Implementation and comparison of pathfinding algorithms in a dynamic 3D space

Bachelor Thesis

for attainment of the academic degree of B.Sc.

Carina Krafft

[redacted]

University of Applied Sciences Hamburg
Faculty of Design, Media and Information
Department Media Technology

First Examiner: Prof. Dr.-Ing. Sabine Schumann

Second Examiner: Prof. Dr. Edmund Weitz

Hamburg, 18.03.2019

Thema der Arbeit

Implementierung und Vergleich von Pfadfinde-Algorithmen im dynamischen 3D-Raum.

Schlüsselwörter

Dynamischer 3D-Raum, Computerspiel, Gegner UFO, A*/ARA* (Anytime Algorithmen), D*/D* Lite (dynamische Algorithmen), AD*

Kurzzusammenfassung

Diese Arbeit konzentriert sich darauf, einen geeigneten Pfadfinde-Algorithmus für dynamische 3-dimensionale Spielumgebungen zu finden, die zum Beispiel für UFOs in Weltraum-Spielen verwendet werden können. Hierfür werden verschiedene Pfadfinde-Algorithmen vorgestellt, implementiert und ihre Ergebnisse verglichen. Die Hauptaufgabe hierbei ist einen Pfadfinde-Algorithmus zu finden, welcher in einer dynamischen Umgebung gute Leistungen erzielt.

Topic of the bachelor's thesis

Implementation and comparison of pathfinding-algorithms in a dynamic 3D-space.

Keywords

Dynamic 3D-space, computer game, enemy UFO, A*/ARA* (anytime algorithms), D*/D* Lite (dynamic algorithms), AD*

Abstract

This bachelor thesis focuses on finding a suitable pathfinding algorithm for dynamic 3-dimensional game environments, to be used for example as UFO AI's in space-themed games. For this purpose different pathfinding algorithms will be introduced, implemented, and their results compared. The main focus hereby lays on finding a pathfinding algorithm, which performs well in a highly dynamic game environment.

Contents

List of Figures

List of Abbreviations

AI Artificial Intelligence

VR Virtual Reality

UFO Unidentified Flying Object

API Application Programming Interface

1 Introduction

1.1 Motivation

"Pathfinding-algorithms": For most people not affine to computers its a terribly scientific sounding term. In reality, though, they are nothing to be afraid of. Quite in the contrary - most of us use them every day without even thinking about it.

Most people immediately think of technology when hearing the word "pathfinding-algorithm". They are not entirely wrong. Pathfinding algorithms are used in a wide variety of modern technologies. The most commonly known application is probably the navigation system. But the use of pathfinding algorithms stretches far beyond that. Whether it is routing information through networks, directing enemy AI's[1] in video games, or steering autonomous driving vehicles - researches and developments of new and improved pathfinding algorithms are made in a wide variety of fields.

To fully understand the potential of pathfinding it is worth while taking a step back and looking away from technology for a moment. Pathfinding starts with simple, human tasks, such as getting to work each morning. Commuting in larger cities, there are several routes and several means of transportation to choose from. Some take longer, some are more crowded, some are more demanding, and so on. Depending on our current mood or situation we regard each attributes, such as time, crowds, activeness, with different significance and then choose the path avoiding most of the unwanted attribute. In a way we perform a pathfinding algorithm. Mostly we are not aware of this process, though. We do not spend hours processing all the data or actively running complex algorithms in our mind. We just "know" what path to take.

Humans are able to process large amounts of data and make a decision based upon them in split-seconds. Looking at a simple map with a marked starting and ending point we intuitively know the fastest way from start to finish. Admittedly in this regard one might say that nowadays computers can make this decision as fast as humans. Our real superiority over computers becomes obvious, though, when being confronted with a dynamic environment. When facts change, we can adjust our perception quickly and adapt to the new circumstances.

A computer has not got this luxury. Even though great progress has been made and by now most computers are better at static pathfinding than humans, the challenge of finding paths in dynamic environments remains.

Since high interactivity in computer games often leads to highly dynamic games, they are an

[1] AI: Artificial Intelligence

excellent use case for pathfinding algorithms in dynamic environments. One example is the formerly developed space themed 3-dimensional game "Lost in Space VR", which was the initial inspiration for this bachelor's thesis. In "Lost in Space VR" the player has to safely fly a UFO through an asteroid field, by avoiding or shooting asteroids. During the development of this game the idea was to implement enemies that would try to catch the player, whilst avoiding asteroids on their path. For this, they would need to use a navigation system capable of finding the shortest path from their current position to the player. Since, throughout the whole game, the player and the asteroids are moving constantly, this pathfinding system would have to perform well in a highly dynamic 3-dimensional environment. At the time "Lost in Space VR" was developed, time constraints did not allow for developing a custom solution for the enemies' navigation system. Also, no off-the-shelf solution for a task with these specific requirements was available. Therefore, no enemy system was implemented and the task of finding a suitable pathfinding algorithm for dynamic 3-dimensional game environments was postponed until this bachelor's thesis.

1.2 Thesis goal

This bachelor's thesis will expand on the topic of pathfinding, especially in dynamic environments, by implementing a suitable pathfinding and navigation system in a dynamic 3-dimensional game environment.

To find the most suitable pathfinding algorithm for this task, a selection of algorithms will be introduced, implemented, and their results compared. Primarily an abstract game environment will be used for implementation and testing, so the complete game environment can be controlled. The ultimate goal, though, is for results to be tested in a space-themed game setting similar to "Lost in Space VR". Furthermore, close attention will be paid to creating results transferrable to other game settings.

Please note that over the course of this bachelor's thesis only a selection of pathfinding algorithms will be addressed. To address all possible pathfinding algorithms, would go beyond the scope of this thesis. Moreover, previously developed game mechanics and assets will be used and there is no focus on extending or balancing gameplay.

1.3 Structure

The aim of this thesis is to identify which algorithm is most suitable to be implemented in the highly dynamic game environment of "Lost in Space". For this, multiple pathfinding algorithm will be implemented and compared.

After having laid out the motivation of this thesis, in chapter two (2) the game "Lost in Space" is introduced (2.1). Also, the pathfinding algorithms A*, ARA*, Theta*, D*, D* Lite, Field D*, and AD* are explained on a theoretical level (2.2). Chapter two concludes by giving an overview of already existent and commonly used pathfinding systems in games and especially those available for the game engine Unity (2.3).

The thesis then continues by drafting a concept for the project execution in chapter three (3). Here also the abstract test environment, which will be used for implementation and testing of the algorithms A*, ARA*, Theta*, D* Lite, and AD*, is introduced (3.1), as well as the setup of the search graph (3.2).

Chapter four (4) starts by describing the implementation of the game "Lost in Space" (4.1.1) and the setup of the abstract test environment (4.1.2), followed by the development of the cell-grid and waypoint-based search graph (4.2). Once the search graph options are set up, the implementation of the pathfinding algorithms is presented (4.3). First overall implementation decisions are discussed, including the use of a background thread, the design of the open, closed, and incons list, and the chosen focussing heuristic. Afterwards the implementation process of each algorithm individually is briefly discussed. Chapter four concludes by describing the overall process of the pathfinding system adapting to changes in a dynamic game environment (4.4). This includes the update of the search graph, the pathfinding algorithm managing and adapting to changes in the search graph, determining new start and goal nodes, and moving the AI through the dynamic game environment.

Lastly all algorithms are tested and compared in chapter five (5). Comparison is done in two steps, mainly comparing each algorithms path length, number of expanded nodes, time required for computation, and the success rate of the AI reaching its destination.

Chapter six concludes this thesis (6). It gives a summary of the thesis (6.1), draws conclusions from experiences made throughout the implementation process and by comparing the algorithms (6.2), names remaining issues and flaws (6.3), and gives a look ahead of possible optimizations for the pathfinding system (6.4).

2 Foundation

The following chapter will start by introducing the game "Lost in Space", which presents a highly dynamic 3-dimensional game environment and thereby is an interesting use-case to test the results of this bachelor's thesis. It will then present a selection of A*-based and D*-based pathfinding algorithms and proceed to give an overview of current standards in pathfinding methods used in games. The chapter will conclude with the introduction of a couple of tools used to implement pathfinding in the Unity game engine.

2.1 Lost in Space

The game used for testing the results of this bachelor's thesis will be called "Lost in Space" and is based on the formerly developed project "Lost in Space VR". As the name suggests, "Lost in Space VR" was a VR[2] -game developed by a group of students, including the author of this bachelor's thesis, as part of the course "Virtual Systems" at the University of Applied Sciences Hamburg. The goal of this game was to make the players feel as if they were flying. To achieve this, they were standing on a custom-made balancing board functioning as a controller to steer a UFO through an asteroid field. Additionally, the player could choose between two weapons to destroy asteroids. A gameplay trailer of the game can be found here: `https://www.youtube.com/watch?time_continue=3&v=uhm9h2Gmv98`

Since the focus during the development of "Lost in Space VR" was set on implementing the main gameplay aspects, such as flying and shooting, as well as the VR-application, time constraints did not allow for implementing an enemy system. Implementing a basic enemy in this game would be an interesting use-case for testing the results of this bachelor's thesis, though. The enemies' AI would have the task to try to catch the player, whilst avoiding collision with asteroids. Thereby conclusions drawn from this bachelor's thesis can be tested in a realistic game environment.

Since "Lost in Space VR" requires specific hardware such as VR-glasses and a custom controller, a new version of the game will be developed, called "Lost in Space". It will differ from the previous version in the way, that it will no longer be in VR. Thus, steering the player will be done via keyboard or controller input. Due to the previous use of a custom controller the player was limited to steering left and right and its UFO moving on an orbit around the centre of the field. To simulate the sideward movement, the UFO moves in the according direction on the orbit. With the orbit being wide enough, this feels like moving to the left and right. To simulate the forward movement of the player's UFO, the asteroids move towards the centre of the game

[2] VR: Virtual Reality

field. If enemies had been implemented in this version, they would have started at a certain distance to the player and moved towards them.

Figure 1 indicates the movement of all game components with arrows. Due to the player only moving horizontally on the given orbit, no vertical movement of any game components was required, limiting all movements to two dimensions.

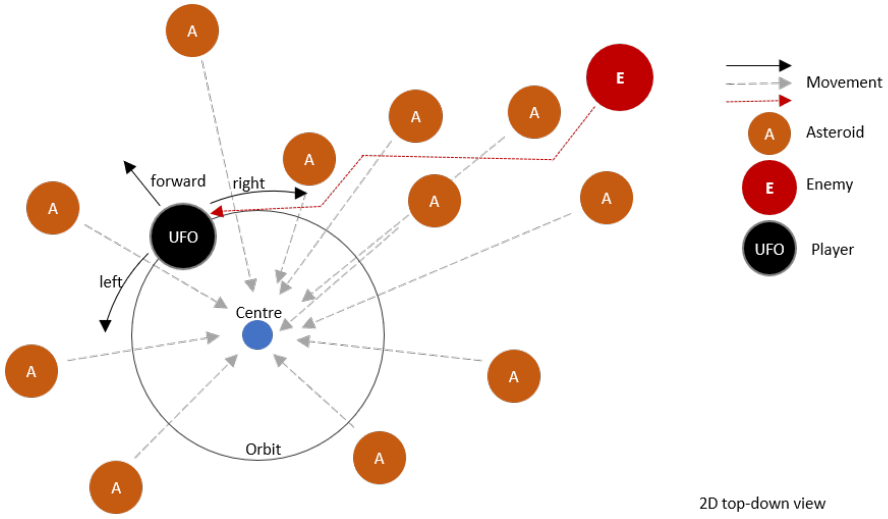


Figure 1: Lost in Space VR 2D top-down view

Removing the constraints of VR and the custom controller, in “Lost in Space” players can have a lot more freedom in their movement. In addition to steering left and right, they can now also steer up and down. This results in the player moving freely in all three dimensions through the game field, no longer constrained to an orbit. This additional freedom does not only affect the player’s movement, but also all other game components’, especially the asteroids’. Therefore, their concept of initiation and movement has to be reconsidered and changed accordingly.

Most importantly, the asteroids’ movement towards the centre of the game field to simulate the player’s forward movement is no longer required. Therefore, they can now move in seemingly random directions throughout space. To limit the number of asteroids needed in the game scene, they can only exist within a certain radius around the player, forming a sphere around it. Enemies will spawn at the rim of this sphere and move towards the player.

Figure 2 indicates the movement of all game components using arrows. Thereby it only shows the game from a 2D top-down view. In the real game the enemy has to be able to pass asteroids from all angles, meaning not only left and right, but also above or beneath. Looking at this figure it can be seen clearly, that almost all game components are moving constantly. Therefore, the requirement for a suitable enemy pathfinding is, that it has to perform well in this highly dynamic 3-dimensional game environment.

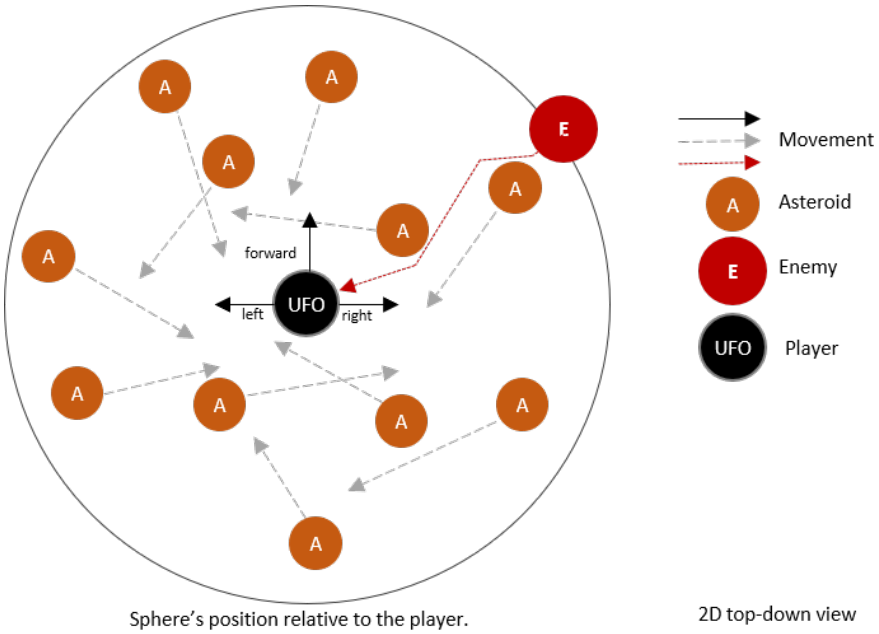


Figure 2: Lost in Space 2D top-down view

2.2 Pathfinding algorithms

In this chapter a selection of pathfinding algorithms will be presented. Since the task of this bachelor's thesis is the implementation of a suitable pathfinding system in a dynamic computer game, these algorithms were chosen for a few important characteristics, such as being efficient, being able to react to dynamic environments, or not restricting movement to grid edges.

Throughout this bachelor's thesis following notation will be used:

n:	"node", state
n_{start}:	start node
n_{goal}:	goal node
$c(n_1, n_2)$:	arc cost of moving from n_1 to n_2
$b(n_2) = n_1$:	backpointer of node n_2 is node n_1; n_1 is parent of n_2

2.2.1 A*

The A* (pronounced "A star") search algorithm was first published in 1968 [HNR68]. It was developed by computer scientists Peter Hart, Nils Nilsson and Bertram Raphael at the Stanford Research Institute, now SRI International, as part of their work on "Shakey the Robot" [Com19].

A* can be seen as a generalization and extension of the Dijkstra algorithm, and belongs to the class of the informed search-algorithms. It uses estimation functions, called heuristics, to focus its search and thereby enhance its performance. A* is widely used as a pathfinding algorithm to find the optimal path between two points, called nodes. [Mar15]

The A* algorithm uses the evaluation function $\boldsymbol{f(n)}$ to estimate the total cost of the path going through n to the goal.

$$\boldsymbol{f(n) = g(n) + h(n)}$$

Hereby $\boldsymbol{g(n)}$ is the actual cost of the path from the start node to n, and $\boldsymbol{h(n)}$ is the heuristic function, which means it is the estimated lowest cost from n to the goal node [Wel12]. A heuristic is called **admissible,** if it never overestimates the actual path cost. Additionally, a heuristic is **consistent** (or monotone), if its estimated cost to the goal is always less or equal to the cost of travelling to any neighbouring node plus the neighbouring nodes' estimated cost to the goal (see Figure 3) [Mar15].

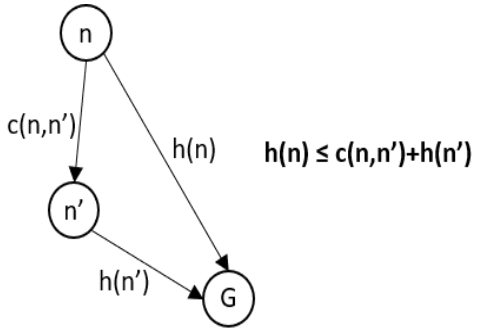


Figure 3: Consistent heuristic [Wel12]

An important concept of the A* algorithm is the use of an open list and a closed list. The open list, in some literature also called "fringe" [Wik18b] , "fringe list" [Mne18] , or "open set" [Wik18a], contains a list of nodes that are children of already expanded nodes, but have not been expanded themselves, yet. The closed list, also known as "closed set" [Wik18b], contains all nodes that have already been processed [Cho10].

Meeting some additional criteria, the A* algorithm is proven to be **complete**, **optimal**, and **optimal efficient**.

Complete: If a solution exists, it will be found [Mar15].

Optimal: If the heuristic is consistent, the solution found will be the optimal one [Mar15]. In cases where no closed list is used, an admissible heuristic is sufficient for optimality [Wel12].

Optimal efficient: Using the same heuristic, no other algorithm can find the solution expanding less nodes [Mar15].

The time complexity of A* is strongly dependent on the used heuristic. In the worst case it is $\mathbf{O}(\boldsymbol{b}^{\boldsymbol{d}})$, where $\boldsymbol{b}$ is the branching factor and $\boldsymbol{d}$ the depth of the solution. [Mar15, Wik18b]

The following graphs demonstrate the structure of the A* algorithm.

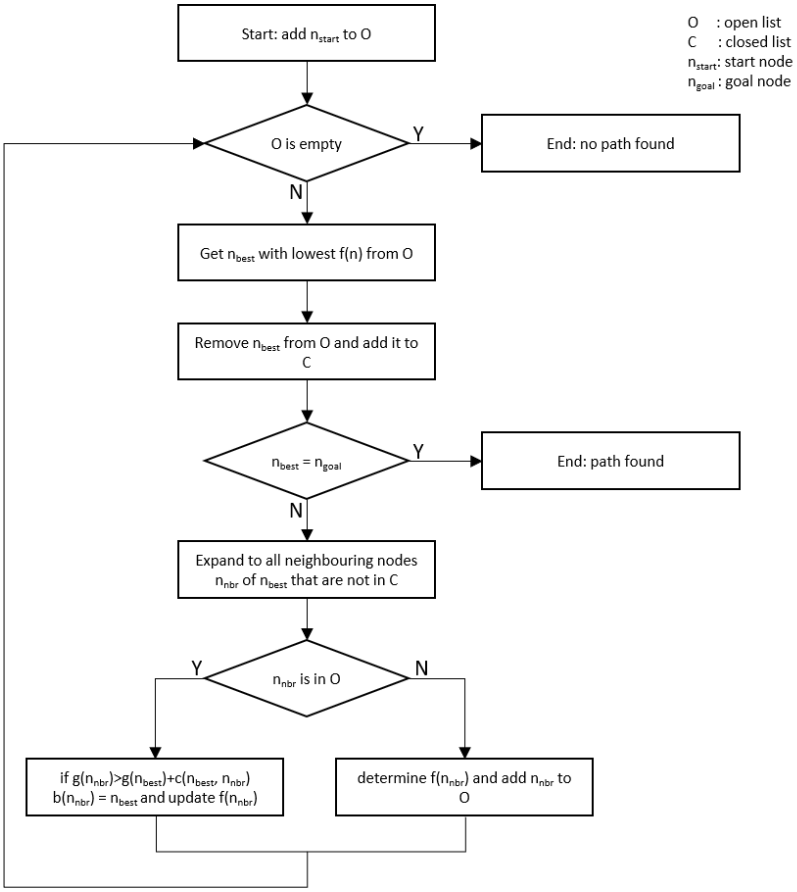


Figure 4: Structure of A* algorithm [Cho10]

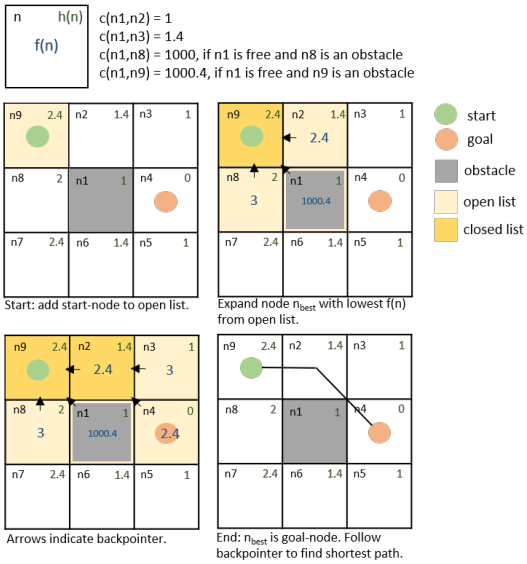


Figure 5: Example of A* algorithm [Cho10, Cod12]

2.2.2 ARA*

The ARA* (Anytime Repairing A*) algorithm was presented in 2003 by Maxim Likhachev, Geoff Gordon and Sebastian Thrun [LGT03a]. The authors state:

> "[ARA*] is an efficient anytime heuristic search that [...] runs A* with inflated heuristic in succession [and] reuses search efforts from previous executions in such a way that the sub-optimality bounds are still satisfied." [LGT03b, p. 2]

This statement names a number of characteristics.
For one it says that ARA* is an **anytime algorithm**. Anytime algorithms are algorithms, which can return valid solutions, even if they are interrupted before ending. In general, they quickly compute a sub-optimal solution and improve its accuracy over time. Thereby they provide a trade-off between computation time and the quality of the solution. [DB88]

The other important characteristic is the use of the A* algorithms with an **inflated heuristic**. As described in 2.2.1 A* is optimal given it uses a consistent heuristic $h(n)$. Inflating this heuristic by a factor $\boldsymbol{\varepsilon > 1}$ mostly helps directing the search and thereby saves time by expanding fewer nodes. [Lik10] The algorithm now uses:

$$\boldsymbol{f(n) = g(n) + \varepsilon * h(n)}$$

Using an inflated heuristic, the found solution is not guaranteed to be optimal, but might be sub-optimal. The **sub-optimality** however is bound by the factor ε. This means the cost of the sub-optimal solution is less or equal to the cost of the optimal solution. [LGT03b]

The third important characteristic mentioned is the **reuse of search efforts**. In other words, instead of running the algorithm from scratch for each execution cycle, it stores previously gained information. ARA* does this by using a third list, called incons (original notation: "INCONS" [LGT03b]) in addition to the open list and closed list, known from classical A*. Nodes are called **locally inconsistent** after their g-value has been decreased and before they are expanded the next time. In consequence all nodes in the open list are locally inconsistent. If a node's g-value is lowered, it is put into the open list, until it is expanded and thereafter put into the closed list. Since every node can only be expanded once per execution cycle, the open list only contains all locally inconsistent nodes, that have not been expanded, yet. Nodes, that become locally inconsistent, but have already been expanded previously, are put into the incons list. [LGT03b]

The sub-optimality bound ε' is calculated as the minimum between ε and the ratio between $f(n_{\text{start}})$ and the lowest non-weighted f-value of all locally inconsistent nodes [LGT03b]:

$$\boldsymbol{\varepsilon' = \min(\varepsilon, \frac{f(\text{goal})}{\min_{n \in \text{O} \cup \text{I}}(g(n) + h(n))})}$$

The following graph illustrates the structure of the ARA* pathfinding algorithm.

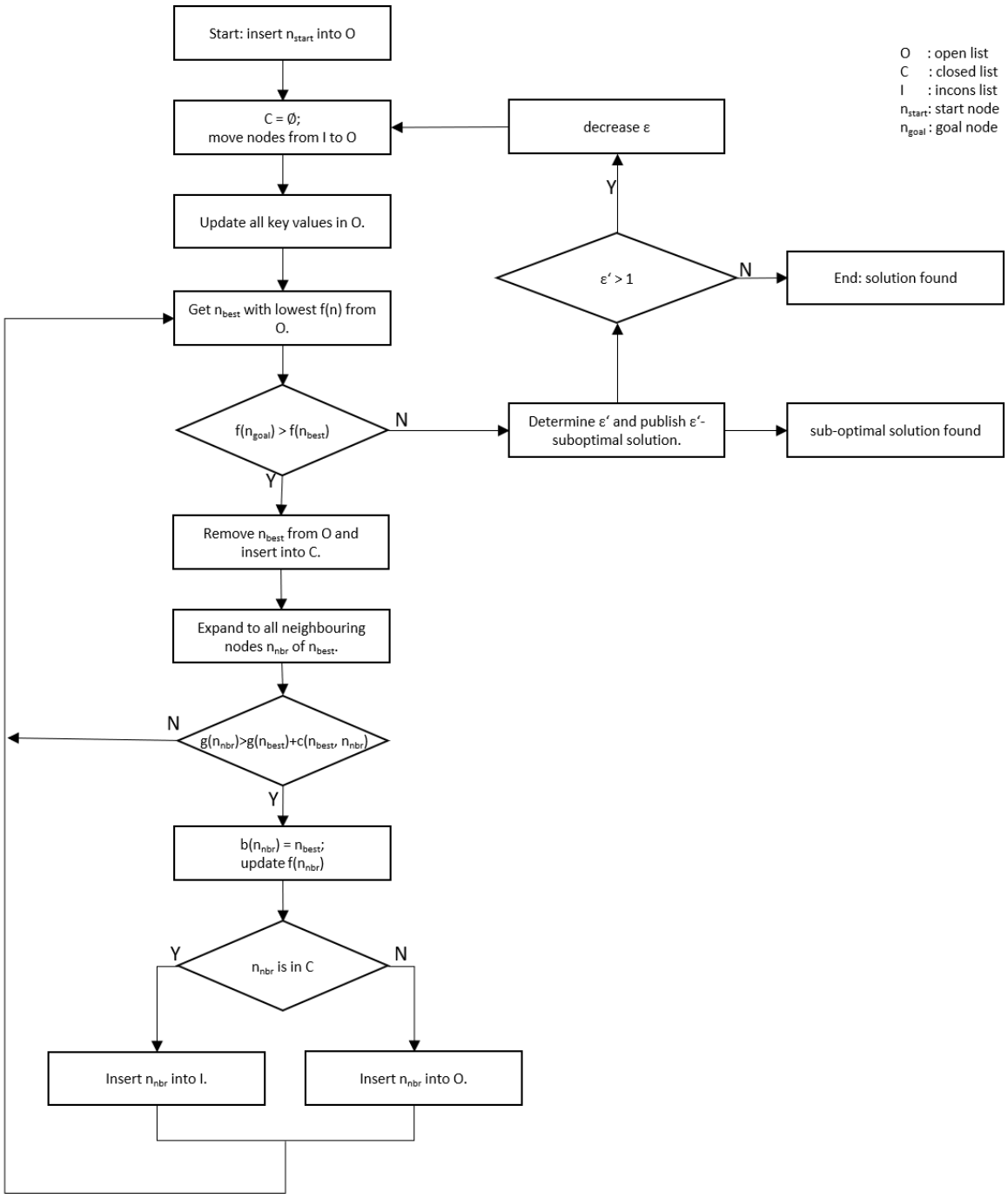


Figure 6: Structure of ARA* [LGT03b]

2.2.3 Theta*

Theta* is an any-angle path-planning algorithm, introduced 2007 by Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner [DN+07]. The authors refer to Theta* as a collective of **Basic Theta*** and **Angle-Propagation Theta***.

As the name suggests any-angle pathfinding algorithms allow their paths to head in any angle. For the Theta* algorithms this means, that even though they are based on A* and therefore propagate information along grid edges, unlike A*, they do not constrain their paths to those grid edges. Allowing the path to deviate from the grid allows for finding shorter and more natural looking paths (see Figure 7). [DN+10]

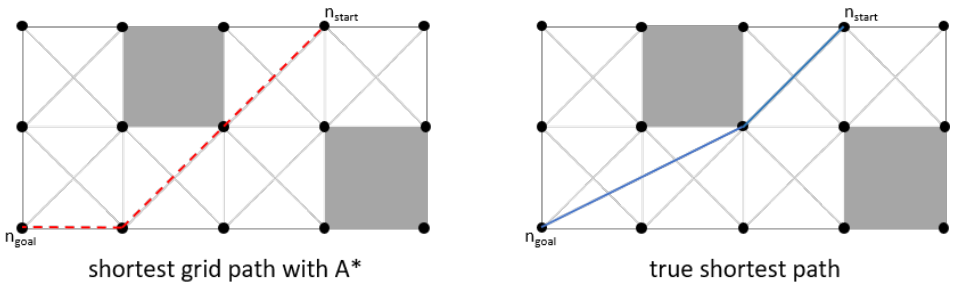


Figure 7: A* path vs any-angle path [DN+10]

Basic Theta* is almost identical to A* and only differs in updating the g-value and parent of unexpanded neighbouring nodes n' of n, during expansion of n. A* only considers the path along the grid edges from n_{start} to n, and from n to n' (see Figure 8 path, path 1) [DN+07]:

$$\boldsymbol{g(n') = g(n) + c(n, n')}$$

Basic Theta* additionally considers the path from n_{start} to the parent of n, parent(n), and from the parent of n to n' in a straight line (see Figure 8, path 2) [DN+07]:

$$\boldsymbol{g(n') = g(\text{parent}(n) + c(\text{parent}(n), n')}$$

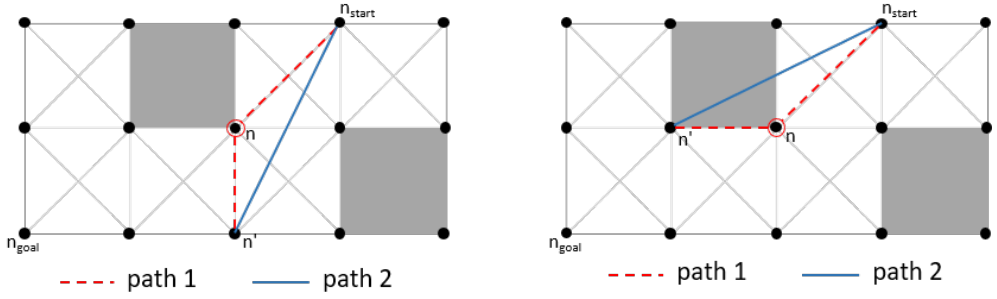


Figure 8: Paths considered by Basic Theta* [DN+10]

Path 2 is always shorter than path 1 and will be chosen from Theta* if it is not blocked. In other words, when there is a line-of-sight between n' and parent(n) (Figure 8, left). Otherwise path 1 will be chosen (Figure 8, right).

The following graphic shows an example trace of Basic Theta*. For simplicity non-relevant extractions of nodes are not shown. The red circles indicate which node is currently being extracted.

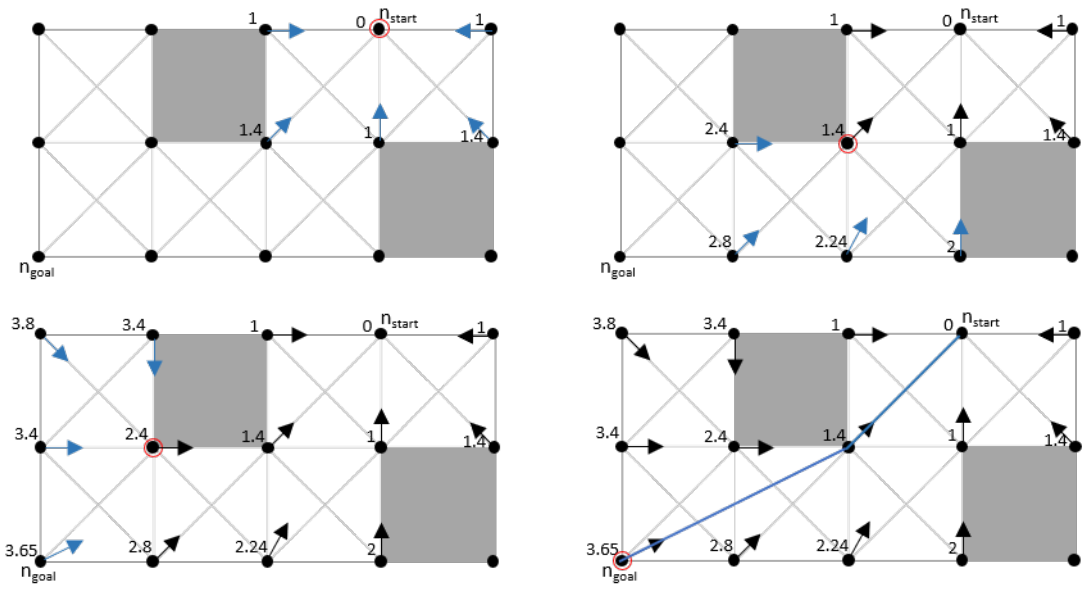


Figure 9: Example of Basic Theta* [DN+10]

Basic Theta* is correct and complete, but not optimal. This means that if there is an unblocked path it will be found and any path that is found is unblocked, but a found solution is not guaranteed to be the shortest one [DN+10]. Experiments have shown, though, that Basic Theta* finds the shortest path in a large percentage of cases. [Nas10]

Angle-Propagation Theta* (AP Theta*) differs from Basic Theta* only in the fact, that it propagates angle ranges and thereby determines whether two nodes have a line-of-sight, without running a line-of-sight check. In doing so AP Theta* improves the worst-case complexity of Basic Theta* significantly. Experiments have shown, though, that overall AP Theta* is slower and finds slightly longer paths than Basic Theta* [DN+10]. Therefore, it will not be considered in this bachelor's thesis. For further information on AP Theta* see [DN+10] chapter six.

2.2.4 D*

The D* pathfinding algorithm was first published in 1993 by Anthony Stentz. It was originally developed for the field of robotics and is a complete and optimal pathfinding algorithm, which is capable of finding paths in unknown, partially known, or dynamic environments. It is able to do so because of its capability to react to changes in arc path costs $c(n, n')$. Reacting to those changes, it updates its found path by reusing previously established data. D* closely resembles A*, except that it is dynamic - hence the name D*. [Ste93]

Unlike the A*-based algorithms presented in previous chapters of this thesis, D* is directed backwards. In other words, it starts its information propagation at the goal node and stops when the position of the robot, the start node, is reached or the open list is empty. Doing so, D* uses the path cost function $\boldsymbol{h(n)}$. Note, that h is not a heuristic, but the sum of path costs from n to the goal, comparable to $g(n)$ in the A* algorithm. [Ste94]

Like A*, D* utilises an open list and a closed list. To identify which list each node is in, they are each associated with a tag $\boldsymbol{t(n)}$, which can take the values NEW, if n has never been on the open list, OPEN, or CLOSED. Nodes on the open list, $t(n) = \text{OPEN}$, are sorted using their key function values $\boldsymbol{k(n)}$. For each node n in the open list the key value function is defined as the minimum of $h(n)$ before modification, and the smallest value of $h(n)$, since n was placed on the open list. In practice this means $k(n)$ is determined as followed when n is being placed or replaced in the open list [Ste94]:

$$\boldsymbol{k(n)} = \begin{cases} \boldsymbol{h_{\text{new}}}, & \textbf{if } \boldsymbol{t(n)} = \textbf{NEW} \\ \boldsymbol{\min(k(n), h_{\text{new}})}, & \textbf{if } \boldsymbol{t(n)} = \textbf{OPEN} \\ \boldsymbol{\min(h(n), h_{\text{new}})}, & \textbf{if } \boldsymbol{t(n)} = \textbf{CLOSED} \end{cases}$$

This key function is an important aspect of the D* algorithm. First of all, each node in the open list is classified by its key function as either a **raise state**, given $\boldsymbol{k(n) < h(n)}$, or a **lower state**, given $\boldsymbol{k(n) = h(n)}$. **Raise states** propagate information about path cost increases, **lower states** propagate information about path cost decreases. Moreover, the parameter $\boldsymbol{k_{\text{min}}}$ is defined as the lowest key function value of all nodes in the open list:

$$\boldsymbol{k_{\text{min}} = \min(k(n)),\ \textbf{with } t(n) = \textbf{OPEN}}$$

Path costs lower or equal to k_{min} are optimal, whereas path costs greater than k_{min} are not guaranteed to be optimal. Lastly k_{old} is defined as k_{min}, prior to the last removal of a node from the open list [Ste94]

As defined by the author the most important functions of the D* algorithms are: PROCESS-STATE and MODIFY-COST [Ste94] . PROCESS-STATE determines the optimal path costs to the goal and MODIFY-COST reacts to changes in the arc costs $c(n, n')$ by updating the arc cost function and placing all affected nodes into the open list. [Ste93]

The following graphs shows the structure of the D* algorithm. Figure 10 illustrates the basic structure of the algorithm and Figure 11 shows a more detailed view of the PROCESS-STATE function. For an example of the D* algorithms have a look at [Cho10] page 38.

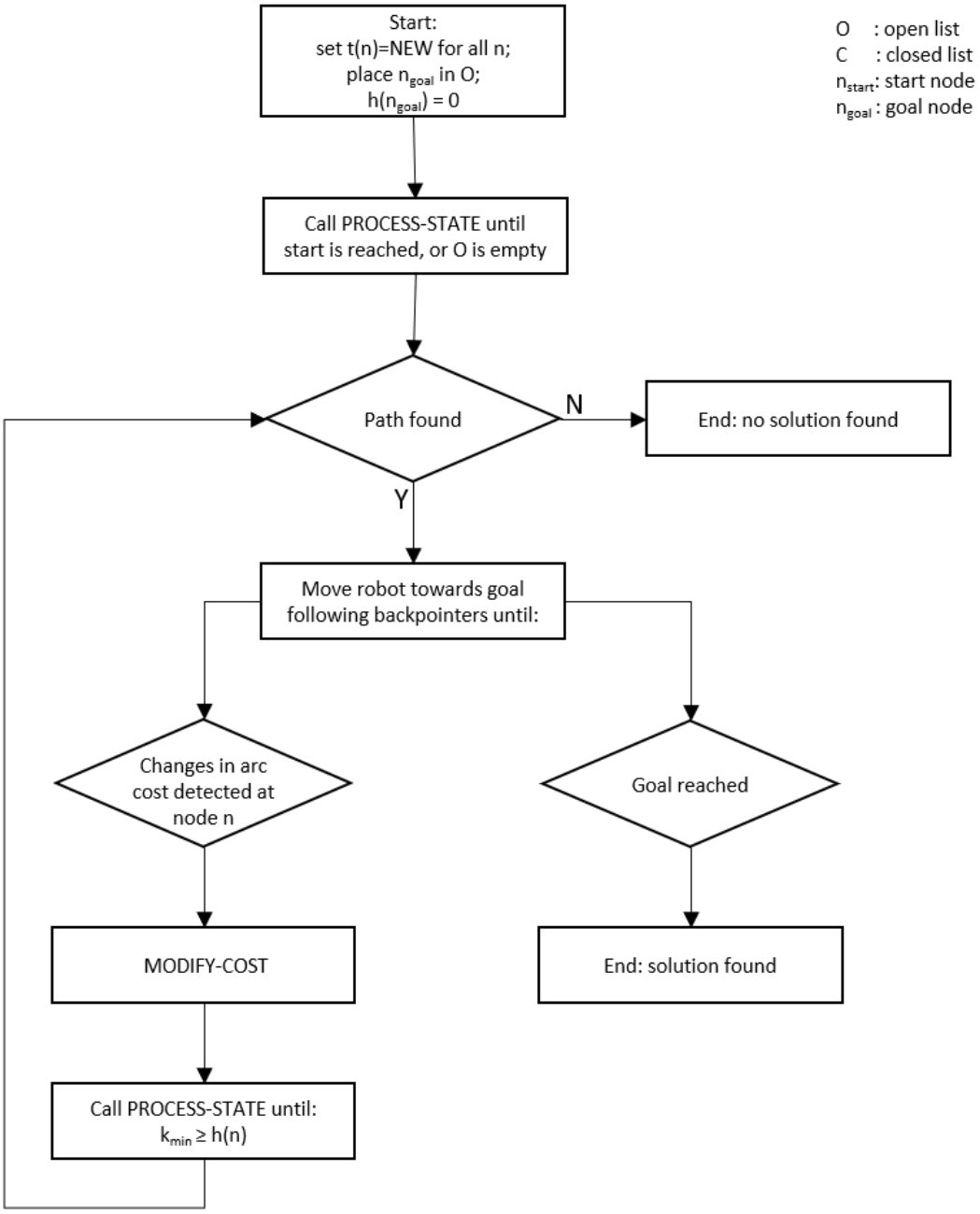


Figure 10: Basic structure of D* algorithm [Ste94]

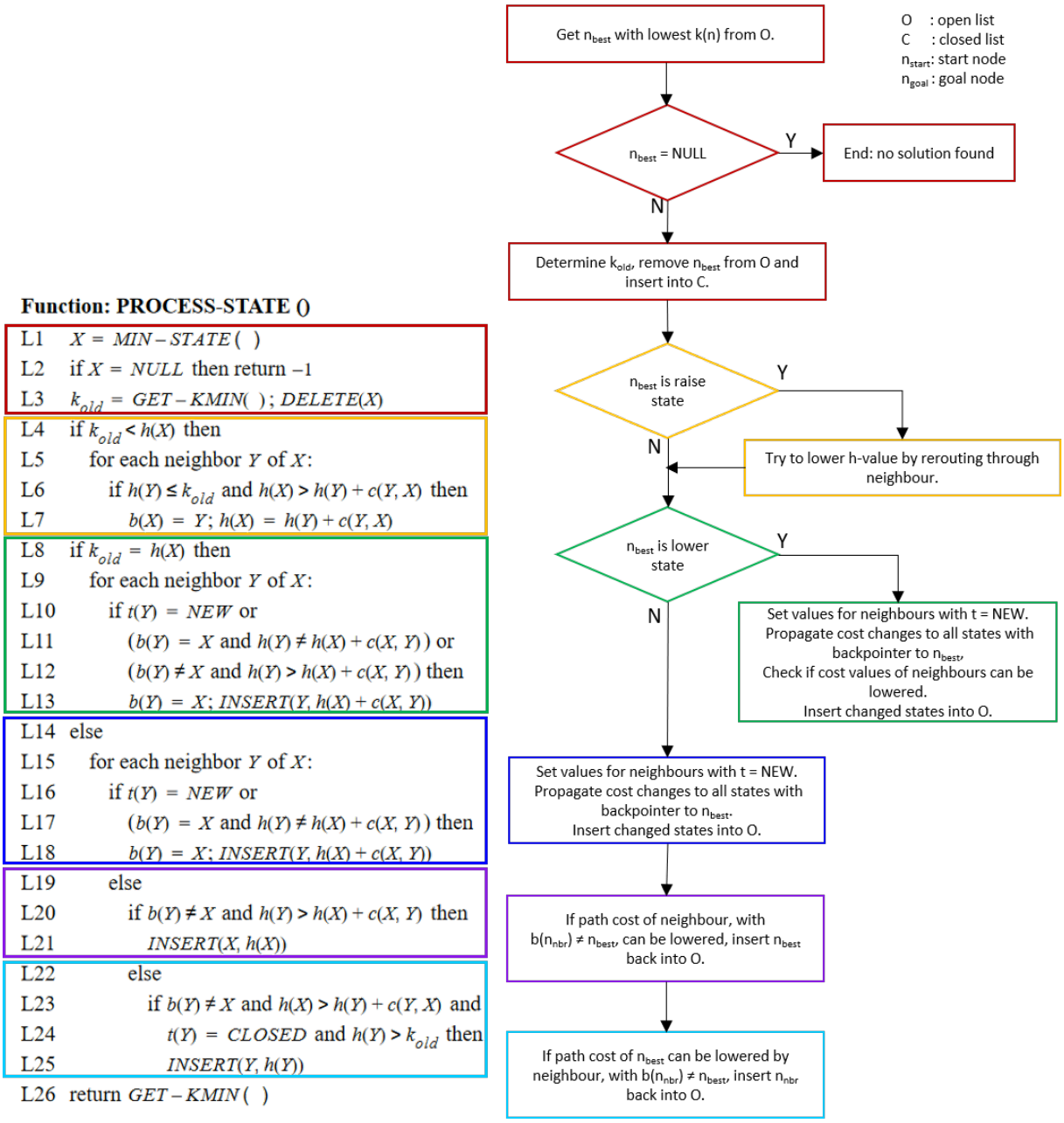


Figure 11: Structure of PROCESS-STATE function [Ste94]

2.2.5 D* Lite

D* Lite is an incremental heuristic search algorithm first introduced by Sven Koenig and Maxim Likhachev in 2002 [KL02c]. Both of those names should sound familiar. Sven Koenig is a co-developer of the Theta* algorithm, and Maxim Likachev was involved in the development of the ARA* algorithm.

Since D* Lite is an incremental heuristic search, it not only uses heuristics to focus its search, but it also reuses information gained from previous searches to find a solution faster. D* Lite was built upon Lifelong Planning A* (LPA*) [KL02a], a pathfinding algorithm developed by Koenig and Likhachev earlier. Moreover, D* Lite uses the same navigation strategy as D* but is generally simpler and shorter than D*. Therefore, it is easily extendable for implementing optimizations, such as inadmissible heuristics. [KL02c]

Before the D* Lite algorithm can be explained, the terminology of **successor** and **predecessor** has to be clarified. Given a directed edge from n to n', it is said that n' is the **successor** of n and n is the **predecessor** of n'. The cost of this edge is denoted with $c(n, n')$. [Cho10]

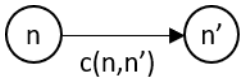


Figure 12: Concept of successor and predecessor [Cho10]

D* Lite contains two cost variables for each node n. One is the g-value $\boldsymbol{g(n)}$, which is an estimated path cost from the goal to n. Estimated in the sense, that the $g(n)$ might change when a shorter path is found. The other one is the rhs-value $\boldsymbol{rhs(n)}$, which is a one-step lookahead of the path cost and is based on the g-values of its successors. Therefore, it is potentially better informed than $g(n)$. [KL02c]

$$\boldsymbol{rhs(n) = \min_{n' \in Succ(n)}(g(n') + c(n, n')}\ \text{[Cho10]}$$

The g- and rhs-values determine whether a node is consistent or not. If $\boldsymbol{g(n) = rhs(n)}$, then a node is **consistent**, otherwise it is said to be **inconsistent**. Furthermore, if $\boldsymbol{g(n) > rhs(n)}$, it is said n is **over-consistent**, whereas if $\boldsymbol{g(n) < rhs(n)}$ then n is **under-consistent**. Unlike most other pathfinding algorithms, D* Lite utilizes an open list, but not a closed list. The open list contains all inconsistent nodes. The order, in which the nodes are sorted in the open list, is determined by their key value.

The key value $\boldsymbol{k(n)}$ of a node is determined by using its g- and rhs-value, as well as a focusing heuristic, and consists of two parts, similar to a primary and secondary key. If there is a tie between two nodes' primary keys, their secondary keys will be used as a tie breaker. [Cho10]

$$\boldsymbol{k(n) = [\min(g(n), rhs(n)) + h(n_{start}, n) + k_m; \min(g(n), rhs(n))]}\text{[KL02c]}$$

As can be seen in the formula, D* Lite uses the heuristic $h(n_{\text{start}}, n')$ and a factor k_m to calculate the key value. The use of $\boldsymbol{h(n_{start}, n')}$is comparable to the use of a heuristic $h(n)$ to calculate the f-value in A*. Hereby the heuristic $h(n, n')$ has to be non-negative and backwards consistent.

The factor $\boldsymbol{k_m}$is needed to avoid having to reorder the open list every time the start node changes, e.g. due to a robot detecting changed arc costs after movement. Updating the path cost, D* Lite considers the current robot location as the start node of the pathfinding algorithm. Assuming the robot moved from a node n to node n', where it detects changed arc cost, $h(n, n')$ is added to k_m, which in turn is then considered when calculating new key values. This is necessary, since after the movement the first key elements (primary keys) of nodes currently on the open list, may have decreased by $h(n, n')$ at the most. Likewise, when new key values are calculated, their first elements would be $h(n, n')$ too low relative to key-values already in the open list. This is balanced by the addition of k_m to all new key values. [KL02c]

The following graph shows a structure of the D* Lite pathfinding algorithm. It does not include the termination condition for the case that no path is found: If after a search $\boldsymbol{g(n_{start}) = \infty}$, then there is no path with finite cost between n_{start} and n_{goal}. [KL02c]

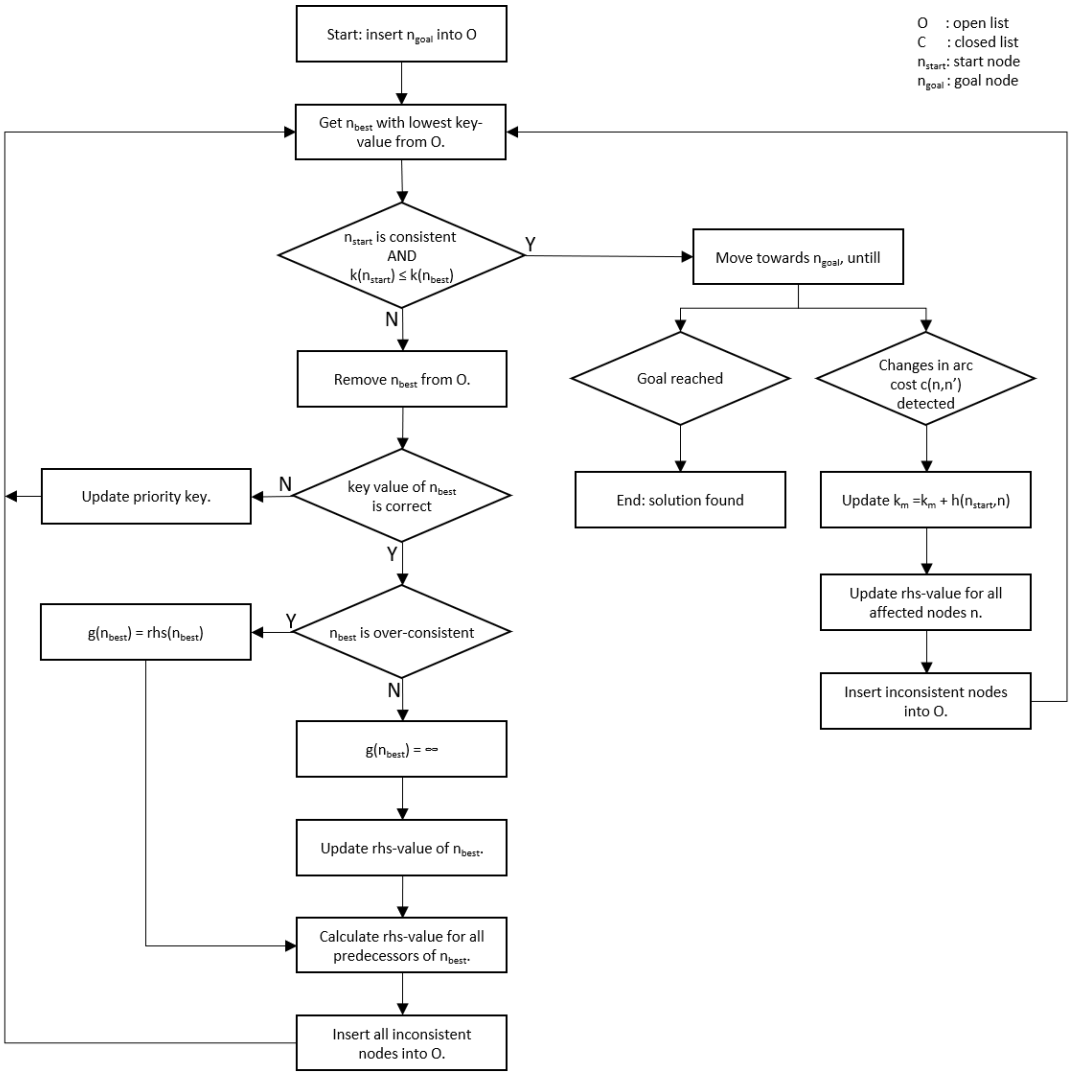


Figure 13: Structure of D* Lite [KL02c]

An example of the D* Lite algorithm can be found as Figure 54 and Figure 55 in the Appendix or at [Cho10] from page 75 onwards.

In general D* Lite terminates, is correct, and at least as efficient as D*. Furthermore, as mentioned at the beginning of this chapter, D* Lite is easily extendable for optimizations. These optimization possibilities include a modification of the termination condition, as well as an optimization when computing the rhs-values of predecessors of over-consistent nodes. For detailed information on optimization possibilities for D* Lite see [KL02b] chapter 5.1. [KL02b]

2.2.6 Field D*

Field D* is an any-angle pathfinding algorithm, which uses liner interpolation to calculate more precise estimates of path costs. It thereby is able to find smooth paths through non-uniform cost grids, not limiting movement to grid edges. This algorithm was first introduced by Dave Ferguson and Anthony Stentz (see D*) in 2005. [FS05]

As the name suggests, Field D* extends D* and D* Lite. Since the core novelty of Field D* is a new method of calculating path cots, it does not change the underlying structure of any algorithm, and therefore can be plugged into a number of other algorithms. Field D* differs from most other grid-based pathfinding algorithms only in the layout of nodes and the method of calculating the path cost value of each node. [FS05]

The first modification of D* is simple. Instead of assigning each node as the centre of a grid cell, nodes are assigned to the corners of grid cells, as seen in Figure 14. [FS05]

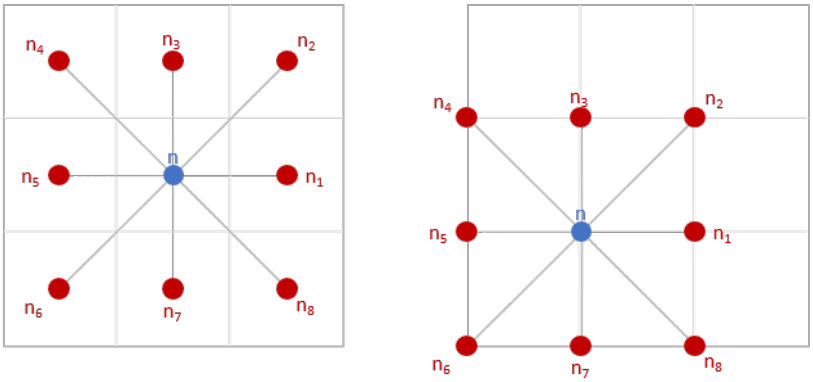


Figure 14: Layout of Nodes [FS05]

The second modification, namely the calculation estimations of path costs utilizing linear interpolation, is more complex.

For classic grid-based pathfinding algorithms $\boldsymbol{g(n)}$ is calculated as

$$\boldsymbol{g(n) = \min_{n' \in n_{nbr(n)}} (c(n, n') + g(n'))} \tag{1}$$

with $\boldsymbol{n_{nbr(n)}}$ being all neighbours of n, $\boldsymbol{c(n, n')}$ the cost of travelling from n to n' and $\boldsymbol{g(n')}$ the path cost of n' [FS05]. This, however, restricts movement to grid edges, which leads to unnatural looking movement and sub-optimal paths, as already discussed in the context of Theta*. To get the actual optimal path cost of a node n, the path must be allowed to pass through any edges connecting two adjacent neighbours of n [FS05] , for example $\overrightarrow{n_1 n_2}$ (see Figure 15).

Knowing the g-values of all possible intersection points $\boldsymbol{n_y}$ of the path with such a connecting edge, $g(n)$ could be calculated by minimising:

$$\boldsymbol{g(n) = c(n, n_y) + g(n_y)} \tag{2}$$

Since there are an infinite numbers of points n_y, it is not possible to calculate each of their g-values individually. Instead linear interpolation is used. To do so, it is assumed that every value $g(n_y)$, with n_y being a point on the edge connecting n_1 and n_2, is a linear combination of $g(n_1)$ and $g(n_2)$

$$\boldsymbol{g(n_y) = y * g(n_2) + (1 - y) * g(n_1)} \tag{3}$$

where $\boldsymbol{y}$ is the distance of n_1 to n_y (see Figure 15), assuming the use of unit cells. [FS06] Of course, using linear interpolation, the resulting path cost value is only an assumption and might not be perfect. According to the authors, though, "this linear approximation works well in practice" [FS06, p. 6]. Knowing $g(n_y)$, equation 3 can now be plugged into 2 and $g(n)$ is calculated as

$$\boldsymbol{g(n) = \min_{x,y}(bx + c\sqrt{(1 - x)^2 + y^2} + yg(n_2) + (1 - y)g(n_1))} \tag{4}$$

with $\boldsymbol{x \in [0, 1]}$ being the distance of travelling on the bottom edge before crossing through the grid cell, and $\boldsymbol{b}$ and $\boldsymbol{c}$ being cell costs as shown in Figure 15. [FS06]

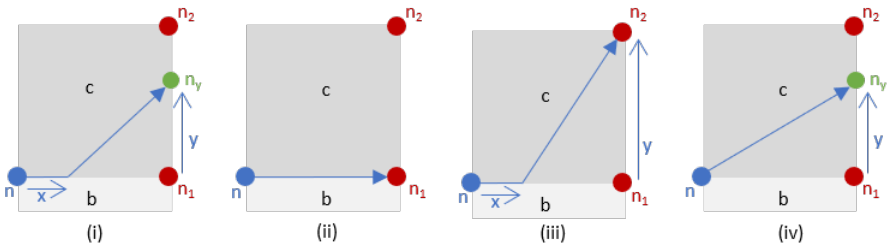


Figure 15: Shortest path of n through edge $n_1 \vec{n}_2$ [FS05]

With (x^*, y^*) being a pair of values for x and y solving the minimization of equation 4, it is proven that at least one of these values is zero or one. If it is cheapest to pass through n_1 and there is no part crossing through the centre of the cell, then $\boldsymbol{y^* = 0}$ (Figure 15 (ii)). If there is any part of the cheapest path crossing through the centre of the cell, though, then this part is as large as possible, resulting in either $\boldsymbol{x^* = 0}$ or $\boldsymbol{y^* = 1}$.Therefore, the path either travels partly along the bottom edge before crossing all the way to n_2 (Figure 15 (iii)) or it crosses directly to a point n_y on the edge between n_1 and n_2 (Figure 15 (iv)). Note, that as a conclusion Figure 15 (i) is not an optimal path. For detailed proof of this circumstance see [FS05] page 5-7. [FS06]

Which of the paths is the most optimal depends on the relative sizes of cell costs c and b, as well as the "difference of path cost between n_1 and n_2" [FS06, p. 6]. For this f will henceforth be defined as:

$$\boldsymbol{f = g(n_1) - g(n_2)}$$

For $\boldsymbol{f < 0}$ it is given that $\boldsymbol{g(n_1) < g(n_2)}$ and therefore it is cheapest to travel along the bottom edge from n to n_1 (Figure 15 (ii)). In this case $g(n)$ is not dependent on y and can simply be calculated as:

$$\boldsymbol{g(n) = \min(c, b) + g(n_1)} \tag{5}$$

If $\boldsymbol{f < b}$, it is cheapest to use none of the bottom edge (Figure 15 (iv)). In this case k is set to $\boldsymbol{k = f}$ and the path cost of n is calculated as:

$$\boldsymbol{g(n) = c\sqrt{1 + y^2} + k(1 - y) + g(n_2)} \tag{6}$$

As can be seen the path cost of n is now dependent of y. To determine the optimal y to minimize $g(n)$ the derivative of this cost (of equation 6) is taken with respect to y and set equal to zero. This results in the optimal value of y, y^*:

$$\boldsymbol{y^* = \sqrt{\frac{k^2}{c^2 - k^2}}} \tag{7}$$

If $\boldsymbol{f > b}$, it is cheapest to use some part of the bottom edge before crossing through the cell all the way to n_2 (Figure 15 (iii)). For this equations 6 and 7 are used where k is set to $\boldsymbol{k = b}$, and y^* is substituted with $\boldsymbol{y^* = 1 - x^*}$. [FS05, FS06]

In cases where $\boldsymbol{f = b}$, the cost of using some part of the bottom edge is equal to the cost of using none of the bottom edge. Which cost is used is then dependent on the implementation of the algorithm. [FS05, FS06]

The full code of the ComputeCost function as stated by the authors is shown in Figure 16. In addition to all cases named above it is also taken into account, that if $\boldsymbol{c < b}$, or respectively if $\boldsymbol{c < f}$, then the path cost of n can be calculated as

$$\boldsymbol{g(n) = \sqrt{2} + g(n_2)} \tag{8}$$

(see Figure 16 line 10 and line 15). [FS05]

ComputeCost(s, s_a, s_b)
01. if (s_a is a diagonal neighbor of s)
02. $\quad s_1 = s_b; s_2 = s_a;$
03. else
04. $\quad s_1 = s_a; s_2 = s_b;$
05. c is traversal cost of cell with corners s, s_1, s_2;
06. b is traversal cost of cell with corners s, s_1 but not s_2;
07. if $(\min(c, b) = \infty)$
08. $\quad v_s = \infty;$
09. else if $(g(s_1) \leq g(s_2))$
10. $\quad v_s = \min(c, b) + g(s_1);$
11. else
12. $\quad f = g(s_1) - g(s_2);$
13. $\quad$ if $(f \leq b)$
14. $\quad\quad$ if $(c \leq f)$
15. $\quad\quad\quad v_s = c\sqrt{2} + g(s_2);$
16. $\quad\quad$ else
17. $\quad\quad\quad y = \min(\frac{f}{\sqrt{c^2 - f^2}}, 1);$
18. $\quad\quad\quad v_s = c\sqrt{1 + y^2} + f(1 - y) + g(s_2);$
19. $\quad$ else
20. $\quad\quad$ if $(c \leq b)$
21. $\quad\quad\quad v_s = c\sqrt{2} + g(s_2);$
22. $\quad\quad$ else
23. $\quad\quad\quad x = 1 - \min(\frac{b}{\sqrt{c^2 - b^2}}, 1);$
24. $\quad\quad\quad v_s = c\sqrt{1 + (1 - x)^2} + bx + g(s_2);$
25. return v_s;

Figure 16: ComputeCost-function of Field D* [FS05]

As already mentioned, this method of calculating the path cost can be used with a number of other algorithms. In their paper Ferguson and Stentz base Field D* on D* Lite, therefore the same is applied in this bachelor's thesis. [FS05]

In this case the ComputeCost function is used to calculate nodes' rhs-values. Field D* thereby finds paths, which are optimal with respect to the used linear interpolation assumption. Once the optimal path cost has been determined, the path can be extracted. This is done by iteratively calculating the next point n_y to move to, using the same linear interpolation principles explained above. Due to the linear interpolation only being an assumption, some precautionary steps should be taken whilst extracting the path, for example taking a one-step lookahead into account. [FS05]

A 3-dimensional version of this algorithm has been developed by Joseph Carsten, Dave Ferguson, and Anthony Stentz in 2006 [CFS06].

2.2.7 AD*

Anytime Dynamic A* (AD*) is a heuristic-based anytime replanning algorithm, first presented in 2005 by Maxim Likhachev (see ARA* and D* Lite), Dave Ferguson (see Field D*), Geoff Gordon (see ARA*), Anthony Stentz (see D*) and Sebastian Thrun (see ARA*). [LF+05]

As the name suggests AD* is a pathfinding algorithm that combines the advantages of anytime algorithms, used to compute sub-optimal solutions in short times, with those of replanning algorithms, used for dynamic environments. To be precise, AD* combines a backwards version of the anytime algorithm ARA* with the replanning algorithm D* Lite. Thus, it is able to compute a sub-optimal solution quickly, which gets improved over time, and is also able to react to changes in the dynamic environment. [LF+05]

ARA* is explained in chapter 2.2.2 of this bachelor's thesis. It is a pathfinding algorithm, that uses a heuristic, inflated by a factor $\varepsilon > 1$, to quickly find a ε-sub-optimal solution. ARA* reuses information from previously run searches to increase its efficiency, but it does not consider dynamic environments. If any arc costs are changed, it has to recalculate everything from scratch. AD* uses a backwards version of ARA*. This means the heuristics used now estimates the distance from a given node to the start n_{start} instead of the goal node n_{goal}. Additionally, at the beginning n_{goal} has to be inserted into the open list instead of n_{start}.

The replanning algorithm D* Lite is explained in chapter 2.2.5 of this thesis. It is a pathfinding algorithm, designed to perform well in partly known or dynamic environments. Detecting changes in any arc costs, it replans its found path reusing information from previous searches. It thereby is quite efficient in its replanning, but always calculates until the optimal solution is found.

In real-world applications, as well as in game applications, it is often necessary to work with only partially-known or dynamic environments, and still get results very quickly. AD* is an interesting option in this regard, since it breaches the gap between these two requirements. [LF+05]

As already mentioned, AD* combines ARA* with D* Lite. AD* copies the use of an open, closed, and incons list from ARA*, but stores and calculates the g- and rhs-value of each node like D* Lite. Nodes get inserted into the open list when they become inconsistent, $\boldsymbol{g(n) \neq rhs(n)}$, the first time and are removed from the open list once they get expanded. All of the already expanded nodes are stored in the closed list. Nodes that become inconsistent but are already

in the closed list are inserted into the incons list. The priority of nodes in the open list is based on their key value. AD* determines the key value similar to D* Lite but differentiates between under-consistent and over-consistent nodes. For under-consistent nodes to propagate their changed cost to affected neighbours, their key has to be calculated using a non-inflated heuristic. [LF+05]

$$\boldsymbol{k(n)} = \begin{cases} \boldsymbol{[rhs(n) + \varepsilon * h(n_{start}, n); rhs(n)],} & \textbf{if } \boldsymbol{n} \textbf{ is over-consistent} \\ \boldsymbol{[g(n) + h(n_{start}, n); g(n)],} & \textbf{if } \boldsymbol{n} \textbf{ is under-consisten} \end{cases}$$

Figure 18 shows a simplified structure of the AD* algorithm. The assumption is made that the algorithm is used for a robot and that the robot starts moving, following the shortest path towards the goal as soon as a sub-optimal solution is found. The robot follows the sub-optimal solution and thereby updates n_{start} of the algorithm, whilst the algorithm keeps optimizing by decreasing ε. Since the key values of nodes in the open list are updated each time ε is decreased, it is not necessary to take the movement into account during the calculation of the key, like in D* Lite. Whenever arc cost changes are detected, the dynamic part of AD* reacts to those. If the detected changes are too significant, it is suggested to increase ε or replan from scratch. The threshold of "too significant" has to be determined for every application individually. [LF+05]

The following graph shows the UpdateState function of AD*. It is responsible for updating the *rhs*-values of nodes as well as sorting them into the correct lists. Orange framed objects are part of the algorithm similar to ARA*. Blue framed ones are similar to D* Lite.

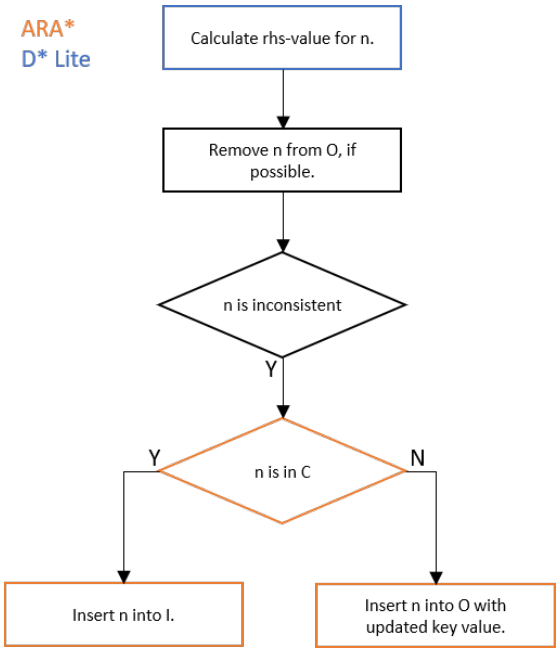


Figure 17: UpdateState-function of AD* [LF+05]

The following graph shows a simplified structure of AD*, which calls UpdateState, seen in Figure 17.

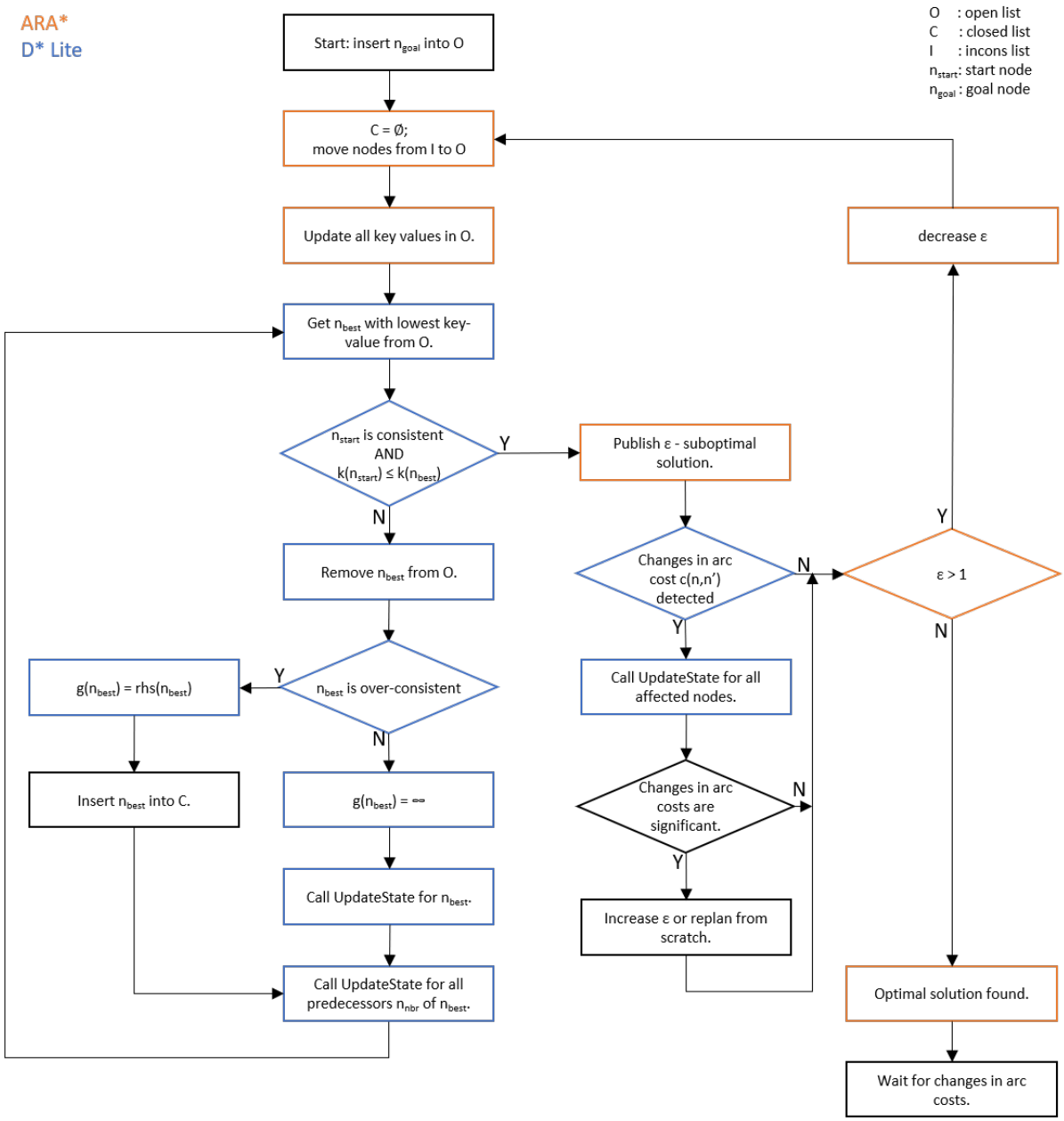


Figure 18: Structure of AD* [LF+05]

2.2.8 Conclusion

In this chapter a selection of pathfinding algorithms has been introduced. With the exception of the A* pathfinding algorithms, which lays a foundation for all other algorithms, they were selected because of their suitable characteristics for their implementation in a dynamic computer game.

In the fast-changing environment of games it is important to use algorithms, which can deliver results quickly and/or are able to react to changes in the game environment. Anytime algorithms, like ARA*, are able to deliver sub-optimal results quickly and improve in the course of time. Dynamic algorithms, like D* and D* Lite, can react to changes in the game environment. The anytime dynamic algorithm AD* combines both these qualities.

Another interesting feature to use in games is the characteristic of certain algorithms to not limit movement to grid edges, allowing for smoother looking paths. Two algorithms possessing this characteristic are Theta* and Field D*.

2.3 Currently used systems

Most of the pathfinding algorithms presented in chapter 2.2 were originally developed for the field of robotics. A* for example was developed as part of the work on “Shakey the Robot” [Wik18c] and D* was implemented in Mars Rover prototypes [KL02c]. Next to robotics, non-player characters in games are a common application for pathfinding systems. With the goal of this bachelor’s thesis being the implementation of a suitable pathfinding algorithm in a computer game, the following chapter presents current applications, practices, and tools for implementing pathfinding in games.

2.3.1 Pathfinding in games

In many games pathfinding is one of the first features implemented into an AI. The core of these pathfinding systems is often based on the A* algorithm. Whether it is utilized for moving whole armies over grid-like world maps in strategy games such as “Civilization V” or “Age of Empires”, or for navigating non-player characters in first-person shooters, such as “Counter Strike” or “Left 4 Dead”. [XH11], [Boo09], [Boo04]

For the cooperative first-person shooter “Left 4 Dead”, published by Valve Corporations in 2008, a presentation (see [Boo09]) was published providing information about the game’s AI system. Part of this AI system is a pathfinding and navigation system. Hereby the navigation system is divided into two parts: finding the shortest path and then following that path. For pathfinding, A* is used on a Navigation Mesh. This produces a rather jagged path. Therefore, the movement along the path is smoothed in a second step. For this the AI uses a look-ahead and steers towards a point further down the path, utilizing local obstacle avoidance as it moves. Figure 19 indicated the Navigation Mesh with dotted lines, the shortest path found by A* with arrows, and the path the AI moves along with the blue line. [Boo09]

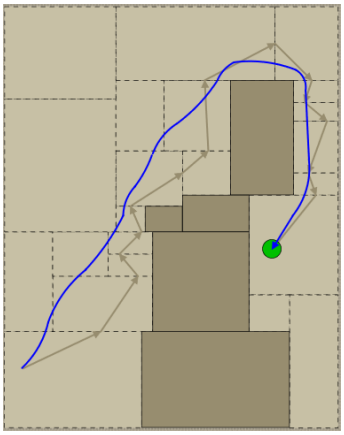


Figure 19: Navigation in Left 4 Dead [Boo09]

As can be seen in this example, it is common practice for navigation systems in computer games to use a global pathfinding algorithm like A*, which delivers a jagged path, and to use algorithms designed to smooth the path afterwards. To avoid or at least simplify the necessity of post-smoothing, any-angle pathfinding algorithms, such as Theta*, can be used. A group at Yeungnam University in Gyeongsan-si, South Korea, compared Theta* with a post-smoothing variant of A*, by running both algorithms over abstracted maps of the games "Dota 2" and "League of Legends". Overall, they found that Theta* results in shorter paths than A*, but also takes longer to find them. The exact method of abstracting these maps and details about their experiment can be read in their paper. [PN+17]

Another approach to pathfinding in modern computer games is the use of hierarchic pathfinding algorithms, for example Hierarchical Path-Finding A* (HPA*), which is described in [BMS04]. This algorithm again uses A* at its core but breaks the search effort into different levels. To do so neighbouring nodes are clustered together to form a more abstract search graph, which can be processed faster than the low-level, more detailed, search graph. Repeating this clustering process, multiple levels can be created. As the name "hierarchical" suggests, these levels are then processed from top to bottom. This means that hierarchical pathfinding algorithms first find a rough abstract of a path, which is then refined with every processed level. [Cha07] Moreover, the algorithm only computes the exact path, using the lower level search graphs, for currently needed clusters. Should the exact path have been computed in all clusters before an AI starts moving along this path, there is a great chance that it has become invalid in some of those clusters by the time the AI arrives at them, due to a dynamic game environment. By only computing exact paths once they are needed, the chance of them becoming invalid is reduced and less search effort is wasted. [BMS04]

The methods used for pathfinding in games described above are designed for moving on the floor, and thereby navigation is more or less restricted to only two dimensions.

2.3.2 Navigation system in Unity

When implementing pathfinding in Artificial Intelligences during the development with the game engine Unity, a commonly used tool is Unity's integrated Navigation System. It is a powerful and easy to use tool, suitable for implementing pathfinding and navigation in a range of game scenarios. Overall Unity's Navigation System consists of a Navigation Mesh, also called **NavMesh**, a **NavMesh Agent**, **NavMesh Obstacles**, and **Off-Mesh Links**, seen in Figure 20. [Uni18d]

Unity's navigation system is primarily developed for navigating agents which are moving on the floor. For this purpose, places in the scene, where agents can stand or move, are defined

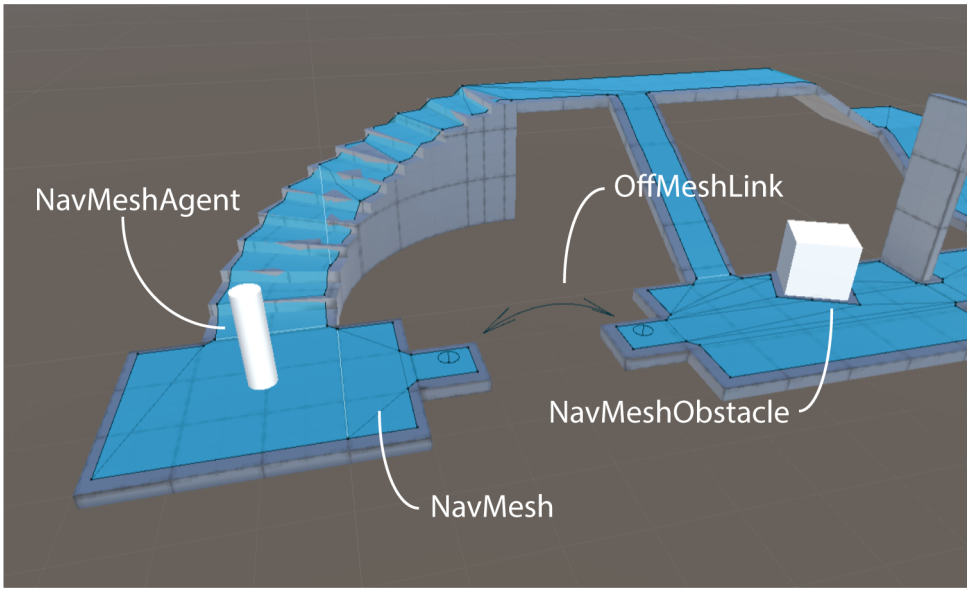


Figure 20: Unity's Navigation System [Uni18d]

as walkable areas. Unity builds walkable areas automatically using the geometry in the scene. These areas are then connected into a surface, which defines the navigation mesh, in short **NavMesh**. The NavMesh, shown in blue in Figure 20, consists of convex polygons, and not only stores polygon boundaries, but also information about which polygons neighbour each other [Uni18a]. Furthermore, areas can be assigned a cost value, for example to discourage the pathfinder to route agents through water [Uni18c]. The process of creating a NavMesh for a scene is often called "baking" and it can be performed before starting the game, as well as during runtime.

Once the NavMesh for a scene has been generated a **NavMesh agent** can be navigated through the scene, from a given starting point to a given goal. It thereby follows the shortest path as well as avoids collision with obstacles or other agents [Uni18d]. Navigating an agent Unity differentiates between two kinds of navigation problems. One is using the NavMesh to determine the shortest path from start to goal. This is a global and rather static process. The second problem is for the agent to follow the determined path, whilst additionally avoiding moving obstacles or other agents. This is a more local and dynamic process. The interaction between these two processes can be seen in Figure 21. [Uni18a]

To find the shortest path using the NavMesh, the start and goal points first have to be mapped to the polygons closest to them. Afterwards an A* pathfinding algorithm is used to determine the shortest path from the start polygon to the goal polygon. The sequence of polygons describing this path is called the "corridor". [Uni18a]

The agent follows this corridor by always steering towards the next visible corner of the corridor.

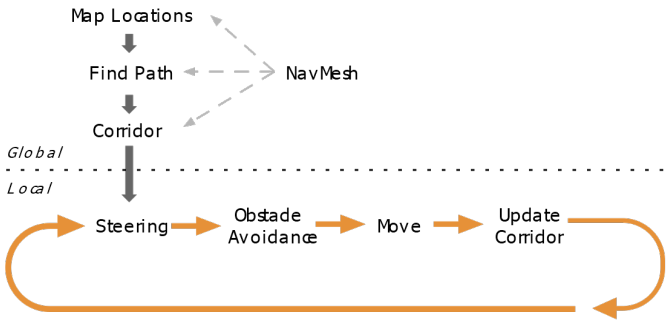


Figure 21: Global and Local Navigation [Uni18a]

For this it first determines the needed direction and velocity to reach the next corner. It then uses reciprocal velocity obstacles (RVO) to predict possible collisions with moving obstacles and adjusts the velocity accordingly. [Uni18a]

In order for the agent to know which obstacles to avoid collision with, they have to contain a **NavMesh obstacle** component. A NavMesh obstacle can impact the navigation either on a global or a local level. For the former it can change the NavMesh by carving holes into it where they touch. Thereby it becomes part of the NavMesh and is considered during the global process of pathfinding. For moving obstacles this means that the NavMesh has to be updated quite often. Therefore, it is recommended to use the latter option for moving obstacles. In this case the agent uses local obstacle avoidance and neither the NavMesh nor the found path have to be updated when obstacles move. The difference between these two options can be seen in Figure 22, where the red line indicates the shortest path the agent is trying to follow and the blue line indicates the rim of the NavMesh. Since local obstacle avoidance does not consider the whole scene, there is a possibility for the agent to get stuck in cluttered environments. Thus, it is useful to enable carving again, once an obstacle becomes stationary. [Uni18a, Uni18b]

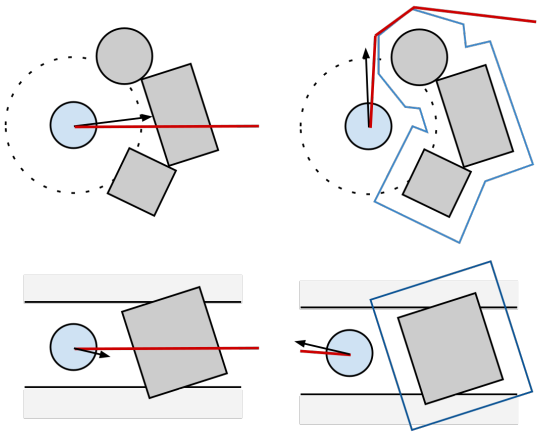


Figure 22: Local obstacle avoidance (left) and carving (right) [Uni18a]

All in all, Unity's Navigation System works well for agents moving on the floor, as is the case in most games where the player steers a human-like or driving character, as in shooters, adventure- or racing games. The Navigation System furthermore supports environments with some dynamic aspects, such as other moving agents or a number of moving obstacles. Mostly, dynamic aspects are supported by utilizing local obstacle avoidance rather than the pathfinding algorithm adjusting to it. For this reason, Unity's Navigation System is not suitable to be implemented in the "Lost in Space".

2.3.3 Path Finder 3D

So far, all pathfinding applications and methods presented in this chapter were designed for navigation in two dimensions, namely on the ground. The game in this bachelor's thesis needs to implement pathfinding in 3 dimensions, though. The Unity plugin "Pathfinder 3D", developed by Graceful Algorithms, enables game developers to do exactly that. "Pathfinder 3D" creates a cell-based search graph setting all cells containing obstacles as impassable. Impassable cells are framed in red in Figure 23.

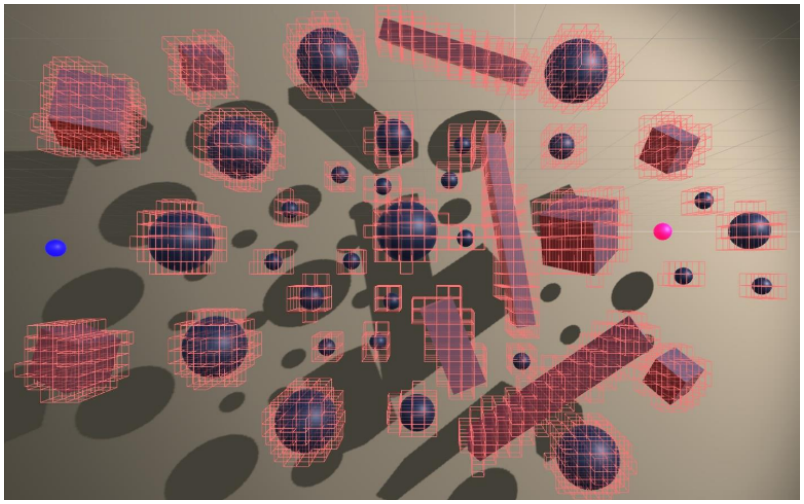


Figure 23: Pathfinding with PathFinder 3D [Gra18]

Once the search graph is generated, the programmer can choose between using a modification of the A* pathfinding algorithm or the Lee algorithm to compute the shortest path between two points in the search space (blue and pink spheres in Figure 23 indicate start and end point). [Gra18]

Even though "PathFinder 3D" enables the user to find paths in a 3-dimensional space, it is not suitable for "Lost in Space" since it cannot react well to dynamic environments. Every time an obstacle is moved, the search graph has to be re-build and the path re-calculated from scratch. Since this process is quite laborious, one should use it as little as possible. [Gra18]

3 Concept

The goal of this bachelor's thesis is to realise a project implementing and comparing different pathfinding algorithms in a dynamic 3-dimenstional space. Considerations made for the execution of this project will be argued in the following chapter.

3.1 Development and testing environment

Since games provide the possibility to control all environmental factors and replicate the exact same setting several times, they are the optimal choice to produce comparable results. For implementation, testing, and comparison of pathfinding algorithms in this bachelor's thesis, an abstract game environment is used. In comparison to the game environment in "Lost in Space", the test environment is smaller and more controllable. It excludes all types of random behaviours, ensuring a replica of the exact same environment for each tested algorithm. Additionally, the test environment is more complex, requiring the algorithm to take more turns and to react to different kinds of dynamic obstacles. Results acquired within this bachelor's thesis will then be tested and verified by implementing them in a real game setting, namely in the game "Lost in Space" described in 2.1.

The development of the game and the testing environment, as well as the pathfinding implementation, will be realised using the game engine Unity (Unity 2018.2.10f1).

3.2 Proceeding

Before any pathfinding or navigation system can be worked on, the game "Lost in Space", as well as the test environment, foremost used for implementation, have to be set up and main functionalities implemented.

In the case of "Lost in Space" this means to first set up the main game functions. This includes the player's and asteroids' movement as well as a shooting and damage system for the player. As a next step the movement mechanic and the damage system of the enemy will be implemented. As a default setting the enemy will steer in the direction of the player. It will take damage upon collision with asteroids and when it is shot at by the player.

The test environment only requires very limited game functionalities. The main task is building the environment and implementing all required movements for dynamic obstacles. The AI's

movement will essentially be the same as the enemy's in "Lost in Space".

Once the environment is set up and the basic functions are implemented, a search space has to be defined and set up for generating a search graph, that the pathfinding algorithms can utilize. Two options of setting up a search graph are either to use a cell-grid, similar to "PathFinder 3D", or waypoints. The former requires the search space to be divided into a cell grid with each cell representing a node. Each node can then be blocked or free. In the case of using waypoints,the search graph is defined by a number of nodes represented by waypoints, which can be placed in the environment either manually or automatically. During execution of this project both methods will be set up and throughout implementation and testing it will be determined which one is more suitable for each algorithm individually.

After completion of these prearrangements the enemy's pathfinding system can be implemented. Most algorithms detailed in chapter 2.2 will be used with the exception of D* and Field D*. These will be excluded since simpler options for both are available. D* is replaced by D* Lite and Theta* replaces Field D*. All other algorithms will be implemented and tested using both the grid-cell and the waypoint search graph. The behaviour of each algorithm will be observed closely and relevant data, needed for comparison in a later step, collected. Throughout the process of implementation and testing, some algorithms might be optimized if it seems appropriate.

As a last step the behaviour and performance of the different algorithms will be measured, analysed, and compared. Conclusions drawn from this analysis will provide information which search graph setup and pathfinding algorithm is most suitable for use in the game "Lost in Space". Moreover, it will determine whether any post-processing of the path or the navigation along the path is necessary.

3.3 Comparison

For comparison of the pathfinding algorithms information about a number of different aspects needs to be collected and compared. On a basic level this includes observing whether the algorithm is able to fulfil its task of catching the player without colliding with any obstacles. Moreover, the total length of the path travelled by the AI from its starting point until it reaches its destination is measured.

For further comparison on a more detailed level the total number of expanded nodes is counted and the total time spent on calculations required for pathfinding is determined.

3.4 Additional considerations

For simplicity only one enemy will be in the game environment at a time. In “Lost in Space” it will spawn at a certain distance from the player, then move towards the player and respawn on collision with asteroids or with the player. In the abstract test environment the pathfinding AI has a set starting position, to which it will also reset upon reaching the player or colliding with obstacles.

Additionally, the conditions on which the search graph has to be updated and the algorithms informed about its changes has to be defined. One possible condition would be to update the search graph once an obstacle has moved more than a certain distance. Due to all dynamic obstacles moving with similar speeds, it was decided to update the search graph in intervals. The length of an interval, and thereby the duration between updates, can be chosen by the developer.

3.5 Expected results

Since a suitable pathfinding algorithm for games like “Lost in Space” has to perform well in a highly dynamic game environment as well as deliver results quickly, the most promising approach is probably the AD* algorithm.

Due to its characteristic of not restricting its found paths to connections between nodes in the search graph, the any-angle algorithm Theta* is also promising approach to finding shorter and smoother-looking paths, than the other pathfinding algorithms.

4 Implementation

Previous chapters have introduced the game "Lost in Space", laid a theoretical foundation about a variation of pathfinding algorithms, and presented two possibilities for setting up a search graph. In this chapter all these theoretical foundations will be put into practice. Fist, the game "Lost in Space" and the abstract test environment will be developed. Following, a cell-grid and a waypoint-based search graph are set up. Lastly, the main focus of this bachelor's thesis is implemented; the pathfinding algorithms A*, ARA*, Theta*, AD*, and D* Lite.[3]

4.1 Test environment

In the following chapter the implementation of the game "Lost in Space", as well as the set-up of the abstract test environment, foremost used for implementation, testing, and comparison of the pathfinding algorithms, will be presented.

4.1.1 Lost in Space

The first step in implementing the game "Lost in Space" is to revisit the earlier version "Lost in Space VR" to determine what assets and scripts can be reused, and which functionalities have to be added or adjusted.

The main adjustment needed for "Lost in Space" is the transition from VR to non-VR, as well as the use of a standard controller instead of the balancing board as a custom controller. Developing a game in VR with this special controller, one had to pay close attention not to overwhelm the player and be aware of the issue of motion sickness. Therefore, to avoid any negative effects on the player, their movement was kept at a relatively slow pace and was quite restricted. Players could only steer to the left and right, whilst the forward motion was simulated by a constant movement of the asteroids. For this asteroids would spawn on approximately the same vertical position as the player and move towards the centre point of the player's movement orbit. To avoid overwhelming the player, asteroids were initially spawned in circles around the centre (see Figure 24). This resulted in asteroids arriving at the player in waves, giving time to collect oneself in-between. Once asteroids hit the player or reached the centre of the orbit, they would respawn on the outer circle and restart their movement. With this concept of player and asteroid movement, the movement in the game is basically restricted to two dimensions.

[3]The result of these implementations can be downloaded from: `https://github.com/CarinaKr/Pathfinding-in-dynmic-3D-space`

Using a standard controller and not playing in VR the player feels a lot more comfortable and confident. Therefore, in “Lost in Space” they are given a greater degree of freedom in their movement, and a faster paced game. Players are advised to use a standard Xbox controller to steer their UFO. They are positioned inside their UFO, seeing the game from the pilot’s perspective. For steering the left analogue stick is used. Moving the analogue stick left and right, the UFO turns to the left and right around its vertical axis. Moving the analogue stick up and down, the UFO tilts upwards and downwards around its horizontal axis. The UFO always moves with constant speed in its viewing direction. This enables the player to move freely throughout the whole game field, expanding movement to all three dimensions. Additionally, the player can look around by using the right analogue stick to turn the camera. Using the X-button, the player can switch between two guns and use the left and right triggers at the back of the controller for shooting. Guns can be re-loaded using the B-button. The player’s UFO takes damage upon collision with asteroids and enemies.

To adapt the asteroids’ behaviour to the faster paced game and the greater degree of freedom, two main aspects have to be adjusted.

First of all, asteroids do no longer have to be spawned in circles. Since players are more accustomed to using a standard controller, no low-time between asteroid waves is required. Thus, asteroids are now spawned in mostly random positions within a certain distance to the player at the beginning of the game (see Figure 25). This results in a more chaotic asteroid field, making the game more interesting and challenging.

Additionally, the player is now moving forwards on their own, so the asteroids do not have to move towards them anymore to simulate the player’s forward movement. Instead asteroids now move in mostly random directions. Moreover, asteroids move with different speeds, depending on their size. Large asteroids move slower than smaller ones.

Looking at the game environment from an outside point of view, it seems as though there is a sphere of asteroids surrounding the player. Throughout the game this sphere moves, keeping the player at its centre. This effect is caused due to asteroids only existing within a certain radius around the player. If their distance to the player gets too great, due to the player’s or their own movement, they respawn at the outer rim of the sphere. Using this method, it is ensured that the number of asteroids present in the game scene is limited to a finite number. It is also the reason for the only constraint to the randomness of the asteroids’ movement. To ensure asteroids do not have to be respawned immediately after appearing, their movement is limited to directions leading them further to the centre of the sphere, in other words decreasing their distance to the player, instead of out of it.

To reduce the number of asteroids needed in the game scene further, tests have shown that a full sphere is not actually necessary. Instead a flattened sphere in vertical direction, meaning a lower value of vertical spread, is sufficient. Tests have also shown that additionally rules have to be implemented to ensure asteroids are evenly spread throughout the whole sphere. Therefore, they are instantiated in overlapping vertical layers, with a given number of asteroids on each layer. The result of this spreading method can be seen in Figure 27.

Figure 24 and Figure 25 show the difference of spawning the asteroids at the beginning of the game from a top-down view. Figure 26 and Figure 27 show the difference of spawning the asteroids at the beginning of the game from a side-view. Asteroids are framed in yellow, the red, green, and blue arrows indicate the x-, y-, and z-axis of the game field and are currently anchored at the centre of the game field.

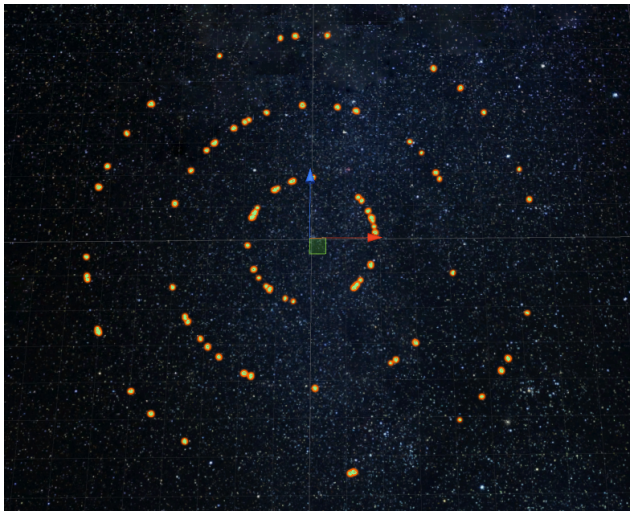


Figure 24: Spawn asteroids in circles (top-down)

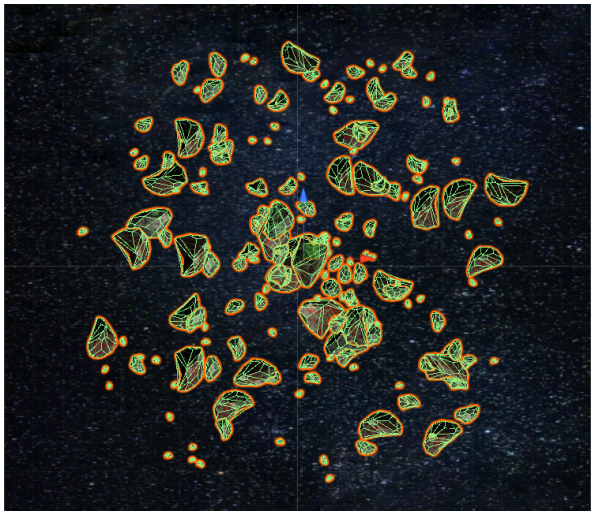


Figure 25: Spawn asteroids in flattened sphere (top-down)

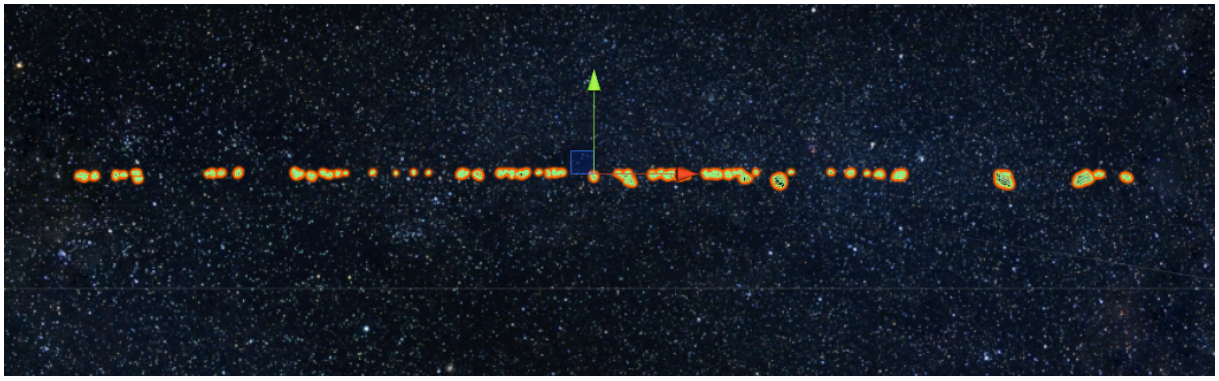


Figure 26: Spawn asteroids in circles (side-view)

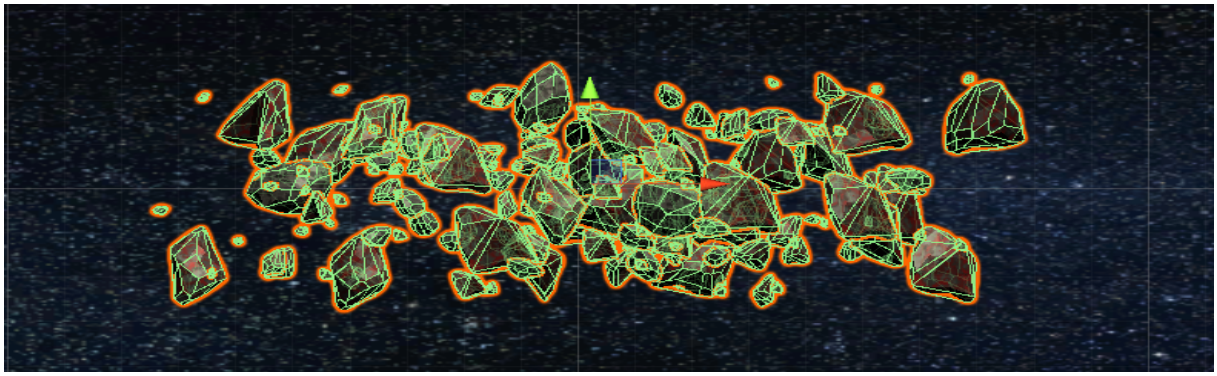


Figure 27: Spawn asteroids in flattened sphere (side-view)

As a last step the enemy's movement and damage system is implemented. Enemies take damage upon collision with asteroids, and if they are shot by the player. Once an enemy has lost all its health, they get respawned at the outer rim of the game field. An enemy's movement can be directed by either giving it the next destination point to move towards, or by turning it to face into the correct direction, in which case it will move in its viewing direction. In both cases the enemy moves at a constant speed. Unless given any other direction the enemy by default moves on a direct line towards the player.

In conclusion it can be said, that from the original game only the overall idea of the space-themed game is transmitted. All visual assets, such as 3D models and textures, can be reused for "Lost in Space", but most of the scripting has to be redone, due to the changed movement concept of the player and asteroids.

4.1.2 Abstract test environment

For implementing, testing, and comparing the different pathfinding algorithms throughout this bachelor's thesis an abstract test environment is used. The main reason for this is to ensure that all algorithms are tested in the exact same setting to produce comparable results. Therefore, the used environment has to be completely controllable and not contain any random factors.

The setup of the abstract testing environment can be seen in Figure 28. Overall the environment is set up in a cuboid shape and consists of different static and dynamic objects. Static objects are white, dynamic objects are color-coded dependent on their type of movement. Green obstacles move with constant speed between patrol points, red obstacles turn around their z-axis, dark blue obstacles rotate around their x-axis, and light blue obstacles appear and disappear on a set timer. For testing purposes, dynamic obstacles do not collide with static obstacles or with each other.

In this environment the pathfinding AI is represented by a dark-green sphere, which starts at the top left corner and tries to reach a goal, represented by a orange sphere, at the bottom right corner. The goal can also be made dynamic by defining patrol points for it to move between.

Please note that this environment is for test purposes only and therefore does not contain any interesting gameplay aspects.

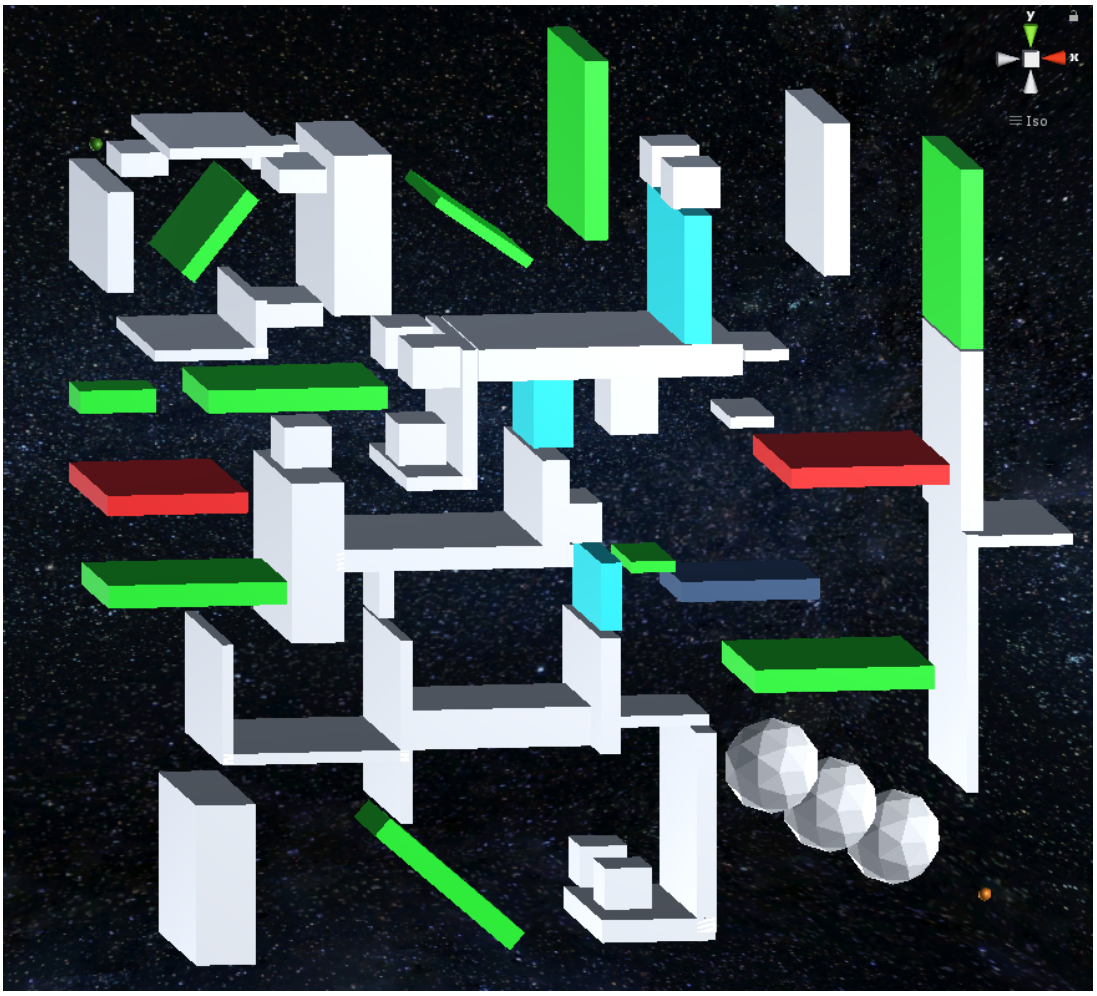


Figure 28: Abstract test environment

4.2 Search graph

Before pathfinding algorithms can be implemented into any environment, a search space has to be defined and a search graph constructed for it. For the abstract test environment, used for implementation, testing, and comparison of pathfinding algorithms in this bachelor's thesis, the search space is defined to be a cuboid with the dimensions 64 units x 56 units x 16 units (X x Y x Z). Due to the development with the Unity game engine, all measurements will be given in "units". Hereby one "unit" equals one unit in Unity. Overall a uniform cost environment is assumed. Therefore, the unobstructed path cost $c(n, n')$ between any node n and its neighbour n' is the distance between them.

Two ways to construct a search graph for this search space are the use of a cell-grid, similar to "PathFinder 3D", or to use waypoints. The implementation of both these options is documented in the following chapter.

4.2.1 Cell-grid

The idea of dividing the search space into a 3-dimensional cell-grid is based on the common use of cell-grids in 2-dimensional pathfinding applications, as can be seen on the examples given throughout chapter 2.3.

In the cell-grid every cell represents a node, with the node's position being the centre of the cell. Due to this correlation, the phrasing "node" and "cell" will be used interchangeably. For the used test environment each cell is defined to be a cube with the side length of 1 unit. Given the search space is 64x56x16 units^3, the complete search space consists of 57344 cells. Figure 29 shows the setup cell grid from an orthographic front view, Figure 30 shows the cell grid from an angled view. Cells are framed in blue and for reasons of clarity only the top left part of the search field is shown. For an image of the full cell grid see Figure 56 in the Appendix.

Each node has a maximum of six neighbours: the node of each connecting cell in x-, y-, and z-direction. There are no connections between cells which are only sharing the position of one vertex, in other words there are no diagonal connections. Figure 31 indicates the connection of a node to all its neighbours with yellow lines. For a full image of the cell grid with neighbouring connections see Figure 57 in the Appendix.

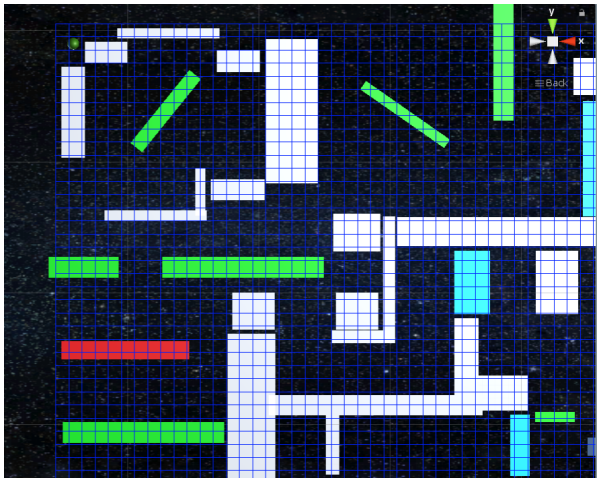


Figure 29: Part of cell-grid (orthographic front view)

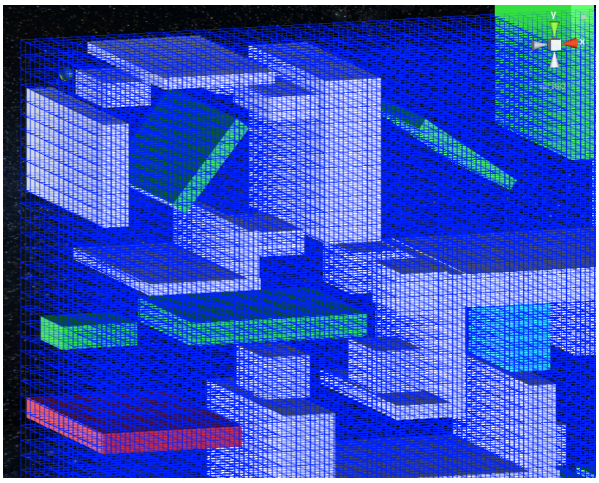
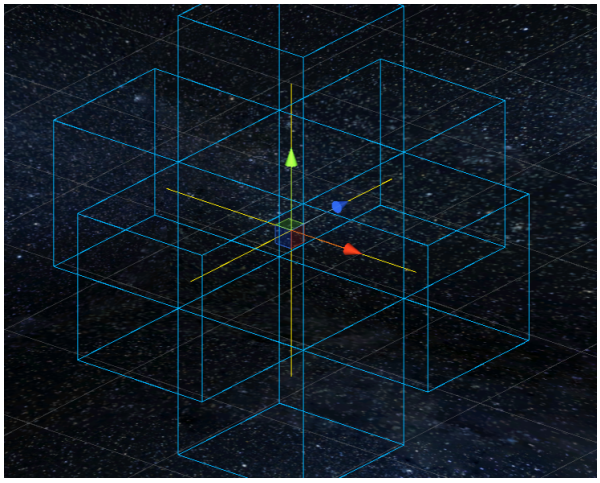
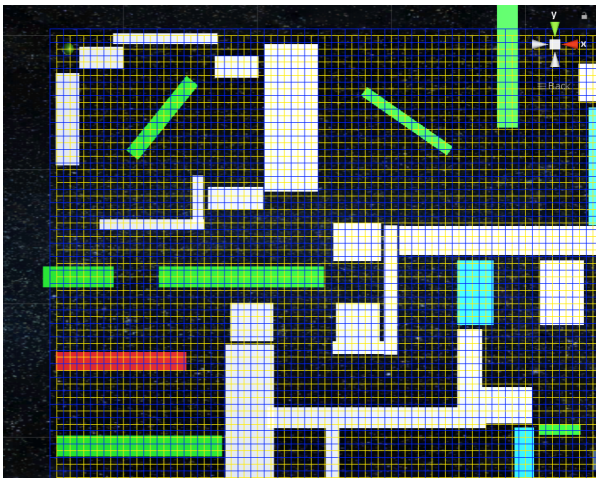


Figure 30: Part of cell-gird (angled view)



(i)



(ii)

Figure 31: Neighbours in 3-dimensional cell-grid

Each cell, and therefore its corresponding node, can either be blocked or free. A cell is defined as blocked, if it overlaps with any obstacle. This means, a cell is blocked once it is even partially obstructed by an obstacle. Figure 32 shows all free cells in green and all blocked cells in red. Overall a uniform cost environment is assumed. Therefore, the path cost $c(n, n')$ from a node n to any of its neighbours n' is either the size of the cell or infinite:

$$c(n, n') = \begin{cases} \textbf{cellSize}, & \boldsymbol{n} \textbf{ is free and } \boldsymbol{n'} \textbf{ is free} \\ \infty, & \boldsymbol{n} \textbf{ is blocked or } \boldsymbol{n'} \textbf{ is blocked} \end{cases}$$

In theory the developer can choose the size of the cells freely. They should pay attention, though, not to make them too large or too small. The cell size has to be balanced between the required computing power and the accuracy of the path. The larger the cells are, the fewer are needed.

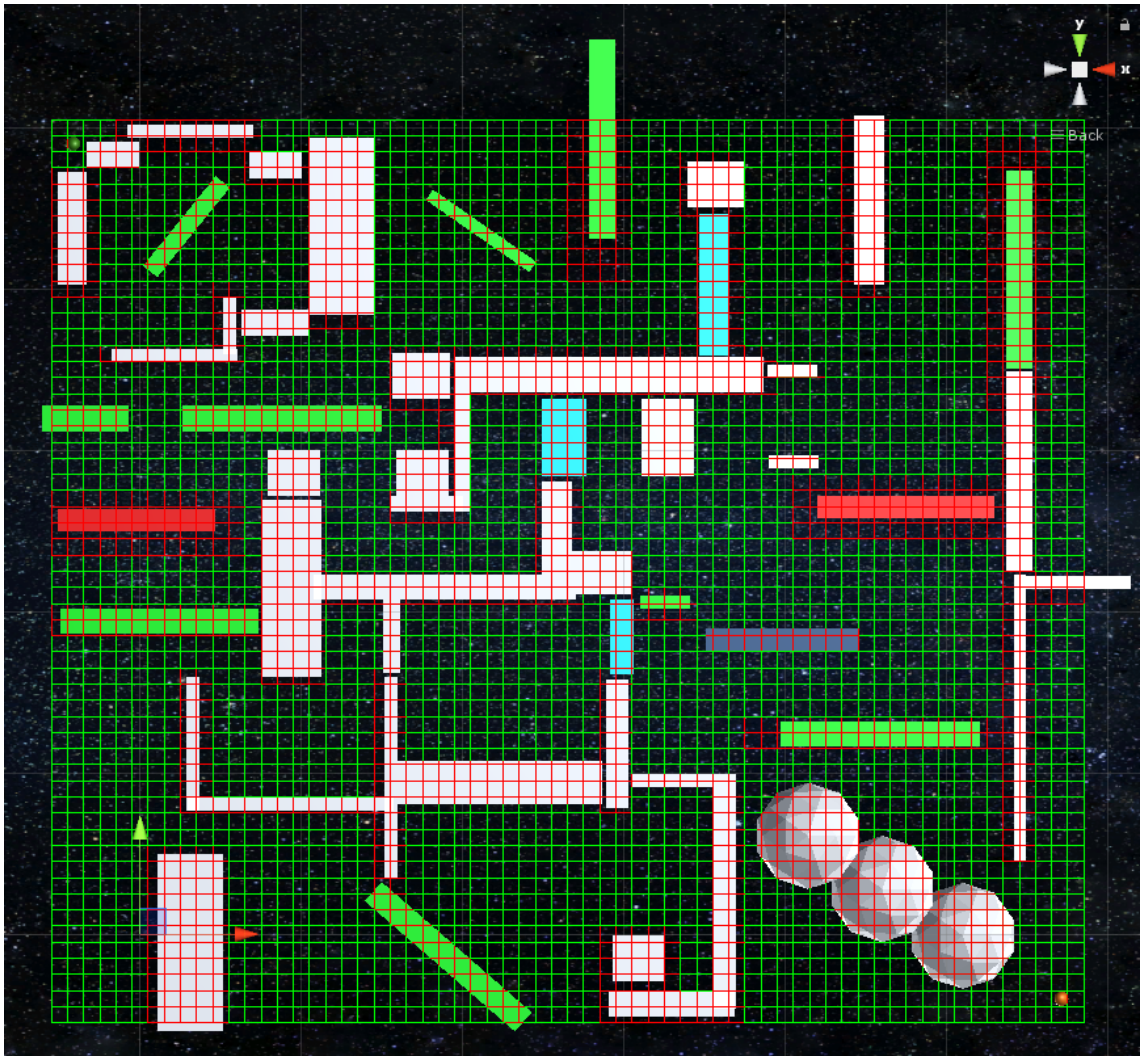


Figure 32: Cell-gird with blocked and free cells

This results in less computing power to be needed. If cells are too large, though, the found path might be inaccurate. Since a node is already marked as blocked if its cell is partially obstructed, larger cells might lead to the pathfinding algorithm considering paths blocked that are narrower than the cell size, but might be wide enough for the player. An additional factor to consider when choosing the cell size is based on the concept of movement used by Unity. In Unity it is standard to take the players centre as its current position. When the player moves from one node to the next, they move their own centre from the centre of one cell to the centre of the next cell. Therefore, if the cells are smaller than the player's radius, the player might collide with obstacles whilst moving to the next node. The other way around, the larger cells are, the more distance will be between the player's position and the border of obstacles.

4.2.2 Waypoints

A second method for creating the search graph is to use waypoints. Generally speaking, waypoints define places in the search field for the player to stand upon, which have connections between each other for the player to travel along. In the case of using waypoints in the context of pathfinding, each waypoint represents a node of the search graph. Therefore, the phrasing "waypoint" and "node" will be used interchangeably.

In theory, waypoints could be placed in the search space by the developer manually. Considering the extent a search space might have, though, and especially in dynamic environments, this process could be quite time consuming. Therefore, creating waypoints for search spaces in this bachelor's thesis is done automatically. To create waypoints in a sensible manner, the idea is used that the pathfinding AI should only have to change directions at edges of obstacles. For this reason, one waypoint is assigned to each corner, also called vertex, of each obstacle and a variable number of waypoints is placed on the edges connecting these vertex-waypoints.

For generating waypoints automatically at each vertex of an obstacle, a mesh, assigned to the obstacles, is used. The developer can choose to either use the obstacle's mesh or to assign a separate mesh, which will only be used for creating waypoints. To successfully generate waypoints around an object that the player can travel in between without collision, the mesh used for generating these waypoints has to resemble, or lay outside of, the mesh used for collision detection. In practise this means, if an object's collider resembles the shape and size of that object, as for example in Figure 33, the object's mesh can be used. Otherwise, for example when using Unity's standard colliders to simplify complex objects or when using convex mesh colliders, a separate mesh needs to be assigned for the waypoint generation. In the following figures the used colliders are framed in green. Figure 34 shows the use of a convex mesh collider. As can be seen in the top left corner, the collider does not resemble the shape of the object completely. Therefore, to avoid collisions of the AI with the object's collider, not the object's mesh but a separate mesh, resembling the shape and size of the convex mesh collider, is used for generating the waypoints.

Additionally, to prevent collision of the AI with obstacles, the player's radius has to be factored in when placing waypoints. Therefore, waypoints are not located on the exact position of a mesh's vertex but offset by the player's radius plus a variable margin.

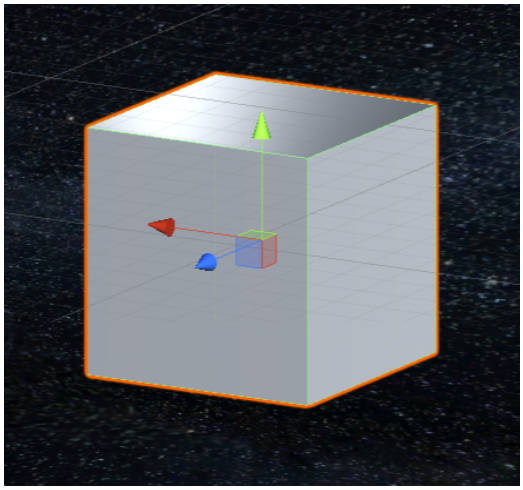


Figure 33: Cube with box collider

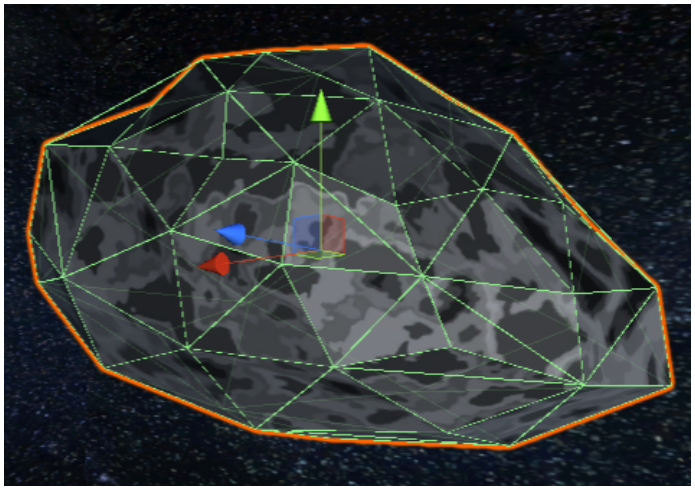


Figure 34: Asteroid with convex mesh collider

To realise the generation of waypoints some functions provided by Unity in regard to dealing with meshes are used. Firstly, Unity includes a function which returns an array of 3-dimensional vectors, listing all vertex-positions of a given mesh. Examining this array, it can be observed that certain positions are listed multiple times. This is caused by the function returning the position of all individual vertices. Since some vertices might have the same position but differentiate in other attributes, such as normal- or tangential-vectors, they are not the same vertex and therefore their positions will be listed individually. Secondly, Unity provides a function returning an array, which contains each vertices' normal-vector. Lastly, as customary, all meshes are constructed of a finite number of triangles. Therefore, Unity includes a function providing information which vertices form the corners of each of these triangles.

Cumulating, combining, and utilizing all this information one knows the position of each vertex, the normal vectors of all faces adjacent to this vertex, and which vertices are neighbours to each other.

Adding up the normal vectors of all faces adjacent to a vertex and normalizing the result, results in the direction in which to offset the waypoint. Multiplying the offset direction vector with the magnitude of the vector $\begin{pmatrix} player\ radius + margin \\ player\ radius + margin \\ player\ radius + margin \end{pmatrix}$ and adding the result to the vertex' position, the position of the waypoint associated with this vertex is calculated. In the following, these waypoints will be called vertex-waypoints and their resulting positioning can be seen as yellow diamonds at the example of a cube and an asteroid in Figure 35 and Figure 36.

Since only assigning waypoints to each vertex of an obstacle can lead to rather rough search graphs, a function was implemented for the developer to choose the maximum distance between waypoints. To create these in-between waypoints, the first step is to determine which

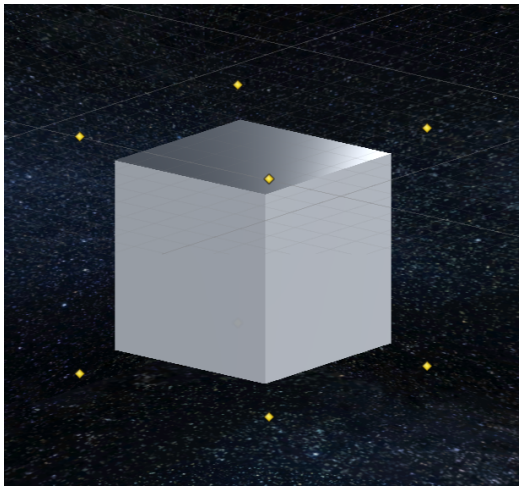


Figure 35: Cube with waypoints

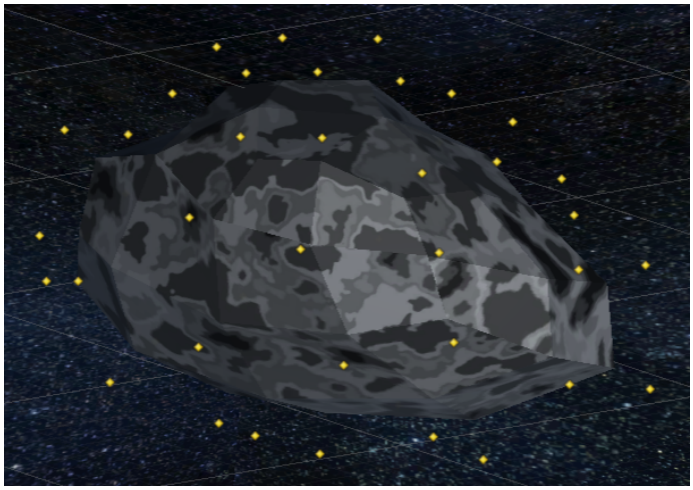


Figure 36: Asteroid with waypoints

vertex-waypoints are neighbours to each other. As stated above, the information about which vertices are neighbours to each other can be gained by utilizing Unity's mesh functions. Vertex-waypoints are defined to be neighbours when their associated vertices are neighbours. If the distance between a vertex-waypoint n and any of its neighbouring vertex-waypoints n' is greater than the defined maximum distance, additional waypoints are placed regularly spaced on the edge connecting n and n'. An example can be seen in Figure 37. For performance reasons the maximum distance between waypoints should be chosen as large as reasonable.

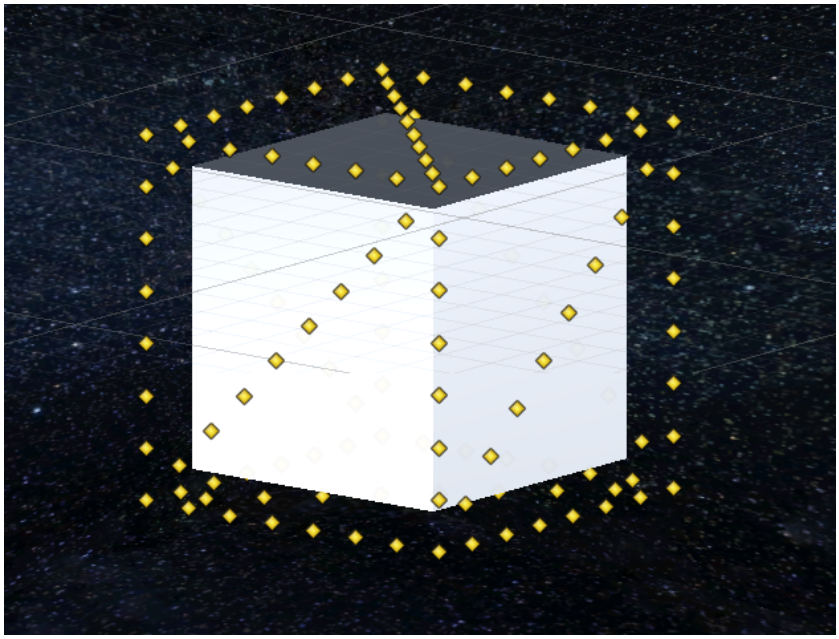


Figure 37: Cube with additional waypoints

The connections between waypoints have to be defined in a next step. In other words, it has to be determined which waypoints are neighbours to each other. Doing so, direct and indirect neighbours are differentiated. Direct neighbours of a waypoint n are neighbours that are assigned to the same obstacle as n. Indirect neighbours of n are waypoints assigned to other obstacles than n.

In order to determine direct neighbours of waypoints, information about which waypoints are on each edge is stored upon instantiation of the waypoints. Hereby waypoints are stored in so called edge lists. Each list starts and ends with a vertex-waypoint and all waypoints in-between them are ordered by their distance to the starting vertex-waypoint of this list. Vertex-waypoints are part of multiple edge list, since they lay on multiple edges simultaneously. Utilizing this information, each closest waypoint n' of each edge list containing the vertex-waypoint n is defined to be a direct neighbour of n. In-between waypoints get assigned each waypoint before and after them in the ordered edge list as a direct neighbour. Additionally, the closest waypoint n' which has a line-of-sight to the in-between waypoint n and additionally is on an edge sharing the vertex-waypoint closest to n is assigned as a direct neighbour to n.

In Figure 38 and Figure 39 connections between waypoints are indicated by yellow lines. Figure 38 shows the connections between waypoints, using a maximum distance between waypoints of 1 unit on a cube with a side length of 5 units. In Figure 39 waypoints are only assigned to vertices. For a better overview, only some of the connections are shown in this image. One can clearly notice the similarities between the pattern shown by connecting lines and the objects collider, seen in Figure 34.

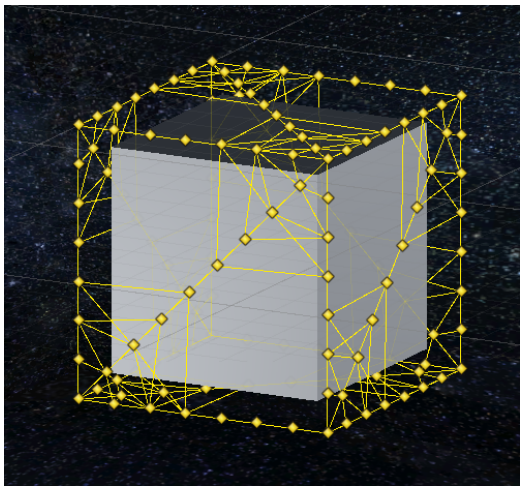


Figure 38: Direct neighbours on cube

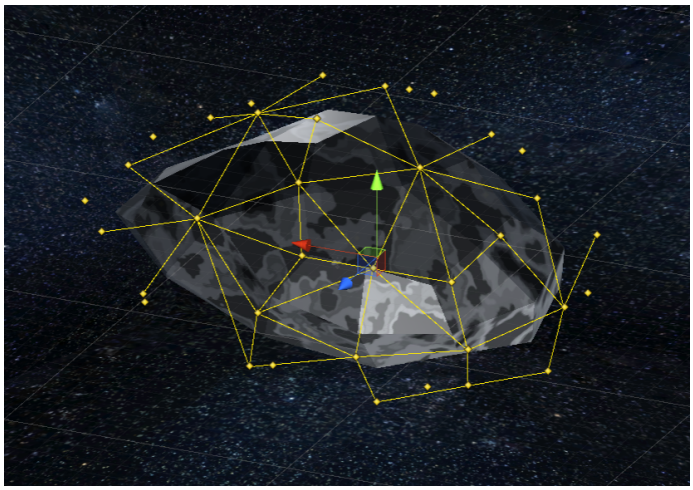


Figure 39: Direct neighbours on asteroid

Due to direct neighbours being relative to their obstacle, they are consistent throughout the whole pathfinding application, even if the obstacle is dynamic. This means, a waypoint n always has the same direct neighbours n' and, as long as the path between them is not blocked, the path cost $c(n, n')$ remains the same.

To create connections between different obstacles, indirect neighbours of waypoints are defined. Each waypoint n gets assigned the closest suitable waypoint as an indirect neighbour n'. A waypoint is suitable, given it is on a different obstacle, not blocked, and the path between n and n' is not obstructed. Similar to cells in the cell-grid, waypoints can be blocked or free. A waypoint is defined as blocked, if a sphere with the player's radius positioned at the wayoint's current position would overlap with any obstacle. A waypoint is also defined as blocked, if it lays outside of the search field. To determine whether the path between two waypoints is obstructed, a line-of-sight check is run between them. To produce usable results, the line-of-sight check must also factor in the player's radius. Figure 40 indicates all neighbouring connections, direct and indirect, in yellow.

Figure 40: Test environment with all waypoint-connections (angled-view)

Since indirect neighbours are very much dependent on the current position and rotation of an obstacle, as well as the position and rotation of all its surrounding obstacles, indirect neighbours change constantly in a dynamic environment. It follows that a waypoint n is not always assigned the same indirect neighbour n'. When the dynamic environment changes, indirect neighbours have to be reassigned. Also, the path cost $c(n, n')$ is not constant, but changes upon movement or rotation of obstacles.

The following figure shows the complete setup of the test environment using waypoints for defining the search graph. In Figure 41 connections between neighbouring waypoints are shown by yellow lines, blocked waypoints are indicated by red wire-framed cubes, and free waypoints by green wire-framed ones. The green frame around the environment defines the border of the search field. For this example, a maximum distance between waypoints of 5 units was chosen. This value is subject to change and might be adjusted throughout the testing of different pathfinding algorithms.

Figure 41: Test environment with all waypoints and connections (front-view)

4.2.3 Conclusion

These are only two options of setting up a search graph for pathfinding in dynamic 3-dimensional spaces. Multiple optimizations and changes could be applied to these methods. For example, the layout of the cell-grid could be optimized by clustering certain areas. Also, the methods used to place waypoints and determine direct and indirect neighbours could be done differently. For example, waypoints could be positioned in the search field using different criteria. Additionally, instead of only assigning one indirect neighbour to each waypoint, multiple indirect neighbours could be assigned, using not only the closest, but also second and third closest suitable waypoints as indirect neighbours. In addition, the placement of the waypoints does not consider any real-world physics, such as gravitational forces two masses might exert towards each other. Since the waypoints are rather close to larger obstacles, this might cause issues in a real-world environment, but can be neglected in the current game environment. Due to this bachelor's thesis focusing on pathfinding algorithms and not on the optimal generation of search graphs, the two methods described in this chapter are sufficient to be used as described for testing purposes.

4.3 Pathfinding algorithms

After the search graph has been setup, pathfinding algorithms can utilize it to compute the shortest path from any given start node to any given goal node. In this chapter the implementation of the pathfinding algorithms A*, ARA*, Theta*, D* Lite, and AD* will be discussed. At first implementation decisions will be named which are relevant for all, or most of, the algorithms, such as the heuristic or the implementation of the open, closed, and incons list. Further on, the implementation process of each algorithm is discussed briefly, detailing the specifics of each algorithm individually. From now on all implemented algorithms will be devided into two groups: A*-based and D*-based algorithms. A*-based algorithms include A*, ARA*, and Theta*. D*-base algorithms include D* Lite and AD*.

4.3.1 General implementation

Some general aspect of implementing the pathfinding algorithms, also in regard to the used Game Engine Unity, will be discussed in this chapter.

4.3.1.1 Multi-threading

Implementing a pathfinding system into a dynamic game environment it not only has to be ensured the pathfinding operates as quickly as possible, keeping up with the changing environment, but it is also required that the overall game experience is not disrupted by it. One of the biggest disturbances to be considered hereby are frame-drops. The pathfinding system, even after being optimized, could potentially require extensive computation time, exceeding the time available for one frame, assuming an ideal framerate of 60 FPS (Frames per Second) and a minimum framerate of 30 FPS. Therefore, the pathfinding system has to be implemented in such a way that it does not block the main thread, on which the majority of the game's functions are operating. Otherwise the overall game would not be able to run smoothly and thereby destruct the player's experience.

Due to the Unity Game Engine generally being frame based and single-threaded, working with multiple threads is more challenging than with other programming languages and development environments.

Instead of working with multi-threading, in Unity a common method of running complex or time extensive operations asynchronously without them blocking the main thread, is to use so called "coroutines". To explain the exact workings of coroutines would be beyond the scope of this thesis. Detailed explanations can be found online, for example at [Uni19a] or [UG16].

To put it briefly, a coroutine is a certain kind of method, whose execution can be paused and later resumed at the same point it was stopped once a certain condition is met. Hereby the method is paused at least until the end of the current frame and can soonest be continued in the next frame. This functionality enables the developer to split up complex calculations or operations and spread them over multiple frames. Whilst the coroutine is pausing the rest of the frame leaves capacities for other game functions to be executed. By running game functions in between the algorithm's calculation, the main thread is not blocked by the algorithm and no frame-drops occur. Please note, that even though coroutines lead to the impression of multiple functions running simultaneously, they do still operate on the main thread.

The challenge, when using coroutines with pathfinding algorithms, is to determine when to break and pause the algorithm's computation. In their core all pathfinding algorithms, described and implemented in this thesis, utilize a while-loop to expand nodes until the goal node, or start node, is reached. Therefore, one option is to simply break and pause at the end of each iteration of the while-loop. Unfortunately, this causes the algorithm to pause until the end of the frame after expanding each node. Computing the pathfinding algorithm would in this scenario take at least as many frames, as it expands nodes, and would be heavily reliant on the overall framerate of the game. For example, using the test environment described in 4.1.2, setting up the search graph by using a cell grid as described in 4.2.1 and running an A* pathfinding algorithm with an inflation factor $\varepsilon = 1$, on average 28295.99 nodes are expanded given a static AI and destination (see Figure 48 (iii)). Assuming an average framerate of 60 FPS, A* would take about 471.6 seconds (which is more than 7.5 minutes) to compute when pausing the coroutine after each expanded node. Even in a static game environment this would be quite a long time for an algorithm to compute before its results can be used by the AI. In a dynamic environment, though, it is highly challenging, if not even impossible, to work with a pathfinding algorithm with this extensive runtime. By the time the algorithm's results can be used, the dynamic environment will most likely have changed significantly and the moving AI will have collided with a moving obstacle. Therefore, utilizing coroutines for the implementation of pathfinding algorithms in a dynamic game environment is not a realistic option.

Since Unity 2017, a new way of implementing asynchronous methods became available - the so-called "async-await" [Ver17] feature. In addition with the "Async Await Support" Asset [Ver18] , which can be downloaded from the Unity Asset Store for free, one can now also execute multiple threads as described in [Ver17]. The downside of this feature is, that even though now functions, such as calculations, can be executed on a background thread, the Unity API can not be used in any of these functions. In other words, the Unity API can not be called from the background thread. This fact has to be considered carefully during the development process and eventually it is the decisive factor of whether a function can be run on the background thread,

or whether a coroutine has to be used. Adapting some functions to this restriction, it is possible to implement most of the pathfinding algorithms to run on a background thread. The only exception is Theta*, since it is reliant on line-of-sight checks, and therefore on Unity's physics (see 4.3.2.3).

4.3.1.2 Open, closed, and incons list

Whether the pathfinding algorithm is run in a coroutine or on a background thread, the open list is a vital part of each of the pathfinding algorithms implemented in this thesis. Just as much the closed list, which all pathfinding algorithms except D* Lite utilize, and the incons list, which is utilized by ARA* and AD*. During the implementation of the pathfinding algorithms, testing has shown that operations requiring interaction with these lists were amongst the most time consuming, and thereby expensive, ones. Therefore, the implementation details of these lists have to be considered carefully. In this chapter, the development process and final setup of the open, closed, and incons list as they are used for all pathfinding algorithms throughout this thesis, will be presented.

The task of the **open list** is to store nodes with their according priority key values. The open list has to be accessed multiple times during each expansion of a node for removing the current node, and adding new ones. Additionally, the node with the lowest priority key has to be determined frequently to know which node will be expanded next. Therefore, the open list should be implemented in a way which allows for fast deletion, addition, lookup, and sorting. Hereby it is important to keep in mind that the priority key can consist of one or two values, depending on the algorithm.

In a first iteration a Dictionary<TKey, TValue> class was used for the open list, with the dictionaries' key being the node (of type Node) and the dictionaries value being the node's priority key (of type double or double[]).

Dictionary<Node, double> (A*-based)
Dictionary<Node, double[]> (D*-based)

One of the main issues using a dictionary as an open list is sorting it. Since the list has to be sorted by the dictionary's values, and not by its keys, a SortedDictionary<TKey, TValue> class was not an option. After testing different options, it was eventually decided that instead of sorting the complete open list and taking the top item, the list would remain in an unsorted state and the node with the lowest priority key would be determined by iterating over the whole list and comparing the current key with the so far lowest one found. This method of

implementation makes the performance of the open list dependent on its length. To increase the list's performance, and thereby reduce the runtime of the algorithm, it was decided not to add nodes with a priority key of infinite to the open list.

Since adding and removing items from a dictionary, as well as finding the next node by iterating over the complete list is too expensive, later in the development the open list was implemented using the "High-Speed-Priority-Queue-for-C-Sharp" published by BlueRaja on GitHub [Blu19]. Using this priority queue two options are available: a "SimplePriorityQueue" and a "FastPriorityQueue". For details on the difference between both queues refer to [Blu19]. Since the "SimplePriorityQueue" did not show much improvement in comparison to the dictionary, the "FastPriorityQueue" is used for this thesis. This priority queue stores a priority for each item on the list. Hereby the priority queue internally implements a sorting algorithm, sorting the list each time an item is enqueued or dequeued. Implementing the open list for A* utilizing the fast priority queue instead of the previously used dictionary, showed a significant improvement of the algorithm's runtime. Therefore, the same implementation was also applied to be the final version of the open list for all other A*-based algorithms.

Using the fast priority queue for the implementation of the open list in D*-based algorithms, is not as straight forward, though. The downside to using the fast priority queue is, that only one value can be assigned as a priority. As much as that is not an issue with A*-based algorithms, a workaround had to be developed to use the same priority queue as an open list for D*-based algorithms. Therefore, it was decided that the open list would only store the first value (primary key) of D*-based algorithms' priority key. The second value is stored as a parameter in the node and updated each time the node's priority changes. Since the primary key is always compared first and the secondary key only needed if two primary keys are equal, it is more important to implement a fast comparison and sorting of the primary key, than of the secondary key. Testing has shown this implementation of the open list to be faster than the previously used dictionary and was therefore implemented in all D*-based algorithms.

The **closed list** stores all node, that have already been expanded. It does not need to store a priority key for its entries and it does not have to be sorted at any point. The main requirement for the closed list is to look up its entries fast, so it can be determined if a node has already been expanded, before it gets added to the open list. Fulfilling all this requirement, a HashSet<T> class is used for the implementation of the closed list.

A HashSet<T> is also used for the implementation of the **incons list**. It is not necessary to store a node's priority key in the incons list, because before each run of the algorithms, utilizing this list, the incons list is added to the open list and the priority key of each node in the open list is updated.

Please note; all lists were optimized up to a point where they work fairly well for the given test environment. There might be more suitable implementations, but to explore all these would go beyond the scope of this bachelor's thesis. If one were to optimize these pathfinding algorithms, reconsidering and potentially optimizing the lists, especially the open list, might be a good starting point.

4.3.1.3 Heuristic

To ensure the algorithms find the shortest path, the right focussing heuristic must be chosen. As explained in 2.2.1 at the example of A* the heuristic has to be admissible and consistent in order for A* to be optimal. Additionally, the chosen heuristic should work well in the cell-grid setup of the search graph as well as when defining the search graph using waypoints.

Requiring the heuristic to work in both search graph setups excludes some of the more common option, such as using a 3-dimensional version of the Manhattan Distance, since these would only work in the cell-grid setup. Instead it was decided to use a Euclidian Distance [Pat19]. Each node's heuristic is the straight-line distance between the node's position and the algorithm's destination. For A*-based algorithms this means the distance between the node and the goal node, for D*-base algorithms the distance between the node and the start node. For all algorithms this heuristic can be inflated by a factor $\varepsilon > 1$.

To avoid the recalculation each node's heuristic each time the node is accessed, the heuristic, as well as the inflated heuristic, is pre-calculated and stored by each node. It is important to store both, the non-inflated and the inflated, heuristic, due to AD* requiring one or the other depending on a node's consistency when calculating its key priority value. Depending on each pathfinding algorithm individually both heuristics get recalculated when the start or goal node changes, the inflation factor in- or decreases, or the algorithm has to be reset.

Having considered implementation decisions relevant for all, or most, pathfinding algorithms, following the implementation process of each algorithm is discussed briefly with regard to its specific characteristics.

4.3.2 A*-based

A*-based algorithms include A*, ARA*, and Theta*. They are all non-dynamic algorithms. This means they can not use results from previous computations to adapt to a changed search graph. Therefore, they have to be reset and recalculated from scratch each time the dynamic game environment changes. Resetting the algorithms mainly includes clearing all lists, determining a new start node and goal node, as well as updating the non-inflated and inflated heuristic.

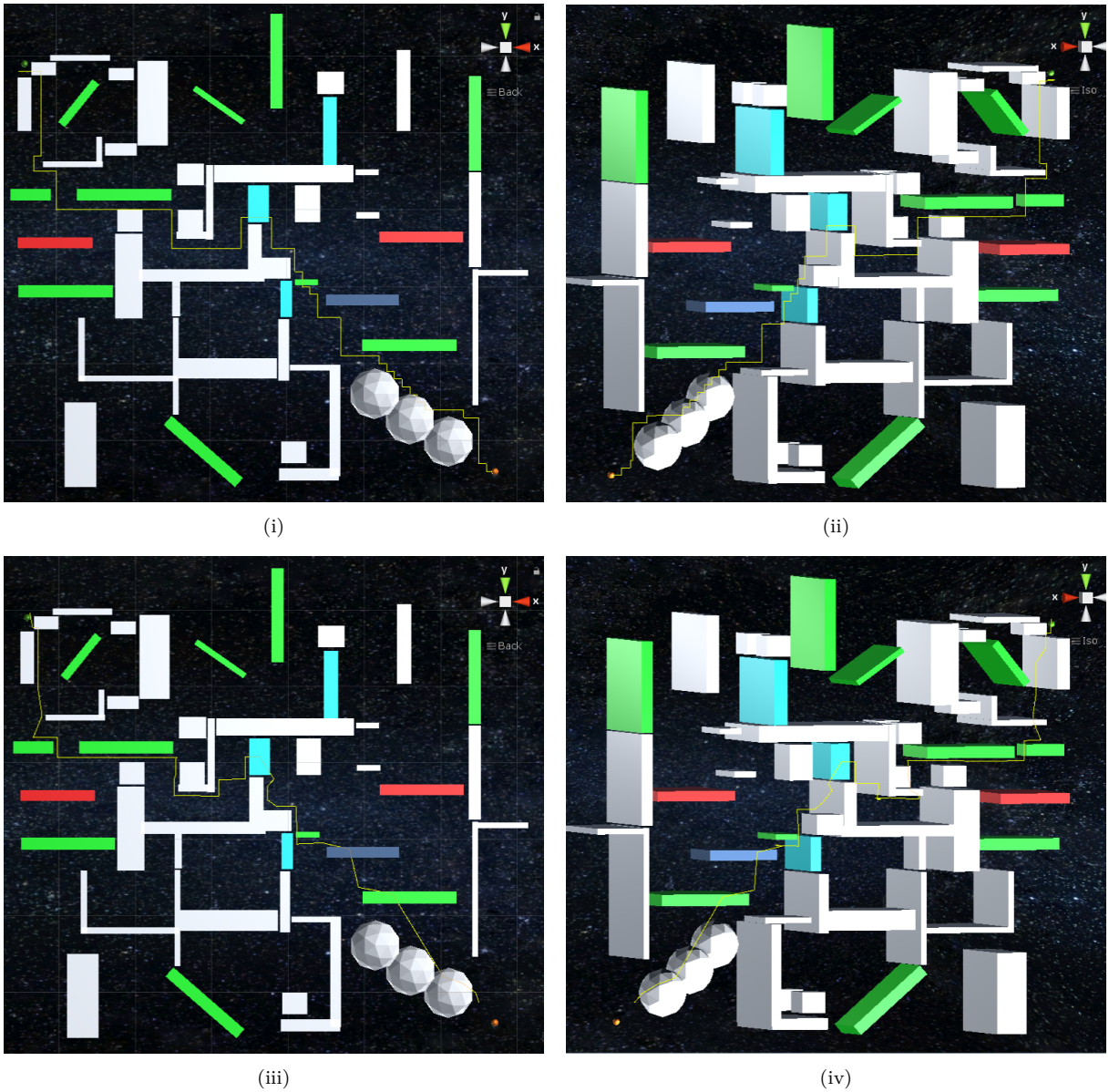
Whilst computing the shortest path each node stores its parent node upon being added to the open list. Should the priority key value of a node be updated, its parent is updated accordingly. This is done so the shortest path from the goal node to the start node, henceforth referred to as "corridor", can be determined as described in 4.4.4.

4.3.2.1 A*

The A* algorithm is implemented following its description in 2.2.1 Due to its rather simple core functionality it is the most straight forward to implement and the developer can be most certain that its calculated results are correct. Therefore, in the development process of this thesis it functioned as a reference to all other algorithms, comparing their calculations to the ones of A* to check their plausibility.

Additionally, A* with an inflation factor of $\varepsilon = 1$ initially had the longest runtime and therefore was used to test all optimization options, such as using the fast priority queue as an open list, since the impact of these optimizations was the more noticeable the greater the runtime of an algorithm was.

In Figure 42 the corridor found by A* is drawn as the yellow line. Figure 42 (i) and (ii) show the corridor found in a cell-grid search graph, Figure 42 (iii) and (iv) in a search graph using waypoints. As one can see clearly at the example of the waypoint search graph, the drawn corridor does not start exactly at the pathfinding AI's position and does not end exactly at the destination. This is due to the pathfinding algorithm only providing a corridor from the start node to the goal node, which are not necessary exactly at the same position as the AI's position and the destination. The method of determining the start node and goal node is described in 4.4.3.



(i) (ii)

(iii) (iv)

Figure 42: A* corridor (top: cell-grid; bottom: waypoints)

4.3.2.2 ARA*

ARA* builds upon A* and was mainly implemented as described in 2.2.2.

The developer can choose a starting inflation factor $\varepsilon_0 > 1$ and a decrease step $\delta > 0$ before starting the pathfinding system. ε will then be set to $\boldsymbol{\varepsilon = \max(1, \varepsilon - \delta)}$ at each run of the algorithm. To adjust for the changed ε, the inflated heuristic gets updated each time before the open list is resorted. Upon detecting changes in the search graph, the complete algorithm has to be reset and recalculated from scratch and ε is reset to ε_0.

Figure 42 shows the different paths found by ARA* with $\varepsilon_0 = 3$ and $\delta = 0.2$ in a cell-grid search graph. Hereby the orange-brown line is the one with the largest ε and the green line is the shortest path. Seemingly blurred lines in the right image are merely corridors laying together very closely.

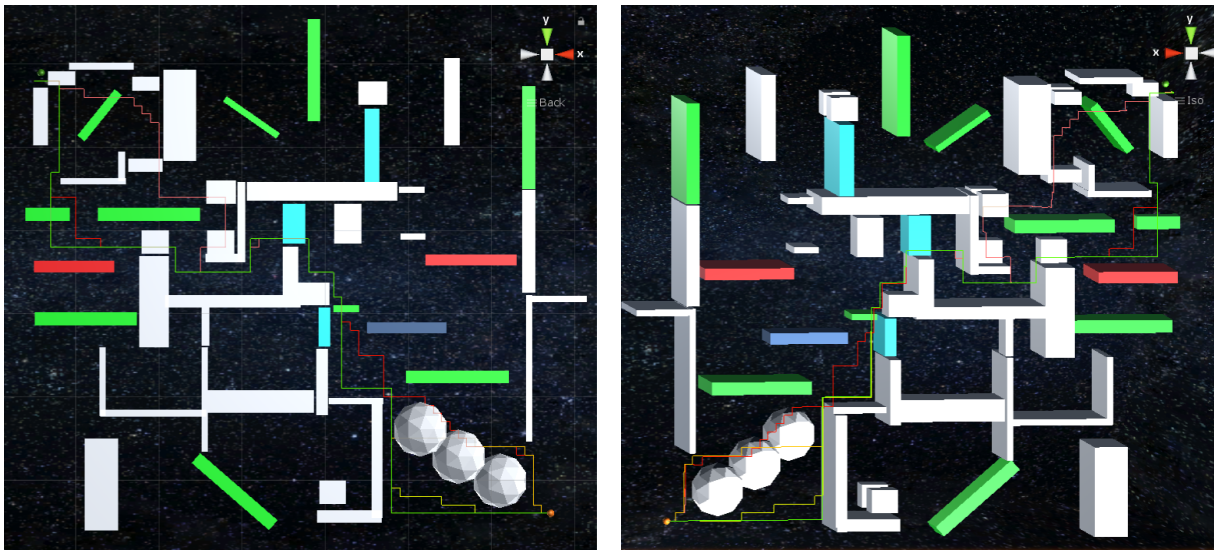


Figure 43: ARA* corridor (cell-gird)

Implementing and testing ARA* in this test environment led to the observation that the initially found path gets redrawn a few times whilst ε decreases. A new shortest path is only found once ε gets closer to 1. For the example in Figure 43 this means, that the orange-brown line is the corridor found with $\varepsilon = 2$, which is the same as the corridor found with $\varepsilon = 3 = \varepsilon_0$ and overlays all previously drawn corridors.

4.3.2.3 Theta*

Theta* is also based on A* and mainly implemented as described in 2.2.3. Unlike A* and ARA*, Theta* is not implemented to run in a background thread, but runs in a coroutine instead. The reason for this is that Theta* relies on line-of-sight checks in order to be able to find shortest paths, which are not restricted to the cell-grid or the connections between waypoints. These line-of-sight checks use Unity's physics and therefore can not be accessed from the background thread.

Initially line-of-sight checks between two nodes n and n' were run by using Unity's *Physics.SphereCast* method [Uni19c]. This method casts a sphere with a given radius along a ray from n to n' and returns which obstacles have been hit on the way. If no hits are returned, one knows that there is a line-of-sight between n and n'. Testing has shown that due to the great number the line-of-sight checks having to be run, simply implementing *Physics.SphereCast*

is too expensive. Therefore, a second method called *Physics.Raycast* [Uni19b] is run first. *Physics.Raycast* casts a ray from n to n' and returns information if any obstacles have been hit on the way. Doing so it does not consider the radius of the player, though, and therefore is less expensive than the SphereCast. Only if the Raycast determines that there might be a free path from n to n', the SphereCast is called to ensure the path is wide enough for the player to pass.

Since both, *Physics.SphereCast* and *Physics.Raycast*, are part of the Unity API, these methods can not be called from the background thread. Theta* is therefore implemented using a coroutine. As discussed earlier, using coroutines and pausing the algorithm at the end of each while-loop iteration, is not ideal. To decrease the runtime of the algorithm it was decided, that it is not necessary to pause the algorithm after each iteration, since this would waste quite a lot of resources given the simplicity of the test environment and overall game functionalities. Instead, the algorithm is only paused after a cluster of iterations. The size of these clusters should be chosen in a way that a minimum framerate of 30 FPS (60 FPS would be ideal) can be achieved. Since the optimal size of the clusters is reliant on the size of the search graph as well as the overall game functionalities, it needs to be adjusted for each game scenario individually. For the test environment used in this thesis and working with a cell-grid search graph, each cluster contains of about 1160 iterations. This results in the algorithm taking about 20 frames to finish its computation, whilst maintaining an average framerate of just under 60 FPS.

For future development it would be necessary to implement a line-of-sight check, which is not reliant upon Unity's physics and therefore can be called in a background thread. This would make the algorithm a lot more adaptable and easier to use in different game environments.

Despite this Unity-related downside of the algorithm, it is a very promising pathfinding algorithm. It finds significantly shorter paths than all other algorithms and it expands fewer nodes than A*, which should make it faster than A*, if they were both implemented in the background thread.

Figure 44 shows the corridor found by Theta* in a cell-grid search graph. It is obvious, that it is not restricted to the cell-grid, like the other implemented algorithms.

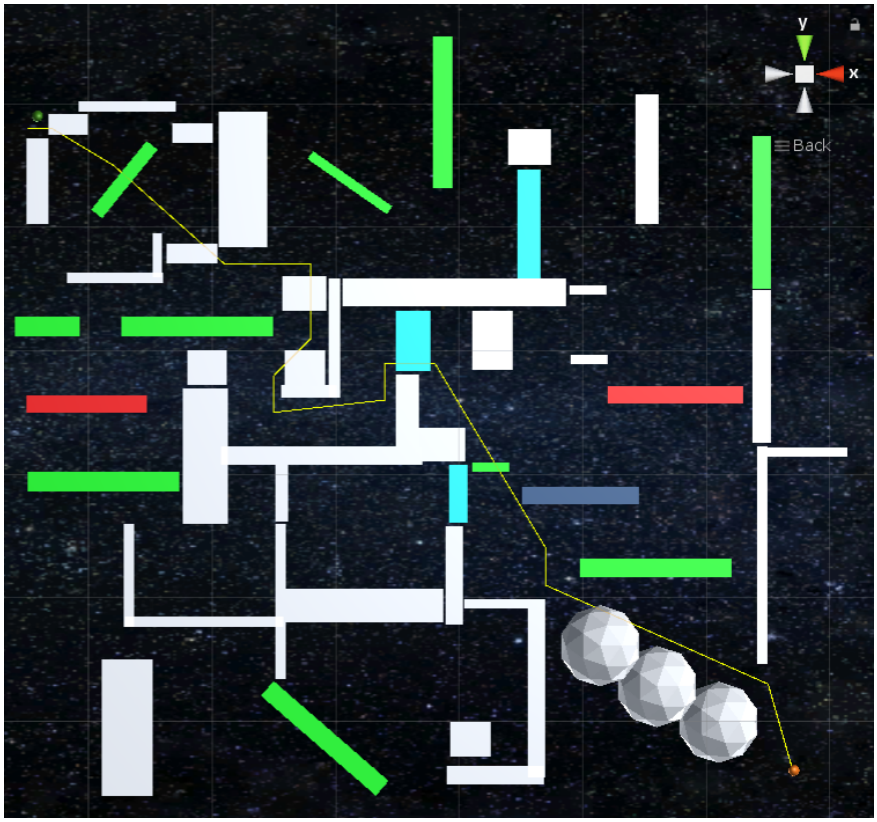


Figure 44: Theta* corridor (cell-grid)

4.3.3 D*-based

D*-based algorithms include D* Lite and AD*. These algorithms are dynamic algorithms and therefore can adapt to changed path costs in the search graph by reusing previous calculations to change their found shortest path. To achieve this they store a list of changed nodes which gets added to whilst the algorithm runs and gets cleared once the algorithm updates all changed nodes stored in that list. D*-based algorithms get reset upon the pathfinding AI colliding with an obstacle or reaching its destination. Additionally, D*-based algorithms have to be reset once any node in the search graph change its neighbours, as is the case when using waypoints (see 4.4.1), and when a new node is defined as the goal node. In addition to g-values, which are also used by A*-base algorithms, D*-based algorithms use rhs-values. Both values are set to infinite at the beginning of the pathfinding process and when the algorithm is reset.

D*-based algorithms use so-called successors and predecessors for their calculations. Successors are equivalent to the neighbours used in A*-based algorithms. For consistent denomination, successors will henceforth be called neighbours.

4.3.3.1 D* Lite

D* Lite is mostly implemented following its description in 2.2.5.

As stated in 2.2.5 a node's rhs-value is defined as:

$$\boldsymbol{rhs(n) = \min_{n' \in Succ(n)}(g(n') + c(n, n'))} \text{ [Cho10]}$$

To optimize this calculation, it is not necessary to calculate the rhs-value based on all successors first and then take the minimum value. Instead a rhs-value needs only be calculated based on a successor, if the successors g-value is lower than the currently lowest rhs-value.

Another optimization is implemented for expanding under-consistent nodes. When under-consistent nodes are expanded their g-value is set to infinite and their own rhs-values, as well as all their predecessors' rhs-values, get updated. Since a predecessor's rhs-value is only going to change if it was reliant on the under-consistent node, it is not necessary to update all predecessors, but only those, whose rhs-value were based on the under-consistent node. To implement this optimization, each node's parent is stored and when expanding under-consistent nodes only predecessors, whose parent is the currently expanded node, get updated. This optimization is also suggested in [KL02b] chapter 5.1.

In addition to these optimizations a change is made to the initial version of D* Lite regarding the calculation of the priority key value. As described in 2.2.5 the key value is calculated as:

$$\boldsymbol{k(n) = [\min(g(n), rhs(n)) + h(n_{\text{start}}, n) + k_m; \min(g(n), rhs(n))]} \text{[KL02c]}$$

Unfortunately, this does not allow the use of an inflation factor $\varepsilon > 1$. Due to the inflation factor, under-consistent nodes which get inserted into the open list after the search graph has been changed do not have a high enough priority. Therefore, the algorithm's termination condition is met too soon and the necessary nodes are not expanded. To avoid this issue, it was decided to calculate the priority key value the same way it is calculated in AD*. It is now calculated as:

$$\boldsymbol{k(n) = \begin{cases} [rhs(n) + \varepsilon * h(n_{\text{start}}, n); rhs(n)], & \textbf{if } n \textbf{ is over-consistent} \\ [g(n) + h(n_{\text{start}}, n); g(n)], & \textbf{if } n \textbf{ is under-consisten} \end{cases}} \text{ [LF+05]}$$

Figure 45 shows the corridors drawn by using D* Lite with an inflation factor $\varepsilon = 2$. In Figure 46 the size of the obstacle marked by a red circle in Figure 45 has been changed and is now blocking the previously found path. Figure 46 (i) demonstrates the corridor found by the next run of the algorithm using the initial method of calculating the priority key. It is noticeable that it does not reach the goal node. Figure 46 (ii) shows the use of the adjusted key.

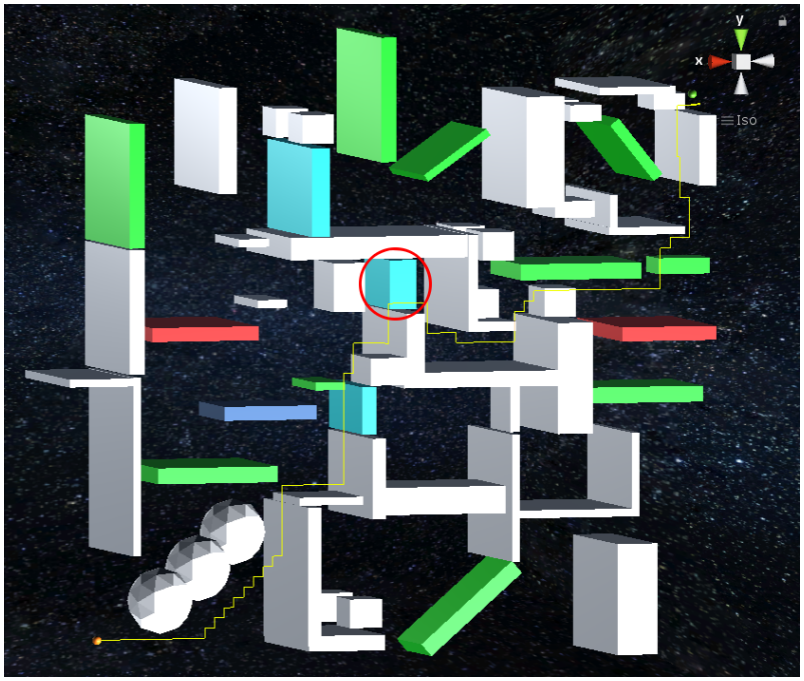
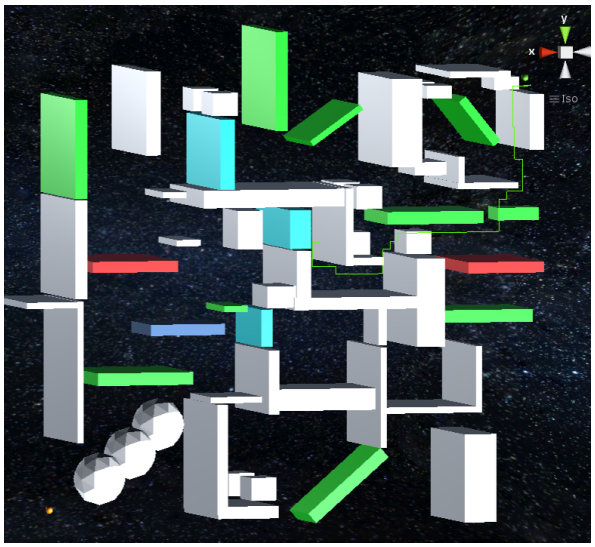
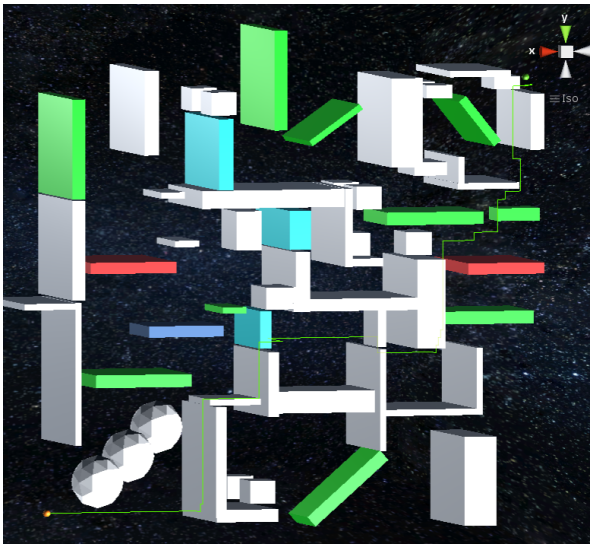


Figure 45: D* Lite with $\varepsilon = 2$ corridor (gell-grid)



(i)



(ii)

Figure 46: D* Lite with $\varepsilon = 2$ (cell-gird)

4.3.3.2 AD*

AD* is implemented following its description in 2.2.7. It combines features of ARA* with D* Lite. As with ARA*, the developer can choose a starting inflation factor ε_0 and a decrease step δ. Upon detecting changes, ε gets reset to ε_0, otherwise $\boldsymbol{\varepsilon = \max(1, \varepsilon - \delta)}$. Additionally, AD* implements all optimizations described for D* Lite in 4.3.3.1.

Figure 47 shows all corridors found by AD* with $\varepsilon_0 = 3$ and $\delta = 0.2$. The light-blue line shows the shortest path found with $\varepsilon = 1$.

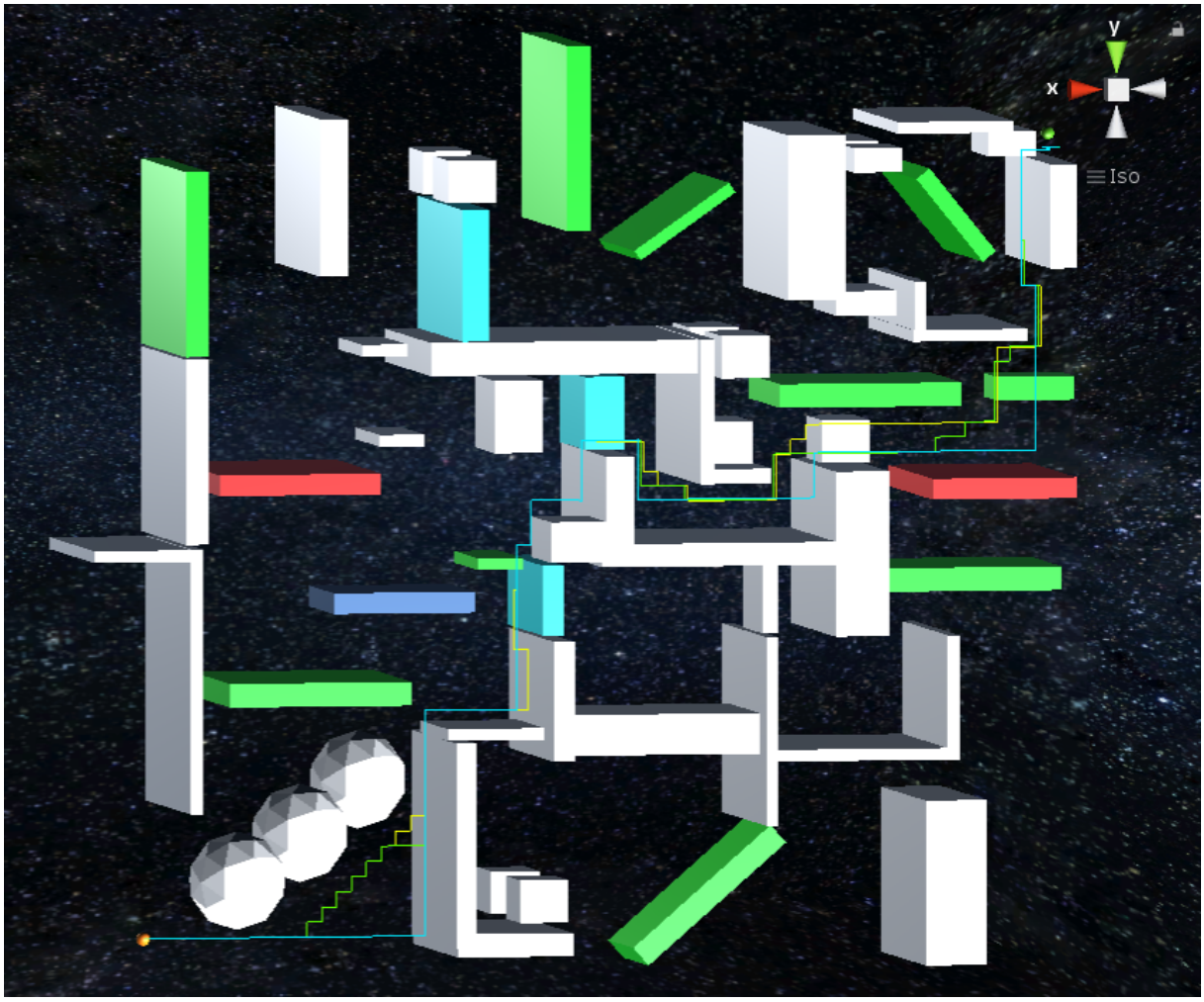


Figure 47: AD* corridor (cell-grid)

4.4 Overall process of dynamic pathfinding

As described in the previous chapter, all implementations of pathfinding algorithms in this thesis are prepared to react to changes in the game environment and therefore to changes in the search graph. Whether this is realised by resetting and recalculating the algorithm from scratch, as is the case with A*-based algorithms, or by them reacting to these changes by reusing results from previous calculations to find the new shortest path, as is the case with D*-based algorithms. The following chapter starts by discussing the process of updating the search graph to represent the changed game environment. Following, the interrelation of the changing game environment and the adaption of the pathfinding algorithms to it will be looked at further. The chapter will conclude by describing the movement of the AI through the game environment and the process of determining the pathfinding algorithms' start node and goal node.

4.4.1 Updating the search graph

For the pathfinding algorithms to react to changes in the game environment, these changes have to be translated to changes in the search graph first.

One decision to make hereby is, when the search graph should be updated and what changes it should react to. If one would work with only a few moving obstacles, or with obstacles moving irregularly or in very different speeds, it might make sense to only update the search graph when obstacles have moved a certain distance and only update those parts of the search graph which contain certain moving obstacles. This is not the case in the test environment of this thesis, though. Here all moving obstacles are moving constantly and roughly at the same speed. Therefore, it was decided to update the search graph in intervals. Each interval the complete search graph is updated. The interval time can be chosen by the developer, but is at least as long as the time required for an update of the search graph.

The main purpose of updating the search graph is to determine changes in path costs between nodes. Doing so, it has to be differentiated between a cell-grid search graph and the search graph based on waypoints. **Grid cells** do not change their position; their neighbours and predecessors always stay the same. Path costs can only change when nodes change their free/blocked status, hereby changing the path costs to the size of the cell, or to infinite. Any node changing its free/blocked status or being the neighbour of a cell changing its free/blocked status recalculates the path cost to all its neighbours and is then defined as having "changed". **Waypoints**, unlike cells, are moving constantly; in addition to their free/blocked status changing also their indirect neighbours and their indirect predecessors might change. Also, it is possible for a moving obstacle to now block the line-of-sight between two direct neighbours, setting the path cost of this connection to infinite. Therefore, all nodes have to recalculate the path cost to

all of their direct neighbours. Should any node detect any changed path costs to any of its direct neighbours, it is then defined as “changed”. Also, all nodes changing the distance to any of their indirect neighbours or getting assigned a new indirect neighbour, recalculate the path cost to their changed indirect neighbour and are defined as “changed”. Since updating the waypoint-based search graph requires multiple frames, it is possible for a dynamic obstacle to change its position during this time, rendering the updated search graph invalid. To avoid this issue, each moving obstacle creates a copy of itself, like a “shadow-self”, which is then used as the basis for updating the waypoint-based search graph. The “shadow-self’s” position gets set to the obstacles position before the search graph starts its recalculations and does not move during the updating process. As this copy is only a “shadow” the AI does not collide with it.

Since assigning new indirect neighbours changes the structure of the search graph, the previous calculations of D*-based algorithms become invalid and these algorithms have to be reset each time the game environment changes.

In both search graphs all “changed” nodes get stored in a list, which is used to inform the algorithm about which nodes have been changed.

Unfortunately, updating the search graph in both setups is reliant on Unity’s physics, for example to determine whether a node is blocked or not. Therefore, it can not be implemented on a background thread, but has to use a combination of coroutines, causing the update to be spread over multiple frames. During this time the overall setup of the search graph is not valid, due to missing information about path costs or temporarily removed connections between waypoints.

To avoid any pathfinding algorithm calculating with an invalid search graph, multiple search graphs are stored. Since the information about the setup of a search graph is stored in each node’s information about its position, its neighbours and predecessors, and the path costs to all its neighbours, multiple sets of these values are stored in each node. The first set is only used in the background by the search graph. The values in this set are the most up-to-date ones and are used and changed when updating the search graph. Since these values may form an invalid search graph at times, the pathfinding algorithm does not have access to them. The second set of values can be seen as a copy, or snapshot, of the last valid search graph. These snapshot-values are updated each time the search graph has finished an update. Due to grid cells never changing their position, neighbours, and predecessors, the corresponding snapshot-values do not have to be updated, when using a cell-grid. Also, only values of changed nodes have to be updated. Storing this second set of information, a valid search graph is available at all times, even when the search graph is currently being updated.

Testing has shown that updating the search graph using coroutines is quite time costly. To reduce the update time as much as possible, coroutines were implemented using a similar method of clustering the calculations, as is emphasized for the implementation of Theta* (see 4.3.2.3).

4.4.2 Recalculating the path

Once the search graph has been updated and a new snapshot is available, an event is triggered relaying information about which nodes have been changed to the pathfinding algorithm. Due to the runtime of the algorithms and dependent on the chosen interval-time for updating the search graph, it is possible for the search graph to update multiple times, whilst the algorithm is running once.

This fact has to be considered carefully when deciding which search graph information the pathfinding algorithm must use for its calculations. Since it is possible for the snapshot to be updated whilst the algorithm is running, this set of values can not be used. Instead, it was decided to create a third set of values, called "public" values. There is no relation between this naming and the values' access modifiers. Rather they are called "public" because they are publicly available to the pathfinding algorithm and ultimately will be used for all computations regarding the pathfinding process. "Public" values are created by copying, also referred to as "publishing", the current snapshot. Creating this last set of values is necessary to completely decouple the pathfinding process from the process of updating the search graph. This allows the search graph to be updated multiple times, taking a new snapshot after each update, without influencing the currently computing pathfinding algorithm. Similar to taking a snapshot (see 4.4.1) only values of changed nodes have to be published, and when using a cell-grid search graph only the path costs have to be published. The process of publishing the latest snapshot is run on the background thread with the rest of the pathfinding algorithm.

Additionally, the possibility of multiple search graph updates during one run of the algorithm has to be considered when managing information about changed nodes. The algorithm might not be able to react to the latest changes, before a new event with information about new changes is triggered. Therefore, it was decided to store incoming information about changed nodes and, if necessary, add it up, until the pathfinding algorithm finishes its current calculations and is ready to react to changes. Once this is the case, each algorithm reacts to all changes accordingly, as described in 2.2 and 4.3, and then computes the next shortest path using the nodes' public values.

4.4.3 Determine start and goal node

Throughout the description of all pathfinding algorithms it was always assumed the start and goal node were known to the algorithm. In order for the algorithm to know the optimal start and goal node, these have to be re-determined frequently to account for the AI and the destination moving. In this implementation of pathfinding system, the start node gets re-determined before each run of an algorithm and the goal node gets re-determined each time the algorithm reacts

to changes in the search graph. Since D*-based algorithms' computations from previous runs are based on a certain goal node, these algorithms have to be reset should a different node be defined as the goal node.

The **goal node** is defined to be the node closest to the destination's current position. To ensure that the algorithm can run on the background thread, finding the closest node can not include any of Unity's physics. For the cell-grid search graph the closest node, and thereby the goal node, is found by determining which grid cell currently contains the destination's position. In the waypoint-based search graph the closest node is found by measuring the distance in a straight line to all waypoints in the search graph and determining the closest one. Even though this method guarantees the goal node to be the closest one to the destination's current position, in theory it can not be guaranteed that there is an unobstructed line-of-sight between them. So far, this downside has not shown any difficulties in its practical application, though. Nevertheless, improving this function would be another reason speaking for the development of a non-physics reliant line-of-sight check.

Similar to the goal node, the **start node** could be defined as the closest node to the AI's current position. This method of determining the start node is only used for the first run and after resetting the pathfinding algorithm, though. At all other times the start node is determined by making a rough prediction where the pathfinding AI's position will be by the time the algorithm has finished its calculations. This prediction is necessary due to the AI constantly following its last known corridor, also during the algorithm being recalculated. If one were to use the AI's current position to determine the start node for the next calculation of the pathfinding algorithm, the AI would have already moved on by the time the algorithm finishes its calculations and would have to move backwards to reach the start node of the new corridor. Since the AI follows the last found corridor, one could be able to predict the node on the corridor, that the AI is walking towards by the time the algorithm finished its calculations and take that node as the algorithms start node. This would result in the AI moving fluently without ever backtracking to reach its new starting position. Unfortunately, this method requires the knowledge of the exact runtime of the pathfinding algorithm, which is not easily predictable, due to its dependency on the search graph, the AI's distance to the destination, the current inflation factor, and many more factors. As a compromise, in this thesis the start node is defined to either be the next node on the current corridor, which the pathfinding AI is heading towards, or the node after that. The backtracking caused by this inaccurate prediction can be minimized by using inflation factors to reduce the algorithm's runtime, when using A*, Theta*, and D* Lite. Using the anytime-algorithms ARA* and AD*, the runtime increases once ε gets close to 1, causing the AI to backtrack. This could be avoided by terminating the algorithm when a certain ε is reached, even if the sub-optimality bound is not yet 1 at this point. If, and when, to terminate the algorithm would need to be

determined for each game scenario individually by extensive testing. Testing for the object constellation in test environment as it is presented in Figure 43 has shown for example, that using ARA* with $\varepsilon_0 = 3$ and $\delta = 0.2$ in a cell-grid search graph, the shortest path is already determined with $\varepsilon = 1.8$.

A thinkable alternative to avoid the inaccurate prediction of the next start node, would be to run the pathfinding algorithm first and to then compare the old corridor to the new one. Using this method the path the AI has to take to transition from the old corridor to the new one, without having to backtrack, could be determined. This option was carefully considered to be implemented in this bachelor's thesis but was dismissed due to the transition proving to be more complex than initially assumed.

For most algorithms implemented in this thesis, choosing the next corridor-node, which the AI is currently heading towards, as the next start node has shown good results and was therefore implemented. Due to dynamic obstacles moving into the current corridor, two optimizations were also added.

The first optimization was made due to the possibility, that the next corridor-node might have become blocked during the last update of the search graph and therefore is not a valid start node. Should this node be blocked, the start node is defined by finding the closest node to the AI's current position, as described above. In the case of using a cell-grid search graph, it is possible that also the closest node is blocked. Should this be the case, it is determined whether any of its neighbours are free and thereby a valid start node. Even though checking the neighbour of the closest node might lead the AI to move diagonally in the cell-grid, it has shown to improve the success rate of the AI reaching its destination.

A second optimization is regarding finding the next start node when using Theta*. Since Theta* is not restricted to connections between nodes, it is possible for the nodes in the corridor to be further apart. Whilst the AI is traversing from its current position to the next corridor-node, which is also the start node for the next run of the algorithm, it can not be controlled and therefore can not change its direction. The greater the distance between two corridor-nodes and therefore the greater the distance the AI has to travel without being controlled, the more likely it is for a moving obstacle to move into the AI's path and block it. Therefore, Theta* only takes the next corridor-node as the start node for its next calculation, if there is a line-of-sight between the AI's current position and that node. Should this not be the case, a new start node is determined by finding the closest node to the AI's current position, as described above.

4.4.4 Movement of the AI

Once the pathfinding algorithm terminates successfully, the found solution has to be translated into a walkable path the AI can follow.

Primarily the AI follows the so-called corridor. The corridor is a list of nodes, that describes the shortest path from a given start node to a given goal node, through the search graph. In the case of A*-based algorithms, the corridor is traced by starting at the goal node and following each node's parent until the start node is reached. For D*-based algorithm one starts at a start node n and determines the next node n' as

$$\boldsymbol{n' = \min_{n' \in \mathrm{Succ}(n)}(g(n)' + c(n, n'))} \text{ [KL02c]}$$

until the goal node is reached. As a last step the pathfinding algorithm informs the pathfinding AI's controller, that a new corridor is available.

Receiving a new corridor, the AI first has to move from its current position to the start node of the corridor. It then moves along the corridor until the goal node is reached or it collides with an obstacle. The AI walks along the corridor by always removing its first node and aiming towards that node's position. Once it reaches this position, it discards this one and aims towards the next node in the corridor. When the AI reaches the goal node it ensures that there is a line-of-sight between its current position and its destination. If this is the case, it moves in a straight line to its destination and gets reset upon reaching it. The AI also resets upon collision with obstacles and informs the pathfinding algorithm, that it has to reset, as well.

Initial tests showed, that the AI had a very low success rate of reaching its destination because of collisions with dynamic obstacles, regardless of the used pathfinding algorithm. The main cause was the obstacles moving into the AI, before the search graph had finished its update. To decrease this risk of collision it was tried to force the algorithm to find paths, which have a "safety distance" to any moving obstacles, especially in the direction of their movement.

In the case of using a cell-grid search graph this was achieved by assigning each moving obstacle a second collider, slightly larger than its first, and especially stretched in the direction of the obstacle's movement. The second collider is taken into account when updating the search graph, marking additional nodes around the obstacle as blocked, but does not affect the AI. Even though this method of implementation might increase the path length by forcing paths to leave extra space to moving obstacles, it increases the success rate of the AI significantly. Therefore, it was decided to implement this method for all algorithms running on the cell-grid search graph.

A similar approach was tested in the waypoint-based search graph, creating nodes in greater distance to their object. This did not show any improvement, though, and was therefore not

implemented in the final version.

Additionally, the behaviour of the appearing and disappearing obstacles (light blue) was altered, so they can only change their status, when the AI is not near them.

5 Comparison

After all pathfinding algorithms have been implemented, in this chapter they will be tested and compared in two steps using the abstract test environment described in 4.1.2. To assess each algorithms' performance the main criteria are the length of its found paths, the total distance the AI travels from its start point to its destination, and the number of expanded nodes. As a third factor the time required for computing the shortest path is considered.

In both comparison-steps each algorithm is tested in the cell-gird search graph, as well as the waypoint-based one. A*, Theta*, and D* Lite are run with an inflation factor of $\varepsilon = 1$ and $\varepsilon = 2$. ARA* and AD* are run with a start inflation factor $\varepsilon_0 = 3$ and a decrease factor $\delta = 0.2$ and $\delta = 0.4$. The interval time between updating the search graph is set to zero. This means, the search graph re-starts updating as soon as it finishes its last update.

All testing and comparison was done using a Predator Helios 300 Laptop by Acer, with a Nvidia GeForce GTX 1060 graphics card and an intel core i7 processor.

5.1 Static AI and destination

The first step in testing and comparison aims at getting a first impression of each algorithm's performance without factoring in any external variance, such as a decreasing path length due to the AI moving towards its destination. Therefore, in this stage all dynamic obstacles in the test environment are moving, but the AI and the destination are static.

The following figure shows the average value for each algorithm's **path length**, number of **expanded nodes**, and **time** needed for computation, in the cell-grid search space (left) and waypoint search graph (right). Each diagram displays the respective values of each algorithm in the form of a bar chart. Each chart is split into a left and right group. The left group represents the non-anytime algorithms with an inflation factor $\varepsilon = 1$, and the anytime algorithms with a start inflation factor $\varepsilon_0 = 3$ and a decrease factor $\delta = 0.2$. The right group in each diagram represents the algorithms with an inflation factor of $\varepsilon = 2$, or with a decrease factor of $\delta = 0.4$ respectively. Path lengths used for calculating the average of non-anytime algorithms are defined as the length of the current corridor from its start node to its goal node. Since anytime-algorithms can find multiple paths whilst their ε is decreasing the corridor found last before ε is reset, either due to resetting or reacting to changes in the search graph, is used for calculating the average path length. In addition, the number of expanded nodes and the time required for calculations are each cumulated throughout the decrease of ε, and all cumulated

values are used for determining the average. For larger printed diagrams see Figure 58 to Figure 63 in the Appendix.

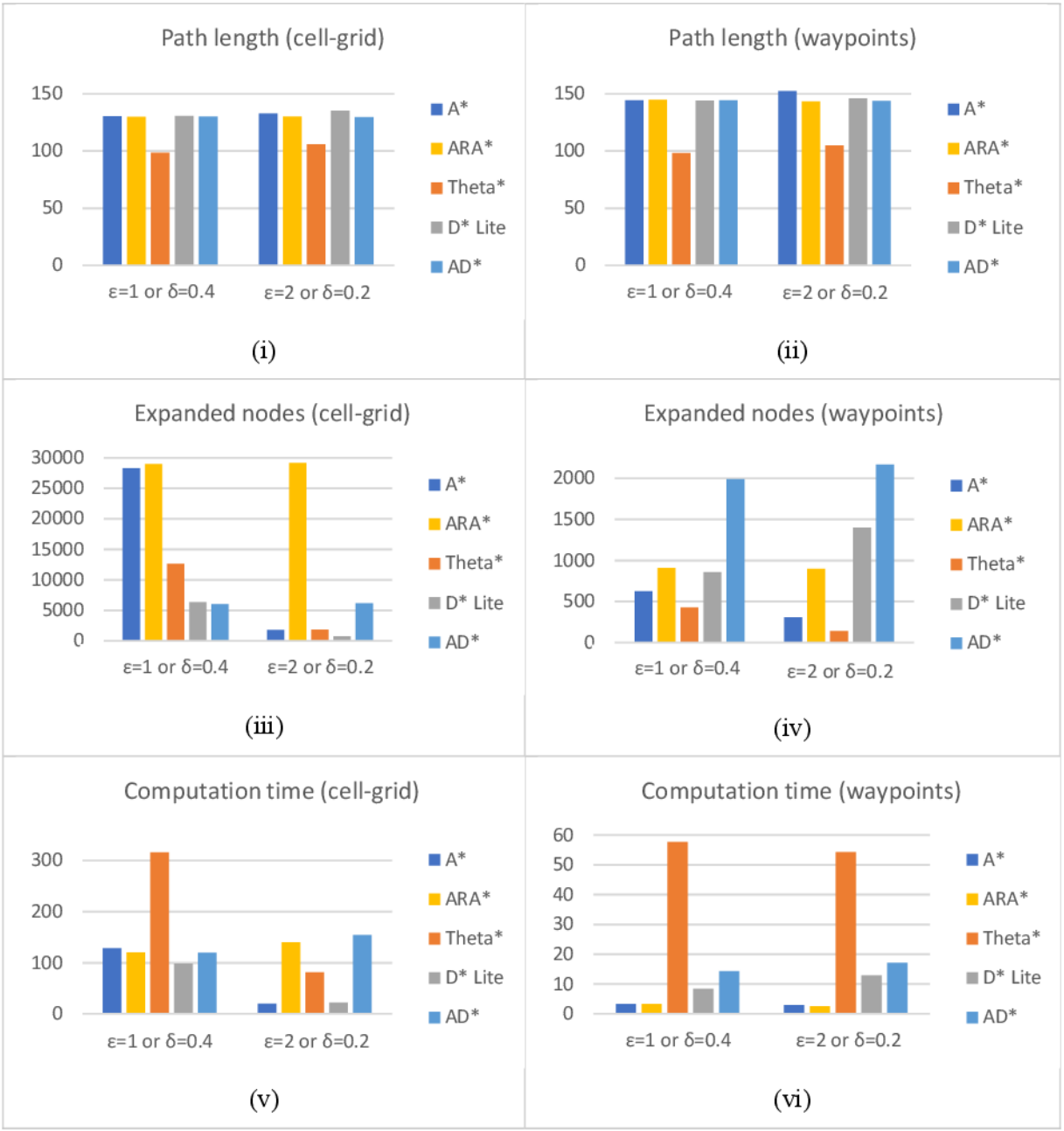


Figure 48: Comparison of path length, expanded nodes, and computation time in a cell-grid and waypoint-based search graph

Starting at the top of Figure 48, the first line shows the average **path lengths** found by each algorithm in the cell-grid search graph (left) and in the waypoint-based search-graph (right). It can be seen, that the path-length differs slightly between the two search-graph setups, whereby most algorithms find shorter paths in the cell-grid setup. For example, A* with $\varepsilon = 1$ finds an

average shortest path of 130.28 units in the cell-grid and an average shortest path of 144.126 units in a waypoint-based setup. The only exception to this is Theta*, which produces almost identical path lengths in both search graphs. Given the path length is measured from the start node to the goal node of a corridor, it is not necessarily the exact distance the AI would have to travel from its current position to its destination. For a more accurate comparison the difference of finding start and goal nodes in both graphs should therefore be considered. As described in 4.3.2.1 and as can be seen in Figure 42, the start node and goal node are mostly closer to the current AI's and destination's position in a cell-grid setup, than in the waypoint search graph. Therefore, given equally long measured path length, the cell-grid search space in most cases produces shorter actually travelled distances of the AI.

Comparing average path lengths of all algorithms utilizing two different inflation factors ε, and respectively two different decrease factors δ, Figure 48 (i) and (ii) also show rather small variance in path lengths with varying factors. Hereby the variance is greater in a waypoint setup of the search graph than when utilizing a cell-grid. A*, for example, only increases its path length from 144.126 units to 152.316 units in this setup (see Figure 48 (ii)).

Moreover, given the same search graph, most algorithms find path lengths in close proximity to each other. The main exception hereof is again Theta*, which finds significantly shorter path lengths than any other algorithm. For example, in a cell-grid search space using an inflation factor $\varepsilon = 1$ A*'s average shortest path is 130.28 units, whereas Theta*'s average shortest path is 98.28 units.

Next, the number of **expanded nodes**, see Figure 48 (iii) and (iv), and the **time** required by each algorithm to complete its computations, see Figure 48 (v) and (vi), are assessed. Firstly, it should be noted that the waypoint-based search graph in its setup used for testing in this thesis consists of 2062 nodes, whereas the cell-grid consists of 57344 nodes. This difference reflects in the number of expanded nodes and therefore also in the time needed for computation. The maximum average number of nodes expanded by any algorithm in the cell-grid is 29141.04, more than ten times the maximum average number of nodes expanded in the waypoint search graph, which is only 2168.02.

Analysing Figure 48 (iii) and (v) one can see some of the algorithms' characteristics described in previous chapters quite clearly. For one A*, Theta*, and D* Lite expand significantly fewer nodes utilizing an inflation factor $\varepsilon = 2$ than with $\varepsilon = 1$. Secondly, Theta* inflates significantly fewer nodes than A* given $\varepsilon = 1$, even though they share most of their basic setup except for calculating the g-value and setting parent nodes. Nonetheless Theta* still does take longer to complete its computations than A* (see Figure 48 (v)), due to it being implemented in coroutines instead of a background thread. It should also be noted, that the dynamic algorithms D* Lite

and AD* expand less nodes than their non-dynamic counter-parts A* and ARA*, regardless of the inflation or decrease factor.

Since nodes in the waypoint-based search graph get reassigned new neighbours frequently, as described in 4.3.3 and 4.4.1, dynamic algorithms have to be reset too often in this setup. Therefore, they also do not expand fewer nodes than their non-dynamic counter-parts (see Figure 48 (iv)). This is in turn reflected by the computation time required by each algorithm shown in Figure 48 (vi). Here the dynamic algorithms take longer to find a shortest path than the non-dynamic algorithm A* and ARA*.

In conclusion, the cell-grid search graph results in overall shorter path lengths, but also requires algorithms to expand more nodes and spent more time on computations than in a waypoint-based search graph. Moreover, dynamic algorithms perform well in the cell-grid setup and for a waypoint-based search graph non-dynamic algorithm, such as A* with $\varepsilon = 2$ have shown to expand fewer nodes and require little computation time. Theta* is an interesting algorithm, due to its short path lengths and low number of expanded nodes. Its extensive computation time might impact the end-result of the pathfinding system negatively, though, and therefore has to be observed over the course of further testing and comparison. Lastly, the average number of nodes expanded by the anytime-algorithms ARA* and AD* in both search graph setups, as shown in Figure 48 (iii) and (iv), varies slightly with the use of different decrease factors δ and mostly tends to expand more nodes when using $\delta = 0.2$.

5.2 Fully dynamic

So far, all testing and comparison has been done in a setup of the abstract test environment, where only the dynamic obstacles are moving, but the AI and the destination are static. In this next step these are made dynamic, too. Hereby the destination follows a pre-defined path travelling along the y-Axis (in other words, up and down) at the left side of the environment. To avoid an infinite chase of the AI and the destination, it has to be ensured the AI is allowed to move faster than the destination. Additionally, all dynamic obstacles also move slower than the AI. Doing so ensures that the AI theoretically is able to avoid collision with obstacles by moving away from them without them catching up.

Being the most decisive factor in choosing a suitable pathfinding algorithm for a dynamic game environment, the success rate of all algorithms tested in both search graph setups will be compared first. Only those algorithms found possibly suitable will then be compared in more detail by determining and comparing an average of the total distance the AI travels from its

start point until it reaches its destination, an average of the total number of expanded nodes, and an average of the total time required for computation of the algorithms.

Figure 49 shows the success rate of all algorithms in a cell-grid search graph (left) and a waypoint-based search graph (right) in percent. The success rate results from the number of successful runs of the AI relative to the total number of runs. Each run of the AI starts with the AI being at its starting position and ends with the AI resetting. A run is defined as a success, if the AI reaches its destination, otherwise it is failed. For larger printed diagrams see Figure 64 and Figure 65 in the Appendix.

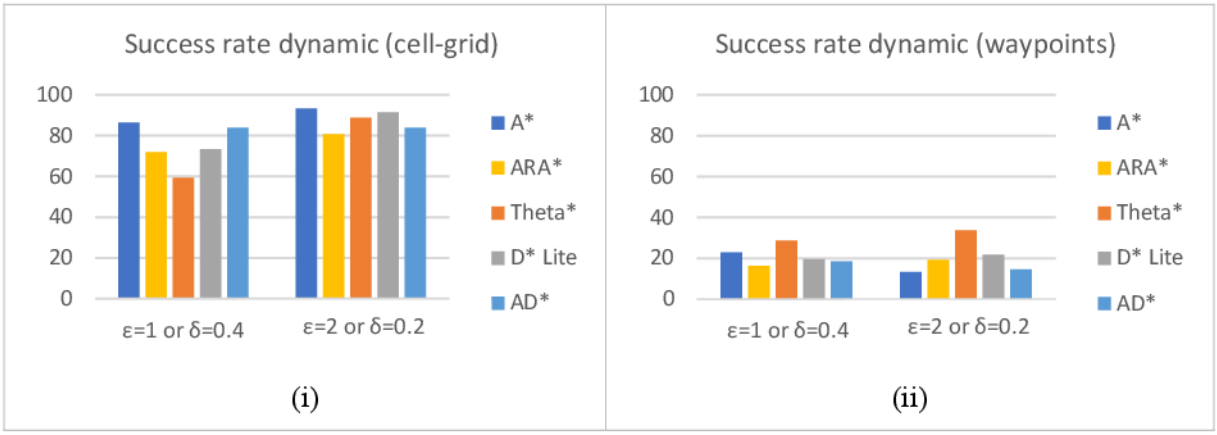


Figure 49: Comparison of success rate in cell-grid and waypoint-based search graph

The graphs in Figure 49 clearly show that the success rate of all algorithms is less than 40% in a waypoint search graph. The highest success rate achieves Theta* with $\varepsilon = 2$, which lays at 33.594%. Observing the behaviour of the AI moving along its corridor, it becomes clear that most collisions were caused by the algorithm not being able to react to changes in the game environment fast enough. Considering that previous comparisons have shown algorithms to require significantly less computation time in a waypoint setup than in a cell-grid, one might have expected the success rate shown in Figure 49 to be reverse. To understand what causes the low success rate in the waypoint setup one should have another look at the computation time required by the algorithm. Figure 50 shows variations of the graphs seen in Figure 48 (v) and (vi). Here a red line is added, indicating the average time needed by the search graph to complete its update and make a new search graph available to the pathfinding algorithm. In the case of using a cell-grid search space, it takes about 282.15 milliseconds to complete the updating process. The waypoint-based search graph takes 327.71 milliseconds to complete its update. As one can see, updating the search graph requires a lot more time than computing the algorithms, except for Theta* with $\varepsilon = 1$ in a cell-grid search graph. Therefore, the runtime of a pathfinding algorithm is not necessarily the deciding factor on how well the pathfinding system can react to changes in the game environment. To optimize the pathfinding system's success

rate, especially in the waypoint-based search graph, the priority should be to first optimize the time needed for the search graph to update.

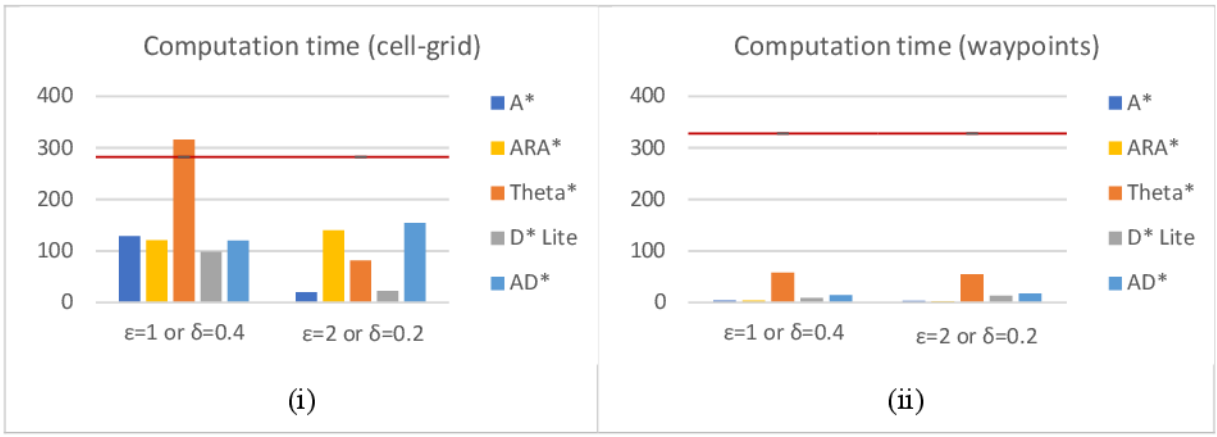


Figure 50: Comparison of computation time and search graph update time

Comparing the success rate of all algorithms in the cell-grid search graph, it can be seen that non-dynamic algorithms are more successful when using an inflation factor of $\varepsilon = 2$ than when running with $\varepsilon = 1$. Additionally, results shown in Figure 49 (i) indicate that the success rate of Theta* is not significantly negatively impacted by its extensive computation time. Especially with $\varepsilon = 2$ it is only little less successful than A* and even more successful than ARA* and AD*.

Given the low success rate of the waypoint-based search graph update, this form of setup is not a realistic option at this state of development and will be excluded from all following comparisons. For statistics on algorithms' performance in the waypoint search graph see Figure 72 to Figure 77 in the Appendix.

Next to having a high success rate, a suitable shortest path algorithm in a dynamic environment should enable the AI to travel as short a distance from its start position to its destination as possible. Figure 51 shows the average distance the AI travels in a fully dynamic game environment utilizing a cell-grid search graph from its start position until it reaches its destination. As before the distance is measured in Unity units and a larger print can be found as Figure 66 in the Appendix. In addition, graphs showing representative data-points of travelled distances can be found in the Appendix, Figure 69.

One of the first noticeable observations is, that the AI travels a shorter distance when A*, Theta*, and D* Lite are utilizing an inflation factor of $\varepsilon = 2$ than when they are not inflated. For example, the average distance of A* with $\varepsilon = 1$ is 177.373 units, with $\varepsilon = 2$ it is 160.109 units. Given the measured results seen in Figure 48 (i) and the general tendency of a higher inflation factor leading to greater path lengths, this result is surprising at first.

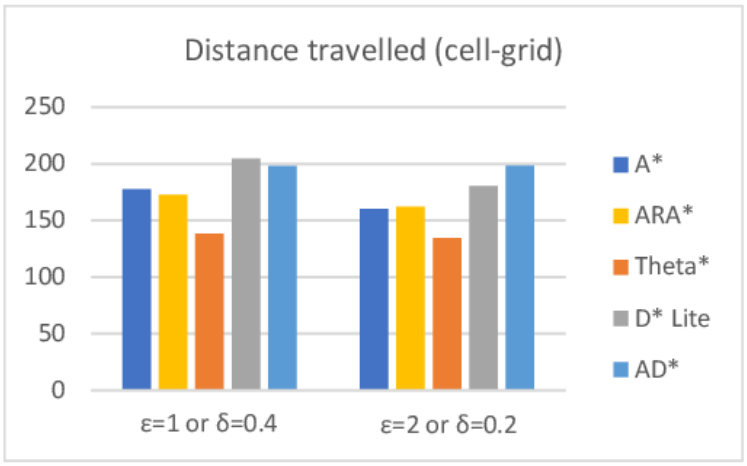


Figure 51: Distance travelled in a cell-grid search graph

Observing the movement behavior of the AI in the dynamic environment it was concluded, that this difference in path length is caused by the AI backtracking. As discussed in 4.4.3 the AI is caused to backtrack to the start node of its new corridor, if this has not been predicted correctly. The longer an algorithm requires to finish its computations, the more likely the AI is to backtrack. As Figure 48 (v) shows, non-anytime algorithms are faster when using $\varepsilon = 2$ and therefore the AI backtracks less, resulting in shorter travelled distances. In addition to causing longer paths, backtracking also causes the AI's movement to feel stuttering and increases the likelihood of collisions with dynamic obstacles.

For further comparison the average total number of nodes expanded and the total time spent on computing a pathfinding algorithm during a successful run of the AI are determined and presented in Figure 52. For larger prints see Figure 67 and Figure 68 in the Appendix. Additionally, Figure 70 and Figure 71 in the Appendix show representative data points of measuring the total number of expanded nodes and total computation time.

Figure 52: Total number of expanded nodes and total computation time in a cell-grid search graph

As expected based on previous observations, A*, Theta*, and D* Lite expand less nodes with $\varepsilon = 2$ than with $\varepsilon = 1$, Theta* expands less nodes than A*, anytime algorithms expand more nodes using $\delta = 0.2$ then with $\delta = 0.4$, and dynamic algorithms D* Lite and AD* expand fewer nodes than their non-dynamic counter-parts. Due to the destination moving, a new goal node has to be defined frequently, causing the dynamic algorithms to reset. Therefore, the difference in expanded nodes between for example A* and D* Lite is not as significant as with the static destination as seen in Figure 48 (iii). For comparison Figure 53 shows the total number of expanded nodes in a setup, where all dynamic obstacles and the AI are moving, but the AI's destination is static. Here the goal node is consistent and the dynamic algorithms do not have to be reset. As one can see in this setup the dynamic algorithms perform much better than in an environment with a dynamic destination.

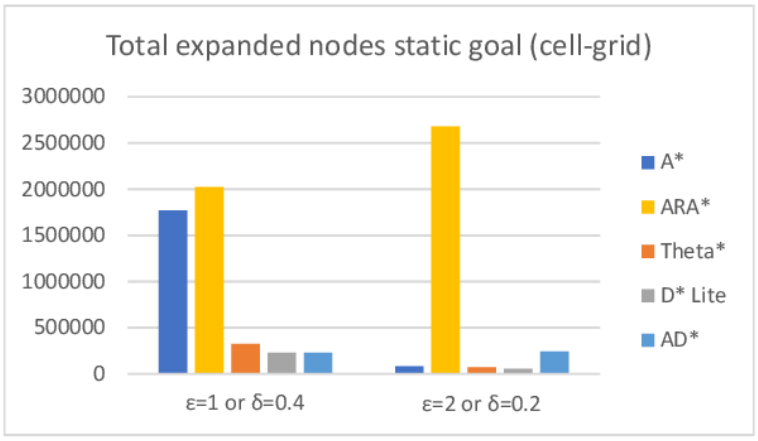


Figure 53: Total number of expanded nodes in a cell-grid search graph with static destination

Considering all factors, the success-rate, the travelled distance, the number of expanded nodes and the time required for computation, two facts become clear. First, using an inflation factor is necessary, and secondly the use of an anytime algorithm is not necessary. The first fact is supported by Figure 49 (i) indicating a higher success rate, Figure 51 showing shorter travelled distances, and Figure 52 illustrating less expanded nodes and lower computation time for A*, Theta*, and D* Lite using $\varepsilon = 2$ compared to $\varepsilon = 1$. The same figures also show, that ARA* and AD* do not have a higher success rate, do not enable the AI to travel shorter distances, do not expand less nodes, or require less computation time than non-anytime algorithm using $\varepsilon = 2$. The unexpectedly low success rate of these two algorithms is caused by the issue of backtracking. Should this issue be resolved, their characteristic of finding a first solution very fast might impact their success rate and travelled path length positively. Therefore, it can be said, that for this fully dynamic test environment choosing A*, Theta* or D* Lite are the best options.

Table 1 shows a comparison of these algorithms listing all factors discussed in this chapter. To gain a quick overview fields are highlighted in green, yellow, or red. Red indicates this value ranking lowest, yellow middle, and green best.

	Success rate	Tavelled distance	Expanded nodes	Computation time
A* $\varepsilon = 2$	93.277	160.109	360037.395	3037.616
Theta* $\varepsilon = 2$	88.722	134.303	203758.367	8050.889
D* Lite $\varepsilon = 2$	91.367	180.144	129771.117	5987.049

Table 1: Comparison A*, Theta*, and D* Lite with $\varepsilon = 2$

In conclusion, using a cell-grid search graph and A* with an inflation factor of $\varepsilon = 2$ showed the best results in the abstract test environment used in this thesis. Theta* and D* Lite are also interesting candidates but do need further optimizations. Theta* does find shorter and smoother-looking paths than any other algorithm, but has to be developed further so it can be run in the background thread in order to be able to compete with those. D* Lite performs well in dynamic environments with a static destination but has to be optimized to increase its performance when reacting to a changed goal node.

This conclusion contradicts the expected results stated in 3.5. As expected, Theta* does produce the best paths, but is currently not the best solution, due to its extensive computing time. Also, AD* is not the optimal solution. The performance of anytime algorithms, such as AD*, is worth reassessing once the issue of backtracking can be resolved.

6 Conclusion

At this point all algorithms have been implemented and compared. Initially it had been planned to use the conclusions draw from the previous chapter to implement the most promising pathfinding algorithm and search graph in the game "Lost in Space", to test it in a real game setup. Due to unexpected challenges developing and optimizing the search graphs, especially the waypoint-based one, and optimizing the implementation-details of the algorithms, this last step can not be performed in the scope of this bachelor's thesis. Therefore, this final chapter will conclude this thesis by first giving a brief summary, then drawing conclusions from the implementation and comparison of the pathfinding algorithms, and finally by looking ahead at extensions and optimizations that could be performed on the existing results and the pathfinding system in general.

6.1 Summary

The motivation for this bachelor's thesis was to identify which algorithm is most suitable to be implemented in the highly dynamic game environment of "Lost in Space". To do so, multiple pathfinding algorithms should be implemented and compared.

In the second chapter (2) of this thesis the game "Lost in Space" is introduced (2.1), and the pathfinding algorithms A*, ARA*, Theta*, D*, D* Lite, Field D*, and AD* are explained on a theoretical level (2.2). Additionally, already existent and commonly used pathfinding systems in games and especially those available for the game engine Unity are presented (2.3).

The thesis then continues by drafting a concept for the project execution in chapter three (3). Here it is also defined, that the algorithms A*, ARA*, Theta*, D* Lite, and AD*, are implemented and compared in an abstract dynamic test environment using two different search graph setups.

Chapter four (4) starts by describing the implementation of the game "Lost in Space" (4.1.1) and the setup of the abstract test environment (4.1.2), followed by the development of the cell-grid and waypoint search graph (4.2). Once the search graphs are set up, the implementation of the pathfinding algorithms is presented (4.3). Here first overall implementation decisions are discussed, including the use of a background thread, the design of the open, closed, and incons list, and the chosen focussing heuristic. Afterwards the implementation process of each algorithm is briefly discussed individually. Chapter four concludes by describing the overall process of the pathfinding system adapting to changes in a dynamic game environment (4.4). This includes the

update of the search graph, the pathfinding algorithm managing and adapting to changes in the search graph, determining new start and goal nodes, and moving the AI through the dynamic game environment.

Finally all algorithms are tested and compared in chapter five (5). This comparison is done in two steps; the first with a static AI and destination, the second in a fully dynamic test environment. Mainly each algorithms' path length, number of expanded nodes, time required for computation, and the success rate of the AI reaching its destination is compared.

6.2 Conclusion

Throughout the process of implementing and comparing the pathfinding algorithms A*, ARA*, Theta*, D* Lite, and AD* in a cell-grid and a waypoint-based search graph using an abstract dynamic test environment, two main conclusions can be drawn.

First, using a waypoints-based search graph is not a realistic option at this stage of development. The upside of the waypoint search graph is its low number of nodes. This causes the algorithms to expand significantly fewer nodes than when using a cell-grid search graph, which is also reflected in the required computation time of the algorithms. Moving the AI through the dynamic environment, though, the highest success-rate reached is 33.594% by Theta* with $\varepsilon = 2$. The main reason for collisions is the time required by the search graph to update. As this is far higher than the time required by any algorithm, it is currently the limiting factor in lowering the time the pathfinding systems need to react to changes in the environment. To improve the success rate of a pathfinding system, optimizing the search graph, especially its update, should be the first priority. Even though in a cell-grid search graph the difference in time between updating the search graph and running the pathfinding algorithms is not as significant, in this case too, optimizing the search graph is necessary to improve the AI's success rate.

Secondly, at this stage of development A* with an inflation factor of $\varepsilon = 2$ is the most suitable algorithm to be implemented in the abstract test environment using a cell-grid search graph. It has a high-success rate, enables the AI to travel relatively short paths, and has a low computation time. Another benefit of A* is its simplicity. It is easy to understand, implement, and maintain.

Other promising algorithms are Theta* and D* Lite. Theta* finds shorter paths then A* and expands less nodes, but at this stage of development has a high computation time. D* Lite expands fewer nodes than A* but also has a higher computation time. In its current implementation it performs well in dynamic environments with a static destination but has to be optimized further for optimal performance in a fully dynamic environment.

6.3 Remaining issues

Having finished the implementation of the search graphs and the pathfinding algorithms within the scope of this thesis, a few issues remain.

The most noticeable is, that any functions utilizing Unity's physics can not be called from the background thread. This does not only affect Theta*, due its reliance on line-of-sight checks, but also the updating of both search graphs and the determining of start and goal nodes in the waypoint-based search graph.

Another issue is the occurrence of backtracking, which in the scope of this bachelor's thesis is not investigated further. At this stage of development, backtracking is merely avoided by increasing the inflation factor of non-anytime algorithms. Backtracking does not only increase the path the AI has to travel, but also makes the AI's movement look staggering and increases the risk of collision.

A third issue are very irregularly and seemingly randomly occurring frame drops. So far, testing has shown these frame-drops to be caused by updating the search graph. As this issue does not appear during every update and not even every time the pathfinding system is run, it is hard to pinpoint the exact source of this issue.

6.4 Looking ahead

Concluding this bachelor's thesis, this chapter will give a look-ahead on next steps and possibilities of extending and optimizing this project in the future.

The most important next step would be to use the conclusions drawn from this thesis to implement a pathfinding system in the game "Lost in Space". The original plan to do so as part of this bachelor's thesis had to be cancelled, due to unexpected challenges occurring during the development of the search graphs and the implementation of the pathfinding algorithms, which therefore demanded significantly more development time than originally scheduled. When implementing any pathfinding algorithm into an actual game environment, one should also consider using a method of post-smoothing the found path to make it look smoother and more natural.

Another interesting next step would be to optimize, or potentially even completely re-design, the used search graphs. The cell-grid search graph used in this thesis has the upside of being simple to understand, implement, and update. This simplicity comes at the cost of having a lot of nodes in the search graph, which in a worst-case scenario all have to be expanded by pathfinding algorithms. Therefore, optimizing the cell-grid search graph by lowering its number of cells

would most likely affect the performance of the pathfinding algorithms positively. To decrease the number of nodes the implementation of a waypoint-based search graph was chosen in this thesis. Even though this did decrease the number of expanded nodes and the computation time of the algorithms significantly, it was not a successful solution. For future use of a waypoint-based search graph the design proposed in this thesis has to either be re-worked or at least the updating time significantly optimized to achieve a higher success rate.

Next to optimizing the updating time of the search graph, also the computation time of the pathfinding algorithms has to be as low as possible to perform well in a highly dynamic game environment. One possibility to decrease the computation time of all algorithms would be to reassess the setup of the open list, especially in the case of D*-base algorithms. Also, to improve the performance of D*-based algorithms in an all dynamic environment, which includes a dynamic destination, optimizations, such as the use of Moving Target D* Lite, presented by Xiacun Sun, William Yeoh, and Sven Koenig in [SYK10], should be considered. To improve Theta* a line-of-sight check not reliant on Unity's physics would have to be developed. Additionally, testing has shown, that by restricting the anytime algorithms ARA* and AD* not to decrease their ε until reaching 1, but only until 1.4, they backtrack less, and the AI moves smoother. Of course, this way they are not guaranteed to find the optimal shortest path but might have a sub-optimality of 1.4.

Thinking more generally a possible option to increase the pathfinding systems' success rate would be to enable them to make predictions about which position dynamic objects will have by the time the algorithm will have finished its computations. This way the shortest path found is up-to-date with the changed environment, instead of based on an old, and no longer accurate, representation of the changed environment.

Another interesting general idea would be to combine an any-angle algorithm, such as Theta* with a dynamic algorithm, such as D* Lite or AD*. Combining the quality of Theta* to find shorter paths and expand less nodes than other non-dynamic algorithms, and the quality of a dynamic algorithm, might lead to interesting results.

In conclusion it can be said, that the implementations and comparisons done in this thesis are merely the very first step in implementing a suitable pathfinding system in a dynamic computer game. Next to all considerations and implementations made in this thesis a few more optimizations and adaptations have to be worked on, before an AI can reliably use a pathfinding system. It should also be noticed, that this thesis only worked with one possible setup of a dynamic environment. Therefore, conclusions drawn from these tests should not blindly be taken for granted for all dynamic environments. They should be seen as one set of experience and function as a starting point for own implementation and testing.

Bibliography

[Blu19] BlueRaja. High-Speed-Priority-Queue-for-C-Sharp, 2019. Retrieved 21.02.2019 from: https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp.

[BMS04] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28, 2004. Retrieved 27.01.2019 from: http://webdocs.cs.ualberta.ca/~kulchits/Jonathan_Testing/publications/ai_publications/jogd.pdf.

[Boo04] Michael Booth. The Official Counter-Strike Bot, 2004. Retrieved 27.01.2019 from: https://www.cs.rit.edu/~jdb/tmp/making_of_official.pdf.

[Boo09] Michael Booth. The AI Systems of Left 4 Dead, 2009. Retrieved 27.01.2019 from: https://steamcdn-a.akamaihd.net/apps/valve/2009/ai_systems_of_l4d_mike_booth.pdf.

[CFS06] Joseph Carsten, Dave Ferguson, and Anthony Stentz. 3D Field D: Improved Path Planning and Replanning in Three Dimensions. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3381–3386. IEEE, 09.10.2006 - 15.10.2006. Retrieved 24.01.2019 from: https://www.ri.cmu.edu/pub_files/pub4/carsten_joseph_2006_1/carsten_joseph_2006_1.pdf.

[Cha07] Alex J. Champandard. Near-Optimal Hierarchical Pathfinding (HPA*), 2007. Retrieved 27.01.2019 from: http://aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/.

[Cho10] Howie Choset. *Robotic Motion Planning: A* and D* Search*. Lecture slides, Carnegie Mellon University, 2010. Retrieved 11.01.2019 from: https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf.

[Cod12] CodeCompile. A* Pathfinding Tutorial, 2012. Retrieved 31.01.2019 from: https://www.youtube.com/watch?v=KNXfSOx4eEE.

[Com19] Computer History Museum. Shakey, 2019. Retrieved 08.03.2019 from: https://www.computerhistory.org/revolution/artificial-intelligence-robotics/13/289.

[DB88] Thomas L. Dean and Mark S. Boddy. An Analysis of Time-Dependent Planning. In *AAAI*, volume 88, pages 49–54, 1988. Retrieved 13.02.2019 from: https://www.aaai.org/Papers/AAAI/1988/AAAI88-009.pdf.

[DN+07] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-Angle Path Planning on Grids. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), 1177-1183*, 2007. Retrieved 14.02.2019 from: `http://idm-lab.org/bib/abstracts/papers/aaai07a.pdf`.

[DN+10] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-Angle Path Planning on Grids. *J. Artif. Intell. Res. (JAIR)*, 39, 2010. Retrieved 14.02.2019 from: `https://arxiv.org/ftp/arxiv/papers/1401/1401.3843.pdf`.

[FS05] David Ferguson and Anthony Stentz. The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-Uniform Cost Environments, 2005. Retrieved 23.01.2019 from: `https://www.ri.cmu.edu/pub_files/pub4/ferguson_david_2005_3/ferguson_david_2005_3.pdf`.

[FS06] Dave Ferguson and Anthony Stentz. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, 23(2):79–101, 2006. Retrieved 23.01.2019 from: `https://www.ri.cmu.edu/pub_files/pub4/ferguson_david_2006_3/ferguson_david_2006_3.pdf`.

[Gra18] Graceful Algorithms. User's manual for "PathFinder 3D": Version 0.3, 2018. Retrieved 27.01.2019 from: `https://gracefulalgs.com/wp-content/uploads/2018/07/User_39_s_manual.pdf`.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968. Retrieved 11.01.2019 from: `https://www.cs.auckland.ac.nz/courses/compsci709s2c/resources/Mike.d/astarNilsson.pdf`.

[KL02a] S. Koenig and M. Likhachev. Incremental A*. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 1539–1546. MIT Press, 2002. Retrieved 18.01.2019 from: `http://papers.nips.cc/paper/2003-incremental-a.pdf`.

[KL02b] S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *IEEE International Conference on Robotics and Automation 2002*, Conference Proceedings Ser, pages 968–975, Piscataway, May 2002. I E E E. Retrieved 19.01.2019 from: `http://idm-lab.org/bib/abstracts/papers/tr-dstarlite.pdf`.

[KL02c] Sven Koenig and Maxim Likhachev. D* Lite. In *Eighteenth National Conference on Artificial Intelligence*, pages 476–483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence. Retrieved 18.01.2019 from: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf.

[LF+05] Maxim Likhachev, David Ferguson, Geoffrey Gordon, Anthony Stentz, and Sebastian Thrun. Anytime Dynamic A*: An Anytime, Replanning Algorithm. 2005. Retrieved 21.01.2019 from: http://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf.

[LGT03a] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. ARA * : Formal Analysis. 2003. Retrieved 12.01.2019 from: http://www.cs.cmu.edu/~ggordon/mlikhach-ggordon-thrun.ara-tr.pdf.

[LGT03b] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. volume 16, 01 2003. Retrieved 13.02.2019 from: http://papers.nips.cc/paper/2382-ara-anytime-a-with-provable-bounds-on-sub-optimality.pdf.

[Lik10] Maxim Likhachev. *A* and Weighted A* Search.* Lecture slides, Carnegie Mellon University, 2010. Retrieved 13.02.2019 from: https://www.cs.cmu.edu/~motionplanning/lecture/Asearch_v8.pdf.

[Mar15] Mitch Marcus. Informed Search. Lecture slides, 2015. Retrieved 08.03.2019 from: https://www.seas.upenn.edu/~cis391/Lectures/informed-search-I.pdf.

[Mne18] Mnemosyne_Studio. A*, 2018. Retrieved 11.01.2019 from: http://mnemstudio.org/path-finding-a-star.htm.

[Nas10] Alex Nash. Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environmentes, 2010. Retrieved 14.02.2019 from: http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/.

[Pat19] Amit Patel. Heuristics, 09.02.2019. Retrieved 23.02.2019 from: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[PN+17] Phuc Tran Huu Le, Nguyen Tam Nguyen Truong, Wonshoup So MinSu Kim, and Jae Hak Jung. Applying Theta* in Modern Game. In *Journal of Computers, vol. 13*, volume 5, pages 527–536. 2017. Retrieved 27.01.2019 from: http://www.jcomputers.us/vol13/jcp1305-05.pdf.

[Ste93] Anthony Stentz. Optimal and Efficient Path Planning for Unknown and Dynamic Environments, 1993. Retrieved 16.01.2019 from: https:/www.ri.cmu.edu/pub_files/pub3/stentz_anthony__tony__1993_1/stentz_anthony__tony__1993_1.pdf.

[Ste94] A. Stentz. Optimal and efficient path planning for partially-known environments. In *Robotics and Automation, '94 International Conference*, pages 3310–3317, Los Alamitos, 1994. IEEE Computer Society Press. Retrieved 16.01.2019 from: https:/www.ri.cmu.edu/pub_files/pub1/stentz_anthony__tony__1994_1/stentz_anthony__tony__1994_1.pdf.

[SYK10] Xiaoxun Sun, William Yeoh, and Sven Koenig. Moving Target D* Lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, pages 67–74, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems. Retrieved 22.02.2019 from: http://idm-lab.org/bib/abstracts/papers/aamas10a.pdf.

[UG16] UG. Coroutines in Unity3D C#, 2016. Retrieved 20.02.2019 from: http://www.unitygeek.com/coroutines-in-unity3d/.

[Uni18a] Unity Documentation. Inner Workings of the Navigation System, 2018. Retrieved 26.01.2019 from: https://docs.unity3d.com/Manual/nav-InnerWorkings.html.

[Uni18b] Unity Documentation. Nav Mesh Obstacle, 2018. Retrieved 26.01.2019 from: https://docs.unity3d.com/Manual/class-NavMeshObstacle.html.

[Uni18c] Unity Documentation. Navigation Areas and Costs, 2018. Retrieved 26.01.2019 from: https://docs.unity3d.com/Manual/nav-AreasAndCosts.html.

[Uni18d] Unity Documentation. Navigation System in Unity, 2018. Retrieved 26.01.2019 from: https://docs.unity3d.com/Manual/nav-NavigationSystem.html.

[Uni19a] Unity Documentation. Coroutines, 2019. Retrieved 20.02.2019 from: https://docs.unity3d.com/Manual/Coroutines.html.

[Uni19b] Unity Documentation. Physics.Raycast, 2019. Retrieved 24.02.2019 from: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html.

[Uni19c] Unity Documentation. Physics.SphereCast, 2019. Retrieved 24.02.2019 from: https://docs.unity3d.com/ScriptReference/Physics.SphereCast.html.

[Ver17] Steve Vermeulen. How to use Async-Await instead of coroutines in Unity3d 2017, 2017. Retrieved 17.02.2019 from: http://www.stevevermeulen.com/index.php/2017/09/using-async-await-in-unity3d-2017/.

[Ver18] Steve Vermeulen. Async Await Support - Asset Store, 2018. Retrieved 17.02.2019 from: https://assetstore.unity.com/packages/tools/integration/async-await-support-101056.

[Wel12] Max Welling. *A* Search*. Lecture slides, Donald Bren School of Information & Computer Sciences, 2012. Retrieved 11.01.2019 from: https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf.

[Wik18a] Wikipedia contributors. A*-Algorithmus — Wikipedia, Die freie Enzyklopädie, 2018. Retrieved 31.01.2019 from: https://de.wikipedia.org/w/index.php?title=A*-Algorithmus&oldid=179400352.

[Wik18b] Wikipedia contributors. A* search algorithm — Wikipedia, The Free Encyclopedia, 2018. Retrieved 31.01.2019 from: https://en.wikipedia.org/wiki/A*_search_algorithm.

[Wik18c] Wikipedia contributors. Shakey the robot — Wikipedia, The Free Encyclopedia, 2018. Retrieved 31.01.2019 from: https://en.wikipedia.org/wiki/Shakey_the_robot.

[XH11] Xiao Yan Cui and Hao Shi. A*-based Pathfinding in Modern Computer Games. 2011. Retrieved 27.01.2019 from: http://paper.ijcsns.org/07_book/201101/20110119.pdf.

Appendix

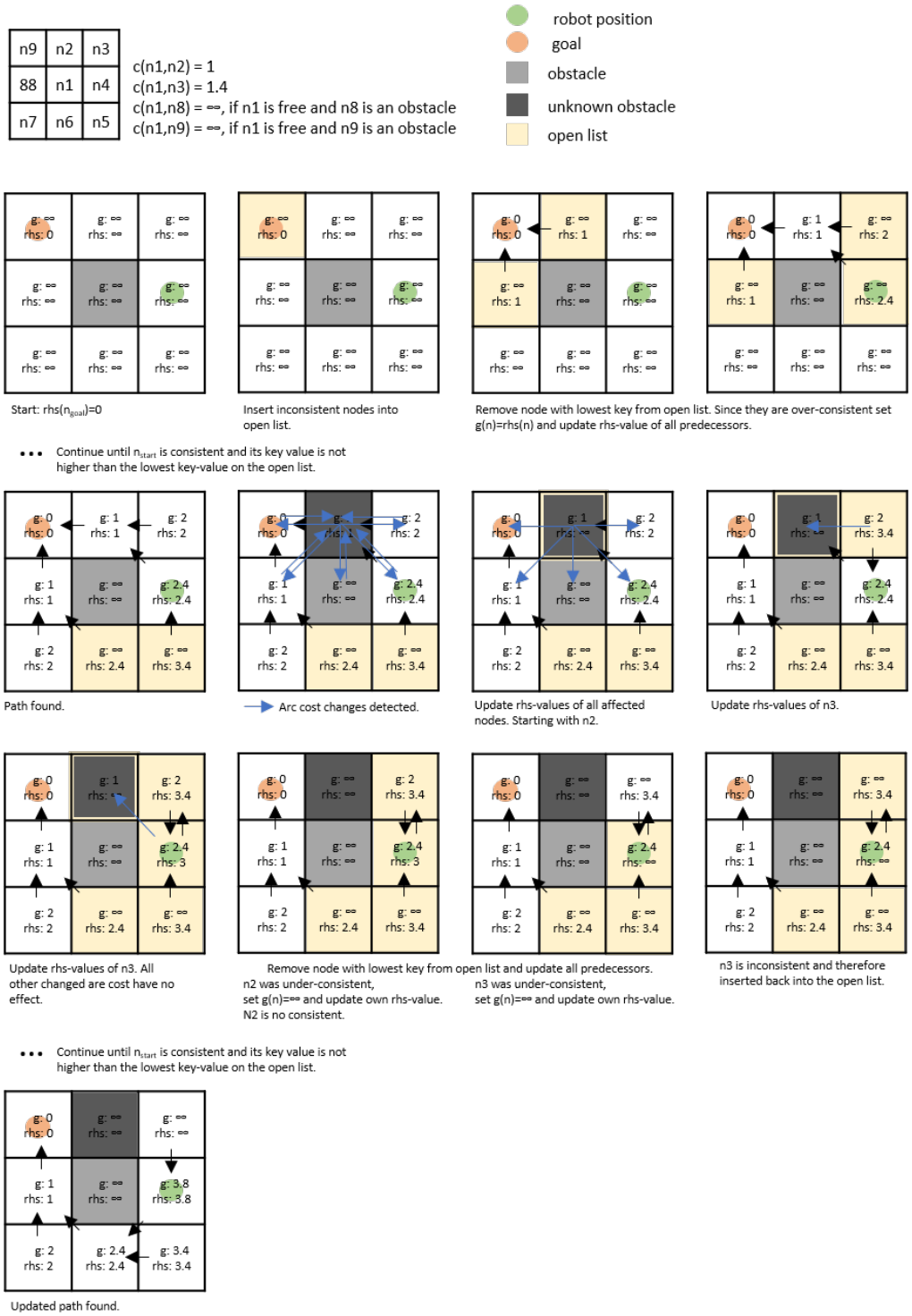


Figure 54: Example of D* Lite with no heuristic [Cho10]

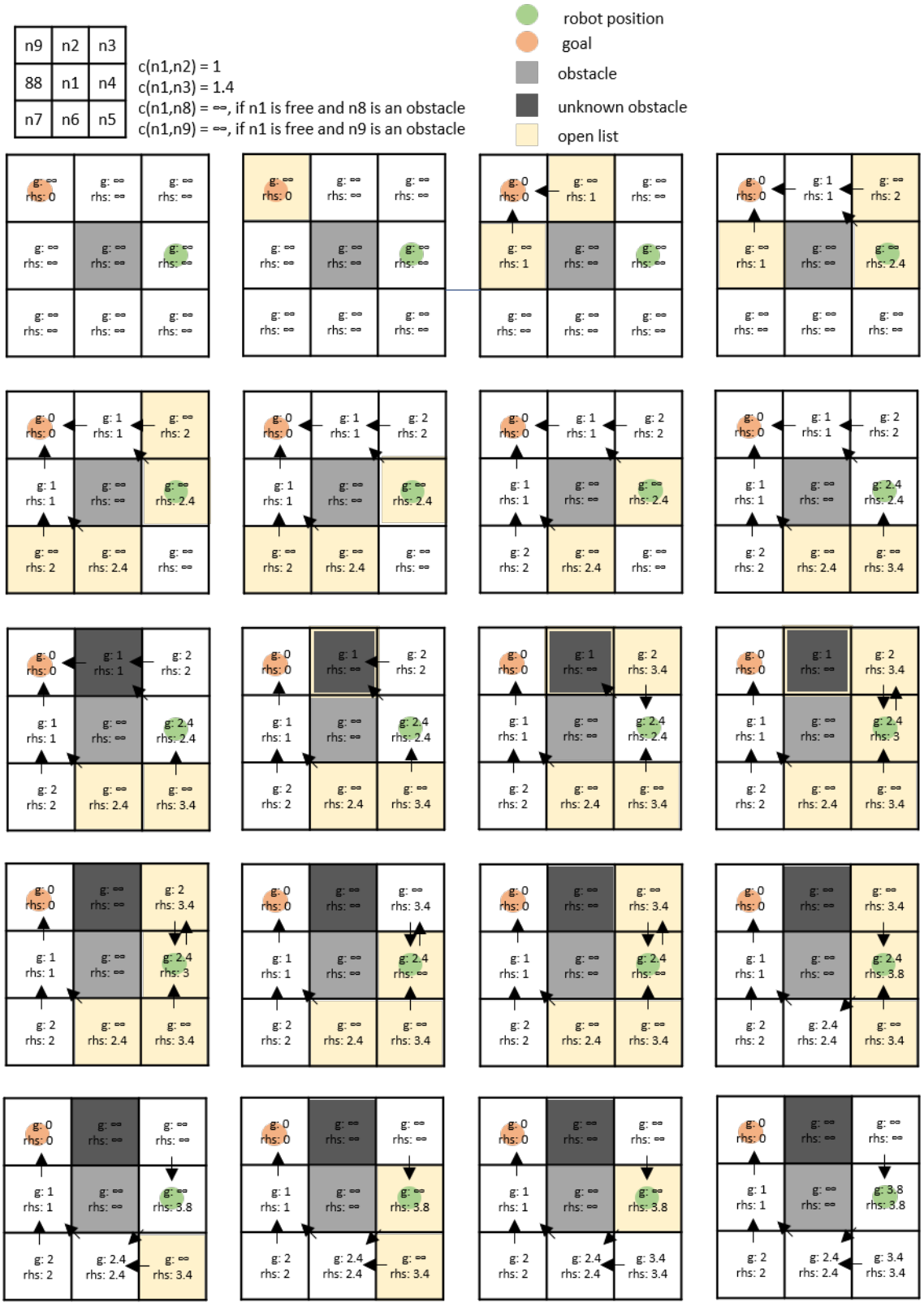


Figure 55: Full example of D* Lite with no heuristic [Cho10]

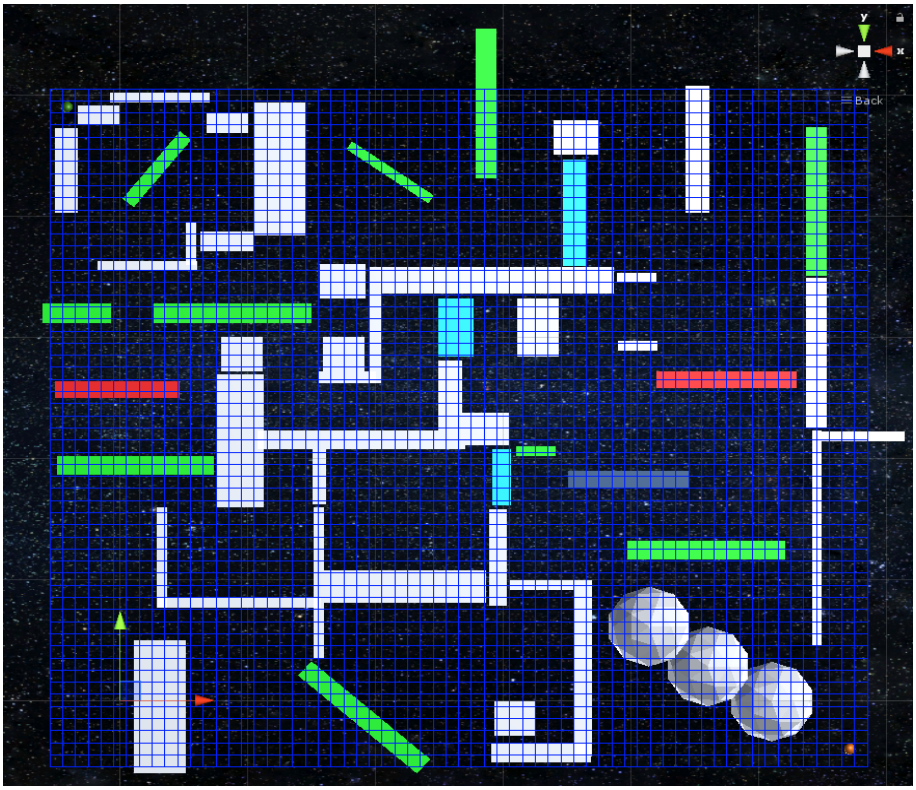


Figure 56: Full cell-grid (orthographic front view)

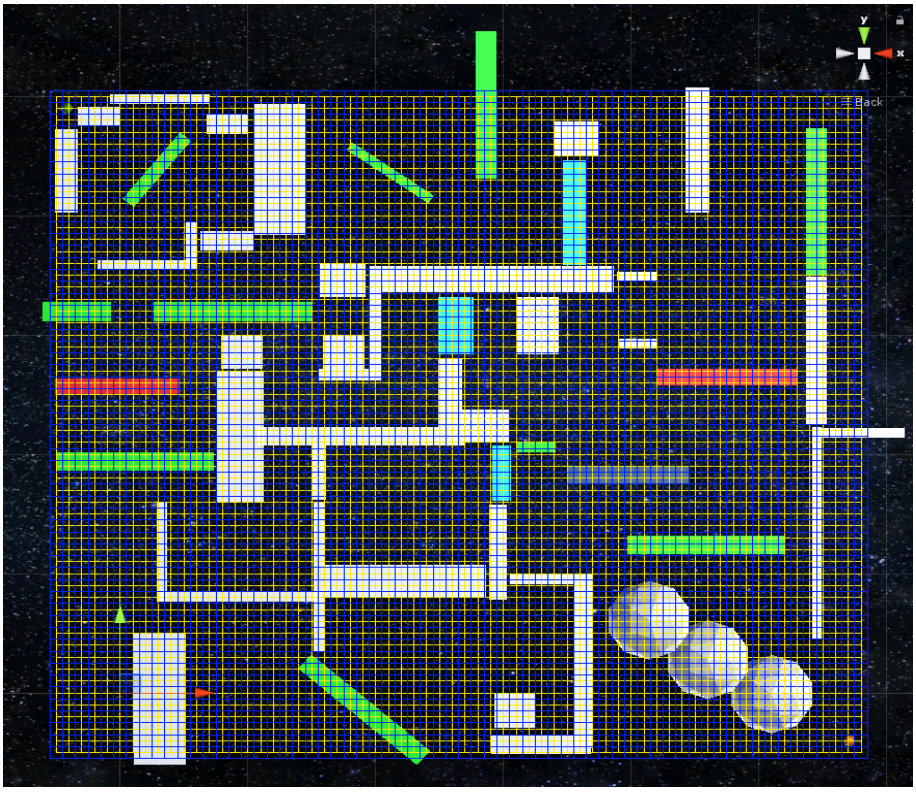


Figure 57: Full cell-grid with connections to neighbours indicated in yellow (orthographic front view)

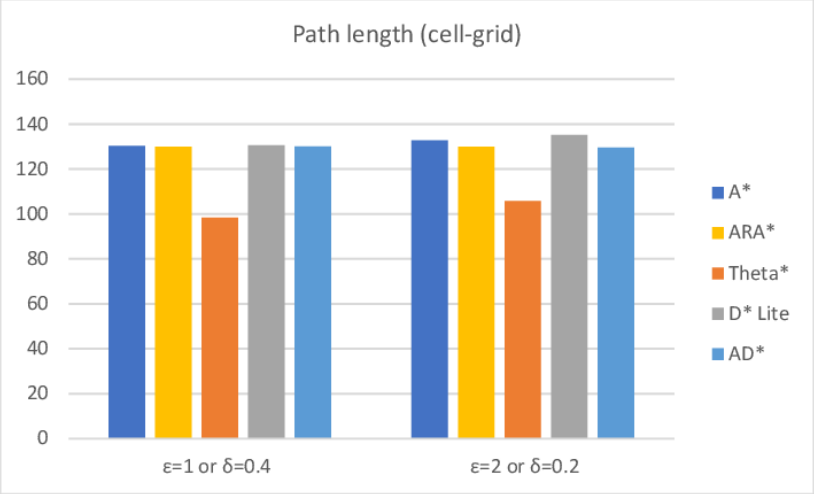


Figure 58: Average path length utilizing a cell-grid search graph

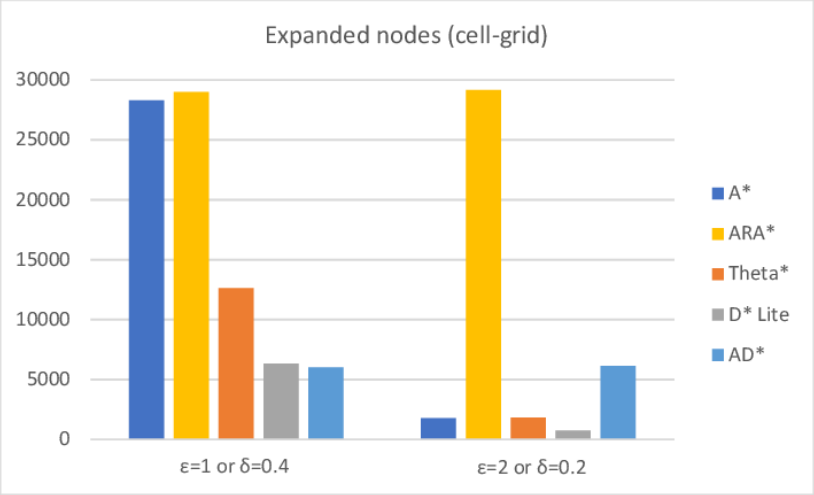


Figure 59: Average expanded nodes utilizing a cell-grid search graph

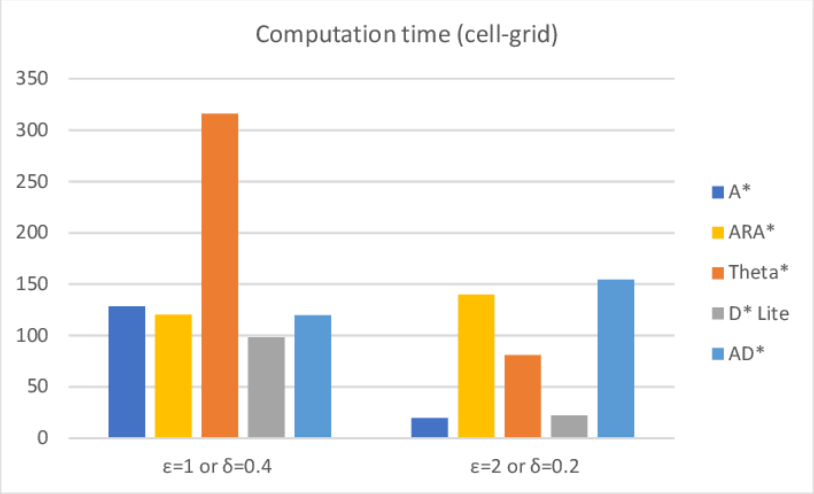


Figure 60: Average computation time utilizing a cell-grid search graph

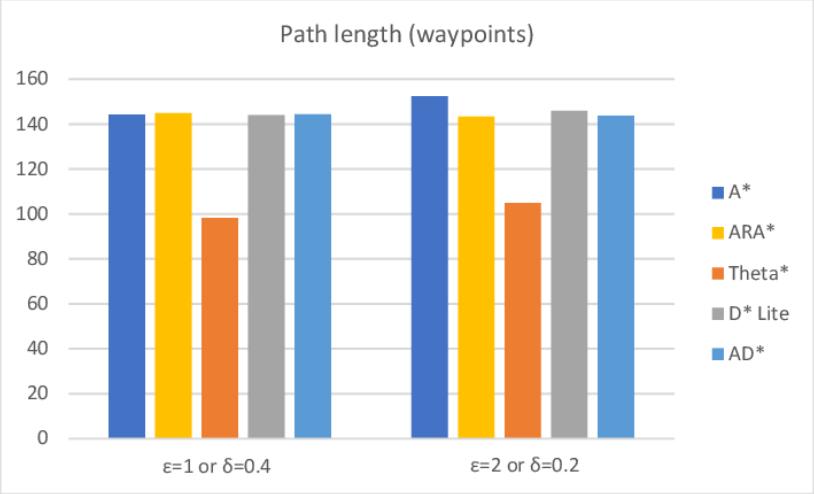


Figure 61: Average path length utilizing a waypoint-based search graph

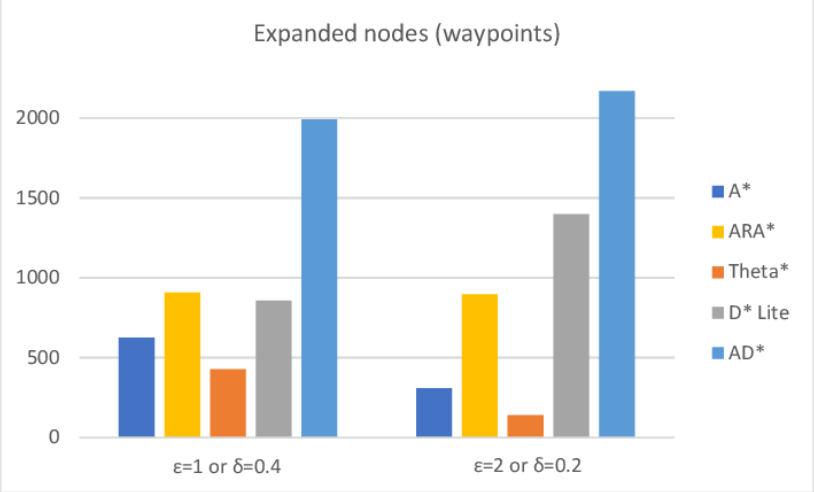


Figure 62: Average expanded nodes utilizing a waypoint-based search graph

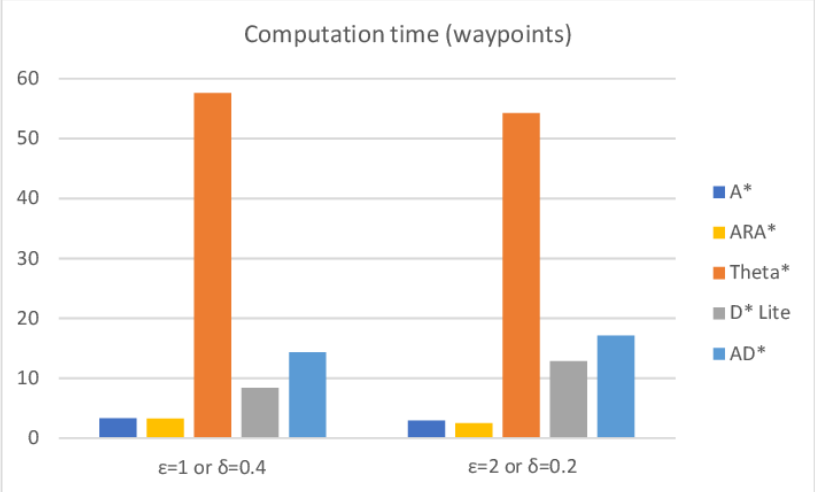


Figure 63: Average computation time utilizing a waypoint-based search graph

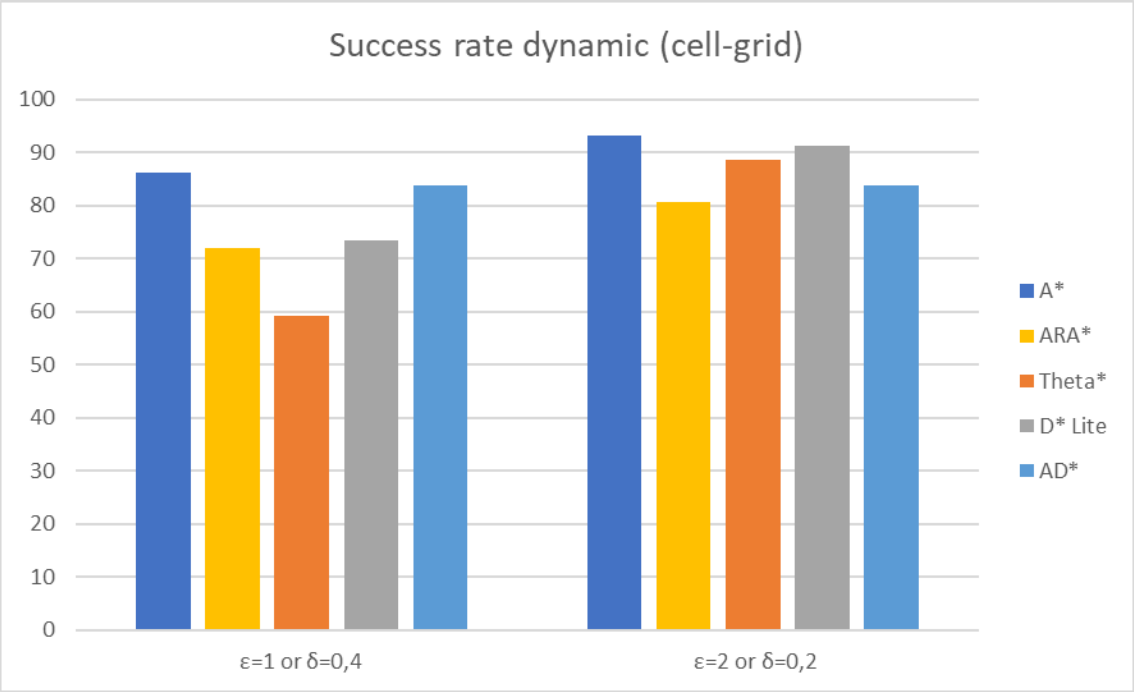


Figure 64: Success rate in an all dynamic environment utilizing a cell-grid search graph

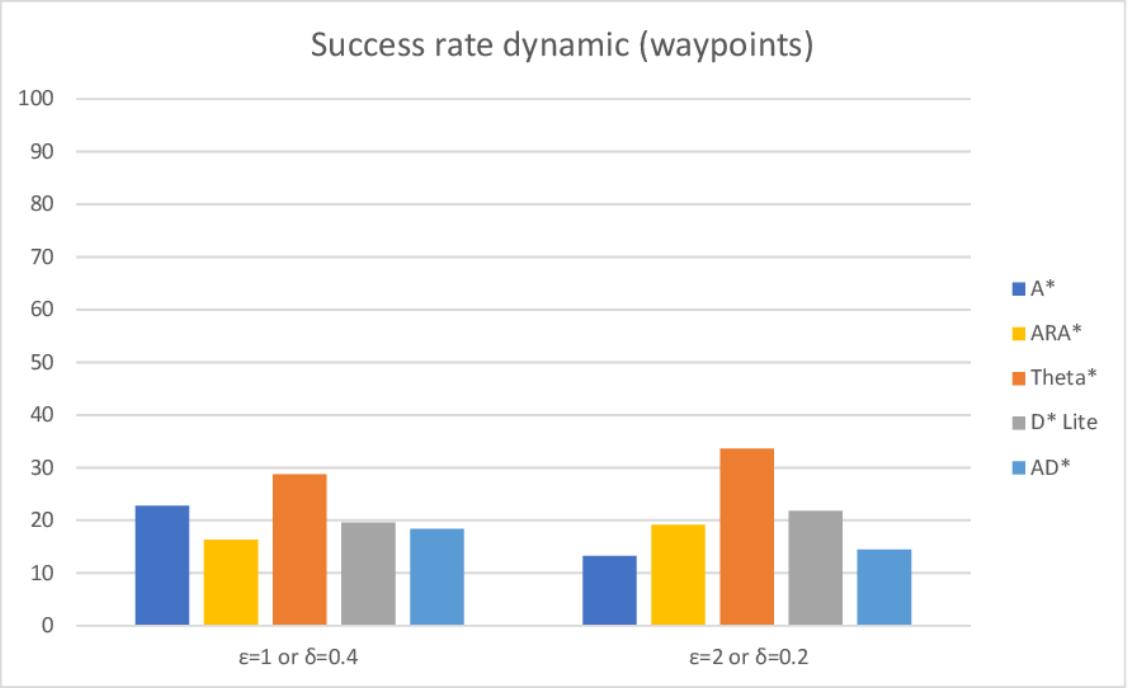


Figure 65: Success rate in an all dynamic environment utilizing a waypoint-based search graph

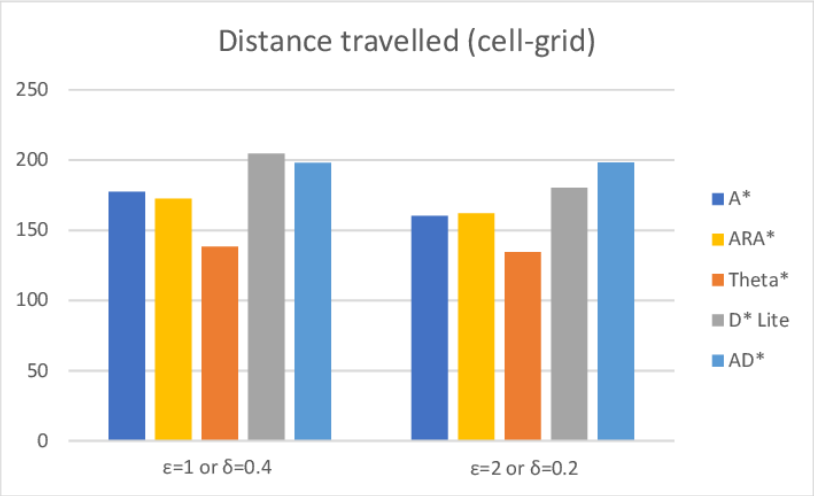


Figure 66: Average distance travelled by the AI in an all dynamic environment utilizing a cell-grid search graph

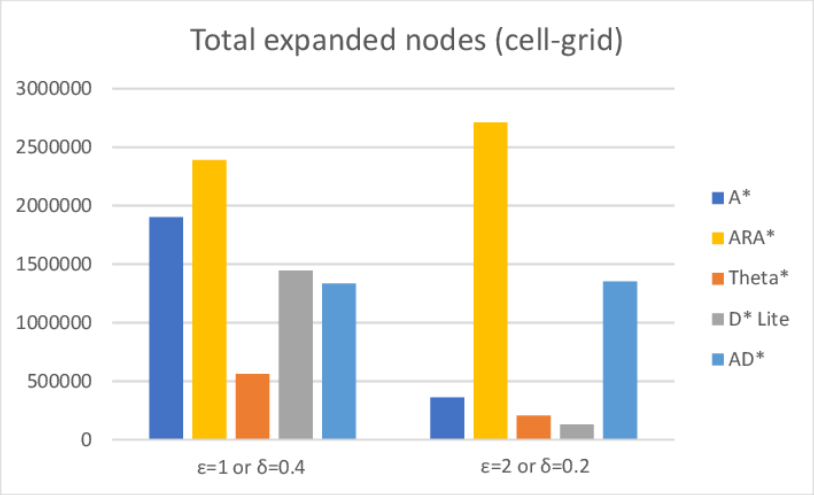


Figure 67: Average total expanded nodes in an all dynamic environment utilizing a cell-grid search graph

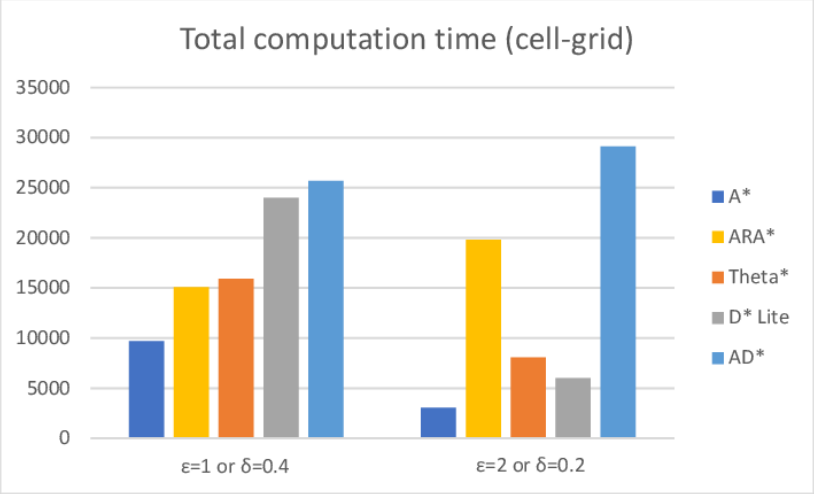


Figure 68: Average total computation time in an all dynamic environment utilizing a cell-grid search graph

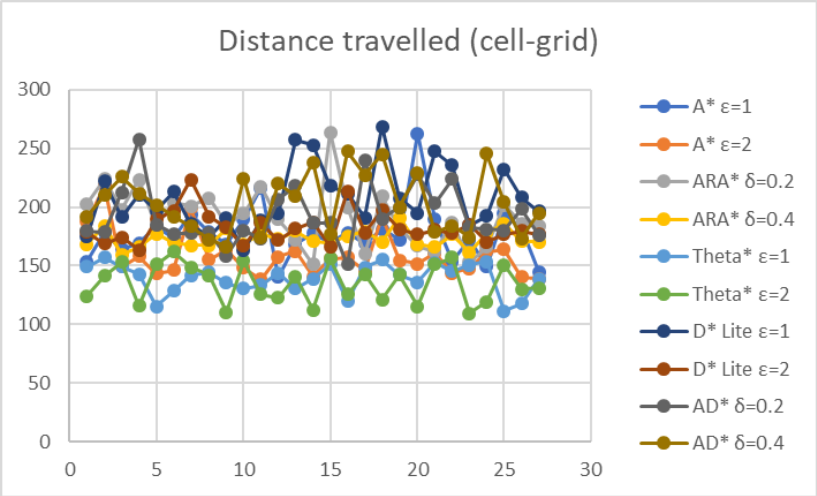


Figure 69: Distance travelled by the AI in an all dynamic environment utilizing a cell-grid search graph

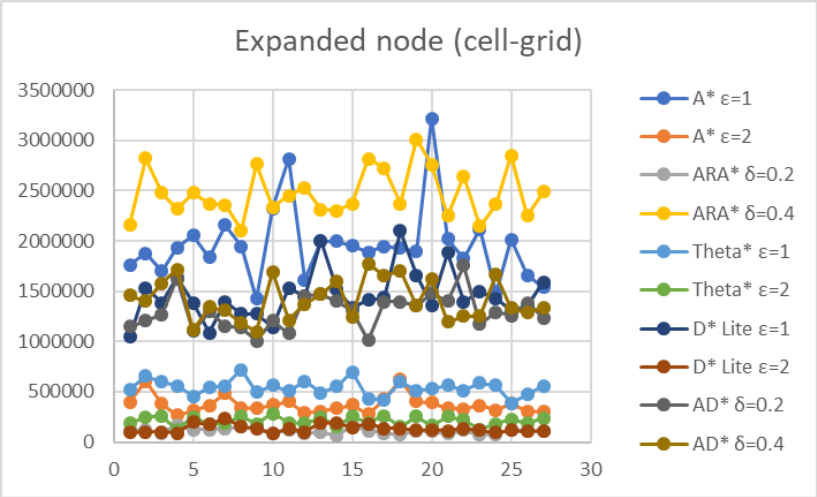


Figure 70: Expanded nodes in an all dynamic environment utilizing a cell-grid search graph

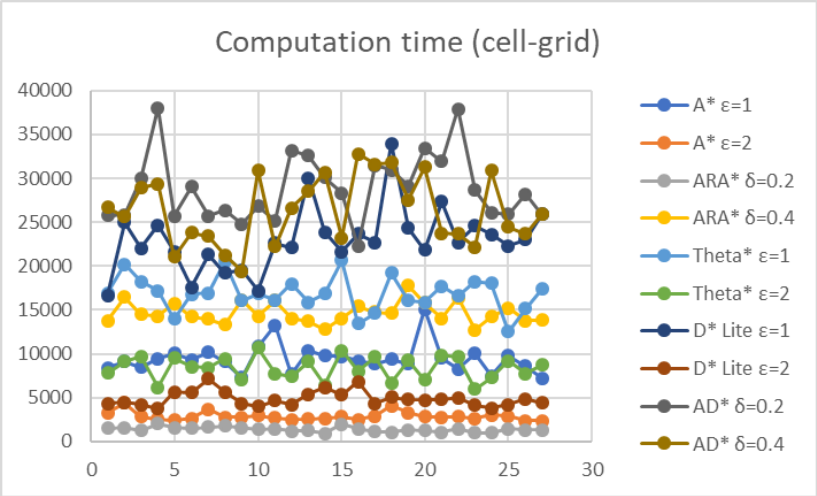


Figure 71: Computation time in an all dynamic environment utilizing a cell-grid search graph

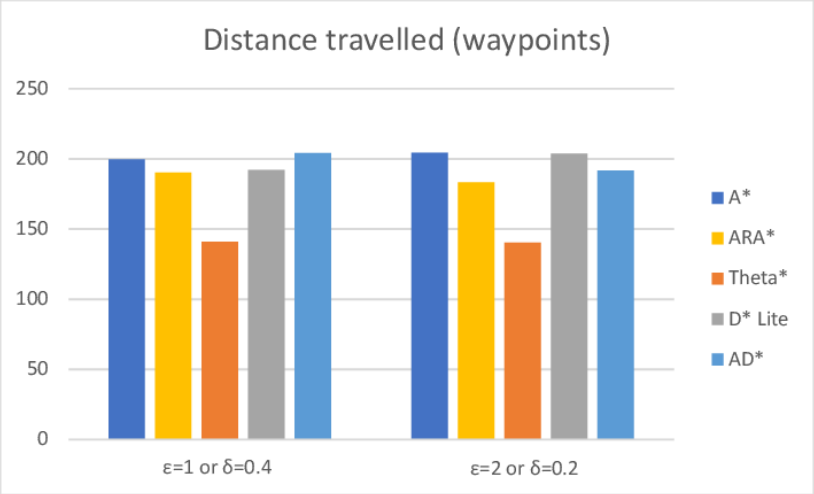


Figure 72: Average distance travelled by the AI in an all dynamic environment utilizing a waypoint search graph

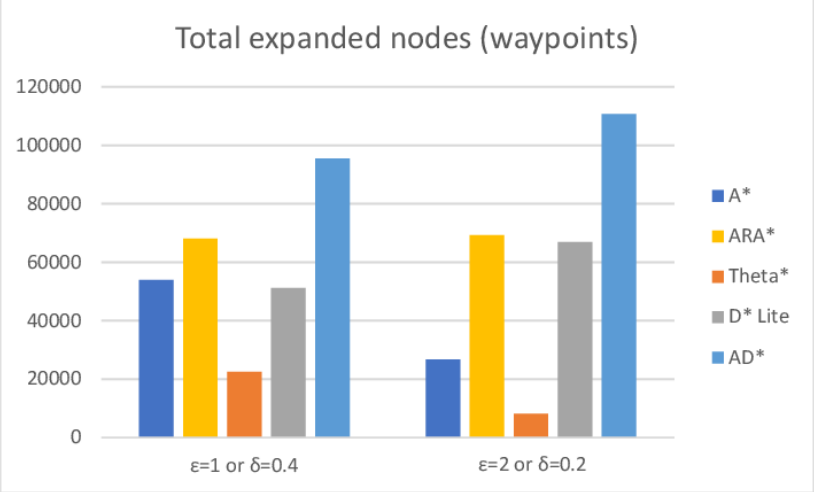


Figure 73: Average total expanded nodes in an all dynamic environment utilizing a waypoint-based search graph

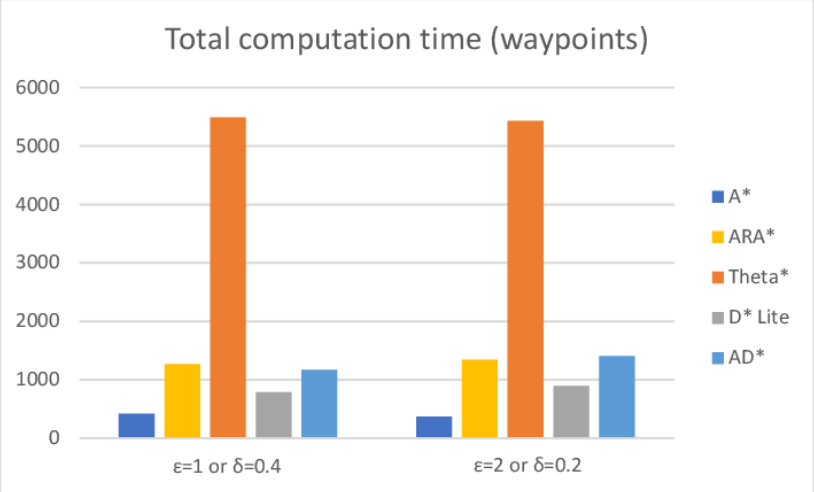


Figure 74: Average total computation time in an all dynamic environment utilizing a waypoint-based search graph

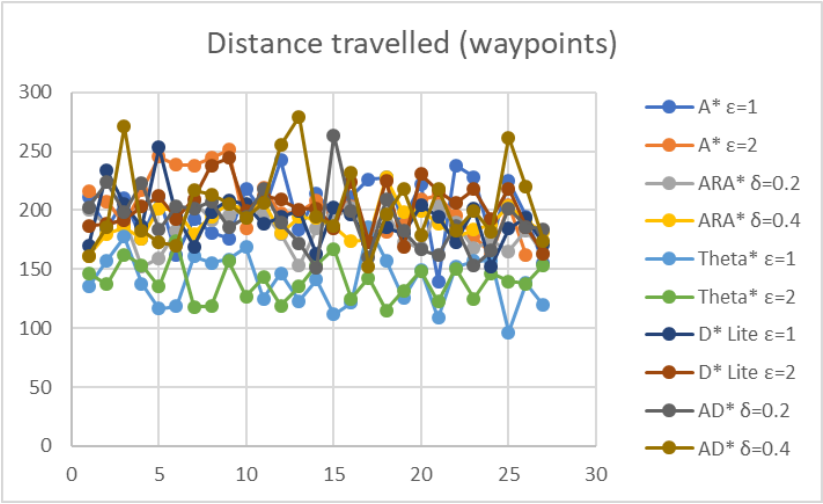


Figure 75: Distance travelled by the AI in an all dynamic environment utilizing a waypoint-based search graph

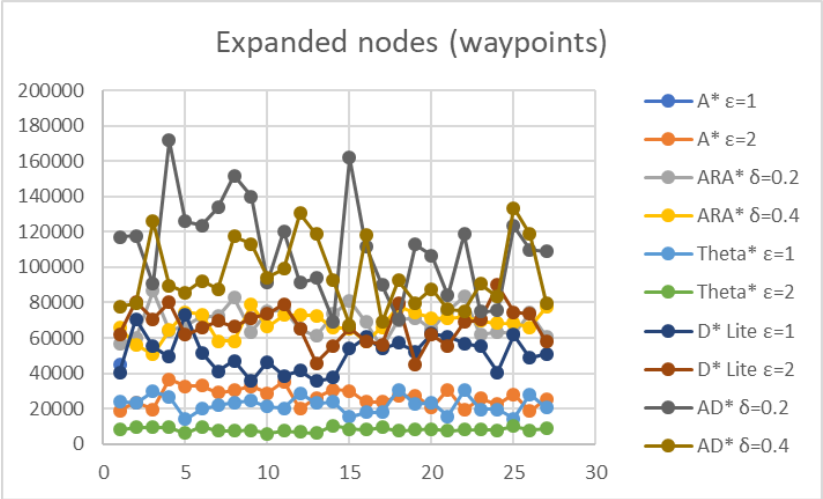


Figure 76: Expanded nodes in an all dynamic environment utilizing a waypoint-based search graph

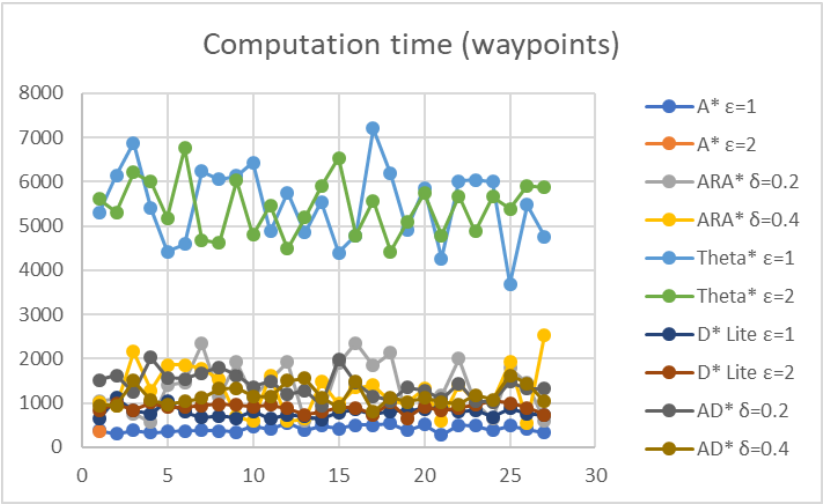


Figure 77: Computation time in an all dynamic environment utilizing a waypoint-based search graph

Hereby I confirm that I completed the presented bachelor's thesis with the title:

"Implementation and comparison of pathfinding algorithms in a dynamic 3D space"

independently, and only with help of the specified resources. All passages adopted from literature or other sources, e.g. websites, are clearly marked as a quote by identifying the source.

Hamburg, 18.03.2019 Carina Krafft

Hiermit versichere ich, dass ich die vorliegende Bachelor-Thesis mit dem Titel:

"Implementation and comparison of pathfinding algorithms in a dynamic 3D space"

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z.B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.