



Pathfinding Algorithm Comparison In Dynamic Congested Environment

Oscar Sandberg

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor Thesis in Computer Science. The thesis is equivalent to 10 weeks of half-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

Oscar Sandberg

E-mail:Ossn17@student.bth.se

University advisor:

Dr. Valeria Garro

Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. Navigation is a crucial aspect in many fields today, such as video games like Real-Time Strategy (RTS) games or real-world scenarios like self-driving robots. Within these areas are dynamic environments, a common sight where multiple agents are moving around to reach different goals, which can cause congestion. Other types of pathfinding algorithms are needed to handle navigation through environments and possible changes in the environment, like congestion, static ones like A*, or dynamic ones like Dynamic A* (D*) Lite.

Objectives. Throughout this thesis, a deep dive is performed into the adaptability and performance of two renowned pathfinding algorithms, A* and D* Lite, when used in a simulated dynamic environment with various numbers of agent densities that cause congestion. To determine which algorithm performs best, analyze the A* and D* Lite results from these simulations during high congestion.

Methods. The main goal is to compare the results from these algorithms gained during the simulations to determine which algorithm performs best—using a custom-designed simulation framework and the algorithms implemented to work for this simulation to consider other agents better when navigating the environment. This systematic experimental approach can evaluate the algorithms for their ability to navigate effectively amidst obstacles and dynamic changes caused by a high number of agents.

Results. The findings revealed that while A* performs well under lower densities, it struggles with specific starting positions during higher congestion levels, resulting in it failing by getting stuck. In contrast, D* Lite consistently adapts to environmental changes, showcasing robustness by recalculating routes on the fly and succeeding in all tested scenarios without failure.

Conclusions. The thesis highlights the limitations of the A* algorithm due to it failing during certain circumstances. D* Lite showed superior adaptability and good performance in dynamic environments, showing it is suitable for applications that require a success rate to environmental changes. The research also points out the importance of using the right algorithm based on the application's specific needs, such as success rate, performance, computational cost, etc. By performing this study has shown areas of focus regarding these algorithms for future work, either for improving the algorithms or further bettering the understanding surrounding them.

Keywords: Pathfinding, comparison, congestion, dynamic environment.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Valeria Garro, for her continuous support of my Bachelor's thesis and her immense knowledge. Her guidance helped me throughout the research and writing of this thesis.

I must acknowledge the support of my family and friends, who provided me with moral and emotional encouragement throughout my studies.

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Table of acronym	3
1.2 Aim and Objectives	3
1.2.1 Aim	3
1.2.2 Objectives	3
1.3 Background	4
1.4 Thesis scope	6
1.5 Ethical and Social Considerations	7
1.6 Outline	8
2 Related Work	9
3 Method	13
3.1 Research questions	13
3.2 Implementation	13
3.2.1 Environment map	14
3.2.2 Simple Agent (BFS)	18
3.2.3 A* agent	19
3.2.4 D* Lite agent	20
3.2.5 Tests and Data collection	22
3.3 Validity and reliability	22
4 Results and Analysis	25
5 Discussion	35
6 Conclusions and Future Work	39
References	43
A Appendix A: Pseudocode	47
A.1 BFS (breadth first search)	47
A.2 A*	48
A.3 D* Lite	48

One of the fields in Artificial Intelligence (AI) is pathfinding, which is a rapidly evolving and adaptable field that has been applied in multiple sectors like robotics, transportation, and video games, as talked about in Algfoor et al., [2]. Pathfinding algorithms calculate the best path between a start and end point. The best path can mean many different things to different algorithms; one might want the shortest path, and another might focus on safety in a densely populated area. This ability to navigate in environments to reach their goal is crucial for agents in everything from physical robots like vacuum cleaners [14] to entities in programs. The importance of pathfinding goes far beyond theory in computer science and is applied in many areas of our everyday lives. In the transportation field, pathfinding algorithms optimize routes for navigation in Global Positioning System (GPS), making it more efficient by reducing travel time [18].

We have robotics, which excellently uses pathfinding to move through demanding environments autonomously and safely. Within the robotics industry, pathfinding has been revolutionary, where advanced algorithms were used to handle specific tasks in varied environments, from manufacturing floors in warehouses to the home, as explored in [17].

Similarly, pathfinding is critical in evacuation, which is used to design optimal escape routes in emergencies [5]. Pathfinding algorithms' utility makes them a cornerstone in AI, affecting a large number of applications from virtual simulations to real-world robotics.

First, pathfinding makes the gaming industry more realistic by enabling Non-Player Characters (NPCs) to map a path through complex environments intelligently. On top of making it possible for NPCs to move in complex terrains, it is also a crucial part of AI when it comes to reacting to player actions, which creates a more entertaining gameplay experience. This ability to respond to players and other changes is essential in several game genres, from RTS games to racing games [20]. Another instance where pathfinding algorithms are used in RTS games is in the game Starcraft. They allow units to navigate [7] and coordinate movements in a dynamic battlefield environment.

In the same way, in racing games, dynamic pathfinding helps AI-controlled cars adjust in real time and change their routes to avoid obstacles or changes on the track, like finding a better path to enhance the game's competitive aspect. This article, Sazaki et al. [23], is an example of this, where the authors compare pathfinding algorithms for a racing game. These examples show the versatility of pathfinding in games and are significant in enhancing realism and player interaction during the

gaming experience.

Pathfinding goes beyond navigation to include efficiency and safety features for systems with specialized algorithms for specific challenges. One of these challenges is *collision detection*, which consists of the ability to sense obstacles or agents in the environment that might block its path or cause a collision [25]. Going hand in hand with this is *collision avoidance* like in this article Ahn et al. [1], which is usually employed in some algorithms once a particular obstacle has been identified, it attempts to avoid it. Dynamic pathfinding for *collision avoidance* is especially important when it comes to AI in situations where multiple obstacles and agents come into play, and there is a need to detect and avoid them. Examples of this traffic management are self-driving cars or robotics in warehouses that need to avoid each other.

As stated above, because of the importance of pathfinding, understanding how they work in certain situations and how they compare to each other during these scenarios is crucial. Despite extensive studies on algorithms, there is a significant research gap in testing algorithms' performances in certain situations, such as highly dynamic environments with congestion. Current literature lacks comprehensive comparisons of these algorithms under such conditions. Therefore, this will be addressed by comparing two different algorithms to better understand how they handle these types of situations and perform compared to each other for best use in the future. The scope of this thesis aims to address this research gap, by evaluating the performance of A* and D* Lite algorithms in a scenario with a high density of agents, comparing their ability to handle dynamic changes like congestion environment caused by the agents and analyze their success rates and time performance under varying levels of agent-induced congestion.

Performance is determined by the time it takes for the algorithm to get the goal. This measures how proficient a pathfinding algorithm's performance is at navigating and is also one of the critical aspects of picking a pathfinding algorithm for GPS, games, and more. Another aspect closely related to this is the success rate since you must be able to find a way from point A to point B and be able to get there to determine the performance in time.

While both algorithms are efficient in static settings, comprehensive comparisons are lacking in highly dynamic environments where conditions continuously change, as is often encountered in gaming scenarios. Choosing A* and D* Lite for this thesis provides a comprehensive foundation by covering both static and dynamic pathfinding algorithms. Firstly, A* and D* Lite cover a wide range of pathfinding scenarios, with A* being static and D* Lite suited for dynamic environments. This provides a broader evaluation of which algorithm performs best under the thesis's unique conditions, enhancing the study's comprehensiveness.

Secondly, a specific interest was to see how a well-known game algorithm like A* and D* Lite performs [13] in real-world related scenarios, particularly in highly dynamic and congested environments. This will show how these can work outside a video game context and broaden the understanding of them.

Thirdly, D* Lite was chosen over other dynamic algorithms like D* since it is easier to understand and is more straightforward [12], which makes it easier to implement. The simplicity makes it suitable for the simulations, enabling effective dynamic pathfinding without excessive complexity. Using these algorithms will help

gain a better understanding of how each handles unique scenarios, such as highly dynamic and congested environments. Highlighting their strengths and weaknesses will aid in future work, showing when or when not to use them and what improvements should be made. The goal is also to compare them and learn under which circumstances one or the other should be used. The insight into effectiveness and adaptability provides valuable guidance for developers and researchers to select appropriate strategies for real-time interactive applications. Examples of this are how the authors in Sazaki et al. [23] compared algorithms for racing games or how Crespi et al. [6] analyze Ant Colony Optimization (ACO) algorithms both cooperative and noncooperative for escape in situations which the environment is a labyrinth.

1.1 Table of acronym

AI Artificial Intelligence

GPS Global Positioning System

RTS Real-Time Strategy

NPCs Non-Player Characters

BFS Breath First Search

DFS Depth-First Search

D* Dynamic A*

MAPF Multi-Agent Pathfinding

ACO Ant Colony Optimization

CBS-D* Conflict-Based Search with D* Lite

1.2 Aim and Objectives

1.2.1 Aim

The aim of this thesis is to compare the performance and adaptability of the A* and D* Lite pathfinding algorithms in a simulated dynamic environment with high agent density, causing congestion.

1.2.2 Objectives

1. Set up an environment map that facilitates the congestion conditions and dynamic, including agents with advanced pathfinding algorithms like A* and D* Lite and simple agents that populate the area and create obstacles.
2. Perform tests to gather data on the algorithms' success rates and times to navigate, assessing their performance in navigating through the different test runs.

3. Analyze the collected data statistically to understand performance differences, drawing conclusions about why algorithms may perform better or encounter failures in the simulation.

1.3 Background

Several algorithms have been developed, each with unique strengths and weaknesses when it comes to different navigation challenges. The Breath First Search (BFS), A*, D*, and D* Lite algorithms are widely used and adaptable to a range of scenarios.

BFS is a search and travel algorithm for data structures like trees and graphs. The BFS algorithm explores all nodes at the same level, then moves on to the next level. This makes BFS an effective tool to find the least amount of steps from a starting point to a target point in an unweighted graph [13].

This characteristic ability to find the least steps from start to finish makes it ideal for practical applications such as games, network routing, and AI agent navigation. BFS is particularly efficient in scenarios where all actions have the same cost as mentioned on [21, p. 82] or where the objective is to find the quickest route based on the number of steps. For instance, if there is a maze or grid an agent must navigate to find the fastest route to an exit or through an obstacle course, BFS provides an optimal solution if one exists. This makes it a valuable algorithm for navigating agents in predictable, well-defined environments.

BFS can be applied to an agent to navigate an environment by following these steps:

1. **Start Node:** From the agent's start node (start position), the algorithm initiates the search from the end node (end position).
2. **Queue:** It uses a queue to keep track of all nodes that need to be explored. The start node is placed at the beginning of the queue and then continues from there.
3. **Exploration:** During exploration, the next node at the front of the queue is removed, and its neighbor nodes that have not been visited (explored) before are added to the queue. The resulting search finds the shortest path if one can be found by going from layer to layer.
4. **Marking:** When a node is explored, it gets marked as visited to ensure that nodes are not reprocessed again, avoiding infinite loops or repeated calculations.
5. **Repetition:** This is repeated until all nodes in the queue are explored or the end node is reached.
6. **Path Reconstruction:** Once the end node is found, it backtracks to the start node to gain the shortest path from start to finish.

The A* algorithm is one of the algorithms used in this thesis, and since it will be compared to the D* Lite algorithm, having a good understanding of it will be essential. The A* algorithm is known for its efficiency, robustness, and versatility,

making it popular in various fields such as robotics, video games, and GPS navigation. For example, in [7], it is combined with another algorithm for navigating complex environments in Starcraft. This approach suits static environments where the terrain or other agents remain constant. The A* algorithm utilizes a heuristic search function to gain the shortest distance and lowest cost path balance and determine the most efficient route between two points as described in [21, p. 93]. By having a set of nodes, the function $f(n) = g(n) + h(n)$ is used to evaluate each node. First, the value or cost for reaching the node n from the start node is represented by $g(n)$. Second, $h(n)$ represents the heuristic estimation cost to continue from the chosen node n to reach the goal. Using this heuristic function impacts the efficiency and accuracy of the A* algorithm by prioritizing paths that appear closer to the goal node.

The algorithm can be utilized to navigate a digital agent through the space in a simulation, attempting to find the most efficient exit path from the start position to the end position. This project evaluates the A* algorithm's adaptability and efficacy in passing through a dynamic scenario and its success rate in reaching the end position in such an environment. The results could provide insights that suggest strengths and weaknesses for scenarios with crowded conditions. This, in turn, could lead to modifications to improve the optimization of the algorithm's application in game simulations that mimic real-world scenarios for crowded conditions simulation.

The D* algorithm is an advanced pathfinding method designed to handle dynamic environments efficiently [28]. Unlike more traditional algorithms like A* that require a complete and static map to be optimal, the D* dynamically updates the shortest path when new information becomes available through environmental changes. The process of dynamically updating nodes or states encountered during agent traversal is facilitated through the use of a priority queue. Such nodes are rearranged as the agent moves along its path and changes occur. The queue employs a cost-based organization method, utilizing both the cost to reach the node and an estimated cost to achieve the goal. This ensures that the path remains continually optimized based on the most up-to-date information. This makes D* very suitable for real-time applications where conditions change unpredictably, such as in robotics or autonomous vehicle navigation, as the author introduced it for [29].

Both D* and D* Lite have the same strategy for handling changes, where they only update the necessary section of the path needed to reach the goal. The mechanism for updating is helpful for real-time navigation and *collision detection* in scenarios where adaptability is needed. The D* Lite makes use of a set of components as part of the algorithm's attempt to find the best path:

- $g(n)$ This is the cost from the start node to node n , representing the actual cost of the best-known path from agent start node to n . It is updated as shorter paths are discovered during the algorithm's execution.
- $rhs(n)$ This stands for "right-hand side" and represents the minimum cost to reach node n from any predecessor plus the cost to move from that predecessor to n . In D* Lite, this represents a kind of "one-step-lookahead" for the future cost of moving from n towards the goal. The rhs -value is used to update the g -value if new, more cost-effective paths are found.

- $h(n, s_{\text{start}})$ This is the heuristic estimated cost from node n to the start node, which is the goal node in the context of D* Lite since it determines the path from end to start. The heuristic estimation is used to guide the search direction towards the start (which acts as the goal in a reversed sense).
- k_m This variable increases each time the graph or environment changes to accommodate and reflect the new situation. k_m is used to adjust the key calculation to reflect that the costs in the graph might have changed.

The key of a node, represented as $\text{key}(n)$, is crucial for determining the order in which nodes are processed and explored by the D* Lite algorithm by prioritizing nodes based on the estimated total cost of a path passing through them. The key setup as a vector or tuple is composed of two components:

$$\text{key}(n) = [\min(g(n), \text{rhs}(n)) + h(n, s_{\text{start}}) + k_m, \min(g(n), \text{rhs}(n))]$$

As described in [12], the algorithm runs to determine the best path; it keeps the priority queue, which keeps the nodes organized based on their updated keys, as seen above. When changes are detected in the graph or environment, such as new obstacles or cost changes, relevant g and rhs values are updated, and as a result, their keys are also recalculated, which may lead to an improved or worse path. The algorithm continues to run until the priority queue is empty or until the path from start to goal is found based on the last update in the environment.

A data structure is used to store the minimum of g and rhs values for a specific node and a heuristic estimate (h-value) from that node to the goal. This structure reflects both the cost to reach the node and an estimate of the goal, adjusted dynamically as the environment evolves.

The D* Lite algorithm ensures optimality by continually adjusting the g -values for nodes that do not match their rhs -values. This results in either an improved or degraded calculated path throughout the graph. When a change, like when an obstacle or something that changes the cost to reach a node, happens in the environment, the D* Lite modifies the graph's edges and rhs values and rearranges the priority queue accordingly.

D* Lite's adaptive nature makes it efficient in real-time applications, like video games, where the environment can change unpredictably from frame to frame. Agents or NPCs can adjust their paths dynamically, providing a seamless and realistic gaming experience [22]. It can be applied in 2d games by switching out the graph for a grid and the nodes in the graph for cells in the grid. This study evaluates how easily the D* Lite can be implemented for a 2D simulated crowded game environment and its performance during it.

1.4 Thesis scope

Pathfinding research is a continuous work of trying to bridge theoretical algorithms with real-world applications, like robots navigating through environments. Pathfinding, whether in gaming or real-world applications, has shown the importance of solutions that are both resilient, able to handle large-scale systems, and adaptable to the unpredictable nature of dynamic environments.

Previous research often focuses on collision avoidance or time efficiency, e.g., [16], [25], and [31]. Having looked through relevant research, articles, and literature, I have not found any that apply these algorithms in environments with a high number of agents moving around and comparing them. This thesis will test and compare algorithms for this event to gain a comprehensive understanding of how these algorithms are fair in those scenarios and compare them to each other.

The research project is a step towards a better understanding of how pathfinding strategies perform in crowded and densely populated settings like a crowded arena in a computer game. By setting up a comparison between D* Lite and the A* algorithms and implementing it from scratch to work as well as possible in a simulated dynamic environment with a high density of agents moving about causing congestion. The goal is to test and compare both their ability to find their way out of such a situation and how fast they are in doing it. Then, a statistical analysis of the gathered result will be performed, and observations over all the test runs will be analyzed to find future changes for improvement.

Focusing on a 2D environment was deliberate for this research since a 2D environment effectively represents agents moving along the ground in the same way as they would in a real-world scenario, making it a perfect model for this study. The findings from this study would be applicable to 3D environments even though they are from a 2D simulation since even in a 3D environment, agents navigate on a planar surface. Using a 2D grid system makes the work more interesting to other workers and researchers since it is the most commonly used in game development for pathfinding. The reason for this is the ability to represent a space for navigation in a simple and efficient way, making it easy then for development, as talked about in [13]. This ensures that the findings and insights gained can still be relevant for other environments and can be generalized for real-world applications where movement planning is kept in two dimensions, even if the environment itself is three-dimensional.

1.5 Ethical and Social Considerations

The social and ethical aspects of this thesis work and their implications are not straightforward. Even as autonomous systems become increasingly integrated into our daily lives, AI and pathfinding must be considered. However, since the thesis and its analysis are focused on game simulation, there are not many social and ethical considerations for now. However, it should be pointed out that this is a narrow study, and before findings from this are implemented for use in games or particularly in real-world applications, further research and trials should be performed to adapt and verify it.

The most important aspects of pathfinding algorithms are safety and reliability, especially in autonomous systems such as robotics, evacuation, and others. Once proven safe, though, further testing could apply these algorithms to real-world systems for situations like evacuation playing or crowd control. For starters, an example of such work is [9], which takes graph traversal algorithms and uses them to plan emergency evacuations by creating evacuation routes in real-time conditions. In the article, [4] explored a multicriteria analysis to develop evacuation routes in buildings. The analysis prioritizes safety over the shortest path by considering factors such as

route width, restricted areas, and floor roughness.

This raises additional ethical considerations regarding pathfinding algorithms' priority of human rights and life, especially in scenarios involving large numbers of people. During an emergency evacuation, people's safety must take priority over efficiency, which is usually a key aspect of pathfinding. Ethical use of pathfinding in applications requires transparency, accountability, and fairness to not cause harm or violate privacy. These ethical ideas should be utilized when developing pathfinding algorithms that perform effectively and contribute positively to societal well-being.

1.6 Outline

The outline is a structured overview of the thesis, from an exploration of pathfinding in artificial intelligence to a detailed analysis of specific algorithms in a specific simulation:

Chapter 1: Introduction This chapter sets the stage for the thesis and its research, highlighting the importance of pathfinding in various sectors, such as transportation, video games, and robotics. It clarifies this study's context, relevance, and motivation alongside the research gap and questions.

Chapter 2: Literature Review Here, a deeper dive into related research related to the thesis on pathfinding algorithms, emphasizing A*, D* Lite, or related somehow. Current methodologies and techniques will be looked into what they focus on, and their solution, which will help identify gaps concerning dynamic environments and high-density scenarios.

Chapter 3: Methodology This chapter details the approach taken to compare the performance of pathfinding algorithms. It explains the design, data collection, and analytical framework used to ensure the validity and reliability of the research.

Chapter 4: Results The results of the experiments conducted will be explained using the data collected during the study. We'll use descriptive and inferential statistics. After that, a complete analysis of the results will be performed.

Chapter 5: Discussion In the discussion, we'll discuss the findings and how they relate to the initial research questions. We'll examine what the results mean in the fields of AI and pathfinding and how they can be used in real-life situations.

Chapter 6: Conclusions and Future Work The final part of the thesis will sum up key insights gathered from the simulation and analysis of the data and also bring up limitations during the study. Lastly, improvement areas that have been discovered during the project and how they can improve the research in the future will also be discussed, as will how these findings can impact the field of pathfinding.

Research into different aspects of pathfinding has exploded over the last years, with everything from Multi-Agent Pathfinding (MAPF) to collision detection and avoidance or just improvements to already existing algorithms for specific applications. Given this breadth of work, guidelines are essential, and this is what Stern et al. [30] aim to provide. Their work focuses on MAPF, it presents a framework for solving problems, like planning paths for multiple agents without collisions. It also brought up the standardized terminology for describing MAPF, but that is true for all areas and helps with making research comparable and application adaptable. They also introduce a benchmark based on grids for evaluating MAPF algorithms to improve future research with robustness by providing standardized measures for algorithm performance.

This article supports the thesis since it revolves around comparing A* and D* Lite pathfinding algorithms in dense multi-agent environments, basically MAPF. by highlighting the difficulties and variations when it comes to agents interacting in a shared space. The difference between the article, which sets benchmarks and establishes terminologies for standardizing research, and the thesis, which aims to assess and compare the performance and adaptability of algorithms in dynamic, agent-dense scenarios. This difference in focus provides a complementary perspective to the thesis, and integrating ideas and standard terminology from the article will enhance the thesis, among other things.

The pursuit of advancing pathfinding has come a long way, but there are still gaps in understanding the performance of these algorithms and developing them further for highly dynamic scenarios, such as densely populated urban areas or crowded computer game environments.

One article that works in that area is written by Sliacka et al. [27], which discusses the use of simulations to optimize railway infrastructure, and on the other side, this thesis examines the performance of pathfinding algorithms in a dynamic 2D simulation environment. Both studies focus on efficient pathfinding in environments where conditions are dynamic by changing quickly and unpredictably. Comparing this article's simulation methodology with my thesis gives a better understanding of the difficulties involved in dynamic pathfinding and how algorithmic adaptations could improve efficiency in real-time scenarios. It also validates the relevance of my study within the broader field of dynamic systems simulation by showing its importance in real-world sectors like transportation.

Another article Neuschwander et al. [16] reviewed also delves into pathfinding, focusing on enhancing the A* algorithm to handle dynamic collision avoidance with

multiple agents in difficult scenarios. This is relevant to the thesis as it highlights the potential of the pathfinding algorithm A* when dealing with multiple agents that coexist within the same environment. The thesis research involves comparing the effectiveness of A* and D* Lite in avoiding congestion and navigating efficiently towards exits. The authors in Neuschwander et al. [16] introduced improvements for handling agent interactions using an adapted version of the A* algorithm. This can be beneficial in informing future modifications for the algorithms or directly used in the thesis simulation comparison. Understanding these extended algorithms gives a foundation for potentially leading to more robust pathfinding solutions suitable for high-density scenarios.

The development of deep learning models within the field of pathfinding has led to more adaptive and intelligent systems, especially when it comes to environments with a heavy presence of dynamic changes. In this study by Sinkar et al. [26], they tackle MAPF with deep learning, and unlike normal algorithms that decide paths for the static environment before running, they do it dynamically. By doing it in real time, the agent will be able to avoid collisions and change its path to navigate past obstacles, resulting in better pathfinding.

This is relevant to this thesis as it works with similar challenges in dynamic 2D game environments, where agents must react to continually changing conditions. However, [26] focuses on a distributed deep learning model that makes local decisions without prior computation, compared to the thesis that explores a comparison between the A* and D* Lite algorithms under similar scenarios. The thesis analysis extends beyond individual pathfinding to learn how these algorithms perform and in which areas they can be optimized for performance and reliability in gaming scenarios that feature high congestion due to a high density of agents.

Another notable advancement mentioned in this article by Jin et al. [10] is The Conflict-Based Search with D* Lite (CBS-D*). This algorithm uses heuristic search with collision avoidance strategies to improve performance in dynamic environments. It is a prime example of how different approaches can result in a better and more well-rounded pathfinding algorithm, especially when there is a high density of agents and obstacles.

The CBS-D* algorithm is an implementation from [10] for improving autonomous robotic navigation in dynamic environments. It combines a D* Lite algorithm with the conflict-based approach to improve pathfinding and collision avoidance success in high-density dynamic environments with moving obstacles. The CBS-D* algorithm is adaptable and has a higher success rate in path planning whilst also focusing on performance optimization, showing a step forward in performance over older algorithms that are limited to less dynamic scenarios.

In this thesis, a simulation where pathfinding algorithms A* and D* Lite are implemented in a dynamic 2D environment. The CBS-D* algorithm aims to improve obstacle avoidance and pathfinding in real-time, which is relevant to my study and could be implemented in my thesis during future work. It provides insights into handling congestion and dynamic obstacles during pathfinding, which could aid in analyzing the result for the thesis comparison. Since the CBS-D* algorithm is an improvement upon the classic D* Lite algorithm, which is used in my thesis simulation, and as a result of the CBS-D* article [10] is directly related.

The article Sasaki et al. [23] is a study that focuses on NPCs in racing games to

find the best path by optimizing their pathfinding algorithm.

This article is closely related to this thesis since it handles dynamic pathfinding because of changes in the environment and obstacles. This also is a core aspect of the work into pathfinding algorithms like A* and D* Lite in 2D game environments done in this thesis.

Both the article and thesis explore the area of advanced pathfinding algorithms in gaming scenarios that demand real-time responses to dynamic changes. However, the article specifically focuses on combining the A* algorithm with a dynamic pathfinding algorithm to improve performance in racing games. It does this by merging two distinct algorithms to optimize NPC behavior in racing situations.

However, the thesis research diverges in scope by assessing how algorithms like A* and D* Lite perform under high-density conditions with numerous agents, like during the exit circumstances of a sports event, which introduces different complexities and performance problems.

The paper Khantanapoka et al. [11] compares seven pathfinding algorithms, amongst them the A*, but also a new algorithm, the Depth Direction A* (DD-A*), in 2D and 3D real-time strategy games. The DD-A* algorithm is shown here to perform pathfinding efficiency better in multi-layer environments, reducing node expansion significantly compared to the other seven algorithms.

This research is relevant as it was also conducted in a dynamic environment and shows why efficient pathfinding is needed in such an environment. The same goes for this thesis, where A* and D* Lite are compared in dense, agent-populated 2D environments. Both the paper and the thesis use a comparative approach to evaluate pathfinding algorithms in their respective environment. However, the paper addresses multi-layer environments and aims to reduce computational overhead, which the DD-A* algorithm does, while this thesis focuses on establishing algorithms within a single-layer 2D environment.

The paper's contribution lies in multi-layer pathfinding, while this thesis offers a more in-depth analysis of the two algorithms' performance in high-density, dynamic 2D environments. These different perspectives enhance our understanding of pathfinding challenges and might suggest possible ways to incorporate them into future algorithm improvements.

In the article, Crespi et al. [6] explore the literature addressing evacuation strategies using ACO algorithms. This research utilizes ACO to compare evacuation paths for emergencies by setting both a cooperative and non-cooperative ant behavior algorithm. This comparison aligns closely with the objectives of this study as both aim to find the best method for navigating under changing circumstances, although with different agent-based procedures. Although the ACO approach and this thesis focus on path optimization, they are different in which algorithms are being used. The ACO emulates the behavior of ants when they are looking for food by prioritizing paths based on pheromone trails.

The thesis, on the other hand, looks at the A* and D* Lite algorithms, which are graph traversal techniques that emphasize getting the shortest paths and, for D*, recalculating them dynamically when needed. This difference highlights the flexibility of pathfinding algorithms in various simulation scenarios, and insight is gained by analyzing these behavioral models within algorithmic frameworks. It can also enrich the thesis strategies regarding pathfinding surrounding interactions and

logic for agents. This is a crucial aspect for applications to make them and their agents seem believable when it comes to their behavior and incorporate them based on real-time environmental studies that inform potential improvements for pathfinding algorithms.

The article Sarbini et al. [22] compares the A* and D* Lite algorithms in controlled video game environments, focusing on their efficiency and speed. This analysis highlights how each algorithm performs under predefined conditions, which is typical in many gaming scenarios where pathfinding needs are predictable and the environment minimally changes.

The thesis takes a more specialized approach by focusing on the performance of these algorithms in highly dynamic environments with significant agent density. The environments are characterized by between 150 and 1300 agents moving around and a map with dense areas, making navigating harder. These characteristics create unpredictable interactions among agents and lead to situations where rapid changes in real time often block paths.

This focus explores the robustness and adaptability of A* and D* Lite in handling real-time congested areas. The thesis aims to identify which algorithm offers the best result in navigating through a map during these conditions and if a more advanced dynamic algorithm is needed. While the article lays the groundwork by assessing algorithmic performance in more straightforward scenarios, the thesis contributes significantly by testing these algorithms under the stringent conditions of a tight arena with hundreds of agents that cause congestion and dynamic obstacles. This fills a gap in current research on pathfinding algorithms by deepening the theoretical understanding of current pathfinding capabilities and, in practice, applying them in new scenarios demanding high adaptability and responsiveness.

Here is outlined the approach and strategy used to compare the A* and D* Lite algorithms in navigating a dynamic 2D game environment with a high number of agents. Going over the methodology used to analyze and perform the comparison.

A comparative experimental approach was used to design and address this question. With Python and the graphics library pygame, a controlled 2D simulation environment was implemented and tested under various agent densities and starting positions [8, 19]. This simulation measures each algorithm's time performance and success rate, directly comparing the two.

The approach's validity lies in its design and ability to produce repeatable and measurable results in controlled conditions. Conditions that keep random and unpredictable features of the simulation away except for those that are being tested. It mimics a real-world scenario in game development where pathfinding efficiency is crucial to navigating a congested 2D environment.

3.1 Research questions

"What are the differences between the A* and D* Lite algorithms regarding time performance and success rate of navigating through a 2D game environment with a high number of agents causing congestion?"

With this thesis, the study aims to understand the differences in performance between these algorithms. The results can help developers and researchers choose the appropriate pathfinding solutions for their work. Knowing how they act and perform would help find a balance between efficiency and resource consumption, particularly in complex interactive applications like multiplayer games or autonomous crowd simulations.

3.2 Implementation

Here is a breakdown of the thesis implementation into the main sections like environment map, simple agent, A* agent, and D* Lite agent. It goes into detail about how each of these sections works and their part in the thesis, and lastly, how the tests are performed and the data collected.

3.2.1 Environment map

A tailored environment was set up as part of this simulation to help answer the research question. It was set up to look like an ice hockey arena with a large white section to represent the ice and a section where agents can not move. Around it, several walls (black lines) are placed to represent stands from which a crowd would stand and watch the game. These stands are narrow and the place where all the agents will start to navigate towards one of the six exits spaced out along the outer wall.

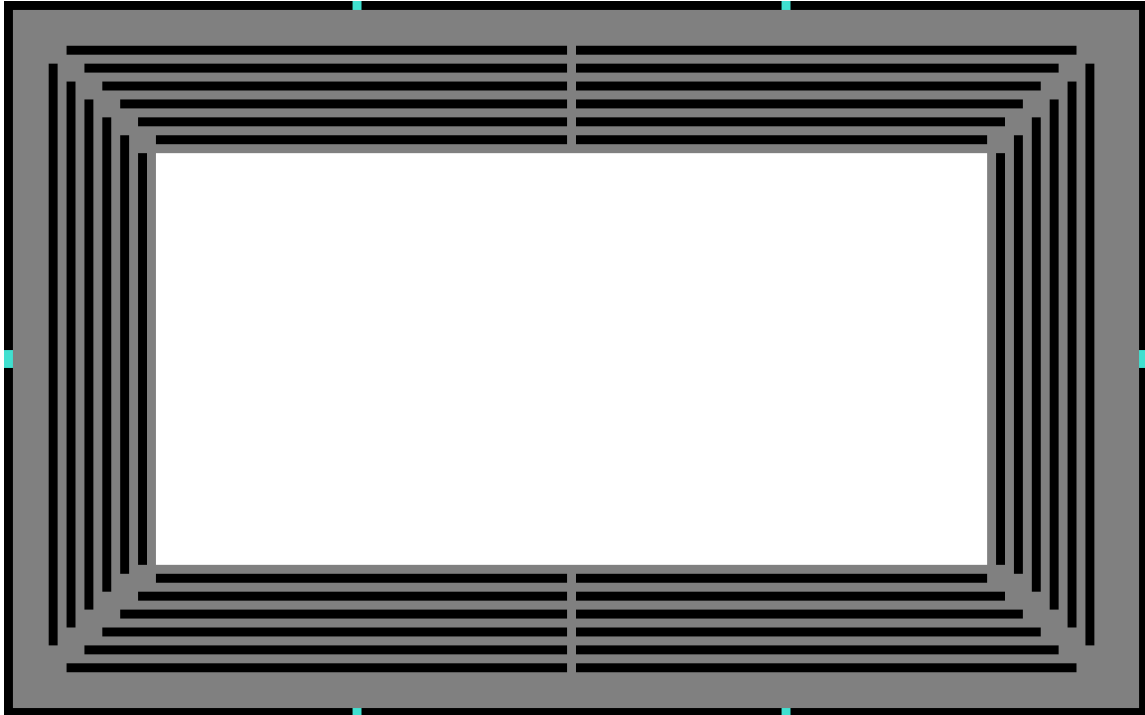


Figure 3.1: Map for game simulation that represents a hockey arena. Black lines represent walls, and the blue areas represent the exits.

This environment consists of a grid system where each cell is represented as a node with distinct properties and states, such as walls, occupied spaces, and passable or icy terrain. Attaching colors to nodes based on their stats makes it possible to draw the map environment and determine which nodes can be traversed or are already occupied. Node states can also be modified in real time to represent environmental changes, like a node becoming occupied.

The grid is set up so that each node (cell) is initialized with coordinates, size, and state, granting direct control over environmental variables. Nodes can represent obstacles like walls, occupied spaces by an agent, and special areas like ice surfaces; these abilities in the node class are used to determine which state a node represents.

Nodes are interconnected, so each can know its immediate surroundings (four-way and eight-way neighbors) depending on the pathfinding algorithm. With all points on the grid connected, it becomes possible to find all the available paths from a specific location. Updating these nodes when it comes to dynamic pathfinding for the D* Lite algorithm is paramount so that the actual changes are represented and

that the algorithm knows about them, such as the sudden appearance of obstacles or the movements of other agents.

This particular design is well-suited for the research question because it replicates real-world situations in 2D game environments where multiple agents interact within a shared space. By emulating these conditions, the simulation provides a solid platform for comparing the efficiency and success rates of the A* and D* Lite algorithms. The nature of the grid and the nodes' ability to update which are occupied or walls or free allows for the examination of each algorithm's adaptability to changes, which is a crucial factor in their practical application in games and other real-time systems.

Using a grid-based model is standard practice in many areas of game development and AI simulation. Having the same framework that peers and industry professionals can easily understand and analyze makes it a good choice. This approach allows for systematic adjustments of variables, making it manageable to conduct controlled experiments and test algorithm performance under various conditions for specific hypotheses.

Discrete grid systems are the chosen method, and even though there are other methods like continuous space simulations, the chosen one is well suited to work with these graph-related algorithms and even if continuous space simulation might have given a more realistic simulation like done in [24] that was not the focus for this thesis. Time constraint for this thesis did not allow for something of that complexity. Lastly, a grid-based approach was chosen because it is commonly used in game development, like in [22]. On top of that, it allows for straightforward use and is simple to implement to show how different algorithms perform under various conditions, like congestion and dynamic obstacles like in [10].

The game class works as the game controller as it handles things like initialization of the game environment, management of game states such as starting, pausing, and finishing the game, and creation and management of game agents, including their positions, movements, and interactions. It has the game loop, which runs the real-time dynamics for the game simulation, like agent updates and the game delay that keeps the agents from moving each frame to once every second. This keeps it from going too fast for visual analysis. The simulation runs for things like odd or harmful behaviors and possible things that can be updated. During the game loop, each agent's move function is called. They all use the same move function as part of the agent's base class, which moves the agent along its designated path if possible. If blocked by another agent, it waits until the next time called to try again. If it can move, it sets its current node state to unoccupied, moves the agent to the new node, and sets its state to occupied.

When agents are initialized, they are given a specific role and placed at various locations. The agent's locations are picked randomly across different stands (starting sections of the game grid around the ice area), as specified in the `setup_agents_random` method as shown in listing 3.1.

Listing 3.1: Code snippet showing how to generate, shuffle, and select positions.

```
available_positions = [(row, col) for row in setting['rows'] for col  
    ↪ in setting['cols'] if (row, col) not in [(astar_row, astar_col)  
    ↪ , (dstar_row, dstar_col)]]  
random.shuffle(available_positions)  
selected_positions = available_positions[:agents_to_place]
```

Within these areas, the placement node is determined by randomly choosing from the available positions in the area and placing agents there until a decided number of agents are filled. The counts for this simulation are 150, 450, 900, and a filled start area of 1380 agents. This random placement simulates varying congestion and interaction dynamics in each game session, reflecting real-world unpredictability.

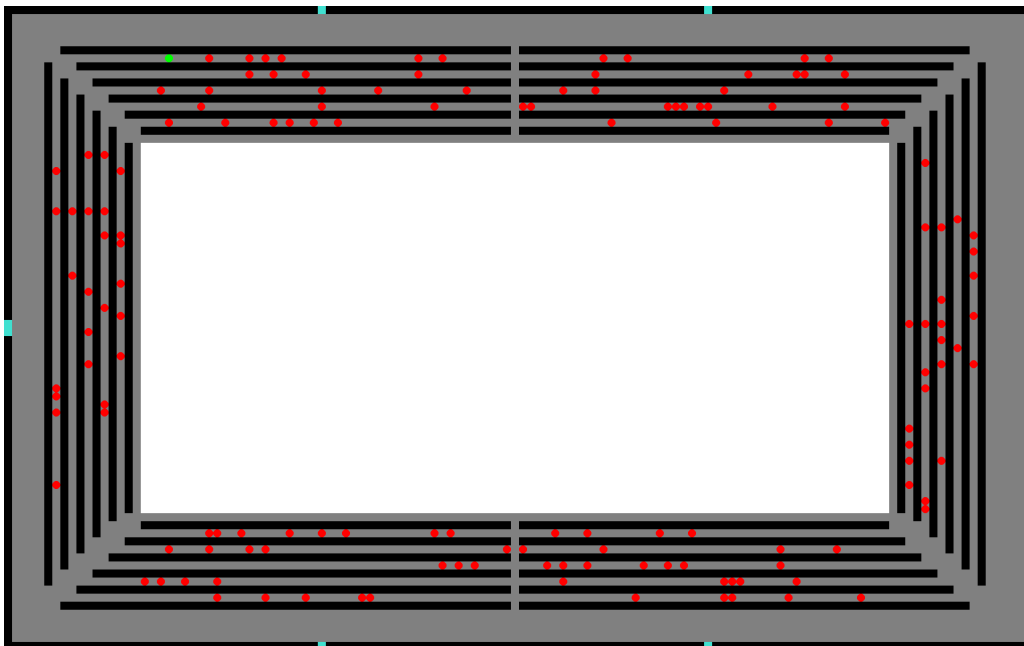


Figure 3.2: Map for one of the runs with randomly placed 150 simple red agents.

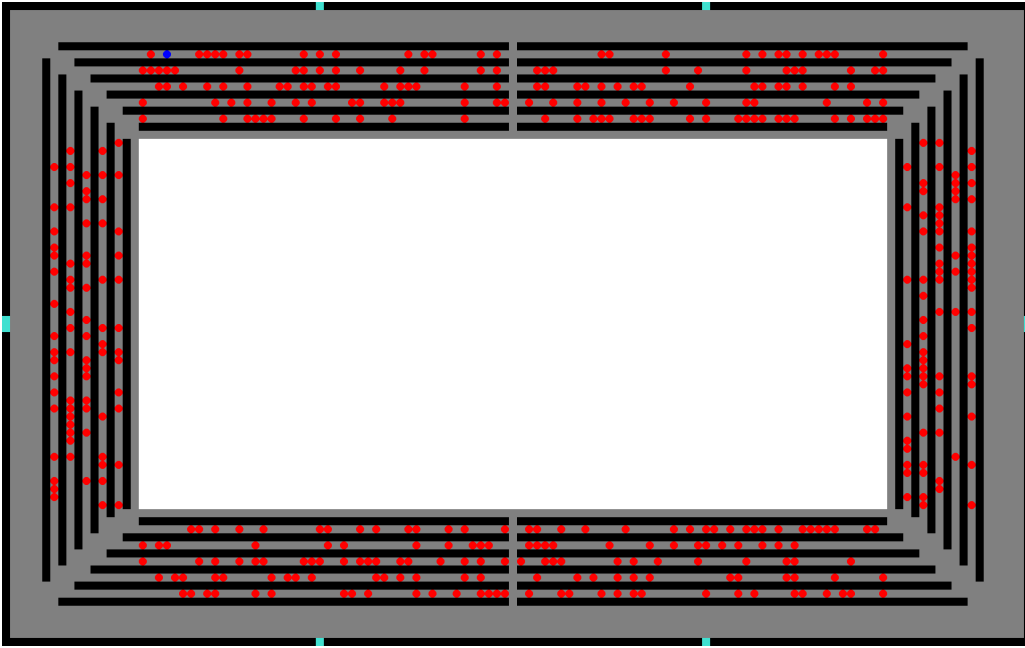


Figure 3.3: Map for one of the runs with randomly placed 450 red agents.

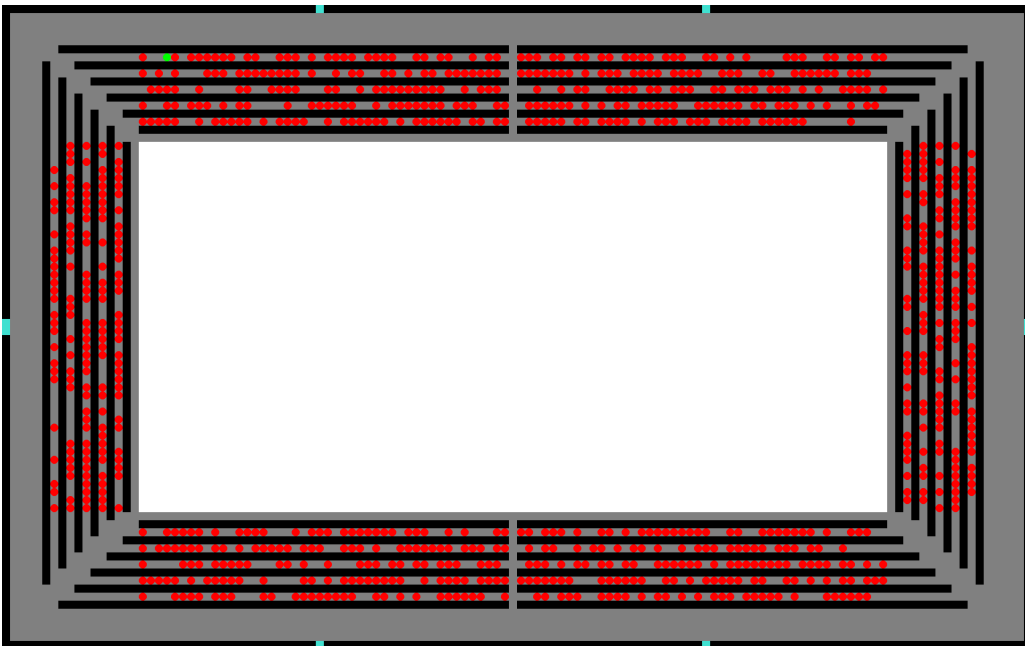


Figure 3.4: Map for one of the runs with randomly placed 900 simple red agents.

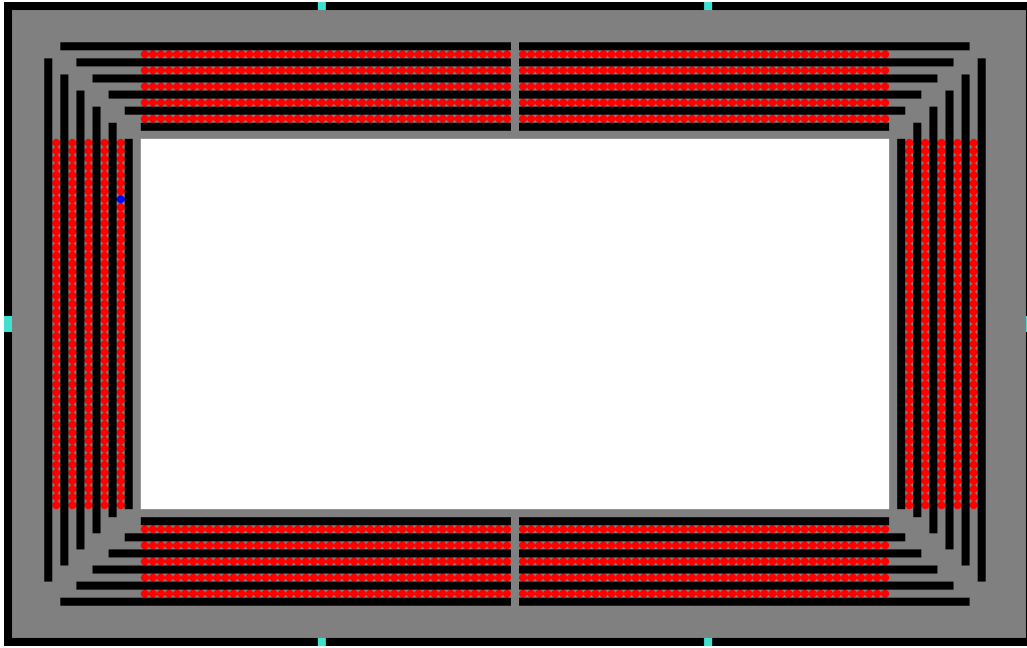


Figure 3.5: Map for one of the runs with randomly placed 1380 simple red agents.

The majority of agents are the so-called simple agents, which use a simple pathfinding algorithm BFS to navigate to the closest exit. Their role is to be dynamic obstacles and cause congestion for the more advanced pathfinding algorithms that are being compared. In each simulation run, one of the agents will be an advanced agent, either A* or D* Lite, and will be initialized on the stands amongst the other agents to find its way out as fast as possible. Running them one at a time will ensure that the test is as just as possible by not having them affect each other.

The simulation map design addresses the research question by being a controlled setting to observe and measure the algorithms' responses to congestion. The map design with these narrow paths in the style of stands garanti congestion and blocked paths, significantly when the number of agents on the map is increased.

3.2.2 Simple Agent (BFS)

The Simple Agent's main task is to be a dynamic obstacle to block and create congestion in the environment. Implementing it with BFS algorithm to navigate towards the closest exit (end node), these simpler agents populate the simulation with varying amounts to test the A* and D* Lite during different levels of obstacle difficulty. By making use of these, the scenario can simulate advanced agents navigating an ever-changing and unpredictable landscape.

Choose to use BFS algorithm for its ability to explore the simulation map and guarantees to find the shortest path to the nearest exit (end node). It is also a relatively simple algorithm to implement and still gets desirable for simulating simple, predictable agent behaviors.

In this context, a unique aspect of the BFS implementation is using heuristic functions for calculations to determine the closest exit (end node). Instead of using some other standard method like Manhattan in [21], or Chebyshev distances from [3],

the Simple Agent uses Euclidean distance [15] to measure actual distances on the game map. This method is used for the A* and D* Lite agents and their diagonal movements, and because it is more realistic and accurate, it is implemented here, too.

Listing 3.2: Python function to calculate Euclidean distance between two nodes.

```
function heuristic(node1, node2):
    dx = absolute value of (node1.col - node2.col)
    dy = absolute value of (node1.row - node2.row)
    return square root of (dx squared + dy squared)
```

Other methods, like Depth-First Search (DFS) lite [21] and random walks [32], were available. However, random walks could introduce too much irregularity and less realistic scenarios for how simple agents affect more advanced agents.

BFS pseudocode over algorithm to find the shortest path between a start node and a target node can be found in the Appendix A.1.

3.2.3 A* agent

The AStarAgent class, derived from the generic agent base class, is initialized with specific starting coordinates within a grid and a set of destination nodes alongside a unique color (green) to visually differentiate it during simulations. The initialization also involves setting up heuristic functions crucial for the A* algorithm, which estimates the cost from the current node to the goal, facilitating efficient path planning.

It has the core functionality of the A* algorithm, which is the ability to determine the shortest path to a designated exit (end node). This starts with finding the closest end node based on the heuristic distance and function utilized by the AStarAgent is Euclidean distances, same as in Listing 3.2, which are suitable for an environment where diagonal movement is allowed, reflecting a more realistic scenario in-game environments where agents may need to navigate around obstacles or other agents dynamically.

The agent uses a combination of g-scores (the cost from the start node to the current node) and h-scores (the estimated cost from the current node to the end node) to manage its pathfinding queue. This ensures that the path with the lowest total estimated cost is given priority. The algorithm used in this simulation is similar to the classic A* algorithm but with a modification that takes into consideration the occupied nodes. An extra cost is assigned to nodes that are occupied, so the A* agent will choose the shortest route with the least amount of agent blocking.

This simulation uses the A* algorithm because they are highly efficient and adaptable in complex environments. This pathfinding method is strong and suitable for examining pathfinding capabilities in crowded scenarios. It provides a comparative analysis against D* Lite to understand which algorithm is better for pathfinding in such scenarios. The modifications made to the standard A* implementation help tailor the algorithm for this particular simulation.

While other approaches may include different heuristic methods or incorporate more advanced and complex agent interactions to simulate more intricate game scenarios, the current methodology adequately addresses the proposed research ques-

tions. It provides a well-balanced mix of theoretical rigor and practical application in the given time span of the thesis.

Pseudocode code shows the main section of the A* pathfinding function, including path reconstruction, which can be found in Appendix A.2.

3.2.4 D* Lite agent

The DStarLiteAgent class uses the base Agent class the same as AStarAgent did, but its algorithm is a modified version of the D* Lite algorithm for dynamic pathfinding comparison. It is efficient at recalculating paths in response to environmental changes, making it ideal for scenarios where obstacles can appear or move. The D* Lite algorithm is crucial to this thesis because it handles dynamic changes in an environment. D* Lite is a streamlined version and just as efficient as the D* algorithm because it can update the shortest path efficiently in real time as the goal or environment changes. As described in [12], it is more straightforward in design and understanding, making it easier to modify for specific applications.

When initialized, the DStarLiteAgent calculates a path to the closest exit (end node), which is determined by the Euclidean distance heuristic function like in Listing 3.2. The function reflects the straight-line distance between points in a grid, the distance as the bird flies. The function ensures that the agent behaves optimally in the simulation by guiding it toward the closest exit.

The DStarLite class follows D* Lite algorithm first by maintaining a list of open nodes and with the use of their g-value (lowest cost to the current node from the start) and rhs values (looks one step ahead to reach current nodes from neighbors) calculates the path. Similarly to the A* to handle the variability of congestion, the algorithms have been modified with an additional cost for nodes occupied by another agent, as shown in listing 3.3 where constant cost is added. Ensuring that the algorithm finds a path and avoids areas of high agent density whenever possible. Leading to possible reduced waiting times and better flow.

Listing 3.3: Function to calculate movement cost between nodes in a grid, considering diagonal movement and occupancy.

```
def get_cost(self, current_node, neighbor_node):
    base_cost = math.sqrt(2) if current_node.row !=
        ↪ neighbor_node.row and current_node.col !=
        ↪ neighbor_node.col else 1
    if neighbor_node.is_occupied:
        return base_cost + self.OCCUPIED_COST
    return base_cost
```

Another improvement to the implementation is its flexibility compared to traditional D* Lite, which allows it to change goals or exit dynamically during the pathfinding process. If a different exit during the simulation becomes more attractive when recalculating due to less congestion or closer proximity, the algorithm shifts the goal. As part of the implementation, when the recalculation is done, it repeats the entire D* Lite process in the agent class, and by doing that, it finds the closest exit from its current position and takes congestion into consideration with the extra

cost.

The traditional D* Lite algorithm was also tailored to fit the specific environment and research questions better. The customized version of the D* Lite algorithm differs from the standard version in that it performs full recalculations every time a blocked node is detected on its current path instead of adjusting the path calculation incrementally with a k_m factor when the graph changes. This decision was based on the high dynamics expected in the environment due to the presence of multiple agents, where changes occur rapidly and require an equally rapid response necessitated by the movement of multiple agents, how it is done can be seen in listing 3.4. Since the only source of dynamic changes in this simulation was the temporary blocking of a node on the current path by other agents, it was deemed appropriate to recalculate the path each time this occurred in order to increase responsiveness.

While the classic D* Lite algorithm uses k_m to adjust the heuristic costs of all nodes when changes happen in the environment to keep the path optimality without recalculating the entire graph, this thesis implementation changes this mechanism with `occupied_cost`. This cost is directly applied to nodes obstructed by other agents, changing the g and rhs values of only those nodes affected by blockages in the path. This method suits the single change dynamic of this simulation with temporary blockages, whereas k_m would be a better choice in a scenario with multiple and varied types of environmental changes. This implementation ensures that the pathfinding remains responsive and efficient, reflecting the specific requirements and constraints of the simulated environment.

Listing 3.4: Pathfinding methods used by D* Lite agent to determine if path blocked and recalculates the path if better one found.

```
def find_path(self):
    end_pos = self.find_closest_end_node()
    start_pos = (self.row, self.col)
    self.path = self.dStarLite.compute_path(start_pos, end_pos
    ↪ )
    self.path.reverse()

def path_blocked(self):
    return any(node.is_occupied for node in self.path[1:])

def move_along_path(self, grid):
    if self.path_blocked():
        self.find_path() # Recalculate the path if it is blocked
    return super().move_along_path(grid)
```

The D* Lite algorithm is a useful tool for studying the performance of dynamic pathfinding algorithms under different environmental constraints, such as narrow passages and agent congestion. This approach helps us understand the efficiency and effectiveness of A* and D* Lite algorithms in managing paths in busy game environments. Using simulations and tests rather than hypothetical discussions can provide actual results and enable scenario comparisons.

Various other methods, like different heuristic functions or more complex agent

interactions, could have been used to simulate different levels of dynamism and complexity. However, the current methodological approach does the job of exploring the research questions. Because of its streamlined nature and being more easily understood than, for example, the D* algorithm made it possible to adapt it to the simulation with the thesis timeline.

Pseudocode code shows the main section of the D* Lite pathfinding function, including path reconstruction, which can be found in Appendix A.3.

3.2.5 Tests and Data collection

A series of controlled simulation tests were performed to evaluate the time performance and success rate of the customized A* and D* Lite algorithms in a dynamic 2D game environment. The objective is to assess how these algorithms perform under various congestion levels caused by different numbers of agents moving within the environment.

The simulations were designed to replicate a real-world gaming scenario where multiple agents move through a narrow space, potentially leading to high congestion and complex navigation. The algorithms were tested with four different levels of agent density: 150, 450, 900, and 1380 agents, which were selected to represent moderate to extremely high congestion within the game environment. To ensure statistical reliability and mitigate any anomalies due to random outliers, the simulation was run multiple times. In the thesis, the two pathfinding algorithms, A* and D* Lite, were tested through multiple simulation runs conducted at twelve unique starting positions for each of the four different agent density levels. Each starting position and agent density level combination was tested ten times for each of the algorithms, leading to 960 total runs. This approach allowed for a comprehensive evaluation of the algorithms' performance across various scenarios within the game environment. First, check the success rate for each run to see if the algorithm was able to find its way out or if it got stuck somewhere. If a path out was found, check time performance; the time it took it from the moment the simulation started to exit the arena was recorded for statistical analysis.

The use of identical scenarios for evaluating the performance of each algorithm would enable a direct comparison of their effectiveness and efficiency. Additionally, simulation would allow testing the algorithms under high-density conditions, which may not be feasible in a physical setup or real-world application due to logistical and resource constraints. By employing simulation, we could quantitatively measure and record critical performance metrics, such as pathfinding time and success rate. These measurements are necessary to address our research questions about algorithm efficiency and reliability.

3.3 Validity and reliability

This research's reliability comes from performing the simulation with different numbers of agents (150, 450, 900, 1380 agents), which helps provide a wide range of scenarios that might be encountered in actual gaming or real-world navigation tasks. The result's validity comes from its ability to accurately simulate a real-world envi-

ronment like a sports arena through game development and test algorithms' navigation abilities in congested areas with metrics for their performance.

To ensure that the algorithm navigation is efficient and broad, the environment contains both obstacles that need to be traversed around and also agent interaction to a high degree because of the high density. The use of twelve random start positions and four different density levels per start position also improves the validity of the test by reducing the likelihood of location bias in algorithm performance. To make sure that the findings do not get skewed by random anomalies or outliers, the amount of data collected needs to be robust, so with that in mind, each simulation scenario was run ten times. Preparing this many times creates a solid basis for statistical analysis and reinforces the conclusion of the data.

The algorithms' performance is based on how well they find the exit out of the simulation environment. This is determined by the algorithm's success rate and the time it takes for it to find its way out of the environment when it is successful. Adding the extra cost of occupied nodes to the algorithm grants them a real-world-like ability to determine the best path based on agents blocking the path. This evaluates the efficiency and adaptability of algorithms, which is required in real-time applications.

A high level of control and repeatability is available by this approach in the thesis, but with the limitation is due to reliance on specific types of congestion and obstacle arrangements since there is only one map/simulation.

That does not encompass all possible game-related or real-world scenarios, even though the result is relevant in similar environments and scenarios, so further validation would be required.

This thesis aims to compare the performance and success rates of the A* and D* Lite algorithms for navigating the simulation. Different agent densities were used during the simulation tests to represent different levels of population and congestion. The results gathered from these runs were the time taken to navigate through the environment and the success rate in reaching the goal. Twelve start positions were spread all over the map to get diverse data, and as a result, 960 runs were performed. The following figures show that the result can vary even for the same algorithm with the same agent density and start position. This is, of course, a result of how the simple agents are randomly spread out over the stands to create a more realistic feel over the game environment area, so sometimes the agent might have several agents blocking their path to the goal, and other times, the path might be more open, especially with lower densities.

One of the factors examined is the success rate for evaluating algorithm performance. Failures are when a simulation run for one of the algorithms does not find a path to the goal. Figure 4.1 illustrates the number of failures for each algorithm across different agent densities.

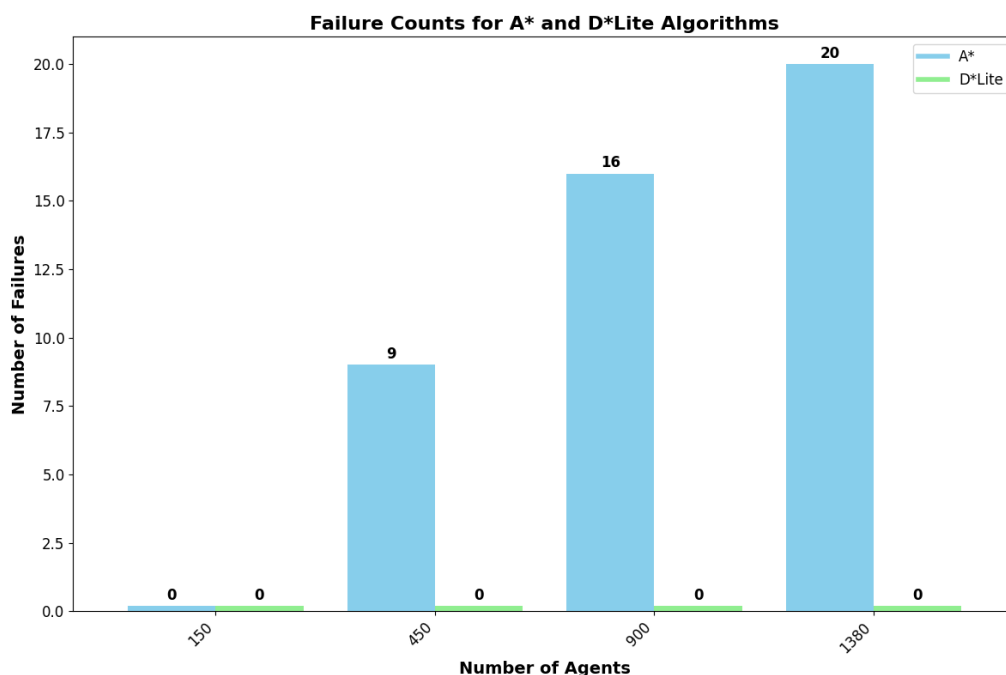


Figure 4.1: Failures for the algorithms per agent group density

A* showed susceptibility to failure, with the number of failures increasing significantly as the number of agents grew. At the highest agent density of 1380, A* failed to find a path in over 20 instances. This highlights a limitation of A* in managing high-density scenarios since it is unable to avoid congestion-related failures like a failure situation when getting stuck. D* Lite, on the other hand, demonstrated remarkable robustness, with zero failures across all agent densities.

In summary, the A* algorithm performs well when the environment has a lower congestion level, but its reliability decreases under pressure, with the risk of failures and, as a result, a worse performance overall. Regarding the D* Lite algorithm, it consistently completes tasks with no failures regardless of the congestion levels or starting positions, indicating its robustness as a dynamic pathfinding algorithm. The results obtained from these simulations can provide valuable insights into the effectiveness of different algorithms used. Based on these insights, one can decide whether to opt for a more complex and resource-intensive algorithm like D* Lite that runs every frame to find the optimal path or a simpler algorithm like A* that only calculates the shortest path once and follows it. Such decisions depend on the specific requirements of the application.

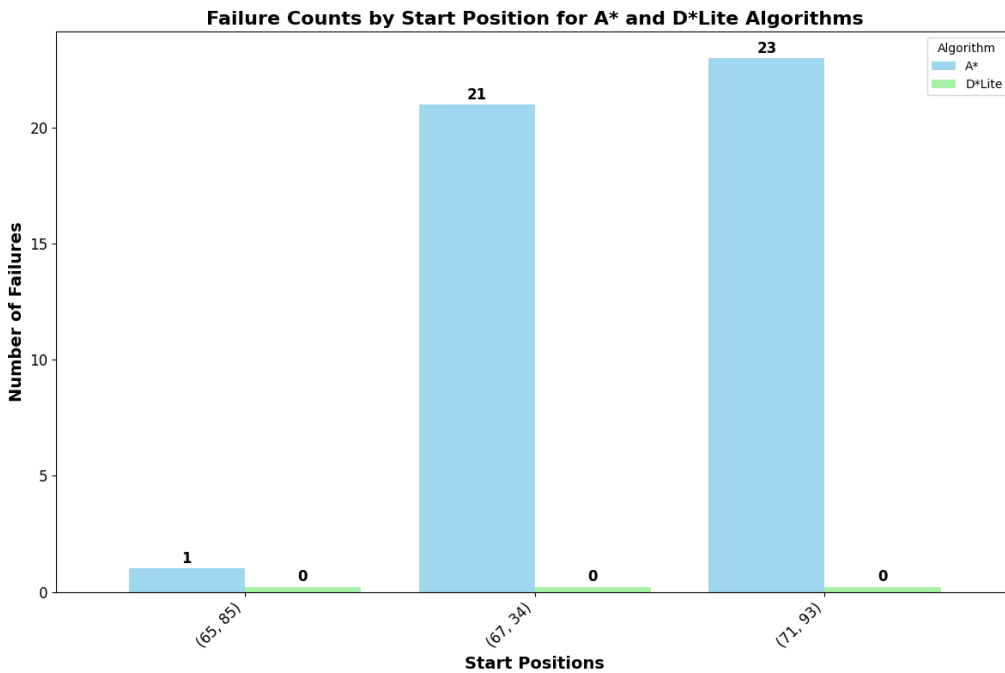


Figure 4.2: Algorithm failures per start position.

Figure 4.2 shows the distribution of failed attempts across different start positions and gives a perspective on which start position for each algorithm causes difficulty regarding adaptability and reliability. The visual analysis of the data highlights a significant difference in the success rate, not just when it comes to the algorithms as previously established, but also how the A* algorithm failures are based on the starting position. It has been determined that some coordinates are more frequent for failures like (67,34) and (71,93). These positions seem to introduce a more challenging navigation scenario for the agents, especially regarding the higher density levels.

Some of the things that cause this could be proximity to obstacles, agent congestion, or a longer and more difficult path to the goal. Interestingly, of the 12 start positions, only three had failures, and as can be seen, two had the majority. For example, (65,85) had drastically less with only one.

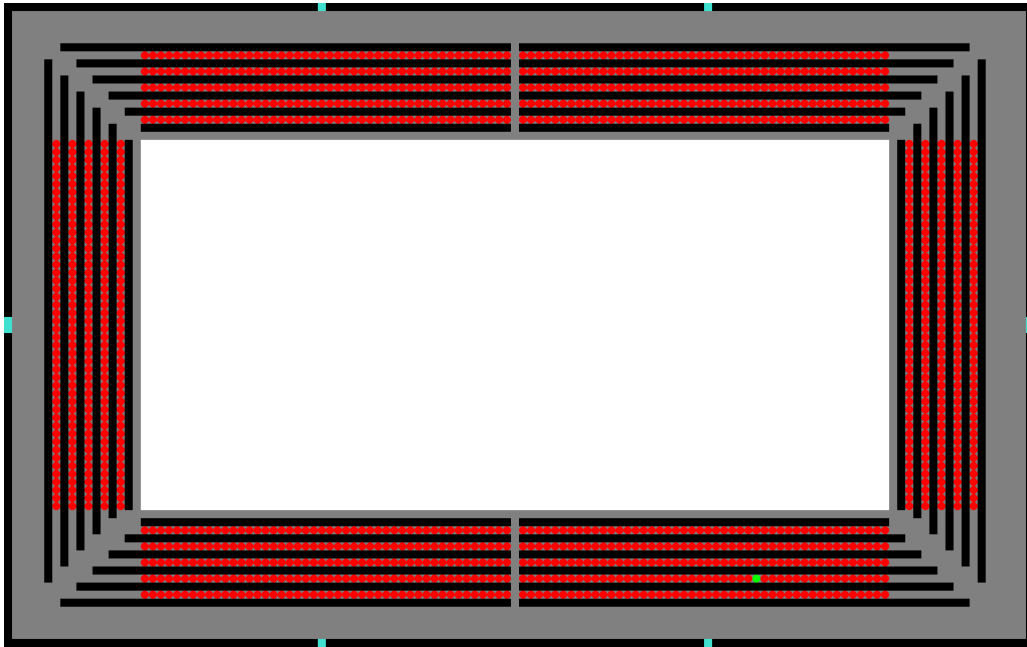


Figure 4.3: Shows troubled start position (71,93) at the start with A* agent.



Figure 4.4: Shows troubled start position (71,93) with A* stuck when all other agents have exited

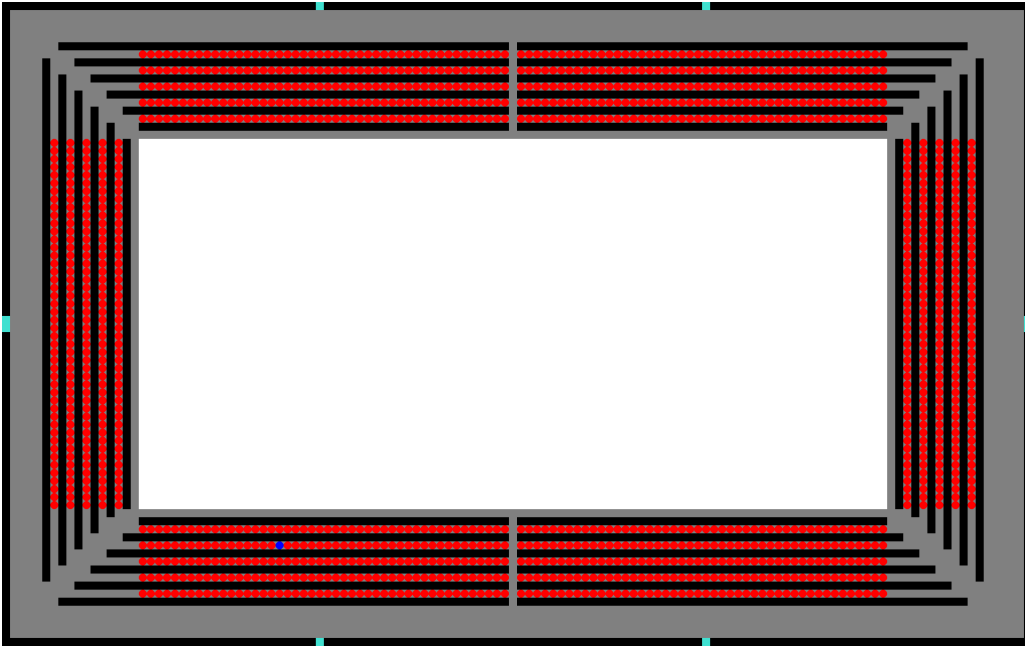


Figure 4.5: Shows troubled start position (67,34) at the start with D* Lite agent.

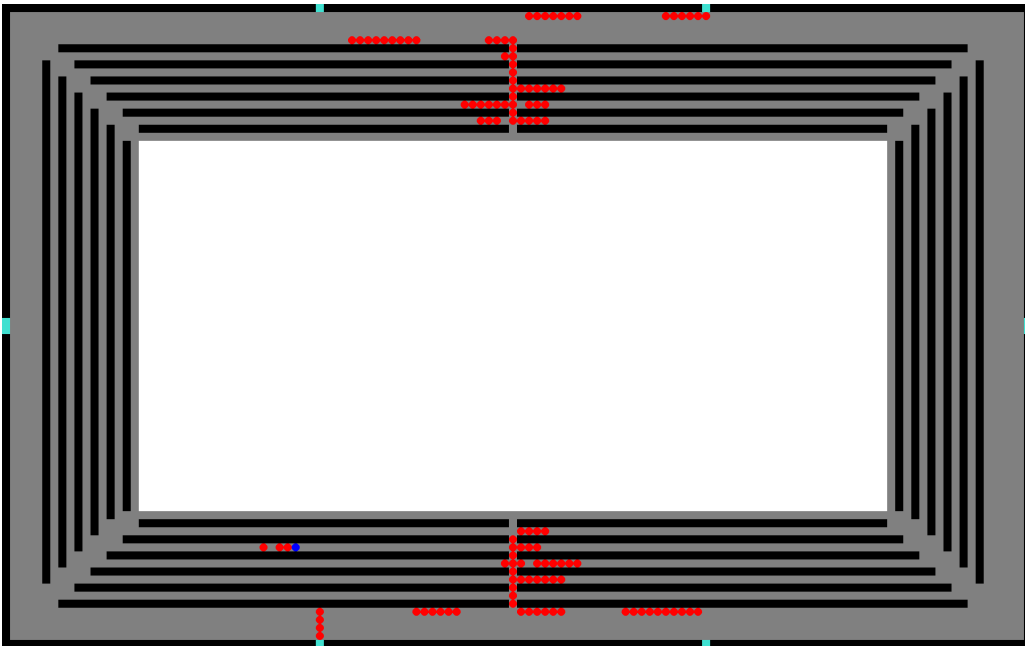


Figure 4.6: Shows troubled start position (67,34) when D* Lite agent has found a new path

The failures and time delays that A* and D* Lite algorithm experience when dealing with the dynamic and congested environment scenario are because of a bad interaction between agents.

When the A* algorithm does its calculations by using a heuristic function and taking the number of other agents into consideration to get the shortest and least populated path to the exit, at the same time, the simple agent that uses BFS finds the

shortest path to its closest exit, and should these paths cross, going in the opposite direction in one of the narrow areas on the stands, that results in obstruction. Since neither the A* nor the BFS consider any dynamic changes or do any recalculations after the first initialization, this leads to an impassable obstacle for them and a deadlock.

This suggests that while A* can be prone to failures at certain challenging start positions, its efficiency is comparable to D* Lite when it is not deadlocked by other agents. This finding emphasizes the necessity of considering these factors when implementing pathfinding solutions in game development or real-world scenarios.

D* Lite did not have a single failure during the test runs, despite agent density or starting position, which shows, as expected, that the algorithm is adaptable to condition changes in the environment. It is worth noting that even though the algorithm can eventually find its way out, it does get stuck for a decent amount of time until enough of the simple agents have cleared out from the area and made another path appealing enough for the algorithm to change it, which makes the pathfinding take a decent amount longer than needed.

The A* and D* Lite algorithms' time to reach the goal was recorded in seconds through multiple simulation runs. Figure 4.7 displays the average run times. In it, we can see that both algorithms have a higher average time when the agent density increases, showing a higher level of congestion and correlation navigation difficulty.

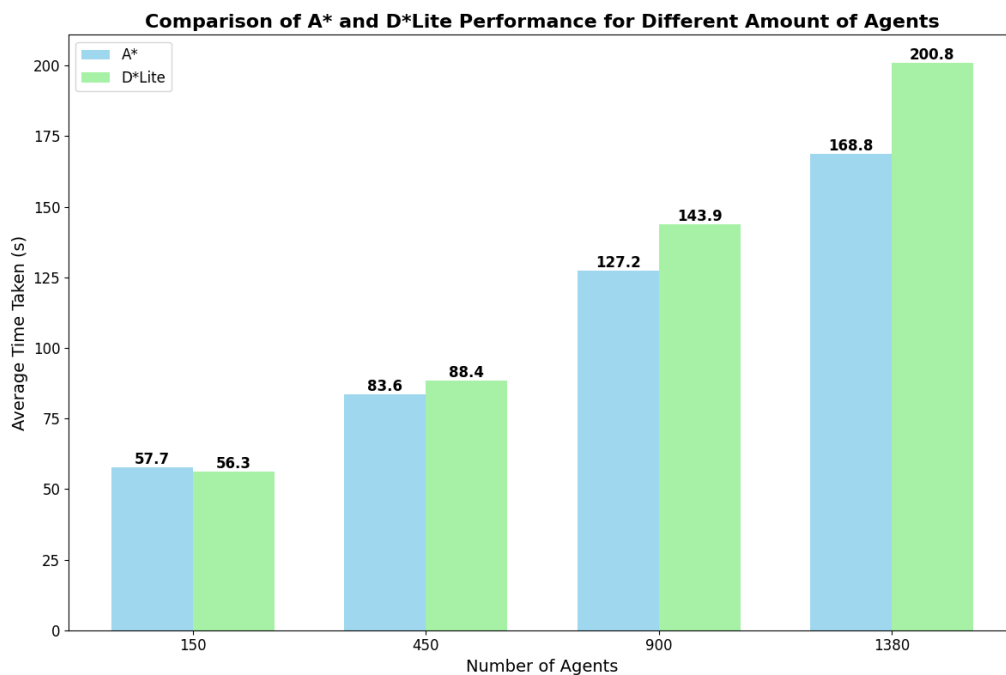


Figure 4.7: Time average for algorithms per agent density

The average completion time performance of the A* and D* Lite algorithm over all four agent density levels is illustrated in Figure 4.7. In this graph, only successful runs have been taken into account, and all failed runs have been excluded to guarantee the viability of the result and not skew it in any way by adding the failures in some way. It shows that the average completion times increased as the density of agents

increased. An expected result that clearly shows a correlation between congestion levels and navigational complexity. In scenarios with lower agent densities of 150, the A* got an average time of 57.7 seconds, and the D* Lite performed similarly by getting a little bit lower with 56.3 seconds. For 450 agents, the A* received an average time of 83.6 seconds, while D* Lite gained a slightly higher average than A* with 88.4 seconds. As the agent density increased to 900 and 1380, the average time for A* rose to 127.2 seconds and 168.8 seconds, respectively. On the opposite side, D* Lite gained a performance average of 143.9 seconds and 200.8 seconds.

This result shows that A* outperforms D* Lite in average completion time for navigating the scenario during different agent densities. However, the D* Lite's ability to circumvent critical failures must be taken into account. As mentioned above, the D* Lite has a perfect success rate for completing all runs, while A* had several failures, particularly specific starting positions at higher densities. These failures are not included in this average time graph, so D* Lite has 45 more runs taken into the graph calculation, most of which from higher density levels, as seen before in 4.1, which has raised the overhaul average time performance of the D* Lite since those took a longer time to navigate through than average. Excluding the failures makes A* look like the better choice from the graph's point of view 4.7, but the failures highlight a significant limitation of A* in highly congested environments.

The boxplot in Figure 4.8 presents the performance of the A* and D* Lite algorithms under varying agent densities (150, 450, 900, and 1380 agents). The boxplot provides several key statistical measures:

- **Median (Red Line):** Indicates the middle value of the dataset.
- **Upper and Lower Quartiles (Box Edges):** Represent the interquartile range (IQR), which includes the middle 50% of the data.
- **Whiskers:** Extend to the smallest and largest values within 1.5 times the IQR from the lower and upper quartiles, respectively.
- **Outliers (Circles):** Data points beyond the whiskers, indicating extreme values.
- **Blue Numbers:** Represent the range (minimum and maximum values).
- **Black Numbers:** Represent the median values.

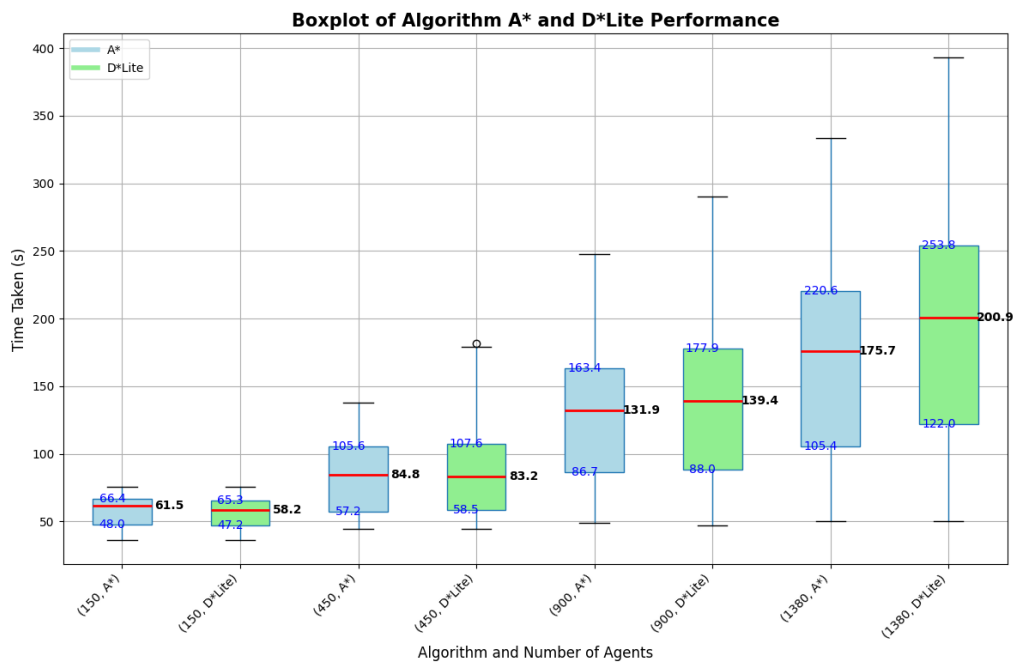


Figure 4.8: Time average for algorithms per agent density

The performance of the A* and D* Lite algorithms across different agent densities in Figure 4.8 can be summarized as follows:

150 Agents

- **A*:** Median time of 61.5 seconds, ranging from 48.0 to 66.4 seconds.
- **D* Lite:** Median time of 58.2 seconds, ranging from 47.2 to 65.3 seconds.
- Both algorithms have similar performance, with D* Lite showing slightly more consistency.

450 Agents

- **A*:** Median time of 84.8 seconds, ranging from 57.2 to 105.6 seconds.
- **D* Lite:** Median time of 83.2 seconds, ranging from 58.5 to 107.6 seconds.
- D* Lite shows comparable performance but with a wider range of whiskers and has an outlier.

900 Agents

- **A*:** Median time of 131.9 seconds, ranging from 86.7 to 163.4 seconds.
- **D* Lite:** Median time of 139.4 seconds, ranging from 88.0 to 177.9 seconds.
- Both algorithms show increased completion times, with D* Lite gaining a higher whisker range and showing a large spread in time values representing the extra work in navigating through all runs.

1380 Agents

- **A***: Median time of 175.7 seconds, ranging from 105.4 to 220.6 seconds.
- **D* Lite**: Median time of 200.9 seconds, ranging from 122.0 to 253.8 seconds.
- Both algorithms here have increased variability due to the extra density, with D* Lite showing higher median times and ranges.

The boxplot in Figure 4.8 shows that as the number of agents increases, median completion times also increase, indicating higher complexity and congestion. D* Lite is generally consistent in lower densities but faces increased variability and computational demands in higher densities. This insight is important for understanding the trade-offs between A* and D* Lite. A* may offer faster average times but with occasional failures, whereas D* Lite provides a more reliable, albeit sometimes slower, solution in highly congested scenarios. By considering both average time and performance variability, it strengthens the findings and grants a deeper understanding.

Another aspect of the result that bears looking into is the performance when the algorithms work as intended without any failures due to deadlocks. In Figure 4.9, the average performance of the algorithms has been calculated, but this time, without starting positions, which caused failures.

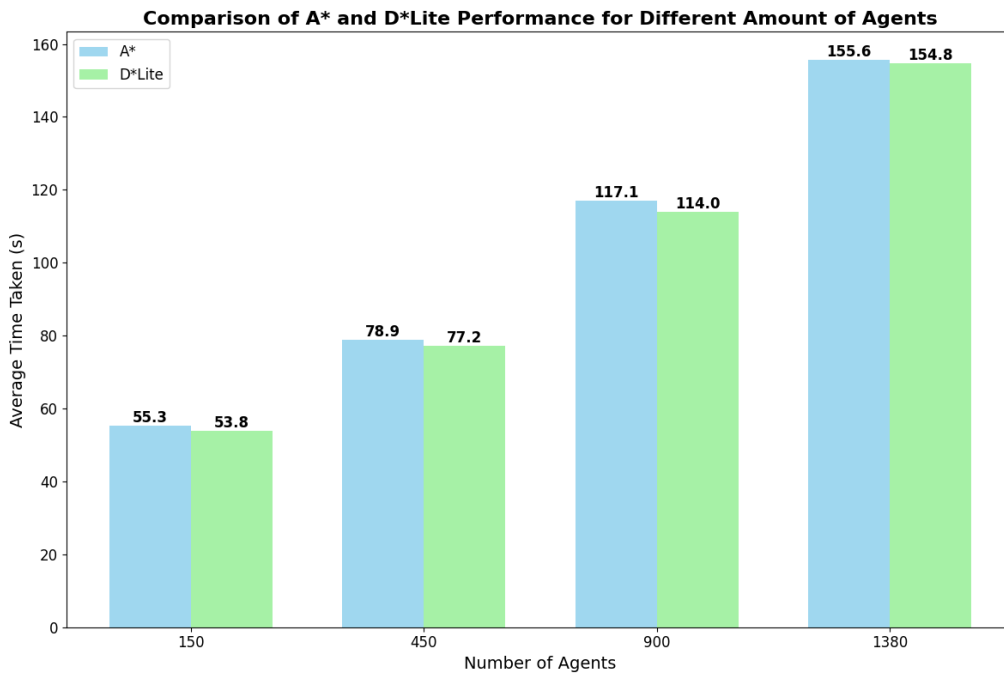


Figure 4.9: Comparison of algorithms average time without tests from start positions that causes failures.

The updated results show a convergence in the average times between A* and D* Lite, particularly at higher densities, as well as an overall lower average performance for both algorithms since all test runs from these troublesome starting positions have been removed. This shows once again that while A* can be prone to failures at certain challenging start positions, its efficiency is comparable to D* Lite when it is not deadlocked by other agents. This finding emphasizes the necessity of considering

these factors when implementing pathfinding solutions in game development or real-world scenarios.

D* Lite algorithms contain their performance here also when problematic starting positions have been excluded, and on top of that, since the problematic positions here were difficult for the D* Lite, this resulted in a much lower average time performance. D* Lite has the ability to adapt to changes and is a reliable choice for applications that prioritize dependability.

It's important to state that even though this shows that the A* can operate on the same level as the D* Lite even in dynamic environments, discounting the significance of removing the data points is important. The fact is that a high number of failures in certain start positions remains a critical consideration, particularly for applications where agents are expected to navigate through scenarios where these types of positions can occur at varying degrees. While the updated figures provide a cleaner comparison, the original data still holds valuable insights into the limitations regarding both of the algorithms.

As part of the visual analysis during all the runs in some of them, when the D* Lite navigating, it cannot pick a better path than its current one if one becomes available. The reason for this is that when a better one becomes available if the current path it is on has no nodes blocked by other agents, the D* Lite algorithm is then unable to recalculate because of its implementation. During the observed times when this happened, it didn't have any major decrements, but if it weren't limited to only recomputing its path to a better one when the current one is blocked, that could have improved its performance.

The research in this thesis examined two renowned pathfinding algorithms, A* and D* Lite, over a series of starting positions on the simulation map, as well as different agent densities.

A reflection on this study explores the intricacies of pathfinding in congested scenarios and the methodological aspects that follow when research is performed on this topic. Using a simulation in this study to approach the reach question effectively captures the complexity of densely populated game settings and makes it possible to set it up in a controlled way to ensure that the result is validated.

When studying these algorithms and evaluating their performance for performance or specific situations like this one, it's important that the study is robust, performing extensive tests with as many variations and controls as possible to avoid anomalies and bias. The thesis was conducted with extensive simulations across various congestion levels to ensure that the observations followed this. However, it must acknowledge the limitations in this thesis posed by having only a singular simulated environment. While a 2D game simulation handles most pathfinding situations, there are limitations and areas where a 3D game environment cannot be accurately represented in 2D. For example, when agents start to fly, for example, with UAVs (unmanned Aerial vehicles), the thesis would have to be expanded upon to a 3D environment, which would require consideration of additional variables such as sensor noise and real-time environmental changes. Even though the study's findings provide valuable insights into the algorithms that could have potential use in real-world applications, the findings are tethered to the specificities of the game-simulated constructed arena but could be expanded to other scenarios such as findings having practical implications for fields like game development, robotics, and autonomous systems.

It hopes that the simulation also sheds light on the pathfinding algorithms' impact on gaming applications or real-world ones. By showing if a more adaptable and robust application like D* Lite, but with higher complexity and computation cost, is a better choice than a more simplistic algorithm like A* that usually is more focused on static environments. This experimental approach of setting up a specific simulation revealed not only a correlation between congestion and navigational difficulties. But also demonstrated the robustness of the algorithms and their ability to handle these types of game simulations with high-level congestion in dense environments. In the result section, it was determined that the average time it took for both algorithms to reach their goal (exit) rose drastically as the number of moving agents in the simulation increased. This result is quite obvious since as more agents obstruct

the environment, it becomes harder to find a clear path through it.

The D* Lite agent shows proficiency in adapting to the circumstances through its dynamic approach to changes. Despite hundreds of test runs with different starting positions and different agent densities, it has a perfect record when it comes to finding its way to the goal (exit) and making it the perfect algorithm for these types of scenarios and when the success rate is a key aspect for the application and developer.

A stark contrast can be seen since A* has a significant failure rate at the highest agent densities compared to the D* Lite. This reveals a limitation within the A* algorithm in handling high-density dynamic environments because of its static approach to pathfinding. This is an essential consideration for application in some areas of game development and several real-world applications when congestion and dynamic changes are a given.

On the other hand, the A* had some problems with certain starting locations, resulting in many failures to reach their goal (exit). This means that compared to D* Lite, it has a harder time dealing with a dynamic changing environment, but only during certain circumstances since most of the starting positions and densities were still handled without problems.

As discussed in the result and analysis section, the A* and D* Lite algorithms show a complex scenario that occurs between algorithm design, agent behavior, and environmental factors. On the one hand, the A* algorithm's failures in densely populated environments underscore its limitation compared to the D* Lite when confronted with dynamic changes. While able to handle congested simulations, its heuristic-based pathfinding cannot handle sudden obstructions, which leads to a failure deadlock state when another agent blocks its predetermined route. On the other hand, D* Lite's dynamic recalculation approach gives it the ability to find another path around the obstacle, avoiding failures but granting time delays as it waits for an appealing path.

The implementation of the D* Lite algorithm is trying to move along its current path, which is the best one. It does this until enough simple agents have moved away, making an alternative route more attractive and cost-effective, and then it starts moving along that one instead. The result is that D* Lite does not fail but may experience increased times to reach the goal due to the nature of its recalculation implementation. For example, at the starting position (71,93), one of the troublesome starting positions at density level 1380 A* failed all ten runs, and D* Lite had an average completion time of 345.077 seconds, which is considerably longer than what would have been needed if it changed path earlier. As can be seen in Figure 5.1, it takes a long time for it to start navigating through the environment, and the majority of the red simple agents have already exited the environment from having been filed with 1380.

A deeper discussion about these observations is warranted when it comes to the design of static vs. dynamic approaches to pathfinding. One of them, the A*, works under the premise that the environment won't change after calculating the most optimal path, which is not an optimal approach in dynamic simulations. On the other side, the D* Lite continuous computation of the environment when nodes on its path are blocked is a more fitting approach when operating in the unpredictable nature of dynamic environments, even if the cost of real-time computational resources

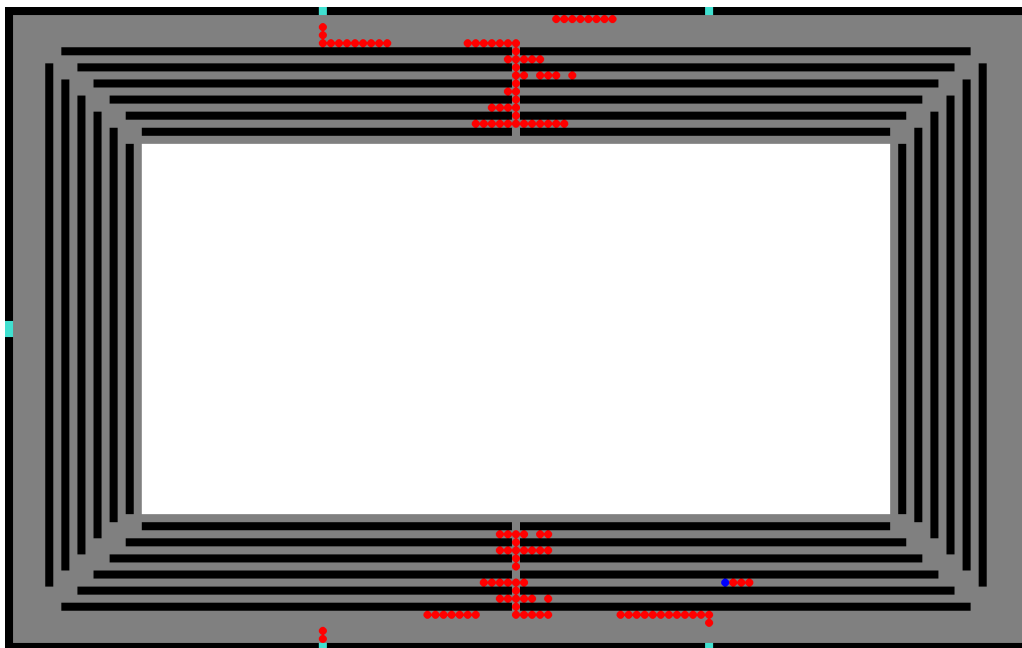


Figure 5.1: D* Lite agent in blue during a test run at start position (71, 93) when new path found and started moving.

and delays result from an ever-going decision-making process.

It might not be as cut and dry as that since it has been determined that the starting position is the main factor in failures for the A*. The A* algorithm fails to work as expected, but it's not always because of bad design. Instead, it might be because it is being used in a non-optimal way or environment. The A* algorithm is a tool that helps find the most efficient path between two points in a static environment. With a few changes to the algorithm to aid in these deadlock situations, like either adding some ability to it so it can determine areas on the map where it can be obstructed by other agents, as seen here, or having some sort of check that if the A* agent hasn't moved for a period of time, it once again tries to find the best path from its current position.

A deep dive was performed in the result section to determine the type of impact these starting positions had on the algorithms' overhaul efficiency, where the starting position data were removed for those that caused failures. This adjustment clearly shows the impact it has on the result, as highlighted in Figure 4.9, by revealing that the efficacy of the A* aligns more closely with that of D* Lite under certain conditions. This finding has an impact not just on theoretical understanding but also on practical implementation when it comes to choosing pathfinding algorithms for game development or other real-world situations where dynamic changes are frequent.

Finally, this indicates that A*, a traditionally considered static pathfinding algorithm, showed a surprising ability to navigate effectively in a highly dynamic, congested environment. This is evident from the median and range of completion and average times, particularly in lower-density scenarios where A* performs similarly to D* Lite without failures. The result also showed that A* was prone to deadlock failures in higher congestion scenarios, which must be considered. Such failures occur when blocked by other agents, which stops its path. To avoid this

main issue, one must either consider the map design and start positions to prevent such problems or the A* could be modified the A* to improve its adaptability in dynamic settings. This underscores how important it is to think over the specific application requirements when choosing A*.

The modified evaluation of time performance in Figure 4.9 also showcased an even lower average time for D* Lite by removing the failure-prone start positions. This points out these types of situations as major weak points for both algorithms in this simulation, and future work should take that into consideration when further testing or improvements are made to the algorithms.

The D* Lite was shown to be more reliable when it comes to performance, with no failures but generally higher completion times in more congested environments. This would be a result of its prolonged recalculations to navigate through all highly populated environments. This emphasizes that D* Lite's adaptability makes it a good choice for environments with a high degree of unpredictability and congestion where the success rate is a key factor. When looking into future work for both algorithms in regard to weak points, the main thing should be the deadlock situations.

The design of the simulation is the rock bed upon which the study builds. It aims to obtain accurate data so that a proper analysis can be performed on the surrounding data. Yet further inquiry into the depth and breadth of different environmental dynamics and their impact on pathfinding could be a great opportunity to build upon this thesis. Expanding the study to incorporate various dynamic scenarios, possibly even different topographies or more complex agent interactions.

Other potential future studies could focus on optimizing A* for dynamic environments or at least being able to handle situations where other agents can cause deadlocks, and the same could be needed for the D* Lite algorithm that also could get stuck for long periods of time in the same way. Another investigation of integrating both A* and D* Lite characteristics to create a hybrid algorithm that combines the efficiency of the A* algorithm in finding the shortest path without the need for constant recalculations with the adaptability of D* Lite in high-density scenarios such to avoid any failures and gaining a successful pathfinding. Advancements in this technology could have significant implications, including the use in heavy-duty game environments with lots of agents where a response and cost-efficient algorithm is needed, as well as in the development of autonomous navigation systems in robotics and vehicular technology.

As we transition from these specific observations to the broader implications for the field of pathfinding, it is evident that the choice of algorithm can profoundly impact the success of navigation tasks. The discussion demonstrates that while a classic A* algorithm may encounter difficulties in certain complex scenarios, its failures are not solely attributable to flawed design. Instead, it suggests that its application needs to be tailored to its strengths—efficient navigation in less dynamic environments.

Chapter 6

Conclusions and Future Work

The research question of this thesis was: "What are the differences between the A* and D* Lite algorithms regarding time performance and success rate of navigating through a 2D game environment with a high number of agents causing congestion?"

Based on the extensive simulations and data analysis conducted, the following key findings address the research question:

- **Time Performance:** Both algorithms showed an increasing average time performance as agent density increased due to increased congestion. The A* algorithm showed faster average completion times than D* Lite, particularly in higher-density environments when looking at higher-density levels.
- **Success Rate:** The D* Lite algorithm outperformed the A* algorithm in terms of success rate. The A* algorithm experienced several failures, particularly in high-density scenarios where it became stuck due to congestion. However, the D* Lite algorithm successfully navigated through all tested scenarios without failure, showcasing its robustness in dynamic environments, but as a result, demonstrated slower average time performance at higher density levels due to difficulty navigating through tough situations.
- **Adaptability:** The D* Lite algorithm's ability to recalculate paths dynamically in response to changing conditions allowed it to handle congested environments more effectively than the A* algorithm. This adaptability is crucial in real-world applications where environments are rarely static. Being a more simplistic static algorithm and easy to adapt, A* still showed an ability to navigate a majority of runs, only having trouble in certain starting positions.

This thesis looked into the A* and D* Lite algorithms when it comes to their ability to navigate within dynamic environments with congestion. During the research into related works and other research, it became obvious that adaptability, collision avoidance, detection, and performance of pathfinding strategies were priorities. This thesis, instead, focused on environments where it is dense either from the environment itself or from the number of agents and also in unpredictable scenarios, which is often encountered in gaming simulations and real-world navigation tasks.

The thesis result helps with building on the current understanding of the static A* and dynamic D* Lite by showing how they react in these environments and what possible modifications can be performed to improve their work even further.

Research shows the critical role of pathfinding algorithms in enabling AI agents to navigate complex and dynamic environments. It highlights the need for algorithms

to be versatile and responsive but also preferably low computational cost. Since, in high-density environments, they might need to recalculate each frame to handle sudden changes in real-world conditions that can grow exponentially if more than one dynamic agent is used. The algorithms should quickly be able to adapt to new situations with the help of a pathfinding strategy so that the agent can safely and efficiently navigate through the application's different terrains. This underscores the importance of choosing the right pathfinding strategy and developing a pathfinding algorithm that is best suited for the task at hand but also keeps up with the pace of gaming and real-world conditions.

From the test simulation and analysis, the result findings form a clear picture where D* Lite shows off its ability to reach the goal consistently and performance to navigate through different degrees of agent densities. Then, on the other side, the A* algorithm shows good results in less congested simulations and is also able to navigate through dynamic obstacles by finding the shortest path and staying on that path, basically waiting in line. A drawback shown in the study for the simpler static A* algorithm was that despite being able to handle a majority of starting positions and agent density variations, it did have several failures where it could not find a path out of the simulation. This can be seen in the result section, but it also mentioned that D* Lite has similar difficulties in the same situations. Even though it didn't have any failures, it did cause delays in its navigation process.

It showed that the pathfinding algorithm's effectiveness is heavily correlated with and influenced by the changing conditions and features of the environment in which it is applied. Therefore, choosing which algorithm and how it is implemented for the task at hand requires careful consideration to match the dynamics and difficulties of the environment.

The thesis is limited by its specific scope of agent densities tested environments. To capture the full idea of the algorithm performance, future research should include additional pathfinding algorithms, a broader range of evaluation metrics, and a more varied set of maps. More maps would, for example, help test the conditions for the deadlock failures further and aid in learning to avoid such, but might also show other troublesome situations not apparent in the current environment. This will ensure a more comprehensive comparison and provide deeper insights into the strengths and weaknesses of each algorithm.

In conclusion, A*'s failures result from using a static pathfinding approach in an environment that is dynamically changing, and agents' movement and interaction with other agents can render a chosen path infeasible. Despite the failures when only looking at the overhaul time to navigate out the environment map with the different agent densities causing congestion, it performed just as well as the D* Lite algorithm or even better average time. D* Lite's approach with dynamic changes can avoid failure, although it results in a longer pathfinding time to reach the exit as it waits for conditions to change and new paths to emerge. So the D* Lite is a safer pick if the success guarantee is what you are looking for. Still, it is a more complex and computation-heavy algorithm, so if an application contains a situation where an agent can end up in a deadlock situation with another agent, then the D* lite is the better pick, and otherwise, A* would be a good pick.

Multiple research possibilities are presented during this thesis comparison. For further research and algorithm improvement, a focus should be on the specific diffi-

culties related to the failures that occurred in connection with specific start positions for A*. Understanding that environmental features can cause increased failure rates can lead to improvements in heuristic-based pathfinding algorithms. A new adaptation or modification to both A* and D* Lite so that they can better handle the situations where agents' paths go in opposite directions and cause deadlocks. By making it possible for the A* and D* Lite to sense when they have been stuck for a longer period of time and then be forced to recalculate a different path. Another approach could be to implement an ability to determine locations where this is prone to happen and avoid them or even log directions and movement of other agents nearby and predict their behavior/path to avoid it.

Other interesting avenues of investigation could be to test these algorithms in other simulations to see how they handle varied environmental conditions and with other types of behavior agents. Doing that could lead to an even deeper understanding of the differences between these algorithms in different scenarios and could help identify further areas of weakness within their implementation. For example, all simple agents that cause congestion in this thesis use the same type of pathfinding algorithm and strategy, trying to find the shortest path to the closest exit. Many different simulations could be set here to continue this work, like using different algorithms for different types of behavior agents such as Theta* [33], Jump Point Search (JPS) [34], and having different goals for different agents and maybe even adding in some random elements either in the agent's movement pattern or in the environment itself with other obstacles to really test the adaptability and robustness of the algorithms. On top of modifying the simple agents to test specific scenarios, the environment can also be modified to represent other areas where congestion is common for some reason or another. For example, other sports events, stadiums, amphitheatres, outdoor demonstrations, etc.

Moreover, using a wider range of evaluation metrics for the comparison, such as computational efficiency and scalability, will capture a better picture of the algorithms' capabilities. Testing in real-world scenarios will also be crucial to validate these findings before they can be applied in different domains, including autonomous vehicles, robotics, or crowd control.

A possible future use for these algorithms or a modified version of them based on this thesis could be when it comes to crowd control. The idea for the thesis came from the tight packed environment in a sports stadium where thousands of people are either trying to get in or out and, as a result, cause large lines of people to wait. Instead, based on a person's seat, an algorithm could be used to determine the best entrance and the shortest path to their seat, which would lessen congestion and reduce the overhaul waiting time.

At the end of this thesis, it has become clear that pathfinding is a far-reaching and complex field. It requires a comprehensive understanding of the environment, how routes are planned and executed, and the ability to adapt to unexpected scenarios. Developers must understand how different algorithms handle these points when choosing. By laying the groundwork, this research hopes to aid in future research in this field, with new possibilities for researchers to improve upon and refine the findings. With further work into AI navigation systems, revolutionary discoveries could be made on how it interacts with people and their surroundings in anything from virtual reality experiences in games to cars navigating the real-world pathways

in cities. The quest to improve pathfinding algorithms is ever ongoing, and more efficient, robust, and versatile versions are discovered all the time, making the potential benefits endless.

References

- [1] J. Ahn, S. Jung, H. Kim, H.-J. Hwang, and H.-B. Jun, “A study on unmanned combat vehicle path planning for collision avoidance with enemy forces in dynamic situations,” *Journal of Computational Design and Engineering*, vol. 10, no. 6, pp. 2251–2270, 11 2023. [Online]. Available: <https://doi.org/10.1093/jcde/qwad099>
- [2] Z. A. Algfoor, M. S. Sunar, and H. Kolivand, “A comprehensive study on pathfinding techniques for robotics and video games,” *International Journal of Computer Games Technology*, vol. 2015, pp. Article ID 736 138, 11 pages, 2015. [Online]. Available: <https://doi.org/10.1155/2015/736138>
- [3] C. D. Cantrell, *Modern mathematical methods for physicists and engineers*. Cambridge University Press, 2000.
- [4] E. E. Castillo Osorio and H. H. Yoo, “Simulation of evacuation route scenarios through multicriteria analysis for rescue activities,” *Journal of the Korean Society of Surveying, Geodesy, Photogrammetry and Cartography*, vol. 37, no. 5, pp. 303–313, 2019.
- [5] P.-H. Chen and F. Feng, “Evacuation in buildings: Path-finding for real-time evacuation systems,” in *Proceedings of the 22nd International Symposium on Automation and Robotics in Construction*. Ferrara, Italy: International Association for Automation and Robotics in Construction (IAARC), September 2005.
- [6] C. Crespi, G. Fargetta, M. Pavone, R. A. Scollo, and L. Scrimali, “A game theory approach for crowd evacuation modelling,” in *Bioinspired Optimization Methods and Their Applications*, B. Filipič, E. Minisci, and M. Vasile, Eds. Cham: Springer International Publishing, 2020, pp. 228–239.
- [7] L. Critch and D. Churchill, “Combining influence maps with heuristic search for executing sneak-attacks in rts games,” in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 740–743.
- [8] P. S. Foundation, “Python language reference,” <https://www.python.org/>.
- [9] A. Gaitanis, A. Lentzas, G. Tsoumakas, and D. Vrakas, “Route planning for emergency evacuation using graph traversal algorithms,” *Smart Cities*, vol. 6, no. 4, pp. 1814–1831, 2023. [Online]. Available: <https://www.mdpi.com/2624-6511/6/4/84>
- [10] J. Jin, Y. Zhang, Z. Zhou, M. Jin, X. Yang, and F. Hu, “Conflict-based search with d* lite algorithm for robot path planning in unknown dynamic

- environments,” *Computers and Electrical Engineering*, vol. 105, p. 108473, 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0045790622006887>
- [11] K. Khantanapoka and K. Chinnasarn, “Pathfinding of 2d 3d game real-time strategy with depth direction a algorithm for multi-layer,” in *2009 Eighth International Symposium on Natural Language Processing*, 2009, pp. 184–188.
- [12] S. Koenig and M. Likhachev, “D* lite,” in *Proceedings of 18th AAAI Conference on Artificial Intelligence (AAAI '02)*, July 2002, pp. 476 – 483.
- [13] S. R. Lawande, G. Jasmine, J. Anbarasi, and L. I. Izhar, “A systematic review and analysis of intelligence-based pathfinding algorithms in the field of video games,” *Applied Sciences*, vol. 12, no. 11, 2022. [Online]. Available: <https://www.mdpi.com/2076-3417/12/11/5499>
- [14] L. Liu, B. Wang, and H. Xu, “Research on path-planning algorithm integrating optimization a-star algorithm and artificial potential field method,” *Electronics*, vol. 11, no. 22, 2022. [Online]. Available: <https://www.mdpi.com/2079-9292/11/22/3660>
- [15] M. Negnevitsky, *Artificial Intelligence: A Guide to Intelligent Systems*. Pearson Education Limited, 2011. [Online]. Available: <https://books.google.se/books?id=8mmpBwAAQBAJ>
- [16] K. Neuschwander, R. Dornberger, and T. Hanne, “Collision avoidance of multiple moving agents by adapting the a* algorithm,” in *2023 9th International Conference on Automation, Robotics and Applications (ICARA)*, 2023, pp. 49–54.
- [17] M.-K. Ng, Y.-W. Chong, K.-m. Ko, Y.-H. Park, and Y. Leau, “Adaptive path finding algorithm in dynamic environment for warehouse robot,” *Neural Computing and Applications*, 2020. [Online]. Available: <https://dx.doi.org/10.1007/s00521-020-04764-3>
- [18] D. Ostrowski, I. Pozniak-Koszalka, L. Koszalka, and A. Kasprzak, “Comparative analysis of the algorithms for pathfinding in gps systems,” *ICN 2015*, p. 114, 2015.
- [19] “Pygame documentation,” <https://www.pygame.org/docs/>, Pygame, 2024, accessed: 2024-02-01.
- [20] A. Rafiq, T. A. A. Kadir, and S. N. Ihsan, “Pathfinding algorithms in game development,” *IOP Conference Series: Materials Science and Engineering*, vol. 769, no. 1, p. 012021, feb 2020. [Online]. Available: <https://dx.doi.org/10.1088/1757-899X/769/1/012021>
- [21] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson, 2016, paperback.
- [22] R. N. Sarbini, I. Ahmad, R. O. Bura, and L. Simbolon, “Development of pathfinding using a-star and d-star lite algorithms in video game,” *Journal of Theoretical and Applied Information Technology*, vol. 102, no. 3, pp. 832–841, 2024. [Online]. Available: <http://www.jatit.org>

- [23] Y. Sazaki, H. Satria, and M. Syahroyni, "Comparison of a* and dynamic pathfinding algorithm with dynamic pathfinding algorithm for npc on car racing game," in *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, 2017, pp. 1–6.
- [24] M. Schader and S. Luke, "Exploring planner-guided swarms running on real robots," in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics. The PAAMS Collection*, P. Mathieu, F. Dignum, P. Novais, and F. De la Prieta, Eds. Cham: Springer Nature Switzerland, 2023, pp. 307–319.
- [25] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0004370214001386>
- [26] M. Sinkar, M. Izhan, S. Nimkar, and S. Kurhade, "Multi-agent path finding using dynamic distributed deep learning model," in *2021 International Conference on Communication information and Computing Technology (ICCICT)*, 2021, pp. 1–6.
- [27] T. Sliacka, M. Varga, and N. Adamko, "Application of the a* algorithm for navigation of workers in simulation models of railway yards," in *2021 International Conference on Information and Digital Technologies (IDT)*, 2021, pp. 319–325.
- [28] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1994, pp. 3310–3317 vol.4.
- [29] A. T. Stentz, "The focussed d* algorithm for real-time replanning," in *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI '95)*, August 1995, p. 1652 – 1659.
- [30] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar, R. Barták, and E. Boyarski, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *Proceedings of the International Symposium on Combinatorial Search*, vol. 10, pp. 151–158, 09 2021.
- [31] C. Wei, K. V. Hindriks, and C. M. Jonker, "Multi-robot cooperative pathfinding: A decentralized approach," in *Modern Advances in Applied Intelligence*, M. Ali, J.-S. Pan, S.-M. Chen, and M.-F. Horng, Eds. Cham: Springer International Publishing, 2014, pp. 21–31.
- [32] F. Xia, J. Liu, H. Nie, Y. Fu, L. Wan, and X. Kong, "Random walks: A review of algorithms and applications," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 2, pp. 95–107, 2020.
- [33] Y. Zhang, Y. Hu, J. Lu, and Z. Shi, "Research on path planning of mobile robot based on improved theta* algorithm," *Algorithms*, vol. 15, no. 12, 2022. [Online]. Available: <https://www.mdpi.com/1999-4893/15/12/477>
- [34] K. Zhou, L. Yu, L. Ziwei, and S. Mo, "Local path planning of driverless car navigation based on jump point search method under urban environment," *Future Internet*, vol. 9, p. 51, 09 2017.

A.1 BFS (breadth first search)

Listing A.1: Pseudocode of the BFS algorithm to find the shortest path between a start node and a target node.

```
Function bfs_to_target(start_node, target_node)
    queue = initialize queue with start_node
    visited = initialize set with start_node
    came_from = dictionary with start_node mapped to None

    while queue is not empty
        current_node = dequeue from queue

        if current_node == goal
            return construct_path(came_from, target_node)

        for each neighbor of current
            if neighbor not in visited and not neighbor.is_wall
                visited.add(neighbor)
                queue.enqueue(neighbor)
                came_from[neighbor] = current_node

    # Return empty path if goal is not found
    return empty path

Function construct_path(came_from, target_node)
    # Reconstruct path from target_node to start_node
    path = initialize empty list
    current = target_node

    while current is not None
        prepend current to path
        current = came_from[current]

    return path
```

A.2 A*

Listing A.2: Python pseudocode code for main A* pathfinding function, including path reconstruction.

```

Function bfs_to_target(start_node, target_node, grid)
    queue = initialize queue with start_node
    visited = initialize set with start_node
    came_from = empty map with start_node mapped to None

    while queue is not empty
        current_node = dequeue from queue

        if current_node == target_node
            return construct_path(came_from, target_node)

        for each neighbor of current_node in four_way_neighbors
            if neighbor is not visited and not neighbor.is_wall
                add neighbor to visited
                enqueue neighbor to queue
                came_from[neighbor] = current_node

    return empty list

Function construct_path(came_from, target_node)
    path = empty list
    current = target_node

    while current is not None
        prepend current to path
        current = came_from[current]

    return path

```

A.3 D* Lite

Listing A.3: Pseudocode for D* Lite pathfinding algorithm

Algorithm DStarLite

CONSTANT OCCUPIED_COST

```

Initialize(grid)
    set grid
    initialize open_list as an empty priority queue
    set g and rhs values for all nodes in grid to infinity

```

```
initialize path as empty list
```

```
Function node_from_position(position)
```

```
return the node in grid at the specified position
```

```
Function initialize()
```

```
for each node in grid
```

```
    set g and rhs values to infinity
```

```
    set parent of node to None
```

```
clear and reinitialize open_list as empty priority queue
```

```
Function update_node(node)
```

```
calculate key for node
```

```
if node exists in open_list
```

```
    remove node from open_list
```

```
insert node with its key into open_list
```

```
Function calculate_key(node)
```

```
heuristic_cost = heuristic(node, end)
```

```
return minimum of g[node] and rhs[node] plus heuristic_cost
```

```
Function compute_path(start_pos, end_pos)
```

```
initialize()
```

```
start = node_from_position(start_pos)
```

```
end = node_from_position(end_pos)
```

```
set rhs value of end to 0
```

```
update_node(end)
```

```
while open_list is not empty
```

```
    current_node = extract node with smallest key from  
    ↪ open_list
```

```
    if g[current_node] > rhs[current_node]
```

```
        set g[current_node] to rhs[current_node]
```

```
    else
```

```
        set g[current_node] to infinity
```

```
        update_rhs(current_node)
```

```
    for each neighbor of current_node
```

```
        if neighbor is not a wall or ice
```

```
            update_rhs(neighbor)
```

```
    if current_node is start and g[current_node] equals rhs[  
    ↪ current_node]
```

```
        break
```

```
return reconstruct_path(start, end)
```

```
Function update_rhs(node)
```

```
if node is not end
```

```
    set min_cost to infinity
```

```
    for each neighbor of node
```

```
    cost = g[neighbor] + get_cost(node, neighbor)
    if cost < min_cost
        min_cost = cost
        set parent of node to neighbor
    set rhs[node] to min_cost
if g[node] does not equal rhs[node]
    update_node(node)
```

Function get_cost(current_node, neighbor_node)

```
    if movement is diagonal
        base_cost = sqrt(2)
    else
        base_cost = 1
    if neighbor_node is occupied
        return base_cost + OCCUPIED_COST
    return base_cost
```

Function reconstruct_path(start_node, end_node)

```
    initialize path as empty list
    current = start_node
    while current is not end_node
        add current to path
        current = parent of current
    add end_node to path
    return reverse of path
```