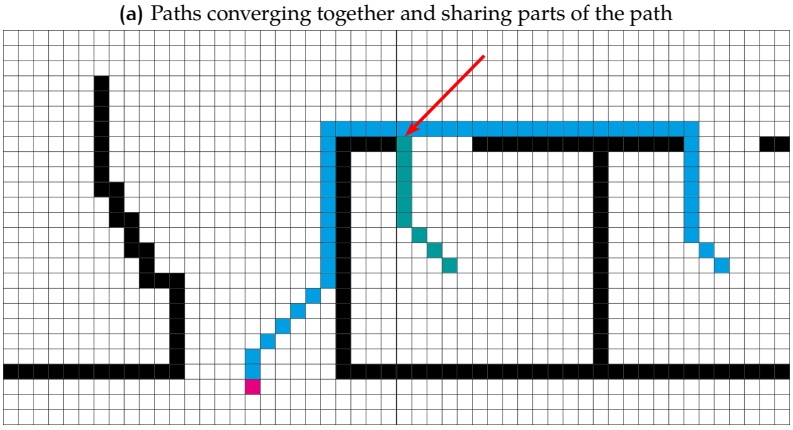
To accurately reflect the occupancy in a building, one voxel should only be able to hold 1 person at the time, which is already an abstraction because of the fact that the voxelsize is much smaller than the occupied space by an actor. An issue however, is that with A*, when looking for the optimal path from multiple locations to an exit, the paths will converge like a river with tributaries in a sort of shortest path graph. Meaning that all paths will inevitably join together. Another side effect is that if paths are not allowed to use the same voxels, a path might think that there is no path to an exit because it has been "cut of" by a path going in front of the door, as is visible in Figure 3.8.

Therefore one should allow paths to share voxels, but try to make sure that they do not constantly share their paths. This can be solved by allowing paths to cross paths, and share voxels between them for one or two voxels, but no more. This can be implemented for all algorithms.

Another option is to use incremental paths, that have a moving starting voxel that travels along the path at the speed at which people would walk, which could be combined with the adage that one voxel can only hold one person/path at a time. Doing this would also allow the use of changing the goal voxel mid-route. To do this, an adapted version of A* will be used, that will in essence be the same as the version of A* described in Section 3.4.3, using the same main loop, but for a time period defined



(a) Paths converging together and sharing parts of the path



(b) A path cut off by another path

Figure 3.8: Some difficulties when not allowing paths to share voxels

by the *voxel speed*, which is defined as the normal walking speed converted to voxel units. Where the voxel speed can be defined as:

$$v_{voxel} = \frac{v_{walking}}{s_{voxel}}$$

Using this one can determine the amount of time a pathfinding simulation must wait before moving one voxel, which can be determined as:

$$t_{voxel} = \frac{1}{v_{voxel}}$$

If, the algorithm on every interval moves the start voxel up one voxel in the path, and sets the previous starting voxel to free again, one could satisfy the one person per voxel rule.

4 IMPLEMENTATION

In this chapter, the implementation of the methodology is presented. First, the testing environment will be presented. Next, the steps taken to setup pathfinding on a grid and on the Morton grid. Thirdly, adapting the algorithms to voxel spaces. All code can be found at GitHub.

4.1 SIMULATING AND TESTING

Starting off the implementation is an environment in which to test all the aforementioned algorithms and data structures, as well as providing visual feedback. Thus, an application will have to be developed to serve as both a testing ground for the algorithms, as providing direct visual feedback on what the algorithms are doing. Additionally, the path data will be outputted by the application for further analysis. As the methodology calls for real-time recalculations, the best way to check the "real time-ness" of the pathfinding is to visualise it, in real time.

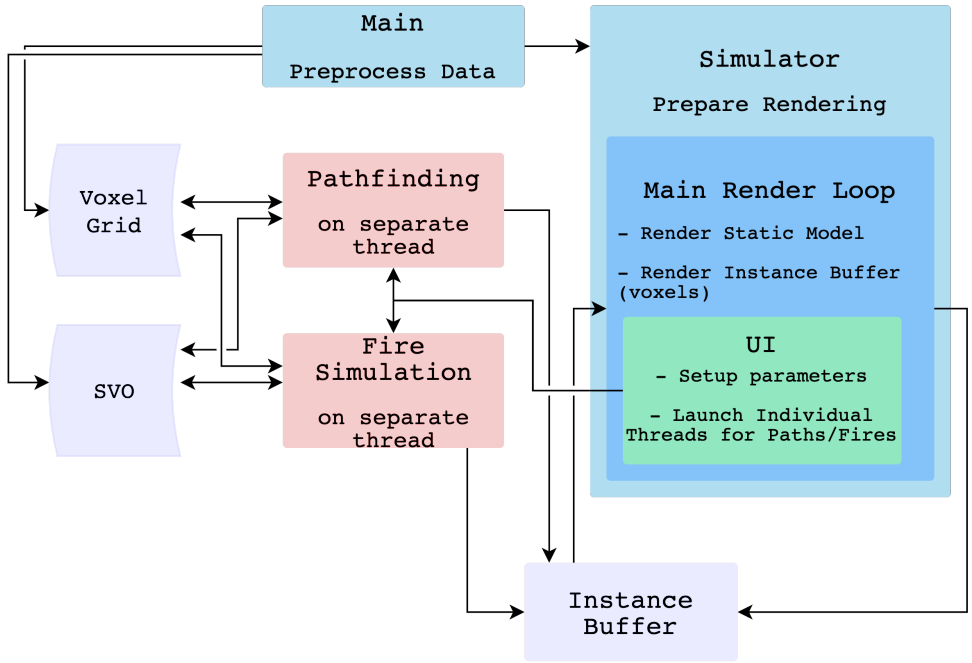


Figure 4.1: A schematic overview of how the methodology is implemented.

The tool to do this all, the C++ programming language is used to build an application using a few dependencies, namely Magnum, which is a lightweight middle-ware layer on top of raw OpenGL, making it easier to setup a basic Simple DirectMedia Layer (SDL) application, while also providing integration with Dear ImGui, which is a "a bloat-free graphical user interface library for C++" Cornut [2020]. The choice for C++ is for its speed, and widespread use in the graphics field, making it easier to find resources, as well as personal preference. However, it should be noted that

in this research C++ means the ISO [2020] standard informally known as C++20, the most recent standardised version of the language. The CXX compiler that is used, is the LLVM AppleClang compiler version 13.1.6.13160021. The full code of the application can be found on GitHub.

4.1.1 OpenGL and Magnum

OpenGL is an open-source, free application programming interface (API) for interaction with the graphics processing unit (GPU) to render anything from games to virtual reality to CAD and information visualisation. Therefore it is the ideal tool for this research, because it is open, and provides the most direct access to the hardware of a computer, which means high performance, so as to not impede with the performance of the real-time pathfinding. Raw OpenGL however, can be quite cumbersome to deal with, especially when one is not an experienced graphics programmer. Therefore, middle-ware abstraction layers over OpenGL exist, with varying degrees of control over OpenGL itself. One of these libraries is Magnum, which makes things easier while allowing control wherever the developer feels the need to do things "their way".

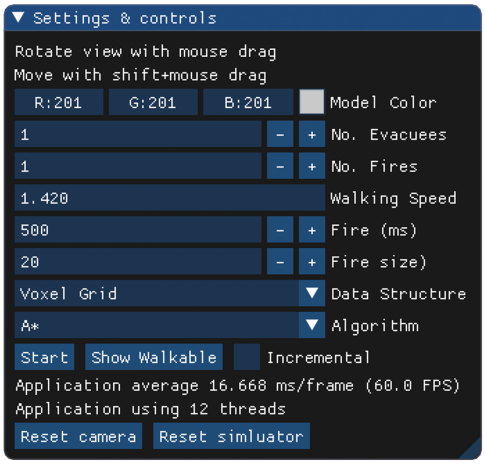


Figure 4.2: The UI designed for the application

In Figure 4.2, we can see that there are a number of simulation parameters that can be changed. The number of evacuees can be changed, as can the number of fires. Apart from this a number of speed parameters can be changed, and lastly, the appropriate algorithm and data structure can be selected. Once everything is set to the required simulation, the start button can be pressed. It should be noted however, that there exists a starting points for each room in the building (30), and that if one for instance were to select only 4 evacuees, then 4 rooms will be chosen at random, with no preference for path length or something else.
Magnum simplifies the OpenGL render pipeline for the user, so no time is wasted setting up buffers, writing simple shaders or other tedious tasks. For this project, the standard Magnum/SDL template application was used as the basis for the simulator application. While the entire dataset is of course a voxelised space, for practical purposes, the static part of the dataset (i.e. the walls, floors etc.) are imported as a mesh (.obj file). This static mesh is rendered in the application as a semi-transparent solid to see inside the building while the paths are being rendered, so the paths are not obscured by the building when running the simulation.

Rendering the paths is a more complicated affair. To do this, an instancing approach was chosen. In graphics, instancing refers to multiple instances of the same geometry

being rendered, thus saving memory by already knowing the to-be-rendered geometry. Instancing in OpenGL and Magnum requires the preparation of a mesh (in this application, obviously a cube will represent a voxel) that will serve as the "original" instance. The instances of this cube will be rendered according to the instance buffer. The instance buffer itself is supplied data by the a dynamic array (a `std::vector<>` called `_instanceData`)which contains the actual render information for the application. All voxels that need to be rendered are added and removed from this array as needed, making it a central part of the architecture of the application. What actually goes into the array is the `InstanceData` struct (see Listing 4.1), which consists of all the information necessary for rendering the voxel. Namely, a transformation matrix containing the position information for the voxel, the normal matrix, a colour value and an identifier, which corresponds to the `id` of the voxel as presented in Table 3.1.

```
struct InstanceData {
    Matrix4 transformation;//a specialised tranformation matrix
    Matrix3x3 normalMatrix;//a regular 3 by 3 matrix
    Color3 color;
    int id{};
};
```

Listing 4.1: Data struct used to render the voxels

4.1.2 Concurrency management

This array is thus a central part of the rendering procedure, and indeed of the entire application, as can be seen in Figure 4.1. This data container, like the SVO and the Voxel Grid, is shared among the many threads that are required to concurrently perform the pathfinding process for all the different starting points. Thus, access to this data must be managed accordingly, to prevent data corruption and data races. What this means is that two threads should not be allowed to access the same data at the same time. For if one thread would, for instance, change a value at a certain memory location while another thread is reading that memory location, data could be corrupted, or simply the wrong value could be retrieved. Therefore, these data containers need to be *thread safe*.

While many possible, and arguably cleverer ways, to prevent this exist, in this application this has been implemented using mutexes and atomic types. A mutex can be considered as a lock on a memory location, if one thread is currently accessing this memory location, it "locks" the mutex, and then when another thread tries to access the data, it sees the locked mutex and knows that it must wait. Only when the first thread unlocks the mutex, can the second thread access the data. This methods prevent data corruption. An atomic type is a special kind of a data type. For instance, an atomic integer is an integer which ensures thread safety without manually using mutexes. Therefore, whenever it is possible, when thread safety is required, first the use of an atomic type was considered, before moving on to the aforementioned mutexes.

4.1.3 Datasets

The dataset to be used is an artificial test building created specifically for this research. Attributes of the model are: three floors, two entrances/exits, two staircases, 34 distinct spaces, a balcony, and a split-level. These attributes have been chosen to simulate areas in real world buildings that do not fit easily in a 2D representation of a building. However, this model is empty, that is, there is no furniture present and there are no doors or windows. The actual size of the building is about $13m$ by $8m$ by $20m$ (x, y, z) with y being up.
This model is then voxelised online at drububu.com, with a resolution of `163x101x256`, resulting in a voxel size of $0.078125m$, and 458174 filled voxels. The voxelisation pro-

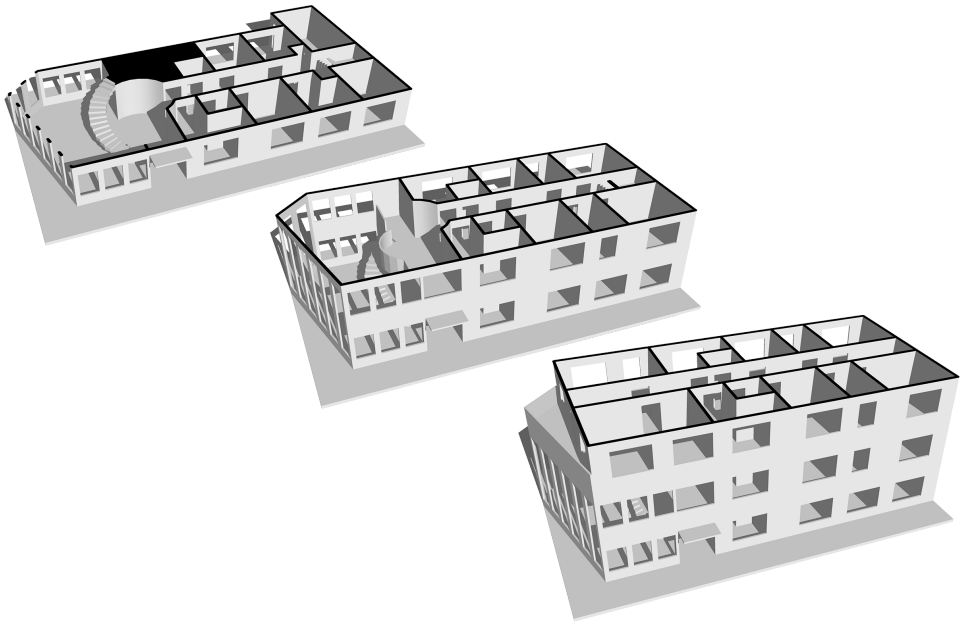


Figure 4.3: A section of each floor of the dataset model

cess outputs a `.txt` file with the index numbers of the voxels that are filled, together with the size (x, y, z) of the voxel grid. This method has been chosen to allow the most simple datasets to be ingested by the programs. Even a list of points can be used, where they will be treated as voxels.

4.1.4 Hardware

All the software designed for this approach is run on the same device. A MacBook Pro (16-inch, 2019), with a 2.6 GHz 6-Core Intel Core i7 and 16 GB 2667 MHz DDR4 of memory. The primary graphics card is the AMD Radeon Pro 5300M 4 GB. The hardware is running the MacOS Monterey 12.3.1 operating system.

4.1.5 Simulation Parameters

The parameters that were used to start the simulations were: 1 to 30 evacuees, with 1 fire in the building and two possible exits.

4.2 A VOXELISED SPACE FROM A MESH

The input `.txt` is ingested by the pre-processing element of the application, with all the voxel indices being loaded into memory by means of an `std::vector<Voxel>`. The voxel class is the first of the three voxel utility classes used by the application, with each of them serving a specific algorithm or data structure. Of these, the `Voxel` class is the most simple. The main properties of this class are visible in Listing 4.2, with a few key functions of interest additionally being shown. The < and > operators have been overloaded to ensure compliance with the way priority queues work in `C++`, namely by comparing things according to *strict weak ordering*. This is a predicate that compares two objects, returning true if the first precedes the second, according to ISO [2020]. Other functions that are key to the class are the `getNeighbours()` functions, while only one is shown in Listing 4.2, one exists for 6-, 18-, and 26-connectivity.

```
struct Voxel {
    int x, y, z, id;
        Voxel(const int &x, const int &y, const int &z) {
        this->x = x;
        this->y = y;
        this->z = z;
        this->id = 0;
    }
    ...

    bool operator<(const Voxel &other) const {
        return std::tie(x, y, z) < std::tie(other.x, other.y, other.z);
    }
    bool operator>(const Voxel &other) const {
        return std::tie(x, y, z) > std::tie(other.x, other.y, other.z);
    }

    std::vector<Voxel> getNeighbours26() const {
        //a vector of all the voxels that satisfy 26 connectivity
        return neighbours;
    }
```

Listing 4.2: The Voxel class

A bounding box is extracted from the vector of voxels as well. With this bounding box, the number of rows for each axis is determined, and an instance of the `VoxelGrid` class is instantiated. This class is the first major data structure that is to be used. It has been described theoretically in Section 3.2, but its implementation will be shown here. As described before, all the information is contained in a single `std::vector<int>`. By correcting for the dimensions of the voxel grid, as shown on line 22 in Listing 4.3, the call operator is defined as such that one can access the correct integer value through (x, y, z) coordinates. Note that the operator asserts that the coordinates are within the grid before returning, because otherwise the return statement will either return the wrong integer value, or get an out-of-bounds index.

```
struct VoxelGrid {
    /**
     * The original VoxelGrid.h file was provided by the GEO1004 course.
     */
  std::vector<int> voxels;
  int max_x, max_y, max_z;
  float mVoxelsize;

  VoxelGrid( int x,  int y,  int z) {
    max_x = x;
    max_y = y;
    max_z = z;
    int total_voxels = x*y*z;
    voxels.reserve(total_voxels);
    for (int i = 0; i < total_voxels; ++i) voxels.push_back(0);
  }

  int operator()(const int &x, const int &y, const int &z) const {
    assert(x >= 0 && x < max_x);
    assert(y >= 0 && y < max_y);
    assert(z >= 0 && z < max_z);
    return voxels[x + y*max_x + z*max_x*max_y];
  }
};
```

Listing 4.3: The VoxelGrid class

After the `VoxelGrid` class is set up, the grid has to be "filled" with the appropriate information. The temporary array holding all the voxels that are filled, is looped through, and for every coordinate, a 1 is stored at the appropriate location in the `VoxelGrid` array.

After this, a function is used to fill the hollow spaces in the floors, ceilings and walls, to ensure a solid voxel model. This function is an implementation of the flood fill algorithm as described in Algorithm 3.2. Starting from the seed that is known to be in the interior, it uses a `std::stack<Voxel>` based approach to fill the entire hollow space. In this dataset, this leads to 280.959 voxels being filled. If we add that to the number of voxels that were previously filled, we now have 739.133 filled voxels, on a total of 4.214.528 voxels, meaning 17% of the voxels are filled or blocked. However, one should note that this is not the search graph, the search graph is only the walkable voxels, which has to be extracted from the dataset next.

The extraction of the navigable space is done in a short function, which has the `VoxelGrid` as its input. This function is straightforward and based on the theorem defined in Section 3.2.3, displayed in Listing 4.4. This function simply loops over all the voxels in the grid, per dimension, and checks if for any given function that is filled, if there is a space of 24 voxels ($\approx 2m$) above the floor that is empty, it will mark the 3 voxels above it as walkable. This function leads to 252.789 walkable voxels, which in total is only about 5% of the total voxel grid, showing the low efficiency of the voxel grid. While it isn't quite accurate that the filled voxels aren't used, because the contours of the filled voxels define the navigable space, they are not traversed anymore.

```
void extractWalkableSpace(VoxelGrid &voxels){
    for ( int x = 0; x < voxels.max_x-1; ++x) {
        for ( int y = 0; y < voxels.max_y-25; ++y) {
            for ( int z = 0; z < voxels.max_z-1; ++z) {
                if(voxels(x,y,z)==1) {
                    bool space = true;
                    for(int i =1; i <24; i++){
                        if(voxels(x, (y + i), z) != 0) space = false;
                    }
                    if(space) {
                        if (voxels(x, y + 1, z) == 0) voxels(x, (y + 1), z) = 2;
                        if (voxels(x, y + 2, z) == 0) voxels(x, (y + 2), z) = 2;
                        if (voxels(x, y + 3, z) == 0) voxels(x, (y + 3), z) = 2;
                    }
                }
            }
        }
    }
}
```

Listing 4.4: Extracting the navigable space

The extraction of the navigable space is the last step in the data preparation for the regular voxel grid, which can now be used for pathfinding and rendering alike, and more importantly to create the SVO.
A full comparison of the size of the `VoxelGrid` and `Voxel` classes can be found in Table 4.1.

4.3 CREATING THE SPARSE VOXEL OCTREE

The SVO is a more complex data structure and thus more complex to create. It is implemented by the `SparseVoxelOctree` class, and maintains two separate nested containers for convenience. One of the key benefits of the SVO is that one needs to only check if a voxel exists in the octree to check if it is navigable. Thus checking if a value exists in a container is one of the key operations performed on the octree. Therefore, it is of paramount importance that this operation is made as efficient as possible, and comparing this between `std::map` and `std::unordered_map`, this operation has time complexity of $O(log(n))$ and $O(1)$ (on average, worst case $O(n)$) respectively. Additionally, the fact that there are two containers, with only the unordered container being used for data modification, means that the ordered container can act as a sort

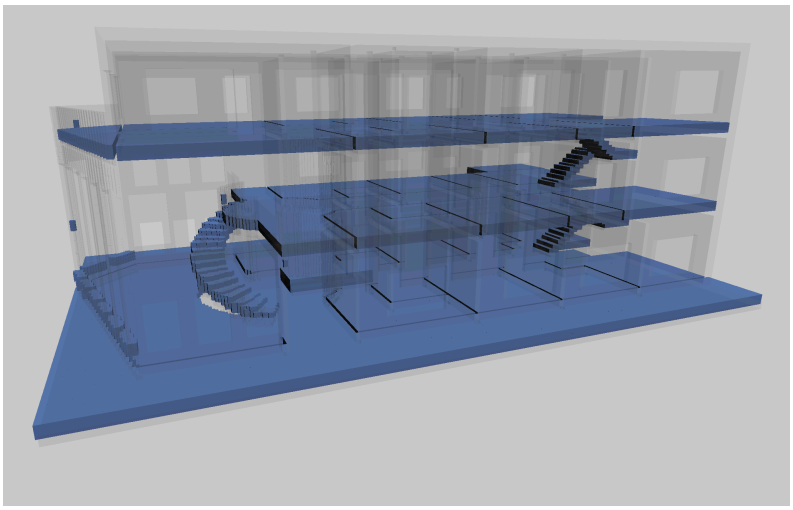


Figure 4.4: The walkable space in the dataset

of backup, so one does not have to rebuild the entire octree from scratch but can just restore it from the ordered container.

```
struct SparseVoxelOctree {
    std::vector<std::map<uint_fast64_t, OctreeNode *>> mTree;
    std::vector<std::unordered_map<uint_fast64_t , OctreeNode *>> mQTree;

    ...
};
```

Listing 4.5: The sparse voxel octree class

The `SparseVoxelOctree` class is in essence just a container class of Morton codes, where only the leaf node levels are used. What it holds is the nodes mapped to their (Morton) index, which has been implemented using the `OctreeNode` class. This class holds all the information about the nodes itself, as can be seen in Listing 4.6. The obvious necessary information is of course the nodes parent, and children, which are both pointers to other instances of the `OctreeNode` class. It holds both the nodes index and (x, y, z) coordinates, as well as some variables required for pathfinding. More importantly, it has a Boolean which knows whether the node has changed (i.e. removed from the map of Morton codes). Because these are the same nodes that make up the path, the pathfinding algorithms know that they have to recompute their path. As soon as a node has changed (i.e. set to non-walkable) the node is removed from `mQTree`, but the node itself is not deleted as it could still exist inside a path, which could lead to invalidated memory. The pathfinding algorithms are therefore required to delete nodes that have changed.

```
struct OctreeNode{
    OctreeNode *Children[8];
    OctreeNode *Parent;
    OctreeNode *PhiParent;
    uint_fast64_t index;
    uint_fast16_t attribute, level, x, y, z, h;
    float g;
    bool isLeaf = false;
    bool isFull = false;
    bool isChanged = false;

    ...
};
```

Listing 4.6: The octree node class

The SVO is created by first generating Morton codes for all the walkable voxels in the regular voxel grid. This is done using the `libmorton` library by Baert [2018]. In

essence, only two main functions of the library are used: to encode coordinates into Morton code, and to decode Morton code into coordinates. The library however is very particular in using fixed-width integer types, which is rather commonplace in graphics programming, thus not unusual, but something to take note of when doing. Thus, the two procedures that are used to convert coordinates to Morton code and vice-versa are visible in Listing 4.7.

```
uint_fast32_t  x, y, z; //unsigned integers of at least 32 bits
uint_fast64_t code; //unsigned integer of at least 64 bits

//encoding
code = libmorton::morton3D_64_encode(x, y, z);
//decoding
libmorton::morton3D_64_decode(code, x, y, z);
```

Listing 4.7: Encoding and decoding morton codes

After Morton codes have been generated for all walkable voxels, the `std::vector<>` holding these codes needs to be sorted (ascending) and then put in a `std::map<>` to ensure sequential first in first out (FIFO) access to the codes, after which the `std::unordered_map` is also filled. Because only the lowest level of the "tree" is used, it is not a full octree implementation, and therefore I will start referring to this structure as a sparse Morton grid from here on out, however, in the code snippets, this will still be called the `SparseVoxelOctree` and `OctreeNode` classes.

4.4 SIZES

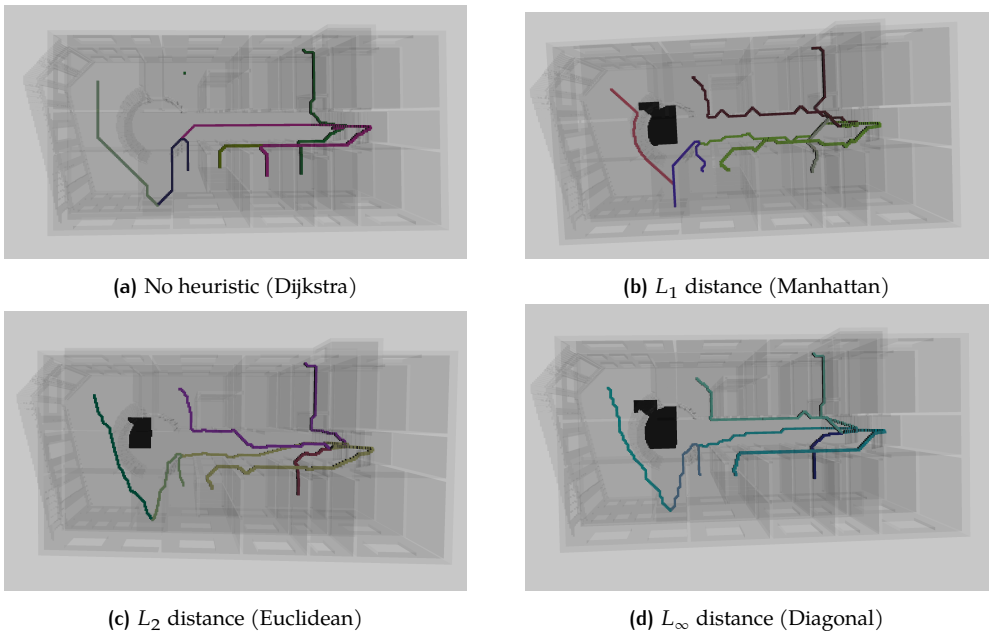
When comparing the two main data structures we can see in Table 4.1, that the `SparseVoxelOctree` is almost 3 times as large as the `VoxelGrid`, as it needs a lot of "scaffolding" to function. However, because we do not use the full `SparseVoxelOctree`, but only the leaf nodes as the `SparseMortonGrid`, we can see in Table 4.1 that this already saves quite some space.

Data object	Size (byte)
Voxel	16
Node	64
OctreeNode	144
VoxelGrid	16.858.112
SparseVoxelOctree	44.307.792
SparseMortonGrid	36.529.056

Table 4.1: Sizes of different data structures and types used for the dataset

4.5 HEURISTICS

As stated in Section 3.4.1, the choice of heuristic is of great importance to the performance of the paths generated by an algorithm, as well as the quality of the paths. Sources like Patel [2022] and Sharma and Kumar [2016] posit that Manhattan distance is the most efficient in terms of computation, as well as an oft stated rule that one should avoid having to compute the square root if one can due to its computational expense. However, in Table 4.2, we can clearly see that Euclidean distance actually has the lowest mean calculation time for a test run where the heuristics are compared. We can also see that Euclidean and Diagonal distance provide more natural looking paths, while a heuristic of 0, which is when A* regresses to Dijkstra, gives the straightforward shortest path. When looking at path lengths, we see that



(a) No heuristic (Dijkstra)

(b) L_1 distance (Manhattan)

(c) L_2 distance (Euclidean)

(d) L_∞ distance (Diagonal)

Figure 4.5: A comparison of the different heuristics in a testing run ($n_{evac} = 6$).

indeed Dijkstra does give the shortest path, with Euclidean and Diagonal both giving shorter paths than Manhattan. Euclidean gives shorter paths with vertical movement included, thus Euclidean heuristic is used for all pathfinding algorithms.

Heuristic	Mean calc time [s]	Mean number of nodes visited	Path length (43 5 106)-(12 5 79)	Path length (45 38 194)-(149 5 210)
Dijkstra	10.67	272960.72	97.94	183.27
Manhattan	7.47	92238.89	103.74	200.77
Diagonal	6.76	115149.48	98.53	187.17
Euclidean	4.92	125856.65	100.18	184.1

Table 4.2: The time statistics of the heuristics tested on A*.

4.6 IMPLEMENTING A*

When using A*, two different approaches are used for the two data structures, and both will be discussed in this section. The main differences between the approaches will be highlighted, but for the most part, the approaches are of course very similar.

The A* pathfinding is launched from the UI, in a separate thread, by the anonymous functions visible in Listing 4.8. These threads detach to separate them from the main thread. As can be seen, both functions are identical except for the data structure and associated mutexes to ensure proper data access. While most of the parameters of the function are self-explanatory, the `_running` and `i` parameters are of note: `_running` is an atomic Boolean, which is a variable that will be set to false if the simulation needs to come to an end, thus signalling all the pathfinding threads that they should stop. `i` is an ID value for every thread, which is unique and will later on also be used to identify a path in the voxel use, as shown in Table 3.1. Every pathfinding thread then finds its starting point by using the its ID value as index from the list of starting points. Next, the closest exit must be determined, which is done by simply checking which

exit is closest, according to the heuristic, which as described in Section 3.4.1, is set to Euclidean distance for A*.

```
std::thread th = std::thread{[this, i] {
    a_star_search_grid(voxelGrid, _mtx, _instanceData, _voxelsize,
                       i, speed, _running, _incremental);
}}; th.detach();//detach threads to allow them to work independently
std::thread th = std::thread{[this, i] {
    a_star_search_svo(voxelOctree, _svo_mtx, _instanceData, _voxelsize,
                      i, speed, _running, _incremental);
}};th.detach();//detach threads to allow them to work independently
```

Listing 4.8: Launching the pathfinding threads

When the algorithm has a start `Voxel`, and a goal `Voxel`, it can instantiate its utility data containers, of which there are three. A priority queue, and two maps, namely `std::map<Voxel, Voxel> came_from` and `std::map<Voxel, int> cost_so_far`. The priority queue is a more convenient wrapper for `std::priority_queue`, that has been implemented in the same manner as Patel [2022]. It has a few key functions, `put(T item, priority_t priority)`, which places an element in the queue with a certain priority value, `top()` which returns the best item of the queue, and `get()` which returns the best item of the queue, and removes that item from the queue. Because the standard C++priority queue returns the items with the largest priority, and with A* we want nodes with the lowest $f(n)$ values, the wrapper changes this by comparing items in the queue with `std::greater` instead of `std::less`.

When these data structures are initialised, the starting `Voxel` is added to the `came_from` map, with it mapping to itself (as in Algorithm 2.2, the starting node is its own parent). Also the starting `Voxel` is added to `cost_so_far` with cost 0, and lastly inserting it into the `PriorityQueue`, meaning the main loop of A* can start. The main A* loop is a while loop, which is true while `PriorityQueue` is not empty or breaks when the current voxel is the goal voxel. In the implementation, the `PriorityQueue` is called the *frontier*, which is to signify that these are the voxels that are currently being "investigated" by the algorithm. The first step in the loop is getting the voxel that has the lowest $f(n) = g(n) + cost()$ value. This will be the current voxel of the loop iteration.

```
while (!frontier.empty()) {
    auto current = frontier.get();    //get the voxel which is currently the "best"

    if (current == exits) {    //check if we have arrived at the goal already
        break;
    }
```

When it has been ascertained that the current voxel is not the goal, the algorithm can continue and explore the neighbouring voxels of the current voxel. Due to the nature of the neighbour access differing vastly between the two data structures, the two different methods can be seen side by side below. With the grid having easier neighbour access, lots of if-statements are needed to see if the neighbour is truly a navigable voxel, whereas with the `SparseMortonGrid`, it is simply the case of checking if the voxel is in the tree, and if it hasn't yet been visited by the algorithm.

```
//get neighbours of current voxel (grid)
for (auto next: current.getNeighbours18()){
    if ((next.x && next.y && next.z >= 0)
    && next.x <= voxels.max_x - 1
    && next.y <= voxels.max_y - 1
    && next.z <= voxels.max_z - 1) {
        if (voxels(next.x, next.y, next.z)
             != 0
            && voxels(next.x, next.y, next.
                 z) != voxelID
            && voxels(next.x, next.y, next.
                 z) != 1
            && voxels(next.x, next.y, next.
                 z) != 3) {
```

```
//get neighbours of current voxel (morton)
    for (const auto &conn:
         getNeighbours18_morton(current->x,
          current->y, current->z, current->
         index)) {
        if(!octree.mQTree[level].contains(
             conn)) continue;
        auto next = octree.mQTree[0][conn];
        if(next->attribute == voxelID)
             continue;
```

Calculating the new, tentative, cost to the neighbouring voxel is done by checking the `cost_so_far` map and looking up the cost it takes to get to the current voxel, and then using the diagonal distance to calculate the cost of going to the neighbouring voxel. If this is cost is not yet in the `cost_so_far` map or smaller than the cost of going from current to the neighbour, the cost is updated or entered into `cost_so_far` and the tentative cost becomes the *cost* of this voxel. Then, the $h(n)$ value for the voxel can be calculated with the Manhattan distance heuristic.

```
//calculate cost from current node to the neighbour
int new_cost = cost_so_far[current] + distDiagonal(current, next);

if (cost_so_far.find(next) == cost_so_far.end()
    || new_cost < cost_so_far[next]) {
    cost_so_far[next] = new_cost;
    int priority = new_cost + heuristic(next, exits);
```

To ensure that this neighbour is not visited again, the neighbour is set to the `VoxelID` value, except when it is the goal node. Before doing this however, because `voxels(...)` is accessed, which is the name of the `VoxelGrid` data object in the this function, and is thus a resource accessed by up to 32 threads at a time. The mutex guarding this object needs to lock it for this thread exclusively. After the neighbouring voxel has been set to the `VoxelID` value, the neighbour can be inserted in the priority queue, and the parent-child in the path is recorded in the `came_from` map. Once all the neighbours of the current voxel are visited, the main loop ends, and a new current voxel is taken from `PriorityQueue`, and this is repeated until either the entire dataset is traversed and the goal is not found, or the goal is found, and the loop is exited via a break statement.

```
//lock mutex on the VoxelGrid
mutex.lock();
if (next != exits) {
    voxels(next.x, next.y, next.z) =
         voxelID;
}
mutex.unlock();
//insert the neighbour in all the
     containers
frontier.put(next, priority);
came_from[next] = current;
```

```
//lock mutex on Morton
mutex.lock();
if(next != exits) {
    neighbours->attribute = voxelID;
}
mutex.unlock();
//insert the neighbour in all the
     containers
frontier.put(next, priority);
came_from[next] = current;
visited.push_back(next);
```

As is visible in the code snippet above, the `SparseMortonGrid` version of A* maintains an extra data container, `visited`, which is just a `std::vector<OctreeNode*>` that holds pointers to all the nodes that have been visited by this thread of the A* algorithm. This container is used solely for cleanup purposes.

After the main search loop has broken, either when the entire dataset has been exhausted or by reaching the goal voxel, the path has to be reconstructed by traversing the voxels backwards from the goal towards the starting voxel. This is done in both the `SparseMortonGrid` and the `VoxelGrid` in very similar functions.

```
std::vector<Voxel> path;
auto current = goal;
while (current != start){
    if(current != Voxel{0,0,0}){
        path.push_back(current);
        current = came_from[current];
    } else {
        std::cout<<"No path found!\n";
        break;
    }
}
path.push_back(start);
std::reverse(path.begin(), path.end());
return path;
```

```
std::vector<OctreeNode*> path{};
auto current = goal;
while (current != start){
    if(current != nullptr){
        path.push_back(current);
        current = came_from[current];
    } else {
        std::cout<<"No path found!\n";
        break;
    }
}
path.push_back(start);
std::reverse(path.begin(), path.end());
return path;
```

These functions find their way back from the goal voxel to the starting voxel trough the `came_from` map, and add these voxels to a path vector. Once the path is done, the path is reversed, but this is actually not even strictly necessary when not dealing with an incremental search algorithm (which this implementation of A* is not).

When the path has been reconstructed, the pathfinding is done, the stats of this run are recorded to a `.csv` file linked to this thread, and the voxels that make up the path are sent to the `_instanceData` rendering buffer, where the main rendering thread will receive them and render them accordingly.

When the data changes however, regular A* has no tricks up its sleeve and has to recalculate the entire path again. And since the `VoxelGrid` data structure has no idea about anything other than the integer values as certain coordinates, when running the simulation for this data structure, the algorithms will simply run on an interval of n seconds, which is the `speed` parameter of the A* function. The `SparseMortonGrid` data structure is a much more intelligent data structure that does *remember* things, and stores things other than a simple attribute value. Consequently, when running the simulation on the `SparseMortonGrid`, a convenience function `pathChanged` is used to determine whether A* needs to recalculate the path or not. This is done by utilising the `isChanged` Boolean member of the `OctreeNode` class, which is only set to true if it is affected by a fire. Thus, the `pathChanged` function simply scans the previously calculated path to see if any of them have changed in the meantime, which is significantly cheaper than recalculating the path.

```
bool change = false;
for(auto &e: path){
    if(e->isChanged){
        change = true;
        return change;
    }
}
return change;
```

If the path does need to be recalculated, in both the case of the `SparseMortonGrid` and the `VoxelGrid` the utility containers `came_from`, `cost_so_far` and `PriorityQueue` are emptied. Before doing that, the `came_from` is used to reset all the voxel to navigable. Then when a new path has been found by A*, then using the `path` vector, the instance buffer needs to be cleared of the previous path. This is done by iterating over the instance buffer and removing elements that match the ID value of the thread. Lastly, the path vector itself can be cleared and filled with a new path. With this process repeating until the simulation comes to an end.

4.7 IMPLEMENTING THETA*

When adapting Theta*, two different approaches are used for the two data structures, and both will be discussed in this section. The main differences between the approaches will be highlighted, but for the most part, the approaches are of course very similar. Theta* is a much more complicated algorithm, that for its main loop in essence, though rather similar to A*, uses more auxiliary functions.

Theta* on the `VoxelGrid` uses a child class of the `Voxel` class used for A*, and described in Listing 4.2. This `Node` class has a few extra members that are required for performing Theta* ($g(n)$, and the parent and child pointers), as well as members that are required for Phi* pathfinding, namely $rhs(n)$ and k, however, this is not implemented at this stage.

```
struct Node: public Voxel{
    Node *parent, *child;
    double  g, rhs;
    std::pair<float, float> k;
    using Voxel::Voxel;

    ..

    std::vector<Node*> getChildren(){
        std::vector<Node*> kids;
        for(auto n: this->getNeighbours26()){
            auto tmp = new Node();
            tmp->x = n.x;tmp->y = n.y;tmp->z = n.z;
            kids.emplace_back(tmp);
        }
        return kids;
    }
```

Listing 4.9: The Node class

When looking at the containers used for Theta*, these are essentially the same as in A*, with `PriorityQueue` and `came_from` being used. `cost_so_far` however, is not used, as for Theta*, the $g(n)$ values are stored with the `Node` class. The initialisation of Theta* is in essence also the same as in A*, with the exit being chosen that is the closest to the starting node, and the starting voxel inserted into `came_from` with itself as its parent. A marked difference is that newly created instances of `Node` are passed to the `initialiseVertex()` function before further use.

```
void initialiseVertex(Node *node) {
    node->g = INFINITY;
    node->parent = nullptr;
}
```

```
bool initialiseVertex(OctreeNode *node){
    if(node->attribute < 11) {
        node->g = INFINITY;
        node->PhiParent = nullptr;
        return true;
    } else {
        return false;
    }
}
```

We can see the different approaches, with new `Node` instances being created on the fly, and the `OctreeNode` objects of course already being inside the `SparseMortonGrid`, therefore the need exists to check if they are not already used by another path thread (attribute integers from 11 and up signify path use).

When all this is done, Theta* employs roughly the same main loop as A*, a while loop that is true while the priority of the best element of `PriorityQueue` is smaller than the $g(n)$ value of the goal voxel (which is initialised to be infinity). The loop gets the best item from the queue, and starts visiting the neighbouring voxels of the current voxels in the same manner as in A*, and then things start to diverge.

```
initializeVertex(next);
came_from[neighbour] = current;
updateVertex(current, neighbour, goal,
     voxels, voxelsize, frontier, voxelID);
mutex.lock();
voxels(neighbour->x, neighbour->y,
     neighbour->z) = voxelID;
mutex.unlock();
```

```
if(initializeVertex(neighbours)) { ;
    came_from[neighbours] = current;
    updateVertex(current, neighbours, goal,
          octree, voxelsize, frontier,
         voxelID);
    visited.push_back(neighbours);

    mutex.lock();
    neighbours->attribute = voxelID;
    mutex.unlock();

    if (neighbours->index == goal_morton) {
        goal->g = current->g;
        goal->PhiParent = current;
        came_from[goal] = current;
        break;
    }
} else {
    continue;
}
```

As is visible above, we can see that the actually relevant pathfinding operations are done in the function `updateVertex`, and that in the case of the `SparseMortonGrid`, if the voxel is not free, and can not be initialised, it cannot be visited. This has consequences for multi-actor pathfinding, because this means that one voxel can only be part of one path at the same time, but this can also sometimes lead to the algorithm not finding a path while there should be one, therefore, when performing pathfinding on the `SparseMortonGrid`, the Theta* algorithm uses an extra looping condition in its outer loop: if a previous path was unsuccessful, try again. This is of course only possible when you *know* that there is a path between start and goal, which should always be the case in the dataset.
Looking at the `updateVertex()` function, it is a straightforward implementation of the function as defined in Algorithm A.1. The main purpose of this function is to compute the cost of visiting a voxel, which is implemented in the `computeCost` function, and if this new cost is better than the old cost of the voxel, it will reinsert the voxel into the queue.

```
float g_old = next->g;
computeCost(current, next, voxels,
     voxelsize, voxelID);
if(next->g < g_old && voxels(next->x, next
     ->y, next->z) != voxelID){
    queue.put(next, next->g + dist(next,
         goal));
}
```

```
auto g_old = next->g;
computeCost(current, next, octree,
     voxelsize, voxelID);
if(next->g < g_old && next->attribute !=
     voxelID){
    queue.put(next, next->g + dist(next,
         goal));
}
```

The main difference between Theta* to A* happens in the `computeCost` function. This is where the algorithm decides whether the voxel is within the line of sight, and the voxels in between can be skipped. In this function, the ray from the current voxels' parent to the neighbouring voxel of the current voxel is traced using the procedure as described in Algorithm 3.5, and if this line does not contain any obstacles, the parent of the neighbouring node will be set to the parent of the current node. However, as this is repeated, more and more "corners" can be cut because the parent voxel is selected, meaning the algorithms keeps moving backward along the parents and checking the LoS. In the code snippet below we can see that these functions are almost identical between the two data structures.

```
std::vector<Node*> line = raytrace3d(
     current->parent, next, voxels, voxelID
     );
if(lineOfSight(line, voxels, voxelID)){
    line.erase(line.begin(), line.end());
    float new_g = current->parent->g + dist
         (current->parent, next);

    /* Path 2 */
    if(new_g < next->g){
        next->parent = current->parent;
        next->g = new_g;
    }
}else{
    /* Path 1 */
    float new_g = current->g + dist(current
         , next);

    if(new_g < next->g){
        next->parent = current;
        next->g = new_g;
    }
}
```

```
std::vector<OctreeNode*> line = raytrace3d(
     current->PhiParent, next, octree,
     voxelID);
if(lineOfSight(line, octree, voxelID)){
    line.erase(line.begin(), line.end());
    float new_g = current->PhiParent->g +
         dist(current->PhiParent, next);

    /* Path 2 */
    if(new_g < next->g){
        next->PhiParent = current->
             PhiParent;
        next->g = new_g;
    }
}else{
    /* Path 1 */
    float new_g = current->g + dist(current
         , next);

    if(new_g < next->g){
        next->PhiParent = current;
        next->g = new_g;
    }
}
```

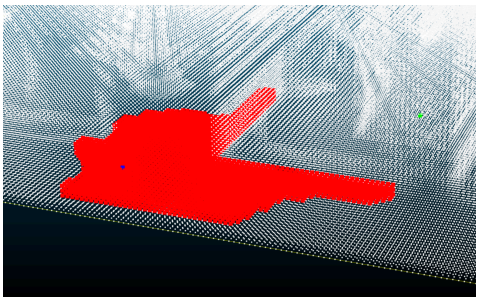
Theta*'s main loop also ends when either the entire dataset has been exhausted, or the goal node has been reached. Like in A*, the path needs to be traced back from goal to start. While this could be done with either `came_from` or the parent pointers, the latter often produces unexpected behaviour, and in the implementation, `came_from` is more solid. Therefore, Theta*'s function for reconstructing the path is identical to A*.

After the path is found and reconstructed, it is sent to the instance buffer in the same manner as in A*, and cleanup is also done in the same manner as in A*.

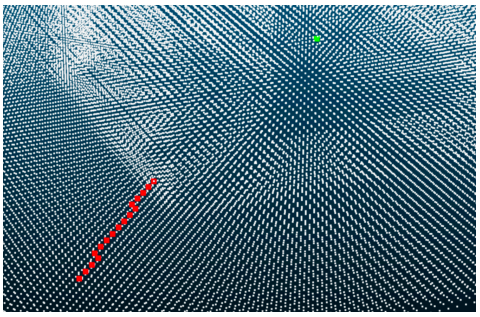
4.8 IMPLEMENTING D*-LITE

Multiple attempts have been made trying to implement D*-Lite, with both a self written approach, using the `Node` and `OctreeNode` classes (because they use the same values) as well as using an open source implementation of D*-Lite. Both were moderately successful in initial tests: the implementations perform well when there are no obstacles that require sharp corners, but when they do, the algorithm gets stuck and breaks down.

However, there is another issue with D*-Lite. D*-Lite has been conceived as an incremental, actor-centred, path planning algorithm, for which each actor maintains their version of the search space, because it requires accurate and up-to-date information about every node in the search space to replan paths. This information is made concrete in the $g(n)$ and $rhs(n)$ values. But since these values depend on the start *and* goal node, every actor needs a *unique* version of the dataset to find their way. What this means on the implementation side, is that `SparseMortonGrid` or `VoxelGrid` would need to be copied to each thread, and then somehow the fire simulation algorithms would need to distribute the simulation information to each thread individually, which is not impossible but rather impractical. The bigger issue is that every thread would require the entire dataset of memory, increasing the memory footprint of the simulation by the size of the `SparseMortonGrid` or `VoxelGrid` for every added actor. For a simulation running paths for the full dataset, meaning a path from every room this would mean an extra $512MB$ when using the `VoxelGrid`, and a whopping $1.4GB$ extra when running the full simulation on the `SparseMortonGrid`.



(a) D*-Lite attempting to find a path from the blue voxel to the green voxel, with explored voxels in red



(b) The path produced by the adapted implementation of D*-Lite

The implementation that was built from scratch employs much of the same structure as the implementation described for A* an Theta*, and is a faithful implementation of the algorithm described in Algorithm 2.4. The implementation has a main function, a `computeShortestPath()` function which comprises of the same main loop as A*, a while loop based on a conditional while loop, and an `updateVertex()` function that will check the $g(n)$ and $rhs(n)$ values for a voxel and its successors and predecessors (i.e. neighbours). However, `PriorityQueue` needs a different priority value, because D*-Lite needs two dimensional keys to function. Thus `PriorityQueue` is now defined as as `PriorityQueue<Node*, std::pair<double, double>>`. The `PriorityQueue` is sorted lexicographical order using its two-dimensional key. When the algorithm was put together, initial experiments found that it had no problem finding paths that were straight or slightly curved, but, as stated before, when sharp corners needed to be made, then the algorithm would run into problems.

In Figure 4.6a we can see clearly that with this implementation, D*-Lite is unable to move past the x and y values of the goal point, and that it gets stuck. This experiment was repeated with all the starting points and all the exits, and similar behaviour was observed. This led to a search for an existing implementation, to eliminate any suspicion of implementation error.

```
while(cur != s_goal) {
        ...
        double cmin = INFINITY;
        double tmin;
        Voxel smin;

        for (i=neighbours.begin(); i!=neighbours.end(); i++) {
            double val  = cost(cur,*i);
            double val2 = trueDist(*i,s_goal) + trueDist(s_start,*i);
            val += getG(*i);

            if (close(val,cmin)) {
                if (tmin > val2) {
                    tmin = val2; cmin = val;smin = *i;
                }
            } else if (val < cmin) {
                tmin = val2; cmin = val; smin = *i;
            }
        }
        n.clear();
        cur = smin;
    }
```

This led to using existing 2-dimensional implementation of D*-Lite by Neufeld [2015], which was subsequently retooled for 3 dimensions, but which alas also generated many problems when dealing with obstacles. In this implementation however, the trouble is in the replanning function, where the path is constructed by using a tie-breaking function that is supposed to prevent the algorithm from simply choosing

a path at 45° from goal to start. However, unfortunately, this is exactly what was observed. In the code above we can see that the value with the lowest $g(n)$ value is selected, and ties are broken using the `val2` parameter. The result is however that the algorithm prefers paths that are close to the the *ideal* line without obstacles instead of preventing this, as can be seen in Figure 4.6b, which is almost a 45° path straight into a wall, not ideal.

4.9 SMARTER PATHS: TIME-AWARE A*

The implementation above describes some useful successes and challenges when implementing the methodology, but as also stated in the methodology, improvements can be made to make the paths better and more useful for answering the research question. By using a version of A* that is time-aware, while simpler than LPA* or D*-Lite, but more complex than A* itself, one could achieve a version of pathfinding that accounts for the other calculated paths as well as the emergency situation in the building. This version of A* is however only possible with the `SparseMortonGrid` data structure, because the information needs stable and reliable information about the voxels across the threads.

The starting point of this implementation is same as for A* that was created for the `SparseMortonGrid` and described in Section 4.6. The first thing to change is the data structure for the path itself, which is changed from a `std::vector<OctreeNode*>` to a `std::deque<OctreeNode*>`. A `deque` is a double ended queue that allows for the data insertion and removal at both the front and the back of the queue. This way, we can reliably access the first voxel in the path, and pop it from the queue without issues, whereas these positions are not guaranteed with a `std::vector<T>`.

```
if(pathChanged(path, mutex) || first || !successfulPath(path)) {
    main A* loop...

    //free all nodes in search space for other paths
    for(const auto &nodes: visited){
        mutex.lock();
        if(nodes->attribute == voxelID) nodes->attribute=2;
        mutex.unlock();
    }
    //reconstruct the path
    path = reconstruct_path_SVO(start, exits, came_from, voxelID);

    //check if the front of the path is in use by another, and then wait the time it takes
          to walk 1 voxel
    if(path.front()->isOccupied) std::this_thread::sleep_for(std::chrono::milliseconds
         (110));

    //set the front of the path to occupied
    path.front()->isOccupied = true;

    //draw the currently occupied voxels
    push_to_buffer(path.front(), pathcol, voxelID, mutex, vSize, _instanceData);
}
```

Listing 4.10: The pathfinding parts of the time-aware A* algorithm

Another thing is that the main A* loop runs while the simulation is running, and only recomputes its paths when the paths have changed, but otherwise does not do anything. This incremental A* must move the starting point continuously, but must do so without recomputing the path, if this is not needed. In Listing 4.10 and Listing 4.11, we can see the essential parts of the code, and see that outside of the main A* loop, it is significantly more complex than regular A*, because it needs to clearly manage its location per at all times and avoid collisions with both other paths and the fire. To handle this, an extra Boolean has been added to the `OctreeNode` class: `isOccupied`,

that is separate from whether the voxel in question is affected by the fire. This class member is only concerned with the occupancy of the paths in the voxel, and of course ensures that it can only be occupied by one path at a time.

```
}else{
    //check if the exit has been (almost) reached
    if(path.empty()) break;

    //if so, release entire path and notify simulator
    if(near(path.front(), exits)){
        mutex.lock();
        start->attribute = 2;
        start->isOccupied = false;
        for(auto all: path){
            all->attribute = 2;
            all->isOccupied = false;
        }
        mutex.unlock();
        std::cout<<"Thread "<<id<<" has exited\n";
        break;
    }

    /*if not yet at the exit, continue walking the path, moving up 1 voxel per 110 ms and
         clearing the voxels behind the current voxel, and drawing them. If there is a
         conflict, wait and repeat.*/
    mutex.lock();
    start->attribute = 2;
    start->isOccupied = false;
    if(!path.front()->isOccupied && !path.front()->isChanged) {
        start = path.front();
        start->isOccupied = true;
        path.pop_front();
        mutex.unlock();

        push_to_buffer(start, pathcol, voxelID, mutex, vSize, _instanceData);


        std::this_thread::sleep_for(std::chrono::milliseconds(110));
    }else {
        std::cout<<"conflict at "<<path.front()->index<<"\n";
        mutex.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(110));
    }
}
```

Listing 4.11: The moving parts of the time-aware A* algorithm

We can first see that this incremental version uses the same `successfulPath` function as the Theta* implementation because this version is more likely to *not* find a path on the first try as it is not allowed to use any voxels that are in occupied by other paths, but it *is* allowed to compute a path that contains voxels shared by multiple paths, as long as it doesn't occupy them at the same time. Inside the main pathfinding loop this is again handled by, checking if a voxel `attribute = voxelID`. But to ensure that only the occupied part of path is set to the correct attribute, the main A* loop must do more bookkeeping, but clearing all the voxels it has used for pathfinding again to be walkable, with the exception of the voxels that now constitute the occupied portion path, which have to be set to occupied. In the real-time visualisation, care has to be taken to allow the temporal nature of this algorithm to be shown. This is done by showing the current voxel with a short tail of a few voxels moving around the dataset. This makes it easy to visually debug and check if the paths are not colliding.

5 RESULTS

In this chapter, the results of the experiments which were outlined in Chapter 3 and implemented according to Chapter 4, will be discussed. To start, a general overview of the performance of the different algorithms will be presented, after which every algorithm/data structure combination will be analysed in more detail.

5.1 OVERVIEW

Algorithm	Success Rate [%][4]	Average memory footprint when running on max [MB]
Idle[5]	-	464.8
A* on Grid	100.00	936.70
A* on Morton	100.00	860.87
Theta* on Grid	16.00	5496.42
Theta* on Morton	50.00	528.00

Table 5.1: The performance of the algorithms.

In Table 5.1, we can see that A* has the highest success rate, with the algorithm always being able to find a safe path before the simulation has ended. Theta* on the regular grid has the lowest success rate, and Theta* on the Morton grid has a 50% success rate. This means that Theta*, on a regular grid and on the Morton grid, is not able find a path before the end of the simulation in 84% and 50% of the runs respectively. A possible explanation for this could be the increased complexity of the algorithm in general compared to A*. Furthermore, the regular grid consumes more memory than the Morton grid, and this could explain why the Theta* on the regular grid has such a low success rate. When the simulator application is idle, we see that it consumes about 460MB of memory, which is quite something. Concerning the algorithms, we see that Theta* on the regular grid has the highest memory footprint, with almost 50GB, and Theta* on the Morton grid has the lowest footprint. A* is more memory efficient when implemented on the Morton grid, with only about 300MB required to run a simulation with 30 evacuees.

Algorithm	Mean calc time [s]	Max calc time [s]	Mean no. of nodes visited	Path length (45 38 194)-(149 5 210)
A* on Grid	3.54	59.28	28330.42	159.61
A* on Morton	5.56	20.02	125560.29	160.00
Theta* on Grid	0.70	16.13	22120.14	178.00
Theta* on Morton	1.92	16.50	26465.43	178.00

Table 5.2: The time statistics of the algorithms and the mean number of nodes visited per algorithm.

4 Success rate refers to algorithms finding paths before the simulation comes to an end.
5 Idle refers to the simulation application running without any algorithms running.

In Table 5.2, we can see that Theta* on the regular grid has the best time performance, and A* on the Morton grid has the worst. This is not a complete representation of the performance of the algorithms, because Theta* on the regular grid fails 86% of its runs, and this is not taken into account for the time calculations. While Theta* on the Morton grid also has good time performance, taking the success rate into account, a similar effect can be seen here, even though the success rate for Theta* on the Morton grid is much higher than on the regular grid. When looking at A*, the regular grid based implementation has the better mean value, but its extremes are higher and lower than the Morton grid counterpart. When looking at the number of nodes visited, we can see that Theta* on the regular grid has the best performance, and A* on the Morton grid has the worst performance. These results for Theta* are however skewed by the low success rate, and are mostly ground floor paths. The 5x increase for A* on the Morton grid is strange, as this is the same algorithm as the regular grid implementation. Looking at the path lengths of a sample path, we see that both algorithms have the same path length regardless of data structure. Theta* should theoretically have shorter paths, but it does not.

5.2 A* ON A REGULAR GRID

For A* on the regular voxel grid, the results are visible in Figure 5.1. In the top right we can see that the computation time rises linearly with increased path lengths. Even though most of the path computation times are still under 10 seconds, the very long computation times $> 30[s]$ only occur when running the simulations with a very high number of starting points, with 30 and 20 starting points being the main contributors of these long computation times. We can see that for comparable paths lengths when running the simulation on only 3 starting points, very good computation times can be achieved, as is evidenced by the purple and dark blue regression lines in Figure 5.1a, which represent 3 and 6 starting points respectively.

In Figure 5.1b, on first glance, we can see a similar pattern to Figure 5.1a, where the number of starting points really affects computation times. We can, however, also see a stable distribution pattern along the number of nodes visited, which indicates that for repeat runs (of which this instance of A* has many), A* is stable and visits the same number of nodes. Also visible is the same phenomenon as in Figure 5.1a, that paths that visit upwards of 100.000 nodes have very long computation times when the simulation runs many different threads, but when only running 3 threads this computation time is comparable with much shorter paths.

Moving on to comparing the number of nodes visited to the path length, the first thing that is very obvious is the clear gulf between approximately 60.000 nodes and 120.000 as nodes visible in Figure 5.1c. This could be all the paths that start from the second floor, which have to check many more nodes, due to their sheer distances from the exits.

In Figure 5.1d, we see the distribution of computation times for the different simulations. A similar pattern emerges where high loads in the simulation lead to high computation times. The distribution of computation times is spread out much more in the high load simulations, with the low load simulations having very stable and reliable computation times, and the high load simulations having both faster and slower computation times. This is probably the case, because due to the nature of the methodology, the high load simulation contain all the paths (both the shortest and the longest) whereas this is not the case for the low load simulations, which could only randomly contain long paths.

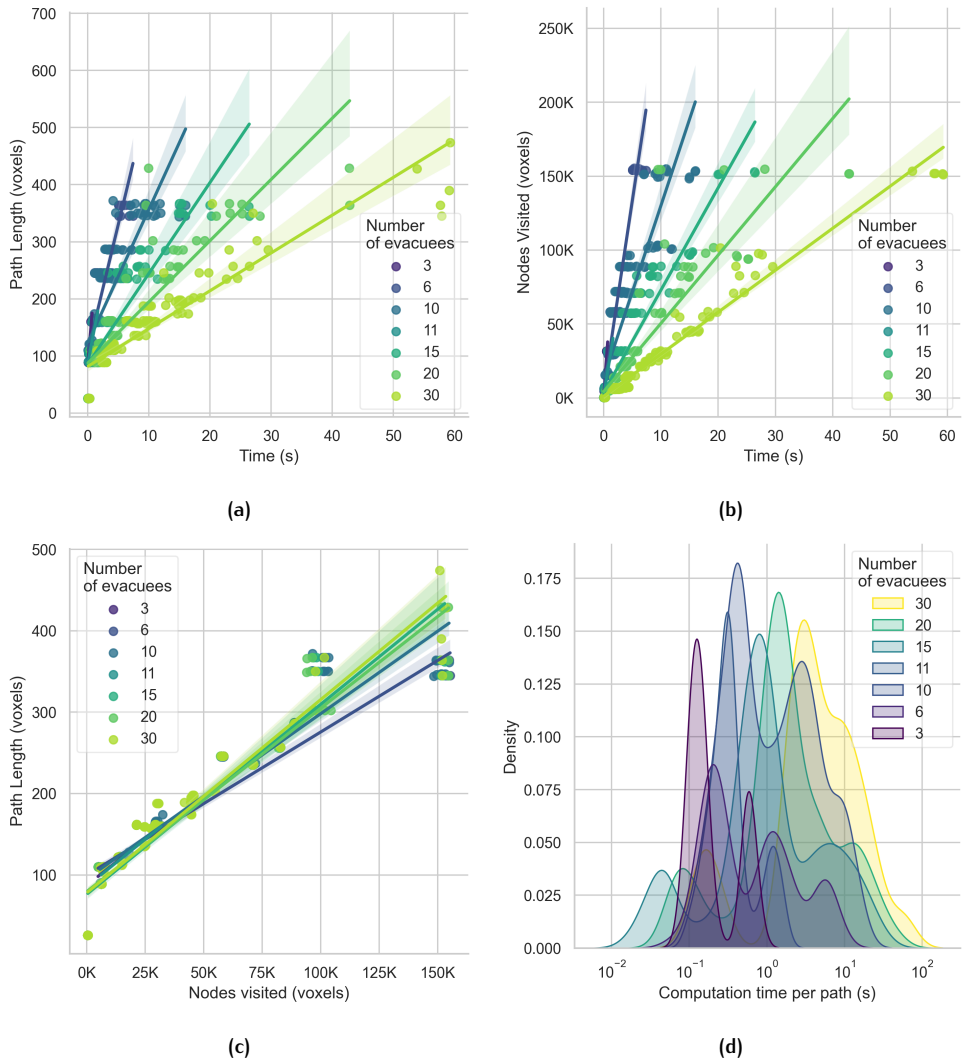


Figure 5.1: Performance results for A* on a regular grid. (a) Time and path length. (b) Time and nodes visited. (c) Path length and nodes visited. (d) Density of time per path on a logarithmic scale.

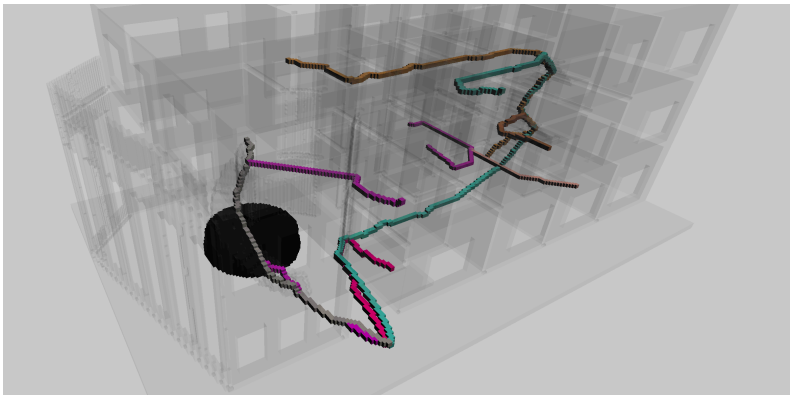


Figure 5.2: Paths generated by A* on a regular grid. Some of the paths, such as purple, are using a route which will soon become blocked.

If we look at what the paths look like, we can see that the paths generated by this method are correct, and seem to be similar to the shortest paths described in Chapter 3 and Chapter 2. We can see that two of the paths in Figure 5.2 are using the semi-

circular staircase, under which the fire in the simulation is situated. This means that the paths calculated by these thread will be different as soon as the voxels containing the path are affected. This changed path route is visible in Figure 5.3, where the purple path now goes through the other staircase, before going to their original exit. Lastly, we can see that voxels are shared between paths.

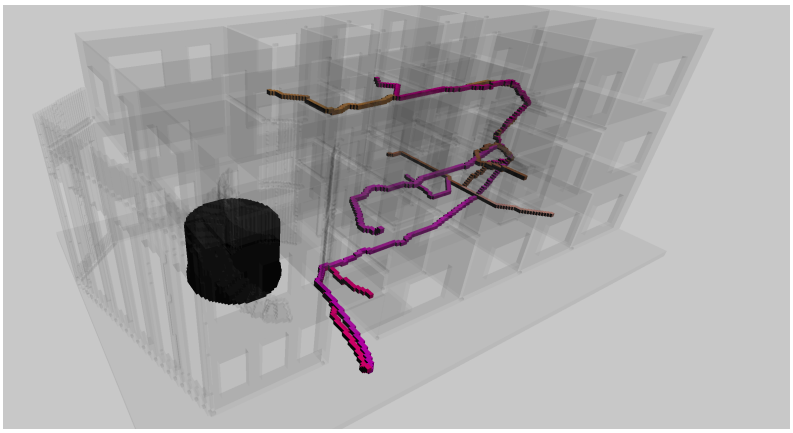


Figure 5.3: Paths generated by A* on a regular grid. The purple path that used to use the semi-circular stairs is now going by another route.

5.3 A* ON AN MORTON GRID

In Figure 5.4, we can see the detailed performance results for A* on the Morton grid. In Figure 5.4a, we see that the general tendency that was present in Figure 5.1a is also present here: simulations with a lower load perform better than high load simulations and longer paths lead to longer computation time, a fact which is exacerbated by the aforementioned simulation load. If we look carefully we can also see that this does not hold true for the simulation with the lowest load (3 starting points), where we can see that the longest path is actually computed in less time than the shorter path. Lastly, if we look at the graph as a whole in comparison to Figure 5.1, we can see that the variance in computation time is lower for the Morton grid than for the regular grid, with the longest computation time being about half as long as the longest computation time on the regular grid.

In Figure 5.4b we see the same pattern to the regular grid based A* algorithm, as expected, with time increasing when more nodes need to be visited, and with the simulation load exacerbating this as well. In Figure 5.4c, we can see a less uniform correlation between path length and nodes visited than in Figure 5.1c, with the lowest load simulation again going against the grain. In general it also seems that longer paths do not necessary lead to much higher numbers of nodes visited for A* on the Morton grid. However, from looking at Table 5.2, we know that A* on the Morton grid visits many more nodes that the regular grid. Thus, if we compare Figure 5.4b to Figure 5.1b, the maximum values for the Morton grid are about 50.000 nodes higher, which is as of yet unexplained.

Lastly, we see that in Figure 5.4d, as expected, a similar pattern emerges as in A* on the regular grid, with the highest peak with the high load simulation being around $10^1[s]$. This is similar to A* on the regular grid, but the bulk of the computation times is well around and below $10^0[s]$, which is better than the regular grid. Overall, looking at the time performance of A* on the Morton grid we can say that it is generally faster than A* on a regular grid, especially when dealing with larger distances. However, its mean is slightly higher than the regular grid.

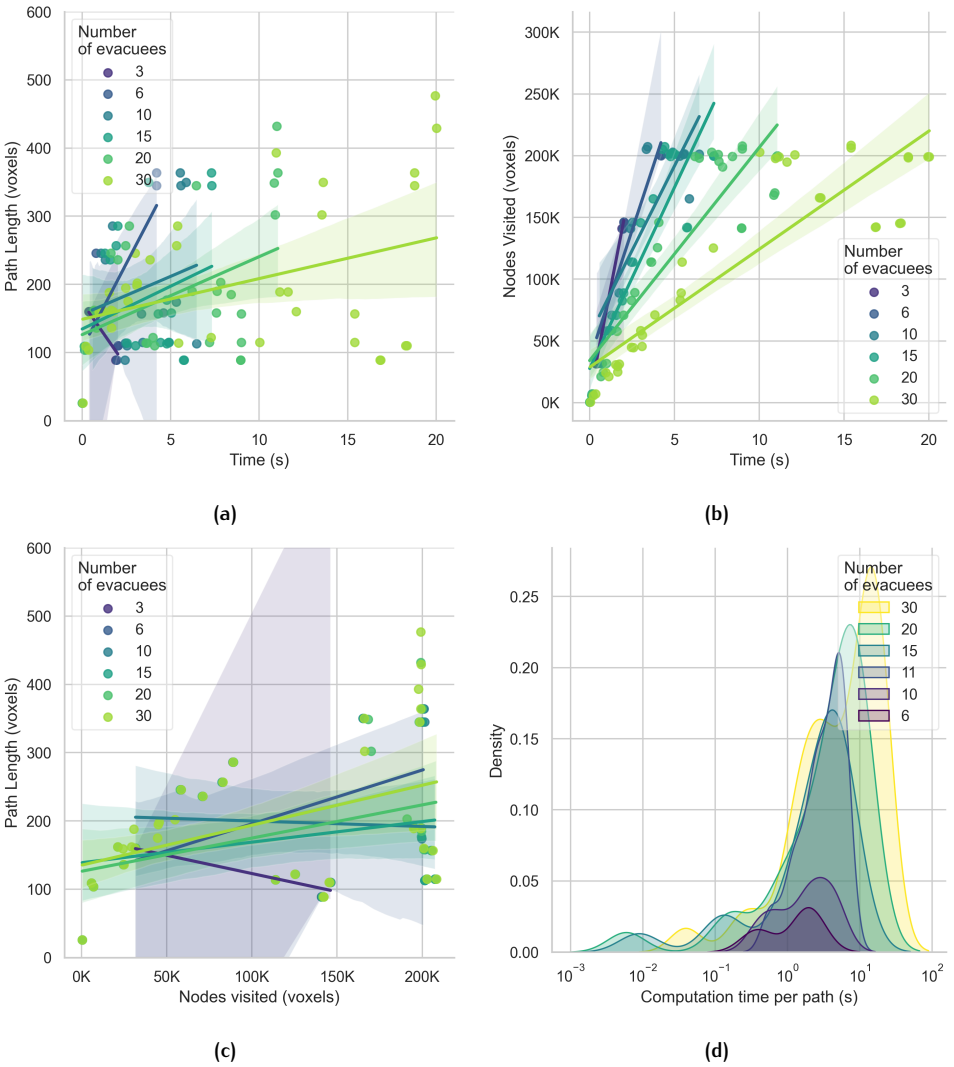


Figure 5.4: Performance results for A* on a Morton grid. (a) Time and path length. (b) Time and nodes visited. (c) Path length and nodes visited. (d) Density of time per path on a logarithmic scale.

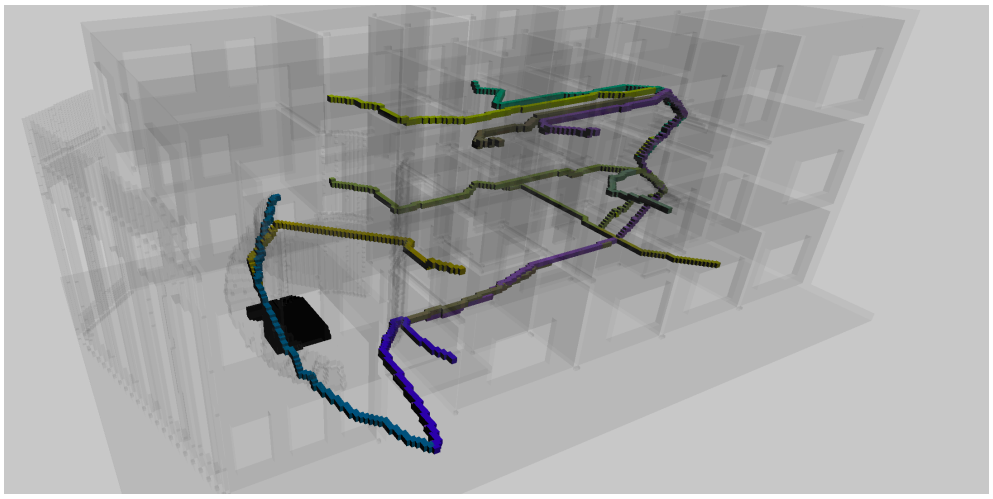


Figure 5.5: The yellow path starting on the first floor in the centre of the image is using the semi-circular stairs.

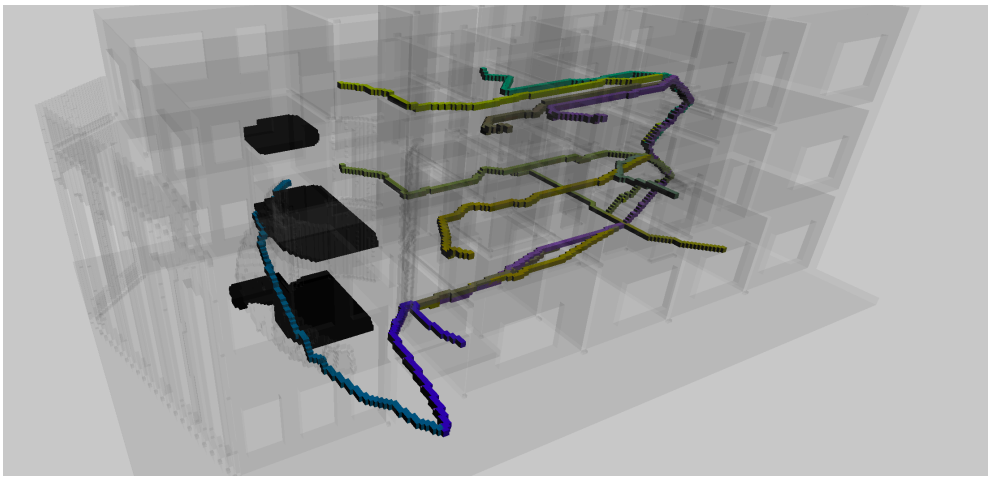


Figure 5.6: The yellow path is not using the other staircase.

When we look at what the paths look like, we can see that the paths follow the same behaviour as A* on the regular grid, which is to be expected. We can also see that for the yellow path in Figure 5.5 is diverted in Figure 5.6.

5.4 THETA* ON A REGULAR GRID

Because Theta* on the regular grid has such an abysmal success rate (only ±16%), there are not a lot of data points to assess its performance. The low success rate of this implementation is probably twofold: the grid generates neighbours on the fly, and Theta* can generate many neighbours while doing its LoS checks. All these extra neighbours come at a computational cost, which slows down the entire simulation, and makes it so that not many paths are found before the simulation is finished. In Figure 5.7, the general tendency is similar to what was seen in A*. What can be noted is that Theta* does not have the highest number of nodes visited (that is A* on the Morton grid), which is may be due to only successful runs being included in these statistics. It is unfortunately not possible to include the failed runs. Theta* visiting less nodes than A* is surprising when looking back at Table 5.1, where the memory footprint of Theta* on a regular grid can be observed to be about 50 gigabytes. However, Theta* has found longer paths in less time than it takes A* to sometimes find paths of equal or shorter length.

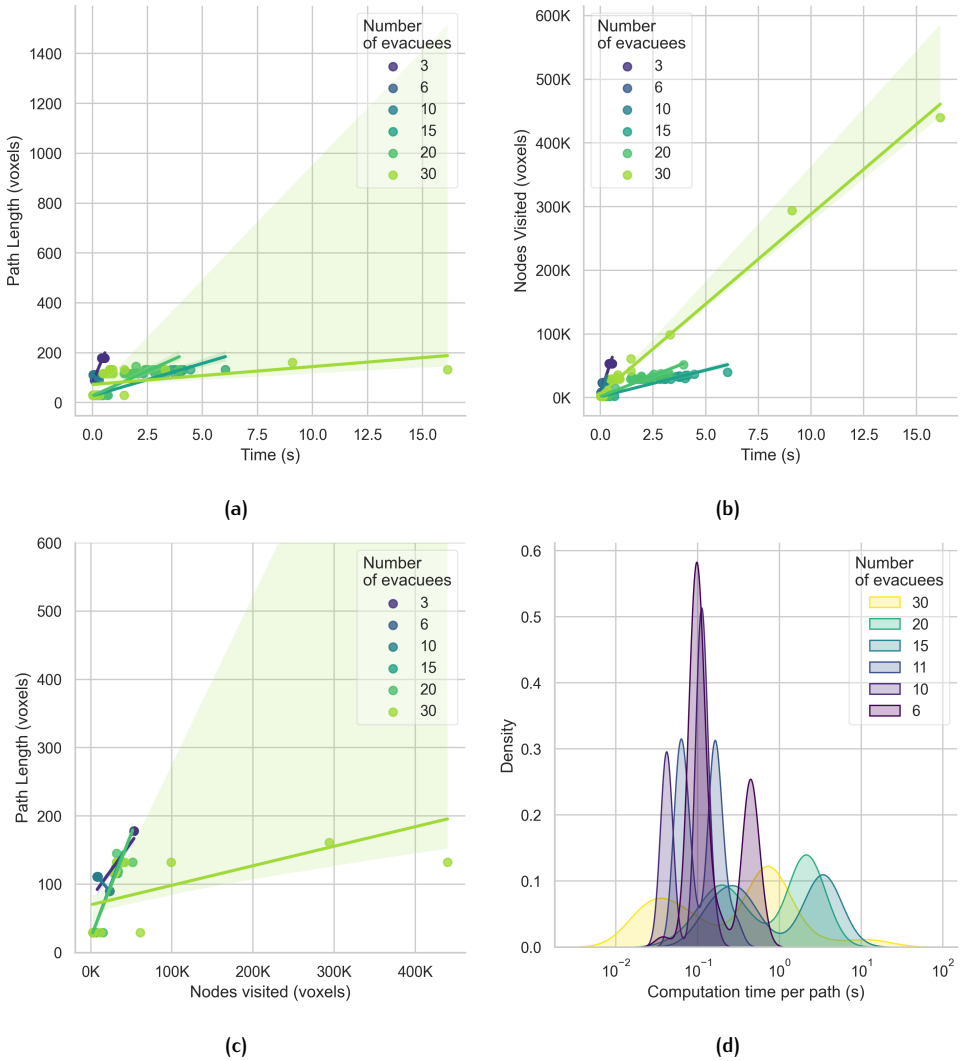


Figure 5.7: Performance results for Theta* on a regular grid. (a) Time and path length. (b) Time and nodes visited. (c) Path length and nodes visited. (d) Density of time per path on a logarithmic scale.

In Figure 5.8 and Figure 5.9, we can see that the path zigzags near door openings, when it seems to think that it can reach its destination sooner like that, and then comes

to its senses and tries the way it was going again. The any-angle nature of Theta* can not be seen, because the algorithm does seem to prefer to travel at 45°. This behaviour is reminiscent of the behaviour seen by using the Manhattan heuristic on A* in the short testing run seen in Figure 4.5, but it is much more extreme.

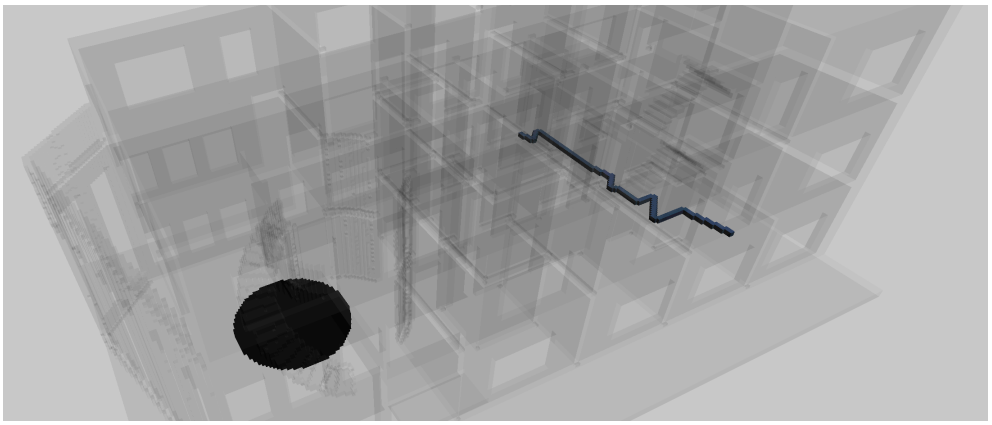


Figure 5.8: Paths generated by Theta* on a regular grid.

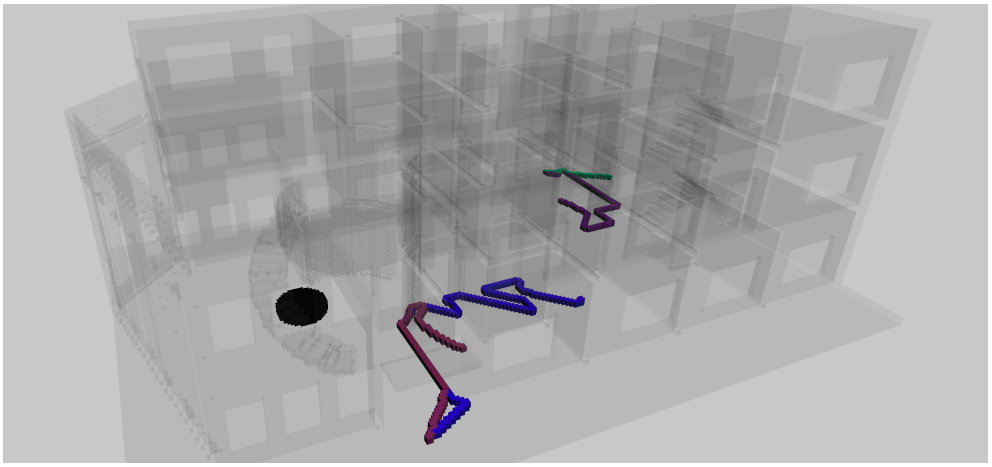


Figure 5.9: Paths generated by Theta* on a regular grid. Exaggerated zigzagging can be observed.

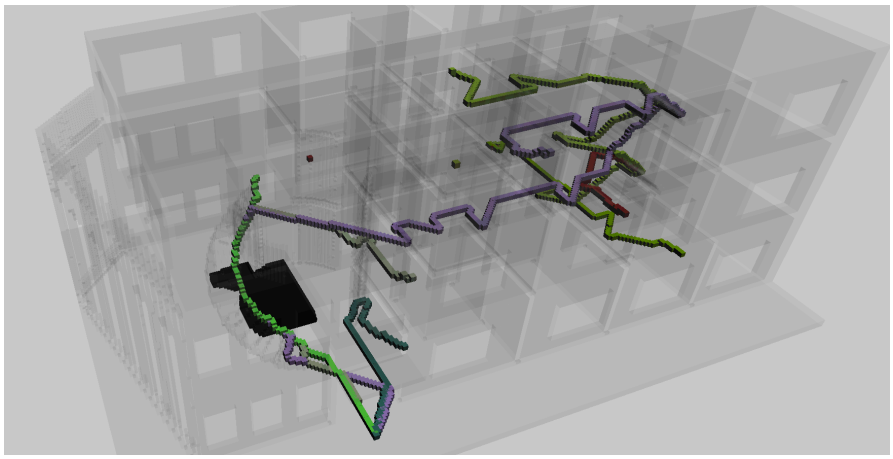


Figure 5.10: Paths generated by Theta* on a Morton grid, the lilac path is travelling in the danger zone.

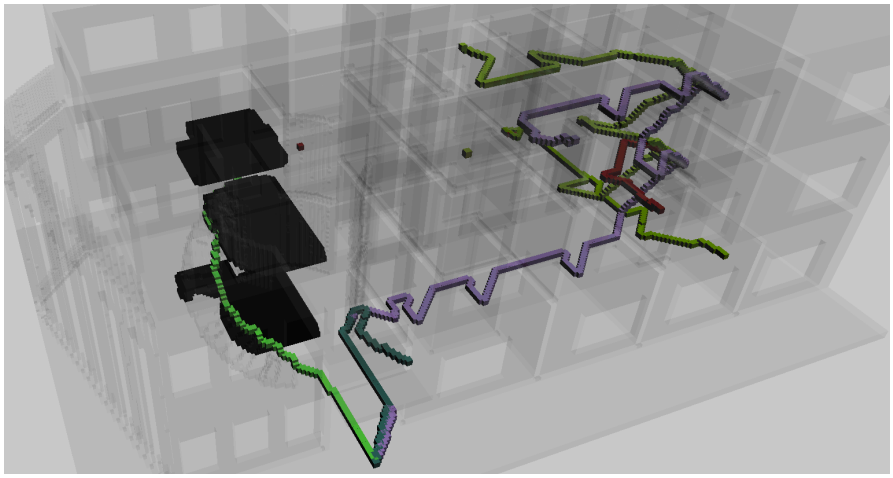


Figure 5.11: Paths generated by Theta* on a Morton grid, the lilac path has recomputed.

5.5 THETA* ON A MORTON GRID

Theta* on a Morton grid has many more data points than its grid implementation, which gives a better view of how the algorithm performs. The performance results are visible in Figure 5.12. In Figure 5.12a, the same relationship between path length and computation time is visible as in the previous implementations. However, it seems that Theta* on an Morton grid is less sensitive for the effects of high simulation loads with regards to performance, as the highest visible time value is 5 seconds. It should be noted however that half of all of the threads have failed to find a path, which could skew the time measurements in favour of "easier" paths, which require less time to compute, even though the starting points are randomised.

In Figure 5.12b we can see the same relationship between nodes visited and time as in the previous implementations, where it seems the effect on simulation load is more pronounced than when solely looking at path length. In Figure 5.12c, Theta* seems to satisfy the linear relationship, that a higher path length leads to visiting more nodes, and though this is not as pronounced as A* on the regular grid, it is not as negligible as A* on the Morton grid.

Furthermore, in Figure 5.12d, we notice a notable inversion of the relationship seen in both A* implementations, with Theta* on the Morton grid having shorter compu-

tation times on higher simulation loads. It should be noted again that Theta* still fails 50% of its runs. The quirks that Theta* had on the regular grid remain for its incarnation on the Morton grid, with exaggerated zigzagging and strange leaps into a room before exiting the room again, as evidenced in Figure 5.10 and Figure 5.11. Again, there is no clear indication that the algorithm is able to compute paths at any angle, and it sometimes even seems to not want to travel in a straight line, which is strange. Lastly, it also makes an unnecessary amount of turns that seem to not make any sense at all.

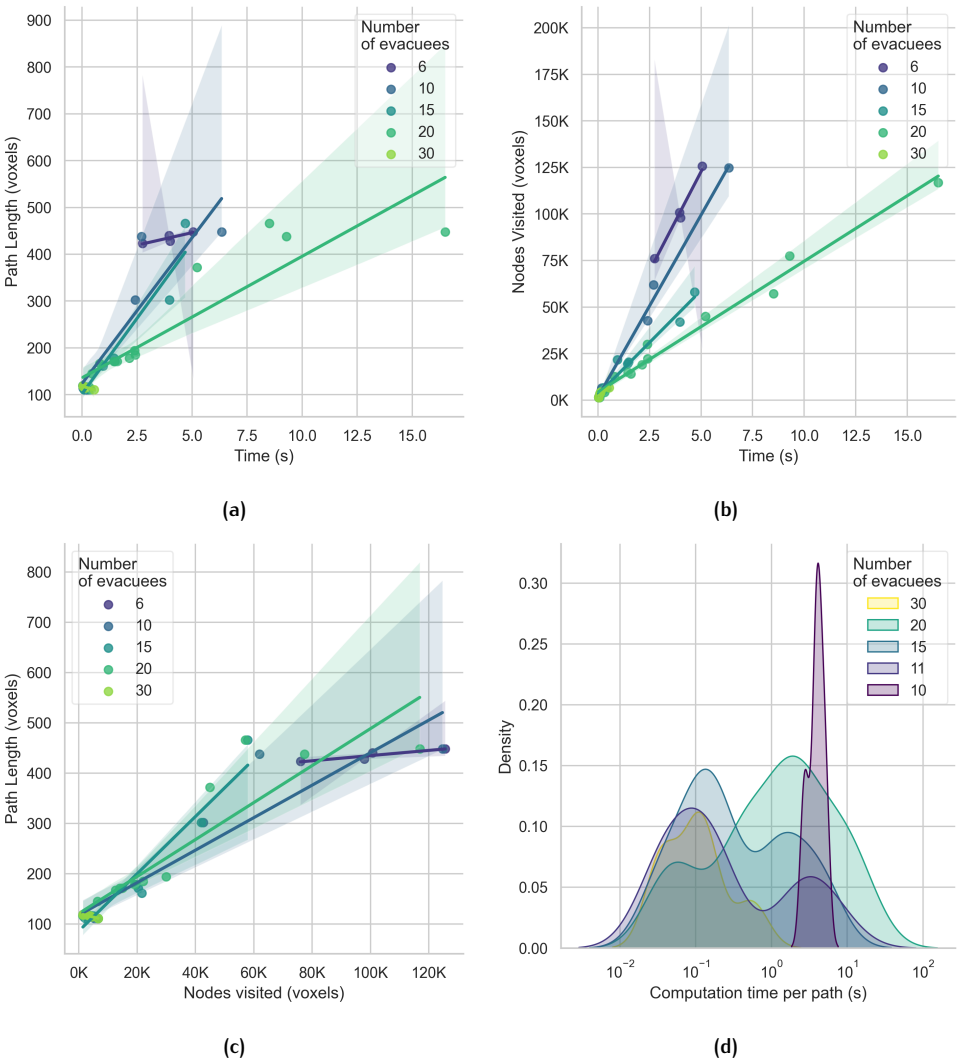


Figure 5.12: Performance results for Theta* on the Morton grid. (a) Time and path length. (b) Time and nodes visited. (c) Path length and nodes visited. (d) Density of time per path on a logarithmic scale.

5.6 SMARTER PATHS

In Section 3.8, it was noted that an optimal solution would require a pathfinding algorithm that was time-aware. This extended variation on A* was implemented using the steps described in Section 4.9. In this section we discuss the results of this implementation.

Algorithm	Success Rate [%]	Average memory footprint when running on max [MB]
Idle	-	464.8
A* on svo	100.00	860.87
Incremental A*	93.40	697.28

Table 5.3: The performance of the algorithms.

In Table 5.4, we can see that the time performance is also good for this time-aware version of A*, as well as achieving a small reduction in the number of nodes that need to be visited per run. This makes sense because nodes which are part of *any* path are not allowed to be revisited, which will automatically lead to less nodes which can be visited.

Algorithm	Mean calc time [s]	Min calc time [s]	Max calc time [s]	Mean number of nodes visited
A* on svo	5.56	0.0059	20.02	125560.29
Incremental A*	4.22	0.0633	12.39	116704.57

Table 5.4: The time statistics of the algorithms and the mean number of nodes visited per algorithm.

In Figure 5.13, we can see a detailed breakdown of the results. If we look at the detailed analysis of the results of the time-aware A* version, we can see a less clear correlation between path length and time per path in Figure 5.13a. This could also be because for all the paths, they can only get shorter over time that is, if a path needs to be recalculated then the path will always be shorter because the starting point will have moved down the path as time has gone on.

Next, in Figure 5.13b, there is a clearer relationship between the time it takes to calculate a path and the number of nodes the algorithm needs to visit. Note that the simulation load almost has no influence on this relationship, in contrast to the other algorithms. In Figure 5.13c, there is a rather unclear relationship between path length and the number of nodes visited. While there appears to be a slight positive linear trend between, this is not as clear as in other implementations.

Lastly, in Figure 5.13d, we see again that simulation load has less of an effect for this time-aware implementation, while also seeing a good time performance, with most of the runs being steady between 10^0 and 10^1.

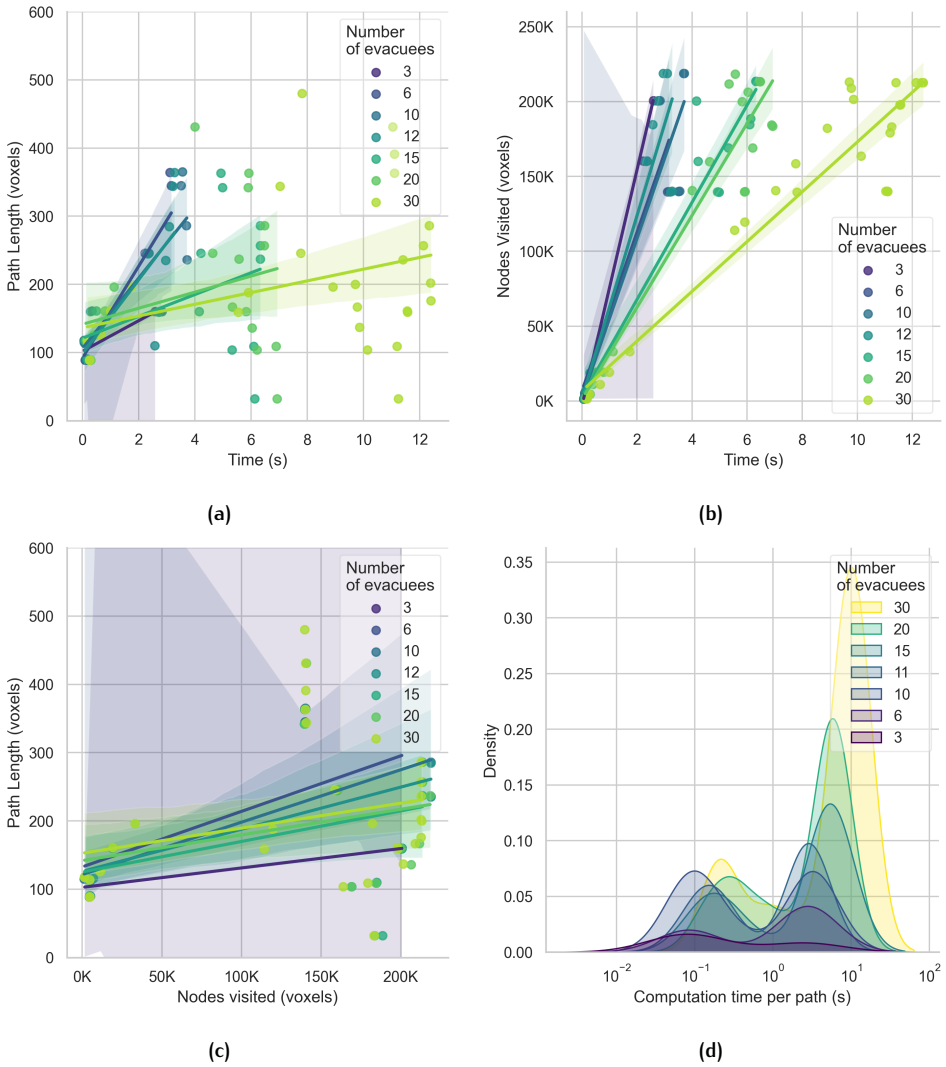


Figure 5.13: Performance results for time-aware A* on the Morton grid. (a) Time and path length. (b) Time and nodes visited. (c) Path length and nodes visited. (d) Density of time per path on a logarithmic scale.

In Figure 5.14, we can see the occupied voxels being rendered in real time.

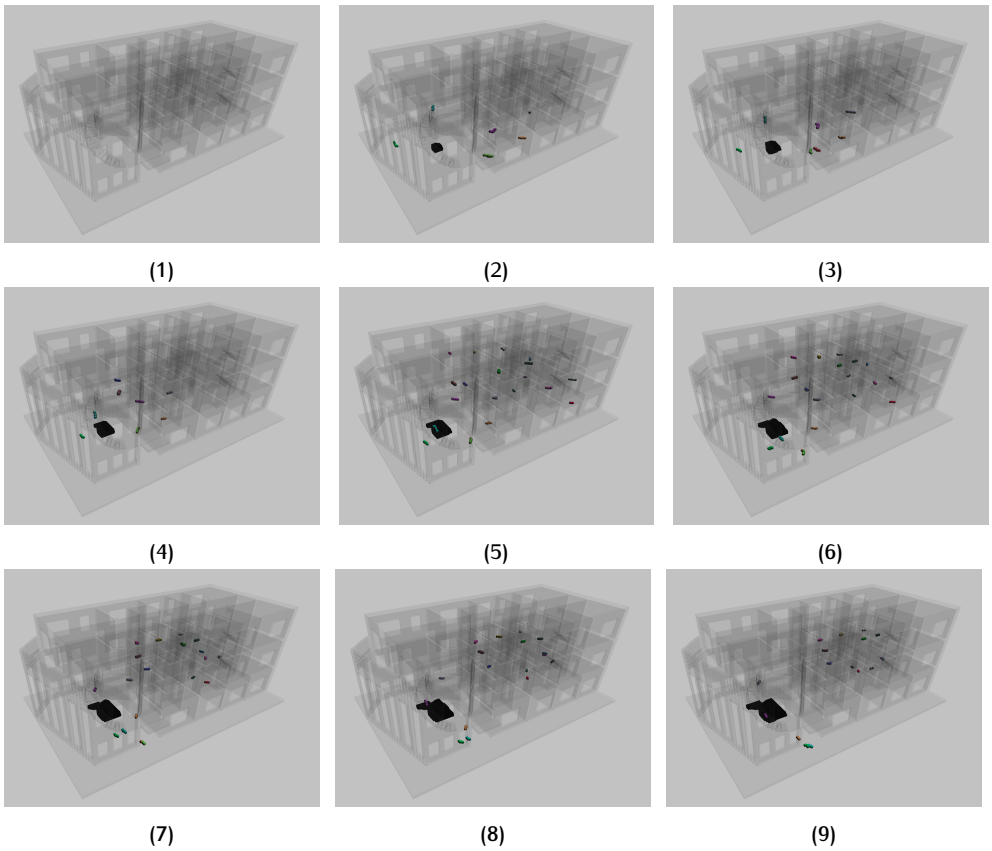


Figure 5.14: The time-aware A* algorithm. The images are in sequence and showcase the paths "travelling" during the simulations.

5.7 ROTATING THE DATASET

Because the grid and the limitations it imposes on movement for the pathfinding algorithms, the dataset was rotated 30° to see if this makes a difference in the paths. A simulation ($n = 25$) was run with A* on the Morton grid on both datasets, with the results recorded separately, and the results can be seen in Table 5.5. On first glance, the results for the rotated dataset look incredible. But due to the nature of the voxelisation of lines in non-axis directions, the rotated dataset has only 179.316 walkable nodes, compared to 253.674 in the regular dataset, which is thus only 70% of the search graph. However, this does not explain why this rotated dataset is 4 times faster than the regular one.

Algorithm	Mean calc time [s]	Max calc time [s]	Mean no. of nodes visited	Mean path length
A* on Morton	6.14	17.18	111519.19	239.75
A* on Morton (rotated)	1.79	5.89	51493.19	175.36

Table 5.5: The time statistics of the algorithms and the mean number of nodes visited per dataset variant.

Additionally, when we look at the paths themselves, we see that they look good, accurate and short. In Figure 5.16, we can also see that the paths are recomputed when the fire spreads, which means that the rotation of the dataset does not affect the correct functioning of the algorithms.

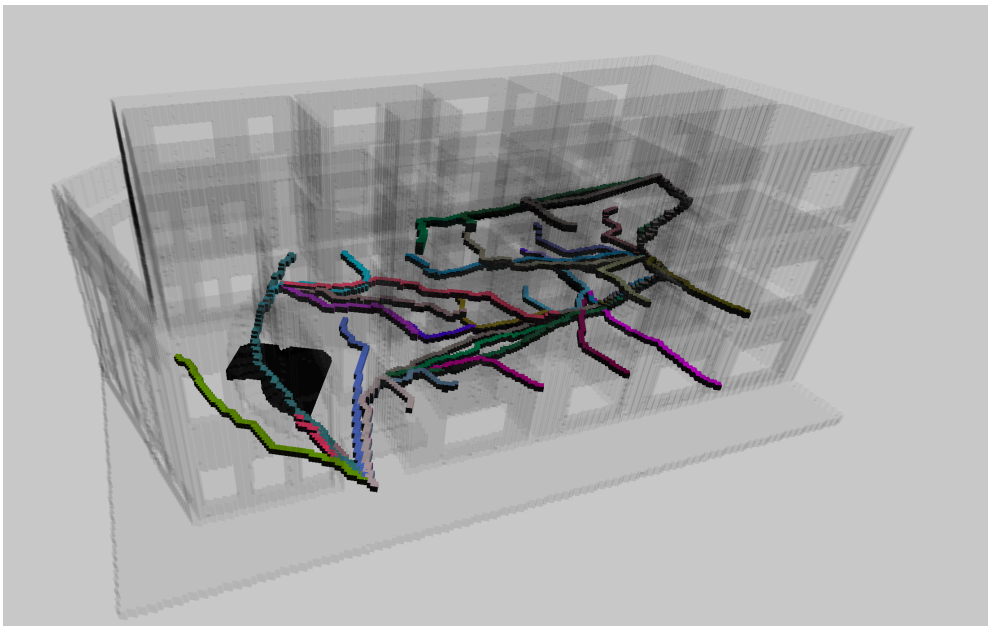


Figure 5.15: Paths in the rotated dataset

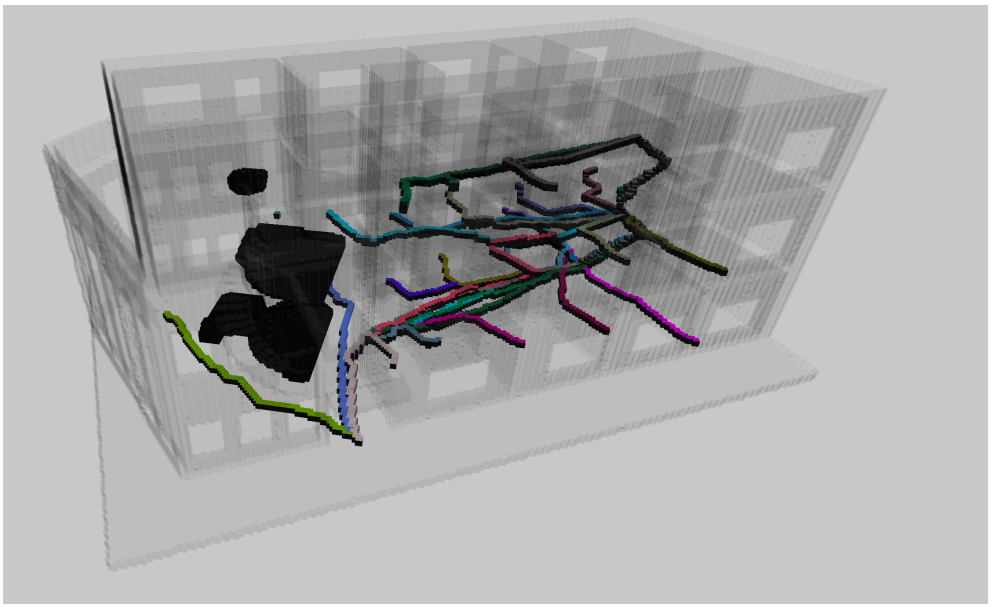


Figure 5.16: Paths in the rotated dataset, note that the paths going down the semi-circular stairs have been recomputed accurately.

6 CONCLUSIONS AND DISCUSSION

In this chapter the conclusions of this research will be presented. First, the research questions will be assessed and a conclusion will be drawn from the research results. Next, the research will be discussed; what went well, and what could be improved. This chapter will conclude with a future work section, detailing steps that could be taken to improve aspects of this research. All code can be found at GitHub.

6.1 CONCLUSIONS

In this section, every research (sub)question will be answered, based on the evidence supplied in the preceding chapters. First, the main question will be assessed, and next, the sub questions will be answered.
To recall the main research question of this thesis:

> *Which algorithm is best suited for multi-actor real-time pathfinding in a dynamic 3D voxelised indoor space?*

And recalling the definition of "suited-ness" as presented in the methodology, where success rate, computation time, the size of the data structure and nodes visited are all taken into account to assess this, we can formulate an answer to the main research question. This answer being:

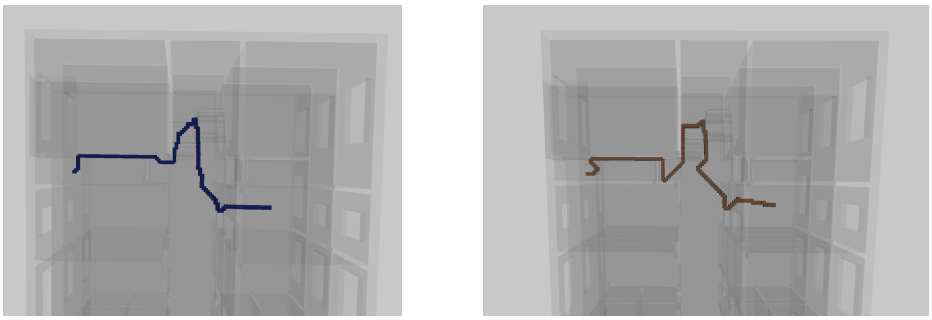
> *A*, with all its limitations, can be considered to be most suited for multi actor real-time pathfinding in a dynamic 3D voxelised indoor space.*

When looking at the five dimensions of suitability in order of importance, A* scores best in the success rate, with all A* algorithms achieving a 100% success rate. This means they are able to find a safe path through the building before the simulation ends. If we consider the context of what these paths are for, the evacuation routes, success rate is of highest importance. But A* does not only excel the rate of success, but also in the quality of its success, being able to handle higher simulation loads on its successful runs. Additionally, A* finds the shortest paths of all tested algorithms. While Dijkstra is of course guaranteed to find the shortest path, and Section 4.5 shows that A* is not guaranteed to do so, it does, on average, find the shortest of all the algorithms that were compared. Furthermore, we can see that A* produces the most natural looking paths, as can be seen in Table 5.2. In Figure 6.1 we see that Theta* is strangely bound to the grid directions, while A* follows a natural looking shortest path. In Figure 6.2 we see that A* does somewhat bend near door openings, but not nearly as much as Theta*.

When looking at the third part of *suited-ness* as defined in the methodology, computation time, we can see that A* performs well. As mentioned in Chapter 5, the time performance of Theta* is markedly better, but this is skewed by the high level of failure. Regardless, a mean computation time of 4.16$[s]$ on the regular grid, 5.56$[s]$ on the Morton grid and 4.17$[s]$ on the time-aware version, is a sufficiently short time when considering an evacuation, especially when considering that the algorithm is running up to 30 pathfinding threads concurrently and visualising them. The maxima of the time computation are a rather high considering the size of the dataset, especially when looking at the grid based implementation, which has a maximum

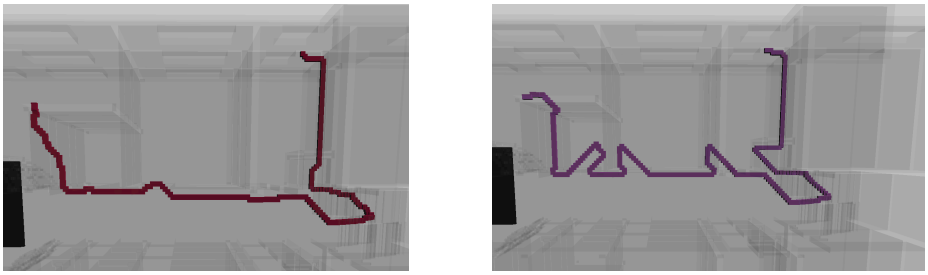
computation time of close to a minute, which is rather a lot when evacuating a building with the size of the dataset that was used. the Morton grid based implementation however already cuts this down to 20.02 seconds, while increasing the mean calculation time, suggesting that it is less prone to dataset size. The fourth marker that was looked at, the data structure size is, slightly harder to measure exactly, and will be answered in more detail when answering the first research sub question, but in general when looking at the mean memory footprint of A* compared to Theta*, we can see that A* performs well, with only a 200–450 megabyte increase in memory while the algorithm is running the maximum load simulation. For Theta* these results are quite distinct, ranging from the low of 528$[mb]$ to the high of 5400$[mb]$. Therefore, also because of its performance instability, the exact data space complexity will be somewhere in the middle, which is still higher than A*.

The last criterion of "suited-ness", the number of nodes visited, is a more concrete measure of assessing the efficiency of a pathfinding algorithm. In this measure, A* does perform comparable to Theta*, except on the Morton grid, when it has an five-fold increase in the number of nodes it processes to find paths, which is also the case when running the incremental version of A* on the Morton grid. This increase in nodes visited is not reflected in the memory footprint of the Morton grid implementation, which is lower than its regular grid-based counterpart. This higher node count does explain why the Morton grid based A* algorithm has a higher computation time, but there is no obvious reason for it to process this many more nodes, as both algorithms are identical.



(a) A* **(b)** Theta*

Figure 6.1: Path zigzagging Theta*. Theta* starts out with a more natural path at a non-grid angle, whereas A* follows a natural path



(a) A* **(b)** Theta*

Figure 6.2: Path zigzagging in Theta*. The zigzagging near doors is very extreme with Theta*.

The first sub question of this thesis was:

> *Which data structure for the voxelised space is best suited for dynamic pathfinding algorithms?*

The answer to this question is slightly more complex than the main research question. When looking at the results, it seems that the Morton grid has the better performance. It has slightly lower computation times than the grid, it is able to handle higher simulation loads, and seems to be less prone to the influence of path length on computation time. When comparing the size of the data structures, the Morton grid is only about one-and-a-half times as large in theory, and requires less memory when the algorithms are being run on it. The key point of the Morton grid is that it is able to store more information over periods of time, which is crucial for simulation style of computing. Therefore, it is able to know which paths are using which voxels, which in turn can be used to signal to those paths that they need to recompute, reducing the amount of path recalculations needed for dynamic pathfinding. While the grid might be smaller and faster (on small datasets), the fact that it does not store anything beyond the current value is not suitable for dynamic pathfinding with multiple actors.

Constructing the Morton grid is currently done by constructing an octree, but this could be simplified by just generating Morton codes for each non-empty voxel, adding these to an array, and then sorting this array. Furthermore, because the Morton grid is essentially an (unordered) map, it could also be used in other infrastructures, like a RDBMS for things like web based approach.

The second sub question of this thesis was:

> *How to include dynamic events into the pathfinding algorithms*

Using a data structure that is able to store multiple attributes, combined with a data structure for which it is possible to have multiple threads accessing and modifying it, it is possible to include dynamic events. The key to this is having pointers to objects in memory, allowing these pointers so be shared across threads, and checking whether the information stored at these pointers has changed. Looking at the memory consumption of the two data structures we can see that the Morton grid approach is less heavy on the CPU, and therefore checking the paths for changed voxels is much more efficient than repeatedly "single-shotting" the paths as is done on the regular grid.

The third sub question of this thesis was:

> *Is it possible to combine evacuation simulation with pathfinding?*

This is possible, when using a data structure that supports the temporal dimension. When using a data structure that is not time-aware, it is not possible to include the information required for evacuation simulations, because the simulator needs to have awareness of where evacuees are at every moment in time, and also needs to be able to access previous states of a voxel. Ideally, this would thus be a data structure that stores a locational position and a temporal position. A hurdle with this, is that, as is the case in the spatial dimension, a high temporal resolution can lead to a lot of data points, and thus a lot of space being taken up in memory.

6.2 DISCUSSION

In this section, some elements that were not implemented successfully, or eventual problems and limitations of the current approach are discussed.

6.2.1 Incremental vs. Non-incremental pathfinding

While it was assumed in Chapter 1 that one voxel should only be able to hold one "person" at a time, this proved to be a limiting factor for most of the pathfinding algorithms, due to the cutting-off phenomenon as described in Section 3.8. It also creates a more fundamental question in this thesis: are we simulating or are we pathfinding? While the word simulation has been used quite frequently to describe the parameters that determine the testing of the pathfinding, it should be clear to note that this research is not concerned with running full evacuation simulations, but rather with testing pathfinding algorithms by running them with certain parameters and a changing environment. Thus, the most correct way to incorporate the occupancy of a building by limiting the occupancy of voxels, but to achieve that, the pathfinding algorithms would have to be adapted to such a degree that they could account for the temporal dimension, or tests would have to be run with full incremental pathfinding algorithms which innately support the temporal dimension. Thus, in this research, both full incremental and non-incremental pathfinding algorithms have been explored, and a hybrid time aware version of A* has been developed that is retains the simplicity of A* while obeying changing occupancy over time.
In Figure 5.13, the entire run of incremental A* has been shown in a storyboard fashion. This version of incremental A* removes old paths from the instance buffers and writes the new paths to the buffer almost 3 times per second, but it is able to keep up, and it does illustrate the benefits of this pathfinding paradigm. Something that still needs to be solved for A* however, is how to deal with multiple potential goals. In the current implementations, every thread selects the exits during the initialisation stage of the pathfinding algorithm from the list of exits (which is only 2 long in this dataset), and it selects this on the heuristic to make sure it matches with distance estimations. However, the incremental version is not able to deal with a changing exit, because it would be confused by the sudden drop in distances, and could not take the "hard work" it has previously done with it, it would need to start anew and invalidate all previously calculated distance and cost estimations. While this is not an impossibility to overcome, it is a challenge to do efficiently. For instance, how often would you do this exit reevaluation? On every loop iteration? Or once every 10 iterations for instance. How can the algorithm even know if there *exists* a path to the other exit. It might be closer in terms of distance, but for the path to go there might be longer than when it would have stuck with its original exit, due to the layout of the building. If we bring this into the context of the research, evacuation simulations, information such as this would be of vital importance, adding intelligence to a complex situation and aiding humans by having the overview of the spatial situations that the people are lacking.

6.2.2 D*-Lite

In Section 4.8, the shortcomings with the 3D approach of D*-Lite were highlighted, with the focal point of attention being the tie breaking function, which works in two dimensions, but fails in three dimensions. While the algorithm is certainly useful, it is also not ideal because of the other shortcoming previously described, namely, the need for every thread to maintain its own version of reality. Though in essence, that is what all the algorithms do to a certain degree, D*Lite does not really function without *a full view*, at least with this implementation. Because theoretically speaking it should not need full knowledge of the dataset, yet most implementations do use

this. The implementation that was created specifically for this research did so with the same on the fly node generation or using the `cost_so_far` and `came_from` maps as A* and Theta* and still ran into the same tie breaking issues, which remain unsolved. Therefore, D*-Lite (and LPA*) have potential, but suffer from flaws that do not make them workable for voxels.

6.2.3 Theta*

Theta*, and the other related any-angle[6] path planning algorithms, are promising because of their power to rise above the grid and plan paths not limited to the arbitrary directions of the grid, in theory. In practice however, these paths did not perform as well as I hoped. While some of this could be due to the more complex nature of these algorithms (i.e. more steps, more computation, more room for things going wrong), there is also the fundamental issue that these algorithms use line of sight analysis that is often designed and tested for a 2D grid. Though not surprising it remains disappointing that extending this to 3D does not produce the desired paths.

6.2.4 Full vs partial octree

As briefly mentioned earlier, the octree that was initially constructed is a full octree, with all the bells and whistles, but due to the limitation of on the one hand, the complex and computationally heavy neighbour generation for dynamic octrees and, on the other hand, the small distances between heuristic values that have a large impact on the path as a whole, the decision was made to only use the nodes at the same octree depth. I would argue that this is enough of an improvement on the regular grid to be a success. Firstly, due to the flatness of the search graph (i.e. the navigable space) the octree only contains 253.674 nodes at the leaf level. In the level above, only 20.133 nodes are "full", i.e. they have 8 children and can thus be used to simplify the space. In the level above, only 2652 full octree nodes remain, and all the other levels (6 left) have no full nodes at all. Therefore, I posit that the gains that could be achieved are not as momentous, and do not automatically outweigh the costs mentioned above while the performance gains on the pathfinding when not using the full octree are still significant.

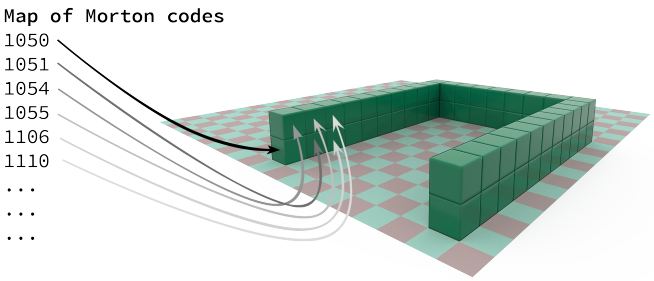


Figure 6.3: A visualisation of what the SVO data structure holds, visible is the sequential sparse data that is ordered by Morton code.

Secondly, while the full potential of the octree might not be used in the pathfinding implementations, spatial clustering of the Morton codes do allow for a type of spatial clustering that the grid does not have. Putting these nodes in a fast hash table-esque container like `std::unordered_map` also allows for very fast lookup, insertion and re-

6 Any-angle pathfinding algorithms is the name Nash et al. give this class of pathfinding algorithms. It is a slightly misleading name, because the paths are still bound to the grid

moval, without the need to rearrange the entire tree above, the integrity of the spatial structure is still maintained due to the Morton ordering. While a regular grid would need much more iteration to find a voxel because of the many redundant and empty voxels, using the lower level of the octree this way skips the limitations of the grid while avoiding the hassle of octree management.

Thirdly, due to the fact that the octree stores sparse information only, as seen in Figure 6.3, we can use the extra space that would have otherwise been used by empty voxels to store more information per node, allowing for the smarter and incremental paths that were seen in this thesis. Therefore, while an octree can certainly have merits for pathfinding in 3D, when the search graph is as flat as the navigable space above the floor, little spatial compression is possible. Combined with the small margins with which the heuristic values are compared, that do not compare across resolutions, this sparse Morton grid data structure suffices.

6.2.5 Rotated dataset

It is very interesting to see the results of the rotated dataset in Figure 5.15 and Figure 5.16, which performs better than when the dataset is aligned with the grid axes. Additionally, the paths are visually also quite appealing. This is unexpected, and should be more thoroughly investigated. A reason for the shorter computation times could be that there are less walkable voxels in the search graph, as mentioned previously. A smaller search graph does not explain the better looking and shorter paths. A reason for that could be that with the Euclidean distance, the shortest path is often found by "cutting corners" instead of travelling in the axis directions, and if there are no lines in the axis directions to begin with, shorter paths are found.

6.2.6 Multiple Fires

While multiple fire locations were selected at the start of this research, and this is a parameter that was mutable in the simulator, all the simulations that were tested with this implementation and recorded in the results section of this report, were done with the fire starting in the same spot. This was done for two reasons. One, it makes comparing the algorithms easier, because that is the only factor that is changing, and two, many other locations would have resulted in blocking off most of the paths because of the narrow hallways, which would have invalidated many path results. The location that was chosen, under the semi-circle stair was chosen precisely because it would force the algorithms to take the other stairwell, and show that the shortest path is not always the safest path. Furthermore, I think that if one would like to test the effect of fire location on the paths, this could be an entire thesis unto itself.

6.2.7 Actor size

In this research, the size of the actor has been set on one voxel, but this could also have been extended to represent the true size of an actor, as per Koopman [2016], which would have made the paths more accurate, but this is limited by the implementation. This could be achieved by adding a buffer to around the currently occupied voxel and blocking these buffer voxels for other paths, an example 2D implementation of this can be seen in Figure 6.4. This could result in more accurate paths in terms of overcrowding of spaces.

6.2.8 Using RDBMS

In the implementation, many steps were taken to ensure multi thread data access, but one could have consider using a relational database to put all the data in, and

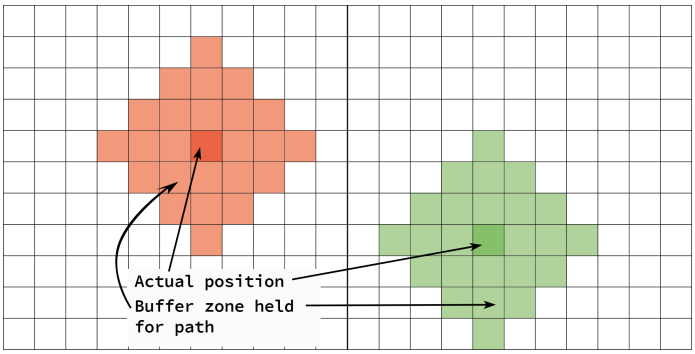


Figure 6.4: A possible solution for including the actor size

query the voxel status there. While this was considered, it was decided not to do this to ensure full control over data access. This decision was also partly based when considering the highly dynamic nature of the voxel data, meaning the insertion and deletion updates would be heavy. Moreover, the data is not so large that it is impractical to store in memory. Considering that pointers are used extensively throughout the implementation to record different inter-voxel relationship, and the fact that this current exploratory research has no reason to be stored non-locally, for this thesis it was concluded that it would be best to focus solely on in-memory data storage.

6.2.9 Multiple exits

An issue with the current implementation is that A* chooses one exit at the start, and cannot find a different exit midway through its computation when the situation has changed, without just deleting all the previous information and starting anew, but then it does not know if this path is actually shorter than the path it would otherwise have taken. This is why D*-Lite and LPA* store an extra value, to check if the situation has improved or worsened. However, other solutions are possible, such as initially calculating a path for the evacuee to every exit, and then choosing the shortest path. But, this is fine when there are only a few exits, but more difficult to manage with larger and more complex buildings. Secondly, sometimes a path needs to be recalculated more than 10 times per simulation, and if the dynamic aspect of the simulation is to be included, every exit should also be reevaluated at every recalculation, significantly increasing the memory load for every path by n_{exits}.

6.3 FUTURE WORK

6.3.1 Extending semantic information

Currently the voxels have options for the algorithms: walkable or not walkable. It could be possible to extend this more, and incorporate it into the decision making process of the paths and by adding more intelligence to the voxels, the algorithms can infer more information from the voxel space and make even smarter paths. For instance, one could think of adding the width of the walkable space to the voxels' attributes, which the algorithms can then use to avoid spots that are potential choke points by assigning a higher cost to these voxels. Currently, travelling from one voxel to the other is only dependent on the distance between the voxel and the next (which is $1 \vee \sqrt{2} \vee \sqrt{3} \cdot s_{voxel}$, depending on connectivity), but more information could be added to this step, at little extra computation cost because if this information is stored in the voxel itself, and the algorithm is looking to calculating the cost of going to that

voxel, then the algorithm is already *at* that voxel.

6.3.2 Path smoothing & map generation

As described earlier, some of the paths can zigzag. There is the possibility of applying path smoothing techniques to the paths after they have been found, but for the A* paths this is not needed. Path smoothing could in this context also be making the paths look more visually appealing. Many different path smoothing options exist as found in Patel [2022], and these come at different costs to the implementation. Another thing, is that many of the paths now skirt the walls or edges of the walkable space, and were you to make these paths useful as a tool for evacuation, a way would have to be found to reposition them to the centre of the navigable space. While these are cosmetics, when thinking of applying the results of this research to for instance, automatic evacuation map generation, the paths need to be logical and communicable for the users.

The next step would then be to integrate this implementation with a positioning system, and develop an augmented reality (AR) or virtual reality (VR) application that can guide real users through a building and use the computed path to guide the evacuees to the exits. How the paths look, is of great importance for these kinds of implementations, and therefore research should be done as to how the paths will can best be presented. For instance, is it logical to have many paths skirt the wall because this is simply the true shortest path, or would users misunderstand this?

6.3.3 Dataset size

All the tests and simulations in this thesis has been done on a single dataset, which is not ideal. Ideally, I would have liked to use a second, larger and different indoor dataset to test the success of the solution on that dataset. While some results do point to the Morton grid based implementation being able to manage that, because the longer paths do have computations that are comparable to the shorter paths, it would have been interesting to see the solution on another dataset that differs from this one not only in size, but also one with other attributes of the indoor space.

6.3.4 Different resolution

Further research could be done on the influence of the resolution of the voxel grid. In this research, a single voxel size has been used for all the paths and both datasets. Research could be done on if the resolution size has influence on the quality of the paths. An interesting option could be to see what the lowest possible resolution is at which this type of pathfinding (dynamic, with multiple actors) can still be done, because a lower resolution will invariably lead to lower computation times.

6.3.5 Using a full octree implementation

As previously stated, this implementation only uses one level of the octree as a sparse Morton grid. Therefore, all the voxels have the same size. In further research, a true octree implementation could be done that passes the following criteria: dynamic neighbour generation (pre-computed neighbours take up too much space) and live updating of tree when the search graph changes due to events. This implementation could be compared with a sparse Morton grid, or a regular grid, and the path quality as well as the computation speeds could be analysed.

6.3.6 A dynamic 3D maze

Currently, this implementation is rooted firmly in the evacuation context, with a fairly regular spatial, albeit artificial, dataset. To fully check the capabilities of this method of dynamic multi-actor pathfinding, a stress-test could be devised in the form of a dynamic 3D maze, to see how well the implementation holds up. Mazes are particularly difficult for pathfinding algorithms, because often a simple distance heuristic can be misleading in a maze that has many dead ends winding paths.

6.3.7 Choice of heuristics

In Section 4.5, it is shown that the choice of heuristic has a very large influence on the length and quality of the paths, as well as the computation time. Considering that these are only the most commonly used distance based heuristics, further research could be done by looking at the influence of heuristic choice for A* in dynamic evacuations, and look into using non-distance based heuristic. Another option is increasing the complexity of the heuristics by incorporating other aspects such as vertical movement, the number of people currently in an area.

BIBLIOGRAPHY

Alattas, A. (2022). *The Integration of LADM and IndoorGML to Support the Indoor Navigation Based on the User Access Rights*. PhD thesis.

Aleksandrov, M., Zlatanova, S., and Heslop, D. J. (2021). Voxelisation Algorithms and Data Structures: A Review. *Sensors*, 21(24):8241.

Algfoor, Z. A., Sunar, M. S., and Kolivand, H. (2015). A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. *International Journal of Computer Games Technology*, 2015:1–11.

Arndt, J. (2010). *Matters Computational: ideas, algorithms, source code*. Springer Science & Business Media.

Baert, J. (2018). Libmorton: C++ morton encoding/decoding library. https://github.com/Forceflow/libmorton.

Baert, J., Lagae, A., and Dutré, P. (2013). Out-of-core construction of sparse voxel octrees.

Botea, A., Müller, M., and Schaeffer, J. (2004). Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, 1(1):1–30.

Brewer, D. and Sturtevant, N. R. (2018). Benchmarks for Pathfinding in 3D Voxel Space. *Symposium on Combinatorial Search (SoCS)*.

Canny, J. and Reif, J. (1987). New lower bound techniques for robot motion planning problems.

Careil, V., Billeter, M., and Eisemann, E. (2020). Interactively Modifying Compressed Sparse Voxel Representations. In *Computer Graphics Forum*, volume 39, pages 111–119. Wiley Online Library, Wiley.

Cohen-Or, D. and Kaufman, A. (1997). 3D line voxelization and connectivity control. *IEEE Computer Graphics and Applications*, 17(6):80–87.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT Press, fourth edition.

Cornut, O. (2020). Dear ImGui.

Dado, B., Kol, T. R., Bauszat, P., Thiery, J.-M., and Eisemann, E. (2016). Geometry and Attribute Compression for Voxel Scenes. *Computer Graphics Forum*, 35(2):397–407.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Flikweert, P., Peters, R., Díaz-Vilariño, L., Vôute, R., and Staats, B. (2019). Automatic extraction of a navigation graph intended for IndoorGML from an indoor point cloud. *ISPRS Journal of Photogrammetry and Remote Sensing*, IV-2/W5:271–278.

Foead, D., Ghifari, A., Kusuma, M. B., Hanafiah, N., and Gunawan, E. (2021). A Systematic Literature Review of A* Pathfinding. *Procedia Computer Science*, 179:507–514.

Gorte, B., Aleksandrov, M., and Zlatanova, S. (2019a). Towards Egress Modelling in Voxel Building Models. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-4/W9:43–47.

Gorte, B., Zlatanova, S., and Fadli, F. (2019b). Navigation in Indoor Voxel Models. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-2/W5:279–283.

Hart, P., Nilsson, N., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

Hoang, L., Lee, S.-H., Kwon, O.-H., and Kwon, K.-R. (2019). A Deep Learning Method for 3D Object Classification Using the Wave Kernel Signature and A Center Point of the 3D-Triangle Mesh. *Electronics*, 8(10):1196.

Holzmüller, D. (2017). Efficient Neighbor-Finding on Space-Filling Curves. *Computing Research Repository*.

Houston, B., Wiebe, M., and Batty, C. (2004). RLE sparse level sets. In *ACM SIGGRAPH 2004 Sketches*, page 137. ACM Press.

Hübner, P., Weinmann, M., Wursthorn, S., and Hinz, S. (2021). Automatic voxel-based 3D indoor reconstruction and room partitioning from triangle meshes. *ISPRS Journal of Photogrammetry and Remote Sensing*, 181:254–278.

ISO (2020). *ISO/IEC 14882:2020: Programming languages — C++*. ISO, sixth edition. Available in electronic form for online purchase at `http://webstore.ansi.org/`, `http://www.cssinfo.com/` and `https://www.iso.org/standard/79358.html/`.

Jones, C. B. (1989). Data structures for three-dimensional spatial information systems in geology. *International journal of geographical information systems*, 3(1):15–31.

Khan, A., Aesha, A. A., Aka, J. S., Rahman, S. M. F., and Rahman, M. J.-U. (2018). An IoT Based Intelligent Fire Evacuation System. In *2018 21st International Conference of Computer and Information Technology (ICCIT)*, pages 1–6.

Kobes, M., Helsloot, I., de Vries, B., and Post, J. G. (2010). Building safety and human behaviour in fire: A literature review. *Fire Safety Journal*, 45(1):1–11.

Koenig, S. and Likhachev, M. (2001). Incremental A∗. *Advances in neural information processing systems*, 14.

Koenig, S. and Likhachev, M. (2002). D* lite. *AAAI*, 15.

Koopman, M. (2016). 3D Path-finding in a voxelized model of an indoor environment.

Kämpe, V., Sintorn, E., and Assarsson, U. (2013). High resolution sparse voxel DAGs. *ACM Transactions on Graphics*, 32(4):1–13.

Lv, C., Lin, W., and Zhao, B. (2021). Voxel Structure-based Mesh Reconstruction from a 3D Point Cloud. *IEEE Transactions on Multimedia*, abs/2104.10622:1–1.

Ministerie van Binnenlandse Zaken en Koninkrijksrelaties (2011). Bouwbesluit 2012.

Morton, G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing.

Muratov, T. and Zagarskikh, A. (2019). Octree-Based Hierarchical 3D Pathfinding Optimization of Three-Dimensional Pathfinding. In *Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control*, ISCSIC 2019, New York, NY, USA. Association for Computing Machinery.

Nash, A. and Koenig, S. (2013). Any-Angle Path Planning. *AI Magazine*, 34(4):85–107.

Nash, A., Koenig, S., and Likhachev, M. (2009). Incremental Phi*: Incremental any-angle path planning on grids. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Neufeld, J. (2015). D*-Lite. Online. Accessed at https://github.com/ArekSredzki/dstar-lite.

Noori, A. and Moradi, F. (2015). Simulation and Comparison of Efficency in Pathfinding algorithms in Games. *Ciência e Natura*, 37(Part 2):230–238.

Patel, A. (1997–2022). Amit's Thoughts on Path-Finding and A-Star.

Samet, H. (1989). Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, 46(3):367–386.

Sharma, S. K. and Kumar, S. (2016). Comparative Analysis of Manhattan and Euclidean Distance Metrics Using A* Algorithm. *Journal of Research in Engineering and Applied Sciences*, 01(04):196–198.

Shi, L., Xie, Q., Cheng, X., Chen, L., Zhou, Y., and Zhang, R. (2009). Developing a database for emergency evacuation model. *Building and Environment*, 44(8):1724–1729.

Staats, B. R., Diakité, A. A., Voûte, R. L., and Zlatanova, S. (2017). Automatic Generation of Indoor Navigable Space Using a Point Cloud and Its Scanner Trajectory. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-2/W4:393–400.

Stentz, A. (1993). Optimal and Efficient Path Planning for Unknown and Dynamic Environments. *INTERNATIONAL JOURNAL OF ROBOTICS AND AUTOMATION*, 10:89–100.

Sundar, H., Sampath, R. S., and Biros, G. (2008). Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708.

Tankyevych, O. (2010). *Filtering of thin objects : applications to vascular image analysis*. phdthesis, University Paris-Est.

Van Bemmelen, J., Quak, W., Van Hekken, M., and Van Oosterom, P. (1993). Vector vs. raster-based algorithms for cross country movement planning. In *Auto-Carto XI*, pages 304–304. ASPRS AMERICAN SOCIETY FOR PHOTOGRAMMETRY AND.

van der Laan, R., Scandolo, L., and Eisemann, E. (2020). Lossy Geometry Compression for High Resolution Voxel Scenes. *Proc. ACM Comput. Graph. Interact. Tech.*, 3(1).

Vörös, J. (2000). A strategy for repetitive neighbor finding in octree representations. *Image and Vision Computing*, 18(14):1085–1091.

Wang, S.-H., Wang, W.-C., Wang, K.-C., and Shih, S.-Y. (2015). Applying Building Information Modeling to Support Fire Safety Management. *Automation in Construction*, 59:158–167.

Wirth, E. and Szabó, G. (2018). Overlap-avoiding Tickmodel: An Agent-and Gis-based Method for Evacuation Simulations. *Periodica Polytechnica Civil Engineering*, 62(1):72–79.

Xiong, Q., Zhu, Q., Du, Z., Zlatanova, S., Zhang, Y., Zhou, Y., and Li, Y. (2017). Free Multi-floor Indoor Space Extraction from Complex 3d Building Models. *Earth Science Informatics*, 10(1):69–83.

Xu, Y., Tong, X., and Stilla, U. (2021). Voxel-based representation of 3D point clouds: Methods, applications, and its potential use in the construction industry. *Automation in Construction*, 126(103675):103675.

A APPENDIX A

Algorithm A.1: Basic Theta*

```
1  Function main():
2      initialise()
3      computeShortestPath()
4      if g(s_goal) ≠ ∞ then
5          return path found
6      else
7          return no path found
8  Function initialise():
9      open ← closed ← ∅
10     initialiseVertex(s_start)
11     initialiseVertex(s_goal)
12     g(s_start) ← 0
13     parent(s_start ← s_start)
14     open.Insert(s_start, g(s_start) + h(s_start))
15 Function initialiseVertex(s):
16     g(s) ← ∞
17     parent(s) ← ∅
18 Function computeShortestPath():
19     while open.TopKey() < g(s_goal) + h(s_goal) do
20         s ← open.Pop()
21         closed ← closed ∪ {s} foreach s' ∈ s_neighbours do
22             if s' ∉ closed then
23                 if s' ∉ open then
24                     initialiseVertex(s')
25                 updateVertex(s, s')
26 Function updateVertex(s, s'):
27     g_old ← g(s')
28     computeShortestPath(s, s')
29     if g(s' < g_old then
30         if s' ∈ open then
31             open.Remove(s')
32         open.Insert(s', g(s') + h(s'))
33 Function computeShortestPath(s, s'):
34     if lineOf Sightparent(s), s' then
35         if g(parent(s)) + cost(parent(s), s') < g(s') then
36             parent(s') ← parent(s)
37             g(s') ← g(parent(s) + cost(parent(s), s')
38     else
39         if g(s) + cost(s, s') < g(s') then
40             parent(s) ← s
41             g(s') ← +cost(s, s')
```

B APPENDIX B

```
static const std::array<int32_t, 256> deltaX=
    {1,7,1,55,1,7,1,439,1,7,1,55,1,7,1,3511,1,7,1,55,
    1,7,1,439,1,7,1,55,1,7,1,28087,1,7,1,55,1,7,1,
    439,1,7,1,55,1,7,1,3511,1,7,1,55,1,7,1,439,1,7,1,
    55,1,7,1,224695,1,7,1,55,1,7,1,439,1,7,1,55,1,7,
    1,3511,1,7,1,55,1,7,1,439,1,7,1,55,1,7,1,28087,1,
    7,1,55,1,7,1,439,1,7,1,55,1,7,1,3511,1,7,1,55,1,
    7,1,439,1,7,1,55,1,7,1,1797559,1,7,1,55,1,7,1,439,
    1,7,1,55,1,7,1,3511,1,7,1,55,1,7,1,439,1,7,1,55,
    1,7,1,28087,1,7,1,55,1,7,1,439,1,7,1,55,1,7,1,
    3511,1,7,1,55,1,7,1,439,1,7,1,55,1,7,1,224695,1,7,
    1,55,1,7,1,439,1,7,1,55,1,7,1,3511,1,7,1,55,1,7,
    1,439,1,7,1,55,1,7,1,28087,1,7,1,55,1,7,1,439,1,
    7,1,55,1,7,1,3511,1,7,1,55,1,7,1,439,1,7,1,55,1,
    7,1};
static const std::array<int32_t, 256> deltaY=
    {2,14,2,110,2,14,2,878,2,14,2,110,2,14,2,7022,2,14,
    2,110,2,14,2,878,2,14,2,110,2,14,2,56174,2,14,2,
    110,2,14,2,878,2,14,2,110,2,14,2,7022,2,14,2,110,2,
    14,2,878,2,14,2,110,2,14,2,449390,2,14,2,110,2,14,
    2,878,2,14,2,110,2,14,2,7022,2,14,2,110,2,14,2,878,
    2,14,2,110,2,14,2,56174,2,14,2,110,2,14,2,878,2,14,
    2,110,2,14,2,7022,2,14,2,110,2,14,2,878,2,14,2,110,
    2,14,2,3595118,2,14,2,110,2,14,2,878,2,14,2,110,2,
    14,2,7022,2,14,2,110,2,14,2,878,2,14,2,110,2,14,2,
    56174,2,14,2,110,2,14,2,878,2,14,2,110,2,14,2,7022,
    2,14,2,110,2,14,2,878,2,14,2,110,2,14,2,449390,2,
    14,2,110,2,14,2,878,2,14,2,110,2,14,2,7022,2,14,2,
    110,2,14,2,878,2,14,2,110,2,14,2,56174,2,14,2,110,
    2,14,2,878,2,14,2,110,2,14,2,7022,2,14,2,110,2,14,
    2,878,2,14,2,110,2,14,2};
static const std::array<int32_t, 256> deltaZ=
    {4,28,4,220,4,28,4,1756,4,28,4,220,4,28,4,14044,4,
    28,4,220,4,28,4,1756,4,28,4,220,4,28,4,112348,4,28,
    4,220,4,28,4,1756,4,28,4,220,4,28,4,14044,4,28,4,
    220,4,28,4,1756,4,28,4,220,4,28,4,898780,4,28,4,220,
    4,28,4,1756,4,28,4,220,4,28,4,14044,4,28,4,220,4,
    28,4,1756,4,28,4,220,4,28,4,112348,4,28,4,220,4,28,
    4,1756,4,28,4,220,4,28,4,14044,4,28,4,220,4,28,4,
    1756,4,28,4,220,4,28,4,7190236,4,28,4,220,4,28,4,
    1756,4,28,4,220,4,28,4,14044,4,28,4,220,4,28,4,1756,
    4,28,4,220,4,28,4,112348,4,28,4,220,4,28,4,1756,4,
    28,4,220,4,28,4,14044,4,28,4,220,4,28,4,1756,4,28,
    4,220,4,28,4,898780,4,28,4,220,4,28,4,1756,4,28,4,
    220,4,28,4,14044,4,28,4,220,4,28,4,1756,4,28,4,220,
    4,28,4,112348,4,28,4,220,4,28,4,1756,4,28,4,220,4,
    28,4,14044,4,28,4,220,4,28,4,1756,4,28,4,220,4,28,
    4};
```

COLOPHON

This document was typeset using LaTeX. The document layout was generated using the `arsclassica` package by Lorenzo Pantieri, which is an adaption of the original `classicthesis` package from André Miede.