Graph search-based pathfinding in dynamic environments

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
May 2024
Joona Mäntysalo

UNIVERSITY OF TURKU
Department of Computing

Joona Mäntysalo: Graph search-based pathfinding in dynamic environments

Master of Science (Tech) Thesis, 58 p., 13 app. p.
Software Engineering
May 2024

Pathfinding is a common problem in fields such as video games and robotics and has attracted a lot of research. In pathfinding, dynamic environments consist of obstacles that can either change their position or the cost of travel. The issues dynamic environments impose on pathfinding are that previously found paths can become invalid or better paths become available. At worst the agent has to constantly compute and update its path, making slow pathfinding methods ineffective.
A* is a well-known and widely used pathfinding algorithm fit for static environments. This thesis explores how A* performs in dynamic environments compared to three of its dynamic variants D* Lite, RTD*, and AD*. These variants also use differing methods and their comparative performance is also measured.
In this thesis, pathfinding in dynamic environments is examined by building a test environment that simulates dynamically changing obstacles. The performance of pathfinding algorithms is measured in this test environment and the results are analyzed by comparing the performance of the algorithms. The effect of different environmental variables on the algorithm performance is also examined, which includes the effect of the grid type, size, and the portion of changing obstacles.

Keywords: pathfinding, graph search, dynamic environment, dynamic obstacles, environmental variables

Contents

1 Introduction

Pathfinding is a common method used in many different areas such as video games, robotics, logistics, and GPS [1] [2]. Its purpose is to provide efficient paths to autonomous agents between two points. Pathfinding algorithms need to be able to fulfill a range of criteria such as fast computations, short path lengths, and obstacle avoidance. Most algorithms focus on giving the shortest path to the goal, but less optimal paths may be settled for in time-sensitive situations [3].

Dynamic environments add a layer of complexity to pathfinding. In many cases, it forces the agent to recompute paths as the previous ones become blocked or inefficient. In complex settings, frequent path replanning can take significant processing power. If the algorithm used is not efficient enough, the agent may need to stand still while waiting for a new path periodically. Dynamically changing obstacles can also introduce new shortcuts to the environment and not taking advantage of it could lead to substantially longer paths, potentially also making the movements of the agent look unnatural. Realistic human-like movement is often desirable in video games [3]. Many environmental variables also affect the speed of different pathfinding algorithms such as the size of the search environment and how large of a portion of the nodes are blocked [4].

Graph search-based pathfinding refers to the use of algorithms that can be used to find paths between start and end points within a graph. Graphs consist of multiple connected nodes and are modeled after the environment. The nodes in a graph

represent specific sections of the environment and have differing costs assigned to them depending on the difficulty of travel through that node. Nodes with obstacles are assigned the cost of travel as infinity or are not given connections to the neighboring nodes, preventing the agent from traveling through them.

A* is a well-known and commonly used pathfinding algorithm used in video games and robotics [1] [5]. It uses heuristics to guide the search towards the goal, skipping the processing of unnecessary nodes which speeds up the search compared to non-guided search algorithms [6]. Due to its popularity, many different modifications and improvements for A* have been developed to obtain better performance in certain scenarios.

This thesis aims to provide insight into how different pathfinding algorithms perform in dynamic environments. A test environment is built which simulates a dynamic environment with randomly changing obstacles. Each algorithm is run under the same conditions and their relative performance is measured and analyzed. The chosen algorithms were A* [6], D* Lite [7], RTD* [8] and AD* [9]. A* was chosen to be tested due to its popularity. The remaining algorithms were chosen to be incremental variants of A*. Incremental pathfinding algorithms have been developed to generally handle environmental changes more effectively than A*. Out of the chosen algorithms D* Lite is purely an incremental algorithm while RTD* also uses real-time aspects and AD* contains anytime aspects.

The effects of environmental variables are also tested and measured. The tested variables were grid size, grid type, and the magnitude of dynamic environmental changes.

This thesis seeks to answer the following research questions:

- How do the selected pathfinding algorithms compare in dynamic environments?

- How do different environmental variables affect the performance of these algo-

rithms in dynamic environments?

Chapter 2 gives background on general pathfinding and how it is done in dynamic environments. In Chapter 3, the chosen algorithms are looked into in more detail. Chapter 4 discusses the specifics of how the tests were conducted. Chapter 5 contains the results of the experiments which are then analyzed. Chapter 6 summarizes the findings of the thesis.

2 Pathfinding

Pathfinding is a common problem in many areas such as video games and robotics. The problem is simple at its core: find a path from point A to point B. However, pathfinding problems we give to computers usually contain additional requirements that need to be satisfied such as minimizing the path length, fast processing times, and obstacle avoidance. These requirements can be quite demanding and varied and as such many different techniques and methods have emerged to solve varied pathfinding problems.

Pathfinding algorithms can be categorized into three categories: graph search-based algorithms, random sampling algorithms, and intelligent bionic algorithms [10]. Of these, graph search-based algorithms are focused on in this thesis.

2.1 Pathfinding methods

2.1.1 Basic algorithms

Basic graph search algorithms such as depth-first-search (DFS) and breadth-first-search (BFS) are some of the earliest and simplest pathfinding algorithms. DFS always expands the deepest node connected to an already expanded node and stops if it finds the goal. BFS always expands the closest nodes to the starting node that have not yet been expanded. Unlike DFS, BFS finds the shortest path but requires more memory [11].

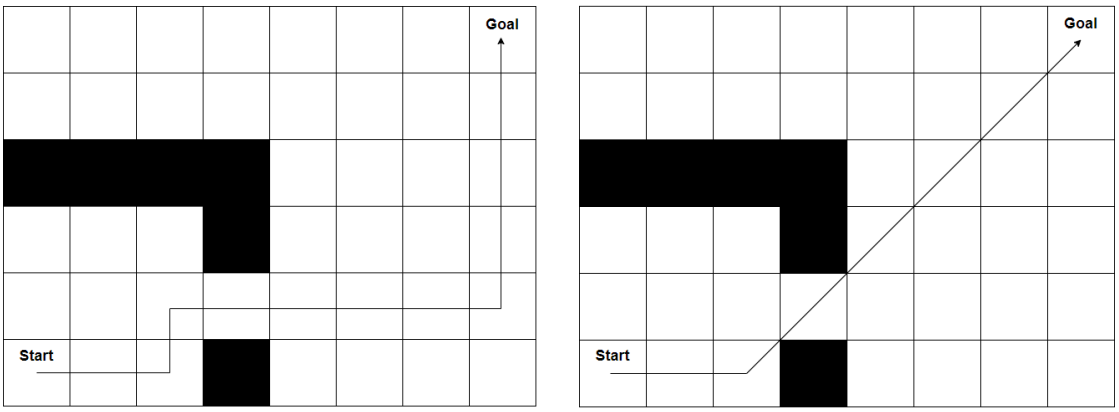
Dijkstra's algorithm by E.W. Dijkstra [12] was developed to solve the shortest path problem in 1959. Unlike BFS, Dijkstra's algorithm takes edge costs of nodes into account meaning it is also able to find the shortest path in a weighted graph. It works by expanding the next unvisited node with the lowest cost and updating the costs of its neighboring nodes if the cost is lower than what was previously assigned from another node. The current cost for a node is the smallest total known cost if traveled through already expanded nodes. This continues until the shortest path is found to the target node or until the shortest path is known to every node in the graph.

A* [6] was developed in 1968 and is based on Dijkstra's algorithm. To increase the speed of the search A* introduced heuristic estimates to guide the search towards the goal node. A* is widely used and is one of the best-known algorithms in video games and robotics [1]. A* is presented more in-depth in Section 3.1.

2.1.2 Graphs of the environment

In order for pathfinding algorithms to work, the interactable environment first must be discretized into a graph. Grids are common graph types to use for pathfinding [13]. Grids consist of polygons that are tiled, creating a graph of connected nodes that represent the environment. Each node on the grid is connected to its neighboring tiles which matches the movement possibilities in the real environment. Grids are often composed of squares, but triangular and hexagonal grids are also used. The nodes in square grids can be either connected to their orthogonal neighbors (4-connected square grid) or also connected to their diagonal neighbors (8-connected square grid) [13] [5]. Figure 2.1 shows an example of the shortest paths found in 4-connected and 8-connected square grids. One disadvantage of grids is that the techniques that use them require considerable memory space [1].

Depending on where the obstacles lay, the nodes on the grid are marked as either



(a) 4-Connected square grid. (b) 8-connected square grid.

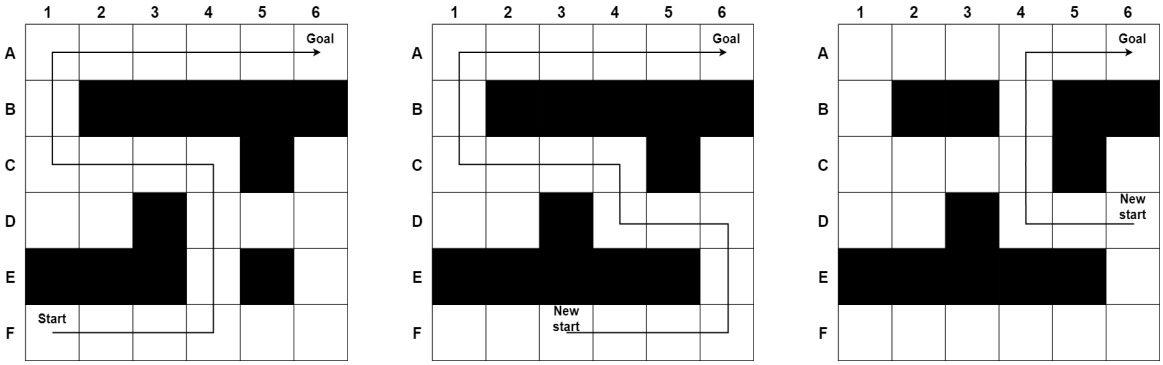
Figure 2.1: An example of shortest paths in (a) 4-connected and (b) 8-connected square grids. When assuming an orthogonal travel cost of 1 and a diagonal travel cost of $\sqrt{2}$, the total travel costs are (a) 12 and (b) 9.07.

open or blocked. The obstacles can also only hinder movement but not completely prevent it. In this case, the node is marked open but given a higher cost of travel. This makes the grid a weighted graph. 8-connected grids are weighted graphs even without any movement hindering obstacles if the diagonal movement cost is made to be more costly than orthogonal movement, typically $\sqrt{2}$ times the cost. The shortest path algorithms will take the cost into account and find the final path based on the total travel cost rather than minimizing the number of cells traveled.

Irregular grids are another possible grid type to use in pathfinding. As the name suggests these types of grids use irregular tiling patterns. Irregular grids include visibility graphs, navigation meshes, and waypoints [1]. A visibility graph is created based on the key points in the environment visible to the agent, such as corners of obstacles [14]. Navigation mesh is a partitioning of the environment into a set of convex polygons [13]. Although the tiling is irregular since each polygon only shares an edge and two corners with adjacent polygons, it guarantees straight-line movement inside the polygons. Waypoints are usually manually crafted graphs that are composed of individual waypoints that represent important points in the environment, such as corners [15].

2.2 Dynamic environments

Pathfinding can be categorized into two groups: static and dynamic [11]. Static pathfinding is performed in static environments, so once the shortest path has been found, there is no need to perform any further computations as it will always be the shortest path for the current problem. Dynamic pathfinding on the other hand is conducted in dynamic environments where obstacles are not fixed on a single location. In this case, once the shortest path is found it may not remain valid as either it becomes obstructed or a new even shorter path opens up. While static pathfinding algorithms can work in dynamic environments, they need to always perform the search from start to finish which is often slow, especially in complex or frequently changing environments. Dynamic pathfinding algorithms avoid this problem by reusing the information gained from previous searches to speed up the current search [16]. Reusing the information allows the algorithm to omit the processing of unnecessary nodes and focus on the ones that were affected by the changes. Figure 2.2 shows an example of how a moving agent finds paths in a dynamic grid.



(a) Initial path (b) Initial path gets obstructed (c) Shorter path becomes available

Figure 2.2: An example of pathfinding of a moving agent in a dynamic grid: (a) Initial path computation and execution by the agent, (b) Adjustment to the path due to the unavailability of cell E4, (c) Recalculation of a shorter path upon the availability of cell B4. Note that in (c), the agent may choose to not recompute a path if the current one is unobstructed, in which case it would continue following the path from (b).

In robotics, information about the environment is usually gathered from sensors attached to the robot [3]. As such, information about the environment outside the range of the sensors is limited. A common way is to assume that there are no obstacles outside the visible range other than the known ones. However, when the agent does come across obstacles it can benefit from using dynamic algorithms. In this manner, static unknown or partially known environments can lend themselves to be more fit for the use of dynamic algorithms than static algorithms.

In video games, the agents usually have access to all the information about the environment [3]. This allows the agents to make more informed decisions in terms of pathfinding, but it also means that the graphs they are dealing with can be quite massive. Updating the path on large graphs can be computationally taxing, which is why algorithms that can deal with dynamic changes are often preferred. Even so, large dynamic graphs can create issues for even optimized algorithms. For example, D*-Lite [7], an algorithm that was developed for unknown and dynamic environments, has a downfall when changes happen near the goal node, far from the agent, which makes repairing the existing path slow [17]. Also, in general, planning a new path when changes are detected far from the agent can lead to unnecessary processing power usage, as it is likely that the environment will change again before the agent can reach the area.

There are two ways of avoiding dynamic obstacles: reacting to the changes or trying to predict the obstacle movements. In the former solution, every obstacle is considered static and when changes happen the graph is updated to represent the new state. This can result in frequent re-planning as no measures are taken to avoid the path colliding with a moving obstacle in the near future. A way to reduce this is to model an area around the known dynamic obstacle as high-cost terrain, making the path likely to avoid the near vicinity of the obstacle [18]. However, this does mean that the paths are no longer guaranteed to be optimal and the path length

can increase substantially while needlessly avoiding areas near some obstacles.

Predicting obstacle movement is a common strategy used in dynamic environments [18] [19] [20]. In this approach, the agent identifies the moving obstacles and is able to predict their trajectory based on current position, direction, and speed. Based on these trajectories, the agent would then be able to plan its own path and dodge these obstacles based on where they will be and not where they are now. If all the dynamic obstacles are known and their movements are accurately predicted the number of times the path would have to be remade would substantially reduce. This however, requires the predictions to be accurate and in many cases, such as with mobile robots that rely on their sensor data, the predicted trajectories have a high degree of uncertainty [18]. In this case, the result can be even worse than not trying to predict the obstacle movements at all, as aside from the frequent need to replan the path, planning around the obstacle trajectories gives the pathfinding problem an extra dimension, taking more processing power.

3 Algorithms

In this chapter, four pathfinding algorithms are presented. These algorithms include A* and three of its variants that were designed to be fit for dynamic environments.

3.1 A*

A* by Hart, Nilsson, and Raphael [6] is a widely used pathfinding algorithm in many different fields such as video games, robotics, and network routing. It makes use of heuristics to make an informed search in a weighted graph and to find the shortest path to a goal.

A* combines the uniform cost search with heuristics. The uniform cost search gives each node s a value $g(s)$ that stores the cost to reach a given cell s. The heuristic function $h(s)$ guides the priority in which nodes are expanded, guiding the search towards the goal. These are combined to a function

$$f(s) = g(s) + h(s), \tag{3.1}$$

which is a representation of the sum of the travel cost to a node s and the estimated travel cost from a node s to the goal node. This makes the $f(s)$ function indicate the estimated travel cost from the start node to to the goal node if traveling through node s.

The g-value is calculated by taking the currently least costly known path from

the start node and adding the costs of each node traveled through. Since g-value is assigned for each node the g-value of node s can be expressed as

$$g(s) = g(s') + c(s, s'), \tag{3.2}$$

where s' is the predecessor node and the function $c(s, s')$ is the travel cost between the two nodes.

Since $h(s)$ is a heuristic estimate of the cost to the goal, it does not have a set definition for different implementations of A*. There are, however, well-known heuristic functions that can be used depending on the situation [21] [5] . For example, the Manhattan distance is suited for grid maps that allow 4-directional movement. If 8-directional movement is allowed then a diagonal distance such as the Chebyshev distance can be a good option. If the movement is not necessarily grid-based and allows for any-angle movement, then the Euclidean distance might be a good fit. Figure 3.1 shows an example of how $h(s)$-values are calculated for nodes on a grid by the mentioned heuristic functions.

procedure HEURISTIC(node)
$dx = abs(node.x - goal.x)$
$dy = abs(node.y - goal.y)$
return $(dx + dy)$

	1	2	3	4
A	3	2	3	4
B	2	1	2	3
C	1	Goal 0	1	2
D	2	1	2	3

(a) Manhattan distance

procedure HEURISTIC(node)
$dx = abs(node.x - goal.x)$
$dy = abs(node.y - goal.y)$
return $max(dx, dy)$

	1	2	3	4
A	2	2	2	2
B	1	1	1	2
C	1	Goal 0	1	2
D	1	1	1	2

(b) Chebyshev distance

procedure HEURISTIC(node)
$dx = abs(node.x - goal.x)$
$dy = abs(node.y - goal.y)$
return $sqrt(dx * dx + dy * dy)$,

	1	2	3	4
A	√5	2	√5	√8
B	√2	1	√2	√5
C	1	Goal 0	1	2
D	√2	1	√2	√5

(c) Euclidean distance

Figure 3.1: Different heuristic functions

For A* to guarantee optimality, two conditions need to be true [22]:

- $h(s)$ is an admissible heuristic, meaning it never overestimates the cost from node s to the goal.
- $h(s)$ is consistent, meaning if the estimated cost for reaching the goal from any node s is never higher than than the cost from any neighboring cell s' plus the cost of moving from node s to node s'. This can be defined as

$$h(s) \leq c(s, s') + h(s'). \tag{3.3}$$

The admissibility ensures that A* does not overestimate the costs, and consistency helps in the efficient exploration of the search space.

```
1: procedure A*(s_start, s_goal)
2:     OPEN.Insert(s_start)
3:     g(s_start) = 0
4:     f(s_start) = h(s_start, s_goal)
5:     while OPEN is not empty do
6:         s = OPEN.Pop()
7:         if s = s_goal then
8:             return BacktrackPath(s)
9:         OPEN.Remove(s)
10:        for all s' ∈ Successor(s) do
11:            if g(s) + cost(s, s') < g(s') then
12:                g(s') = g(s) + cost(s, s')
13:                f(s') = g(s') + h(s', s_goal)
14:                OPEN.Insert(s')
15:                Predecessor(s') = s
16:    return failure (no path found)
```

Figure 3.2: A* Algorithm

A pseudo code for A* is presented in Figure 3.2, where s_{start} and s_{goal} are the start and goal nodes of the search. A* uses a data structure called an open set, which in typical implementations is a priority queue. It keeps track of the nodes available for expansion. Each iteration of A*, the node with the lowest f-value will

be expanded (Figure 3.2, line 6), and each of its neighbors (successors) have their f- and g-values updated and are added to the open set (lines 10-14). Common implementations also include a data structure called a closed set. It keeps track of all the nodes that have already been expanded and allows the algorithm to skip over the nodes that have already been considered.

After the A* search has finished, we still need to extract the path found from it. This can be done by setting up predecessor pointers (often also called parent pointers) for the nodes during the execution (Figure 3.2, line 15). The final path can then be obtained by reversing the path found when following these predecessor pointers from the goal node to the start node.

In dynamic environments the paths found by A* will often become obstructed. Using A*, one approach to maintain a free path is to naively run A* again from the current position. However, although this approach is valid, it runs the risk of requiring a substantial amount of computation time, as the nodes not relevant to the problem will also be searched. The following sections in this chapter will introduce algorithms that were designed to fix this weakness of the A* algorithm.

3.2 D* Lite

D* Lite by Koenig and Likhachev [7] was created to efficiently update paths in an unknown or dynamic environment. It was made to behave the same as D* (Focussed Dynamic A*) by Stentz [23] but to be algorithmically different in a way that is easier to understand and expand upon. D* Lite is built on LPA* (Lifelong Planning A*) by Koenig [24] which was developed as an incremental version of A*. As an incremental search algorithm, D* Lite can make use of the previous searches when computing a new similar search. This can make recomputing new searches fast when changes happen to the search environment as only the nodes affected by the changes have to be reconsidered.

Like A*, D* Lite uses heuristics to guide its search while maintaining optimality as long as the heuristics are admissible. Also like A*, D* Lite uses a priority queue to maintain the order of nodes that it expands. The priority queue contains both nodes that need to be explored or updated. However, unlike A*, D* Lite does not use a closed set to keep track of the nodes that are already explored as it needs to be able to access already explored nodes in subsequent searches if any changes happen in the search environment. To maintain the order of the priority queue D* Lite uses two-part keys to order the nodes. A key can be calculated as follows:

$$k(s) = [min(g(s), rhs(s)) + h(s, s_{start}) + k_m; min(g(s), rhs(s))]. \tag{3.4}$$

The first part of the key is used to order the priority queue and the second part is for breaking ties. In this equation, s_{start} refers to the start node for the search and k_m is a key modifier that allows for reusing the priority queue in later searches. During the first search the value of k_m is 0. The key modifier k_m is explained more thoroughly later in this section.

To allow for incremental searches with a moving agent, D* Lite flips the search direction to start the search at the goal node. Thus, the g-value for a given node is the currently smallest found cost to reach the goal node. This allows the g-values to stay constant even if the agent moves through the graph changing the start node for the next search.

How D* Lite also differs from A* is that together with the g-values it also maintains a second estimate for each node called the rhs-value, which it borrows from LPA*. The rhs-values are look-ahead values based on the g-values of the node's predecessors. It can be calculated as in Equation 3.5:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{otherwise} \end{cases} \tag{3.5}$$

With rhs, D* Lite defines consistency for each node. When $g(s) = rhs(s)$, a node is said to be locally consistent. A node is inconsistent when $g(s) \neq rhs(s)$ which can occur after edge costs get changed. A node is said to be overconsistent when $g(s) > rhs(s)$ and underconsistent when $g(s) < rhs(s)$. When this happens the rhs-values of nodes affected by edge cost changes get new values based on Equation 3.5. If the node is then locally inconsistent, it gets put on the priority queue from where it will be handled only if it proves to be a relevant node for a route in subsequent searches.

D* Lite needs to maintain three invariants:

- Invariant 1: The rhs-value of a node always satisfies the Equation 3.5
- Invariant 2: The priority queue always contains the locally inconsistent nodes where $g(s) \neq rhs(s)$.
- Invariant 3: The nodes on the priority queue are always ordered according to their keys.

Pseudo code for D* Lite is shown in Figures 3.3 and 3.4. The initial search of D* performs exactly like A* performed backward from the goal node to the start node. For the subsequent searches, however, D* Lite uses information gathered from previous searches. Due to invariant 2, every node expanded is locally inconsistent. In ComputeShortestPath (Figure 3.3) method, the $g(s)$-value of a node is set to its rhs-value if $g(s) < rhs(s)$ and otherwise it is set to infinity. To satisfy the invariants 1-3, the nodes possibly affected by this change have their rhs-value recalculated and are removed or added from the priority queue if necessary. This is done by calling the UpdateVertex method (lines 16 and 19). The Main method (Figure 3.4)

```
1: procedure CalculateKey(s)
2:     return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))]
3: procedure UpdateVertex(u)
4:     if u ≠ s_goal then rhs(u) = min_{s∈Succ(u)}(c(u, s) + g(s))
5:     if u ∈ U then U.Remove(u)
6:     if g(u) ≠ rhs(u) then U.Insert(u, CalculateKey(u))
7: procedure ComputeShortestPath()
8:     while (U.TopKey < CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start)) do
9:         k_old = U.TopKey()
10:        u = U.Pop()
11:        if (k_old < CalculateKey(u)) then
12:            U.Insert(u, CalculateKey(u))
13:        else
14:            if (g(u) > rhs(u)) then
15:                g(u) = rhs(u)
16:                for all (s ∈ Pred(u)) do UpdateVertex(s)
17:            else
18:                g(u) = ∞
19:                for all (s ∈ Pred(u) ∪ {u}) do UpdateVertex(s)
```

Figure 3.3: D* Lite: First part

moves the agent forward, following the found path while scanning for changes. If changes are discovered it calls the UpdateVertex (line 20) method to maintain the invariants 1-3.

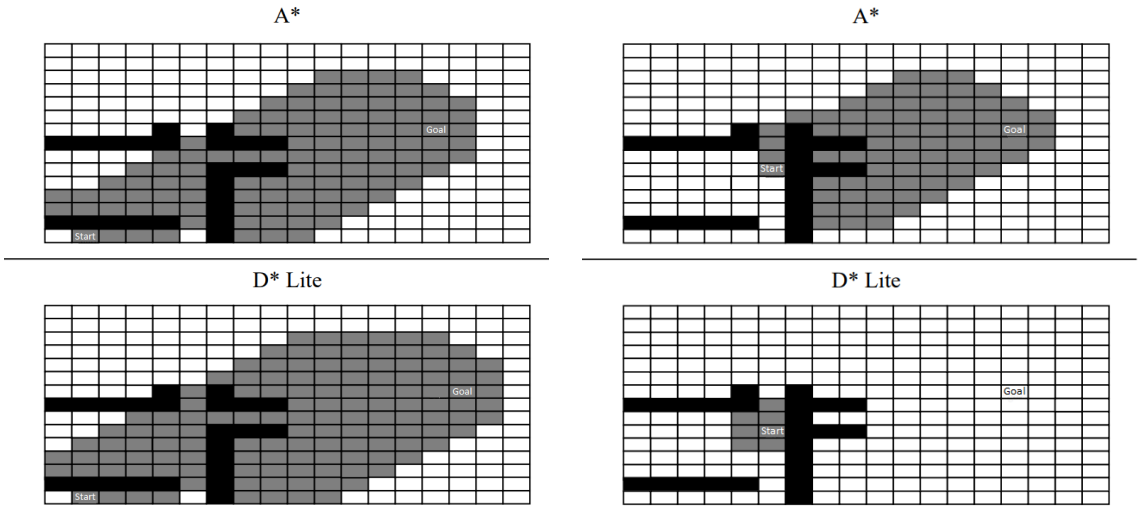
After the initial path is found, the current nodes in the priority queue need to be taken advantage of to speed up the subsequent searches. However, the priorities of these nodes are calculated using a heuristic from the old start node. For the old priority queue to be useful in the new search, keys for each node in it would need to be recalculated. This can prove to be very costly computationally especially since the priority queue will often contain a large number of nodes. After moving through the graph, the most the values of the first part of the keys can be decreased is the heuristic estimate of the distance between the last start node and the current node $h(s_{last}, s_{start})$. Since this value is constant for every node, subtracting it from the key of each node will not change the order of the priority queue, making the

```
1: procedure Initialize()
2:     U = ∅
3:     k_m = 0
4:     for all s ∈ S do rhs(s) = g(s) = ∞
5:     rhs(s_goal) = 0
6:     U.Insert(s_goal, CalculateKey(s_goal))
7: procedure Main()
8:     s_last = s_start
9:     Initialize()
10:    ComputeShortestPath()
11:    while s_start ≠ s_goal do        ▷ if g(s_start) = ∞, then there is no known path
12:        s_start = arg min_{s∈Succ(s_start)}(c(s_start, s) + g(s))
13:        Move to s_start
14:        Scan the graph for changed edge costs
15:        if any edge costs changed then
16:            k_m = k_m + h(s_last, s_start)
17:            s_last = s_start
18:            for all directed edges (u, v) with changed costs do
19:                Update the edge cost c(u, v)
20:                UpdateVertex(u)
21:            ComputeShortestPath()
```

Figure 3.4: D* Lite: Second part

subtraction unnecessary. Instead when a node gets added to the priority queue the first part of the key will get increased by $h(s_{last}, s_{start})$. This is done with the key modifier k_m. At the start of the algorithm, the value for k_m is zero, and whenever a new search is done k_m is increased by $h(s_{last}, s_{start})$. Now all the old values on the priority queue will be comparable with new values added to it, allowing D* Lite satisfy to optimality requirements without needing to recalculate the key for each node in the priority queue.

The incremental nature of D* Lite allows it to potentially expand far fewer nodes during repeated searches than non-incremental algorithms. An example of this is shown in Figure 3.5. The black cells represent obstacles and the gray cells represent expanded nodes. During the first search the number of expanded nodes is around the same with differences coming from how ties are broken for A* [25]. For the



(a) Initial search

(b) Additional search after additional obstacle appeared

Figure 3.5: Node expansions by A* and D* Lite (modified from [25]).

second search an obstacle appears, blocking the currently followed path. D* Lite is able to reuse the information it computed during the first search leading to a lot fewer node expansions, while A* has to compute the whole search from scratch.

D* Lite does have a potentially severe downside in dynamic environments under certain circumstances [17]. It was designed for mobile robots with sensors attached to them scanning the nearby environment. For any environmental changes that occur near the robot, the sensors would detect it and D* Lite is able to efficiently replan a path for it, while any faraway changes would remain unknown until the robot gets closer to detect it. However, if the robot is able to also detect distant changes near the goal, D* Lite has to propagate the changes in g-values all the way to the robot, due to it starting the search at the goal node. This can be more expensive to compute than just starting the search from scratch.

3.3 RTD*

RTD* (Real-Time D*) by Bond, Widger, Ruml, and Sun [8] is a bidirectional search algorithm that combines an incremental global search with a real-time local search. It performs a backward global search from the goal node using an incremental algorithm such as D* Lite [7] or AD* [9]. The global search is periodically suspended by the local search which uses real-time search algorithms such as LRTA* [26] or LSS-LRTA* [27].

A real-time algorithm can search a path within limited time constraints [28]. However, as the algorithm is working with limited resources it means that it cannot guarantee to return an optimal path. Typically the planning and execution phases are weaved together allowing the algorithm to update the path between steps. This allows for fast response times within set time limits in situations where path optimality is not a priority.

A brief description of the two previously mentioned real-time algorithms is given:

- LRTA* (Learning Real-Time A*) performs a search towards the goal node from the current node until a fixed depth is reached [26]. This fixed depth is determined by the computational resources the algorithm is given for each search. To prevent from getting stuck in infinite loops due to not being able to process enough nodes to see out of a local minima, the h-value of the current node gets updated by the minimum h-value of the search frontier [8], which is where it gets the word "learning" on its name.

- LSS-LRTA* (Local Search Space LRTA*) is a variant of LRTA* that uses A* to perform a bounded search towards the goal node [27]. After this search is completed by hitting the lookahead expansion limit, it updates the h-values of all the nodes in the local search space by using Dijkstra's algorithm. The learning is thus faster than with LRTA* which only updates the current node.

LSS-LRTA* chooses the next step by taking the node with the lowest f-value from the open set used by A*, guiding the agent towards the node with the lowest f-cost on the search frontier.

Incremental algorithms make use of the information gathered from earlier similar searches in subsequent searches. This can help reduce search times substantially in dynamic environments compared to running a fresh search to find the goal node when only some parts of the search environment have changed. Further explanations of the incremental algorithms D* Lite and AD* can be found in sections 3.2 and 3.4.

```
1: procedure ComputeShortestPath(GlobalLimit, s_start)
2:     while (U.TopKey() < CalculateKey(s_start) OR rhs(s_start) > g(s_start)) do
3:         if (GlobalLimit = 0) then return EXPANSION_LIMIT_REACHED
4:         GlobalLimit = GlobalLimit − 1
5:         k_old = U.TopKey()
6:         u = U.Pop()
7:         if (k_old < CalculateKey(u)) then
8:             U.Insert(u, CalculateKey(u))
9:         else
10:            if (g(u) > rhs(u)) then
11:                g(u) = rhs(u)
12:                for all (s ∈ Pred(u)) do UpdateVertex(s)
13:            else
14:                g(u) = ∞
15:                for all (s ∈ Pred(u) ∪ {u}) do UpdateVertex(s)
16:    if (rhs(s_start) = ∞) then return NO_PATH_EXISTS
17:    else return COMPLETE_PATH_FOUND
18: procedure ChooseStep(s_start, LocalLimit, status)
19:    if (status = EXPANSION_LIMIT_REACHED) then
20:        return LocalSearch(LocalLimit)
21:    else if (status = COMPLETE_PATH_FOUND) then
22:        return min_{s'∈(s)}(c(s_start, s') + g(s'))
23:    else if (status = NO_PATH_EXISTS) then
24:        return s_start
```

Figure 3.6: RTD* with D* Lite as the global search: First part

Pseudo code for RTD* that uses D* Lite as the global search component is presented in Figures 3.6 and 3.7, where s_{start} and s_{goal} are the start and goal nodes

```
1: procedure MAIN(ExpandLimit, s_start, s_goal, LocalRatio)
2:     LocalLimit = LocalRatio × ExpandLimit
3:     GlobalLimit = ExpandLimit − LocalLimit
4:     s_last = s_start
5:     INITIALIZE()
6:     status = COMPUTESHORTESTPATH(GlobalLimit, s_start)
7:     while (s_start ≠ s_goal) do
8:         s_start = CHOOSESTEP(s_start, LocalLimit, status)
9:         k_m = k_m + h(s_last, s_start)
10:        s_last = s_start
11:        if changed edge costs then
12:            for all u in changed vertices do
13:                for all v in Succ(u) do
14:                    UPDATEEDGECOST(u, v)
15:                    UPDATEVERTEX(u)
16:        status = COMPUTESHORTESTPATH(GlobalLimit, s_start)
17:    return GOAL_LOCATION_REACHED
18: procedure UPDATEVERTEX(u)
19:    if (u ≠ s_goal) then
20:        rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'))
21:    if (u ∈ U) then
22:        U.Remove(u)
23:    if g(u) ≠ rhs(u) then
24:        U.Insert(u, CalculateKey(u))
25: procedure CALCULATEKEY(s)
26:    return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))]
```

Figure 3.7: RTD* with D* Lite as the global search: Second part

of the search [8]. RTD* largely follows the chosen global search algorithm with some modifications to allow for real-time attributes. Firstly, it modifies the global search by introducing *ExpandLimit* and *LocalRatio* variables as parameters. These variables dictate the expansion limits for the local and global searches (Figure 3.7 Lines 2-3). The other modification to the global search is that its execution is interrupted whenever the limit for node expansions indicated by *GlobalLimit* is reached (Figure 3.6 Line 3) and the task of providing the next step for the agent is given to the local search algorithm (Figure 3.6 Line 20). The local search is allowed to search toward the goal node until the limit of node expansions in *LocalLimit* is

reached. Since the global search algorithm is incremental it can return to further process the global search even if the agent has moved from its start location. If the global search finishes naturally, the path found by it will be followed instead (Figure 3.6 Line 22). If the expansion limit is omitted, RTD* would function similarly to D* Lite.

In dynamic environments, RTD* can make use of newly opened shortcuts, unlike other real-time algorithms like LRTA* and LSS-LRTA*, which can reduce the path length compared to these. When measuring against other incremental algorithms like D* Lite, the real-time components of RTD* can allow it to provide the next step for the agent significantly faster when needing to recompute the path, resulting in less idle time. This can however lead to some increases in path length as the next step provided by the local search may not be following the optimal path.

3.4 AD*

AD* by Likhachev, Ferguson, Gordon, Stentz, and Thrun [9] was designed to allow for fast path search times in dynamic environments. To do this it combines the incremental properties of D* Lite [7] and the anytime properties of ARA* [29]. Incremental planning allows AD* to reuse results from earlier searches to provide faster search times for new searches done after any changes have occurred in the environment. By being an anytime algorithm, AD* can provide quick but highly suboptimal solutions which it then can continue to improve if more computation time is provided. The agent can request a solution at any point and the currently best found path is provided.

A*-based anytime algorithms speed up the searches by introducing an inflation factor $\epsilon > 1$ with which the heuristics are multiplied by $(\epsilon * h)$. This speeds up the search but also makes the found paths no longer optimal. The higher the ϵ-value the less optimal the path and faster the search speed. The path cost is still guaranteed

to be within ϵ times the cost of an optimal path [30]. At $\epsilon = 1$ anytime algorithms perform similarly to A* and produce optimal paths.

```
1: procedure KEY(s)
2:     if (g(s) > rhs(s)) then
3:         return [rhs(s) + ε · h(s_start, s); rhs(s)];
4:     else
5:         return [g(s) + h(s_start, s); g(s)];
6: procedure UPDATESTATE(s)
7:     if s was not visited before then g(s) = ∞;
8:     if (s ≠ s_goal) then rhs(s) = min_{s'∈Succ(s)}(c(s, s') + g(s'));
9:     if (s ∈ OPEN) then remove s from OPEN;
10:    if (g(s) ≠ rhs(s)) then
11:        if s ∉ CLOSED then
12:            insert s into OPEN with key(s);
13:        else
14:            insert s into INCONS;
15: procedure COMPUTEORIMPROVEPATH
16:    while (min_{s∈OPEN}(key(s)) < key(s_start) OR rhs(s_start) ≠ g(s_start)) do
17:        Remove state s with the minimum key from OPEN;
18:        if (g(s) > rhs(s)) then
19:            g(s) = rhs(s);
20:            CLOSED = CLOSED ∪ {s};
21:            for all (s' ∈ Pred(s)) do UPDATESTATE(s');
22:        else
23:            g(s) = ∞;
24:            for all (s' ∈ Pred(s) ∪ {s}) do UPDATESTATE(s');
```

Figure 3.8: AD*: First part

ARA*, the algorithm AD* built upon, executes several A*-searches in succession with decreasing ϵ until the execution is stopped by the agent requesting the current best path or until $\epsilon = 1$ after which the path cannot be improved anymore. ARA* speeds up the subsequent searches by reusing the results from previous searches. It does this by using the notion of inconsistency. A node s is inconsistent if $g(s) \neq g(s')+c(s, s')$ where s' is its successor. During the search, each node will be expanded at most once. During the search, a node can become inconsistent due to cost changes in neighboring nodes. After being expanded it will not be reinserted into the open

set, but will instead be inserted to the INCONS list, which contains all the expanded inconsistent nodes. After the search, all the nodes in INCONS will be moved to the open set allowing them to be used in the next search.

```
1: procedure Main
2:     g(s_start) = rhs(s_start) = ∞; g(s_goal) = ∞;
3:     rhs(s_goal) = 0; ε = ε_0;
4:     OPEN = CLOSED = INCONS = ∅;
5:     insert s_goal into OPEN with key(s_goal);
6:     ComputeorImprovePath;
7:     publish current ε-suboptimal solution;
8:     while true do
9:         if changes in edge costs are detected then
10:            for all directed edges (u, v) with changed edge costs do
11:                Update the edge cost c(u, v);
12:                UpdateState(u);
13:        if significant edge cost changes were observed then
14:            increase ε or replan from scratch;
15:        else if ε > 1 then
16:            decrease ε;
17:        Move states from INCONS into OPEN;
18:        Update the priorities for all s ∈ OPEN according to key(s);
19:        CLOSED = ∅;
20:        ComputeorImprovePath;
21:        publish current ε-suboptimal solution;
22:        if ε = 1 then
23:            wait for changes in edge costs;
```

Figure 3.9: AD*: Second part

Pseudo code for AD* is shown in Figures 3.8 and 3.9, where s_{start} and s_{goal} are the start and goal nodes of the search. AD* behaves like ARA* when there are no changes detected in the environment. The Main method continuously calls the ComputeorImprovePath method with decreasing ϵ-values until $\epsilon = 1$. If any changes are detected, it may mean that trying to repair the current solution will become expensive. This is why the ϵ-value can be increased to execute a fast more suboptimal search for the new environment state (Figure 3.9 Lines 13-14). After this, the ϵ-value is decreased again for the next searches. The can also be started

from scratch if it is deemed necessary. Because ARA* does not account for edge cost increases potentially making some nodes underconsistent ($g(s) < rhs(s)$), it means that different key values must be used for them than for overconsistent nodes ($g(s) > rhs(s)$). In the KEY method, the underconsistent nodes have their key values reflecting their old and new costs without being increased by the inflation factor ϵ (Figure 3.8 Line 5).

```
1: procedure MoveAgent
2:     while s_start ≠ s_goal do
3:         wait until a plan is available;
4:         s_start = s∈Succ(s_start) (c(s_start, s) + g(s));
5:         move agent to s_start;
```

Figure 3.10: AD*: Moving agent

For a moving agent, the execution of AD* can be done in parallel with the agent movement. The part for moving the agent can essentially look like the MOVEAGENT method shown in Figure 3.10. Here the s_{start} is shared with the AD* search. The agent can grab the currently best-found solution to move along (or wait until the initial path is found) which will get updated as it gets improved by AD*.

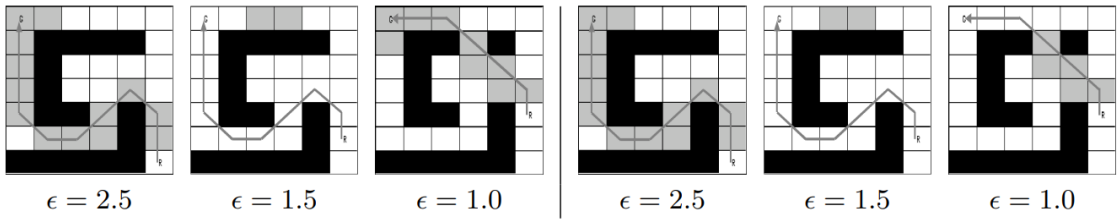


Figure 3.11: Example of ARA* (left) and AD* (right). The expanded cells are marked as grey. The agent starts at the bottom right and moves along the found path. After two steps an obstacle is discovered to have been changed to an empty cell. The first three positions of the agent are shown, each with decreasing ϵ-values. [9]

An example of how ARA* and AD* work is given in Figure 3.11. The first path is computed with $\epsilon = 2.5$. As the agent moves to the next cell, this gives time for the agent to execute another search, this time with $\epsilon = 1.5$. At the third position,

the agent notices that one of the obstacles has changed to an empty cell. ARA* has to replan the path from scratch, while AD* is able to repair its previous solution by needing to expand four fewer cells.

4 Testing pathfinding in a dynamic environment

Dynamic environments affect the agent's approach to navigation. As the environment changes, previously computed paths may become more costly or obstructed and new path possibilities may form. Once the agent is notified of changes in the environment or changes in edge costs of nodes along the current path it is following, it attempts to search for a new path from the current node to the goal. If the changes are not affecting the current path the agent may choose to continue following it disregarding any potential newly formed shortcuts and saving computation time.

4.1 Experimental setup

The algorithms were tested on two different grid types containing dynamic obstacles, which are explained more thoroughly in Section 4.2. The dynamic obstacles are dynamic in a sense that they have two potential states: on and off. In the off-state they are considered as empty cells and can be traversed by the agent. The dynamic obstacles switch between these two states, which is where the dynamism of the grids comes from. During the execution, the agent is aware of every cell on the grid and their states. This enables it to find a path guaranteed to be traversable all the way to the goal cell, provided the obstacles remain unchanged. However, the grid is changing every 10 steps the agent takes, forcing it to compute a new path. Even

if the current path is not affected by the changes, meaning it is not blocked by obstacles, the agent will still compute a new path to find potential shortcuts that would have appeared from the change to the grid.

The agent is only able to move from its current cell to up to four orthogonally adjacent empty cells. The heuristics of the cells are calculated with Manhattan distance. The movement cost into an empty cell is 1, and infinity into an obstacle. The dynamism of obstacles is generated in such a way that there always is a route consisting of empty cells to the goal which means that the agent will never step into an obstacle.

All priority queues were implemented using a binary min-heap. Although the implementations are the same for every algorithm, some differences may occur if using other types of data structures [7], [31]. The framework and the algorithms used in the experiments were implemented using the C programming language. The experiments were run on a machine that uses an AMD Ryzen™ 5 3600 6-core processor. It is worth noting that the results can vary depending on the implementation details and hardware used.

The measured variables for each algorithm were:

1. The average total time it took the agent to get from the start to goal.
2. The average time it took to recompute the path after a change in the environment.
3. The average number of expanded nodes.
4. The average path length the agent traveled.

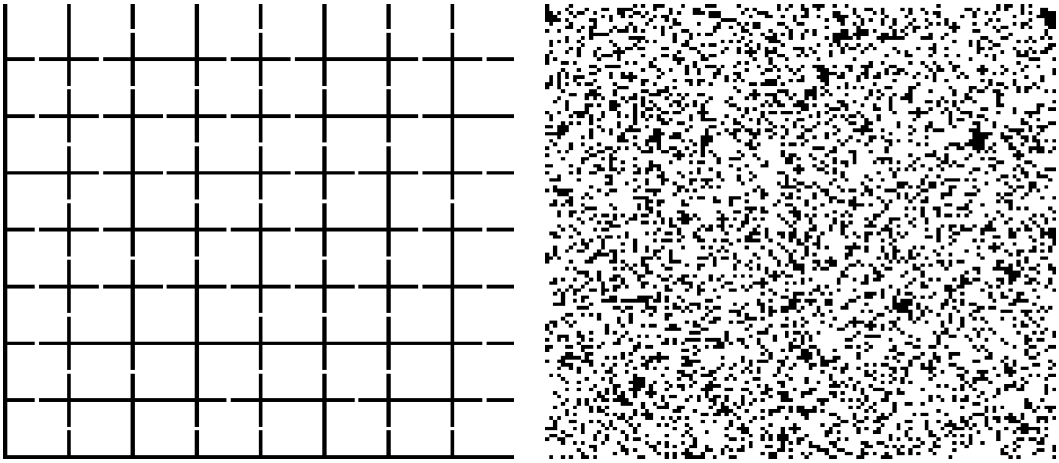
The averages are calculated based on 100 runs for each variant. Measurement 1 only contains the time the agent and the pathfinding algorithm it uses takes. Time taken by external factors such as updating obstacle placement on the grid, is not included. The time measurements were taken using the `Stopwatch` class provided by

the `System.Diagnostics` namespace within the .NET framework. Measurement 2 solely measures the path recompute time. The time taken by the initial search is not included. This was chosen to be measured as it will indicate how long the algorithms need to perform computations before being able to give a next node to travel to. In real-world scenarios, this could mean how long the agent would need to stand still before being able to move to a new node if only using the specific algorithm to guide movement. Measurement 3 is the average number of expanded nodes. Measurement 4 contains the path length the agent traveled. The lengths of found but not fully traveled paths were not measured. This measurement was chosen because it proves valuable not only when comparing optimal and non-optimal algorithms but also in revealing differences among different optimal algorithms in a dynamic environment. Variations may arise when each algorithm sometimes discovers different paths that are equally optimal in length. Given the dynamic nature of the environment, the direction of the initial movement can be impactful for overall travel distance.

4.2 Dynamic grid

Two types of dynamic grids were used to test the algorithms: a grid with rectangular rooms and a grid with random obstacles. The grid types and sizes were based on the suggestions of [32], and are illustrated in Figure 4.1. The grids were slightly modified to better support dynamic obstacles. Both grid types have size variants of 128×128, 256×256 and 512×512 cells. The start cell is fixed at the bottom left of the grid and the goal cell at the top right. Both grid types contain cells that act as dynamic obstacles. These dynamic obstacles randomly change their state into an obstacle or an empty cell every 10 steps of the agent. This causes existing paths to become unavailable while opening new possible paths and forces the agent to recompute its trajectory. The behavior of these dynamic obstacles was first recorded and then replayed for every algorithm so that the environment is the

same for each one. For the dynamic obstacles, the number of cells being obstacles and empty was maintained at around the ratio of 80 : 20. The percentage of dynamic obstacles changing their state each time was tested at 2%, 5%, 10% and 20%. Each variant was run 100 times with different dynamic obstacle behaviors.



(a) Room grid (b) Random grid

Figure 4.1: Grid types of size 128×128 cells

4.2.1 Rooms

In the room grid type, the grid was divided into 16×16 cell rooms. The rooms were separated from each other by 1-cell-thick walls, and each room was connected to each adjacent room with a door. These doors have the possibility to open or close, changing the accessibility of the rooms. The changes to the doors were randomly generated while ensuring that there is a route from each room to the goal, to avoid the possibility of the agent getting stuck in a room or series of rooms that are closed of from the rest of the grid. To achieve this, a randomized depth-first search was done to the grid, which generated a maze-like structure out of the rooms. The currently open doors stayed constantly open, while the rest of the doors were considered as dynamic obstacles that would randomly open or close.

4.2.2 Random obstacles

In the random grid type, obstacles were randomly generated to fill 25% of the cells while ensuring that there is a route from the start cell to the goal cell. Additionally, to guarantee that there always is a route to the goal cell from every empty cell on the grid, every unreachable cell was changed into an obstacle. Each adjacently connected block of obstacles was considered a single dynamic obstacle. Each connected cell in a dynamic obstacle will change states together to simulate different sized obstacles and to ensure that the agent cannot get stuck in the middle of obstacles with no valid path to the goal.

4.3 Tested algorithms

A* was selected as one of the tested algorithms due to its widespread use and established reputation for efficiency in static environments. Three other A*-based algorithms were also chosen due to them being designed to work in a dynamic environment, and also due to their different methods of computing and recomputing paths. These different methods are described in the conference paper [28] as different classes of A* variants. D* Lite is an incremental search algorithm that reuses information from similar previous searches to find the next optimal path. RTD* combines incremental search with real-time searches allowing it to find a new path quicker. AD* is a combination of an incremental and an anytime type algorithm that allows it to find a non-optimal path in a given time frame while reusing information provided by previous searches.

Out of these algorithms, A* is the only one that cannot reuse old search data, requiring it to recompute the path from scratch each time changes occur in the environment. However, this feature eliminates the need to handle old data, which can be beneficial if significant changes render the old data irrelevant.

RTD* combines two different pathfinding algorithms, one being incremental and the other being real-time. Multiple different algorithms of the same type can be used to fulfill the role of its class. In these tests, D* Lite was used as the incremental component and LSS_LRTA* was used as the real-time component based on their performance in the paper RTD* was introduced in [8]. Based on this paper, *local-Ratio* of 50% was used throughout the tests, which indicates how large a portion of the per-step computation limit is given to the local search. RTD* was tested with five different step-limits: 32, 64, 128, 256 and 512. RTD* performs a global and a local search with each step of the agent. This will be included in the total time measurements, but will not be in the measured recompute time, which only count the time it takes to compute the next node to travel to after a change in the grid.

AD* combines the incremental features of D* Lite and the anytime features of ARA*. The anytime features include the ability to compute more optimal paths over time if time allows so. Due to trying to keep the tests fair across all the algorithms no extra time was given for AD* to try and optimize already planned paths. Instead, it was tested with three different static ϵ-values, which influences the trade-off between solution quality and computation time. It was also tested with an ability to optimize paths, as in reducing the ϵ-value periodically to produce more optimized paths. The time taken in these computations is counted towards the total time but not the path recompute time, which only takes into account the time it took to find the initial path after recomputing due to a change in the grid. The ϵ-values measured were 2, 3, and 5. With the changing ϵ-value, it originally started at 2 and was decreased by 0.1 after each step. If any changes happened to the grid, the ϵ-value increased by 0.5. A path with a new ϵ-value was calculated after every step.

Pseudo code for the basic version of D* Lite was presented earlier in Section 3.2. However for these tests, an optimized version of D* Lite found in [7] was used for

both D* Lite itself and the incremental part of RTD*.

5 Results and analysis

This chapter contains the results of the testing and analysis of the performance of the tested algorithms. There were a total of 24 different scenarios that were tested. Each scenario was a unique mix of different grid types, grid sizes, and the magnitude of changing obstacles in each grid change event. Grid types were in two types: Room grids and random grids. Both grid types had three sizes: 128×128, 256×256, and 512×512 nodes. Different obstacle change magnitudes were 2%, 5%, 10%, and 20% which signifies the portion of the dynamic obstacles that were changed after each grid change event. The previous chapter explains the test scenarios in more detail.

The section 5.1 mostly consists of figures showing the results on different grid maps. Section 5.2 contains analysis of the tested algorithms based on these results. Full data of the results can be found in Appendix A.

For each test case scenario, three sets of data are presented as an average out of 100 runs:

1. Total time taken: The total time the agent took from the start node to the goal node.

2. Average recompute time: The algorithm's average time to find the next node for the agent to move into after a grid change event.

3. Traveled path length: The path length the agent traveled using the paths the algorithm provided.

A fourth data set, expanded nodes, was also measured. However, it is not presented here due to the low found significance for these tests and to save on some space. The data for this measurement is still found in the full results presented in Appendix A and will be referred to if found relevant in the analysis.

The results for each test case are given for all the algorithms and their variants on each combination of grid type and size, using the four different grid change magnitudes.

5.1 Graphs of the results

5.1.1 Room 128

Figures 5.1, 5.2 and 5.3 show the results for a 128×128 sized room grid with each grid change magnitude. The different RTD* variations refer to the different step-limits used and the AD* variations refer to the different ϵ-values used.

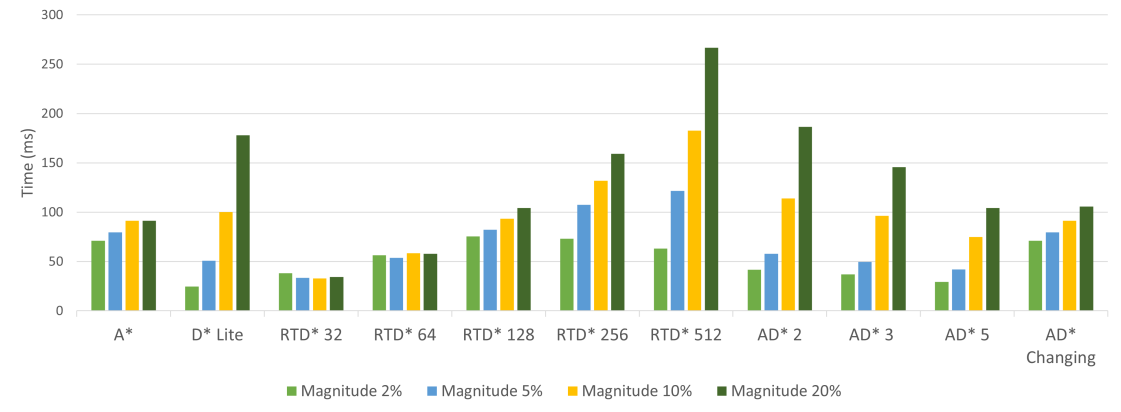


Figure 5.1: Room 128. Total time taken.

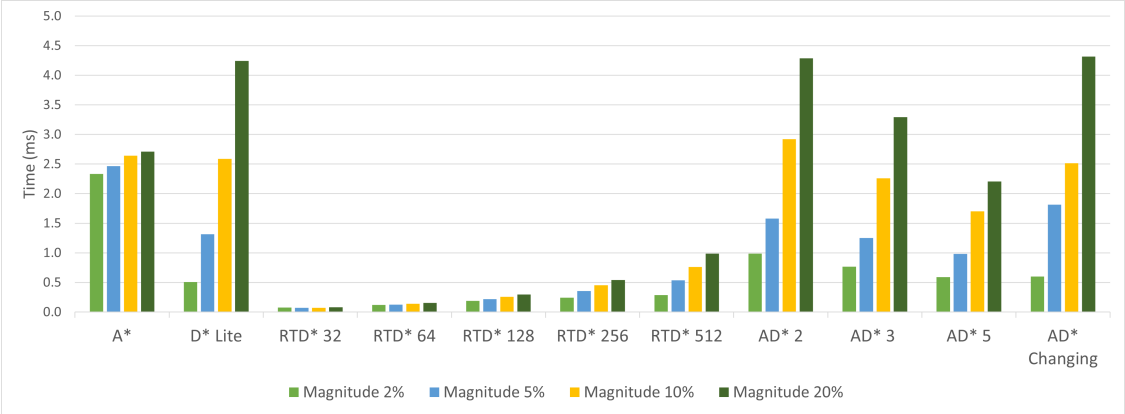


Figure 5.2: Room 128. Average path recompute time.

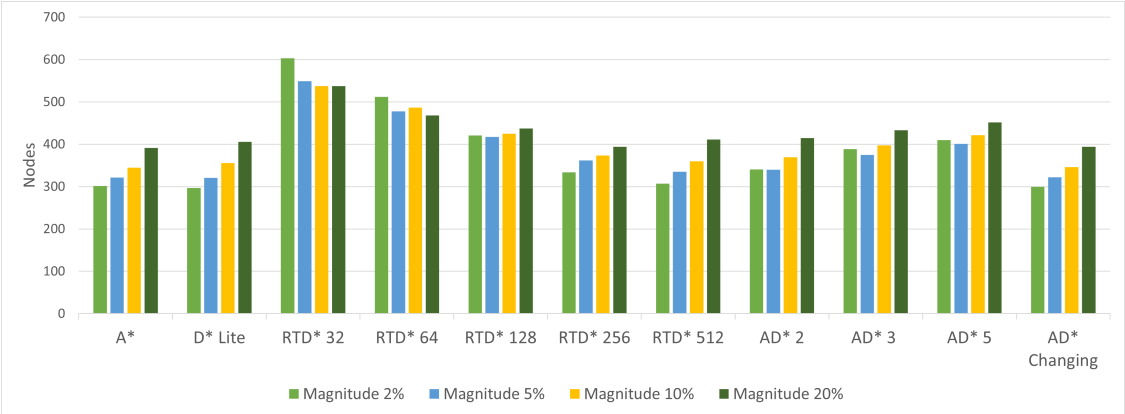


Figure 5.3: Room 128. Traveled path length.

5.1.2 Room 256

Figures 5.4, 5.5 and 5.6 show the results for a 256×256 sized room grid with each grid change magnitude.

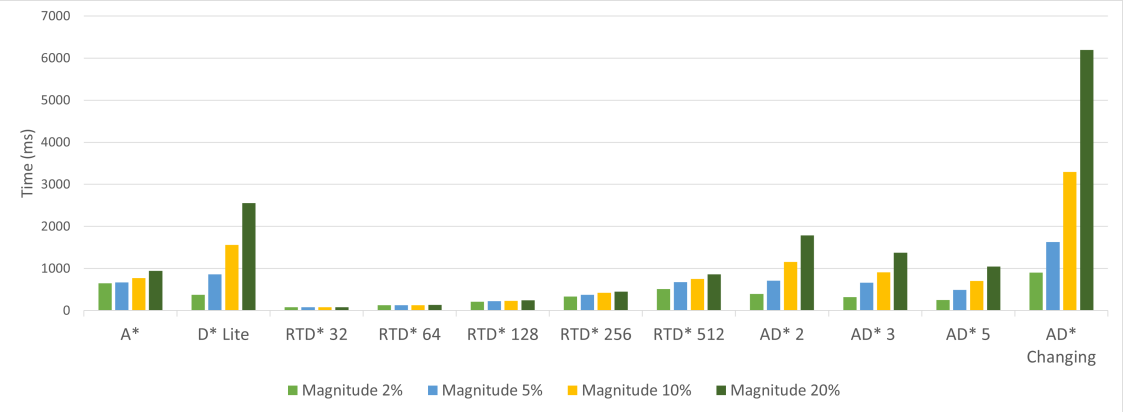


Figure 5.4: Room 256. Total time taken.

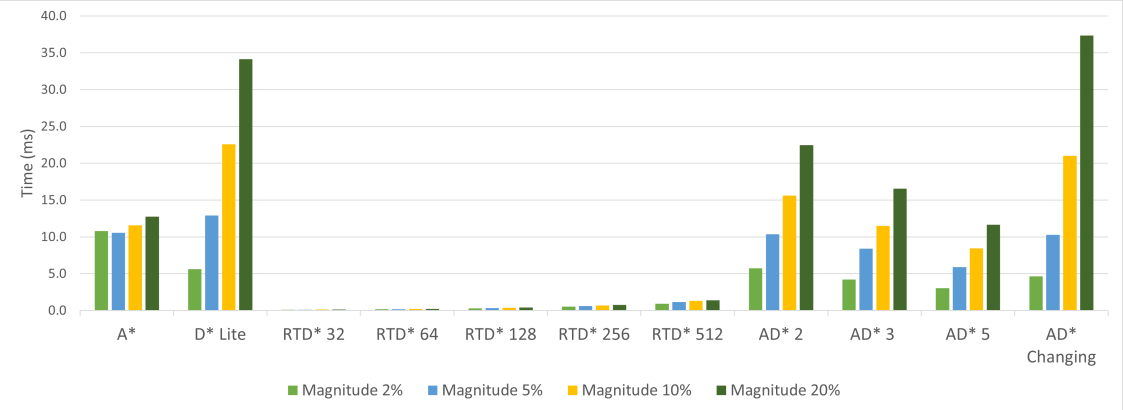


Figure 5.5: Room 256. Average path recompute time.

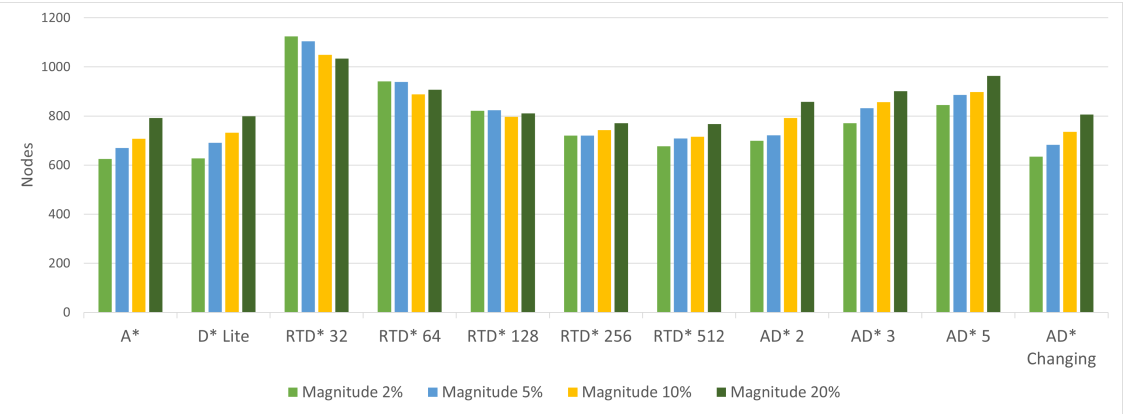


Figure 5.6: Room 256. Traveled path length.

5.1.3 Room 512

Figures 5.7, 5.8 and 5.9 show the results for a 512 × 512 sized room grid with each grid change magnitude.

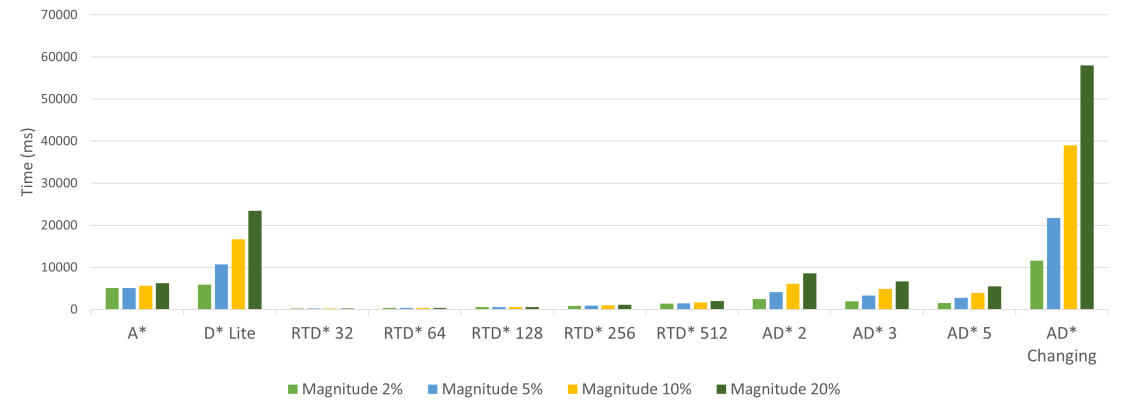


Figure 5.7: Room 512. Total time taken.

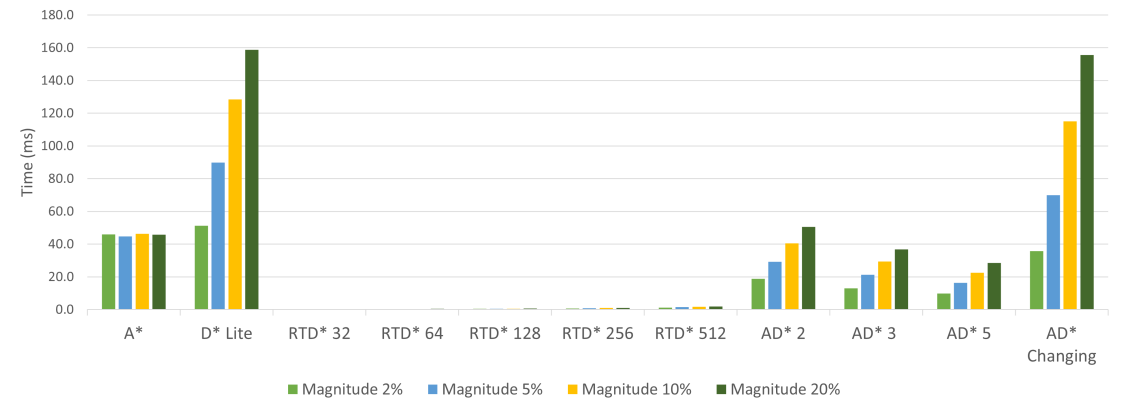


Figure 5.8: Room 512. Average path recompute time.

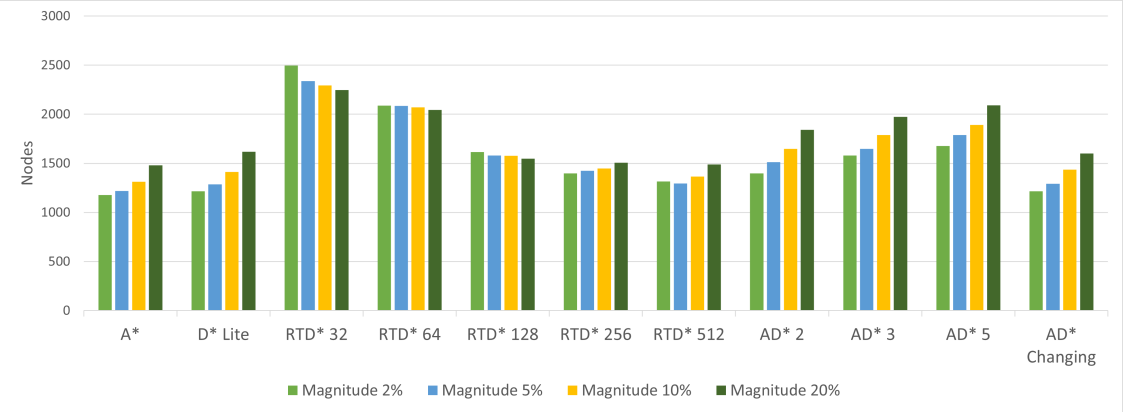


Figure 5.9: Room 512. Traveled path length.

5.1.4 Random 128

Figures 5.10, 5.11 and 5.12 show the results for a 128×128 sized random grid with each grid change magnitude.

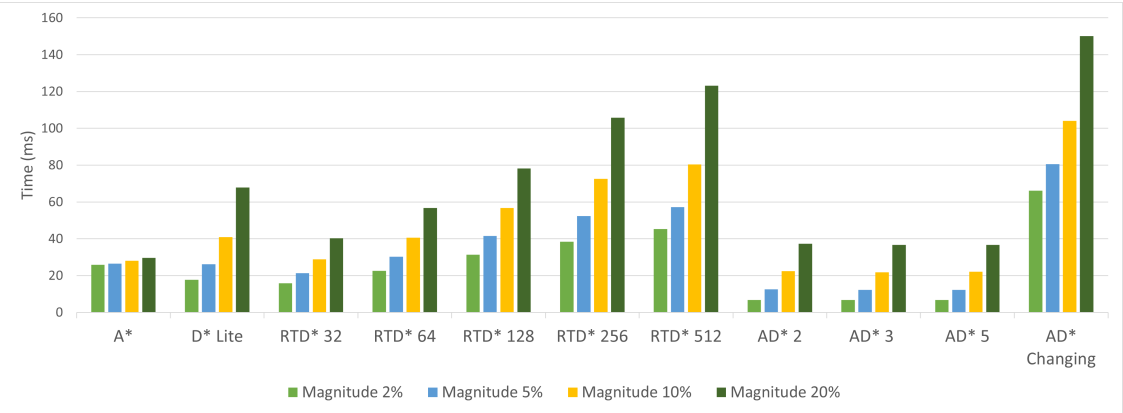


Figure 5.10: Random 128. Total time taken.

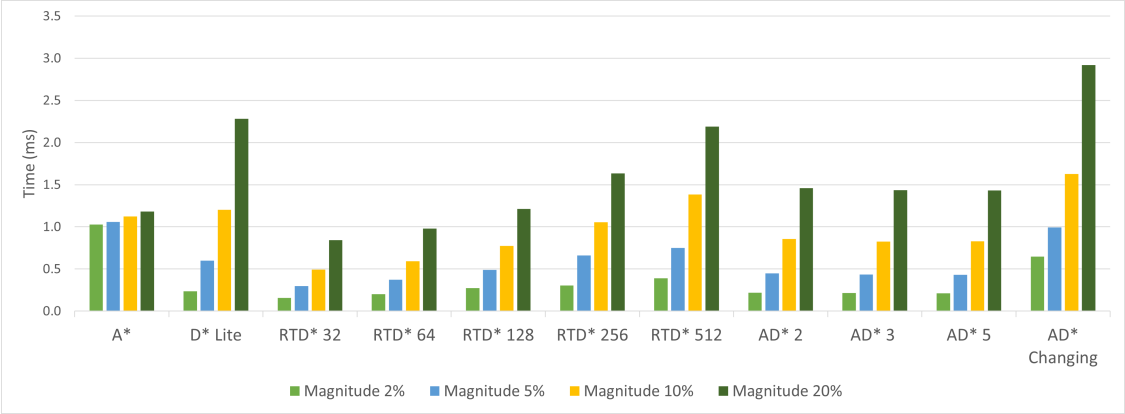


Figure 5.11: Random 128. Average path recompute time.

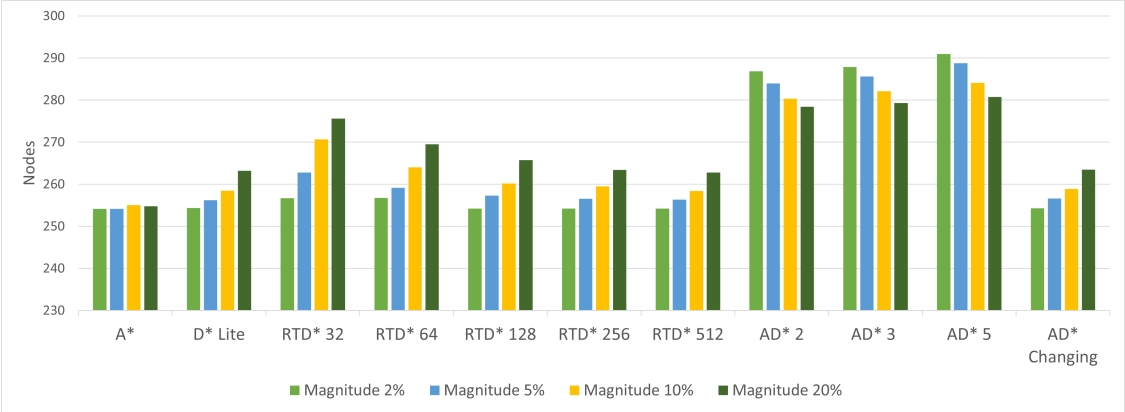


Figure 5.12: Random 128. Traveled path length.

5.1.5 Random 256

Figures 5.13, 5.14 and 5.15 show the results for a 256×256 sized random grid with each grid change magnitude.

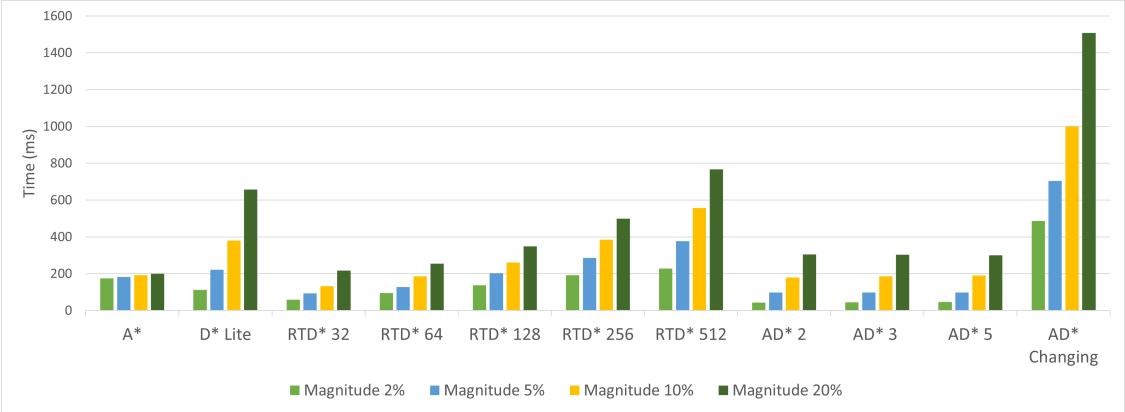


Figure 5.13: Random 256. Total time taken.

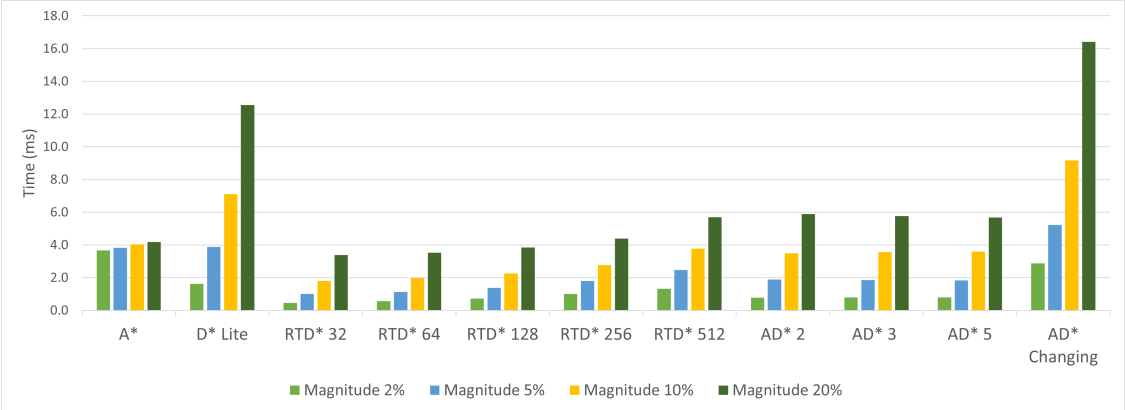


Figure 5.14: Random 256. Average path recompute time.

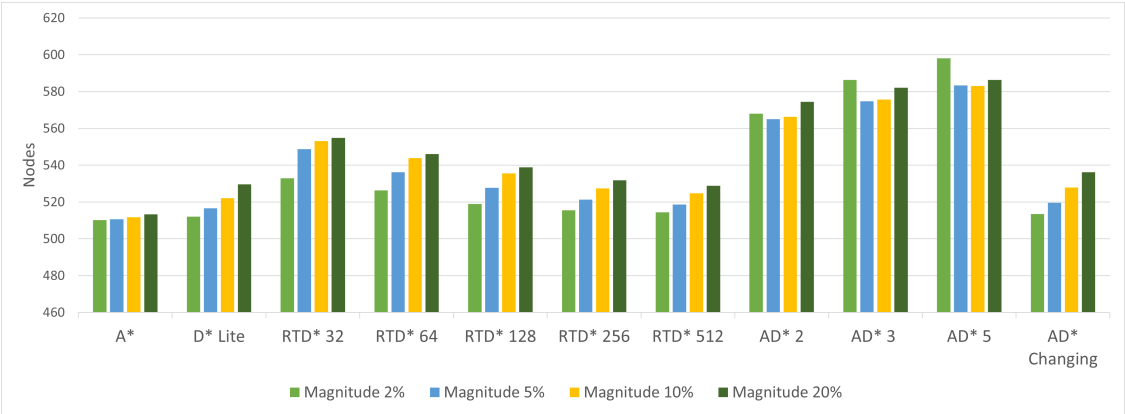


Figure 5.15: Random 256. Traveled path length.

5.1.6 Random 512

Figures 5.16, 5.17 and 5.18 show the results for a 512×512 sized random grid with each grid change magnitude.

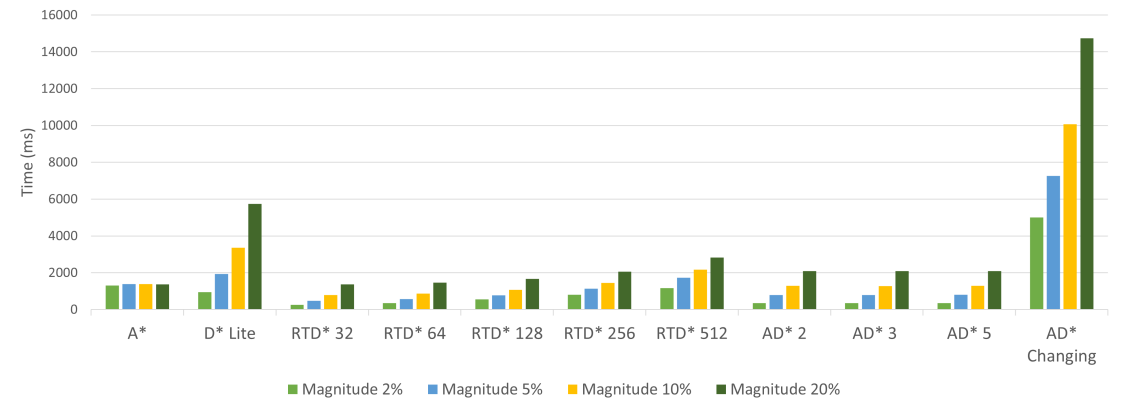


Figure 5.16: Random 512. Total time taken.

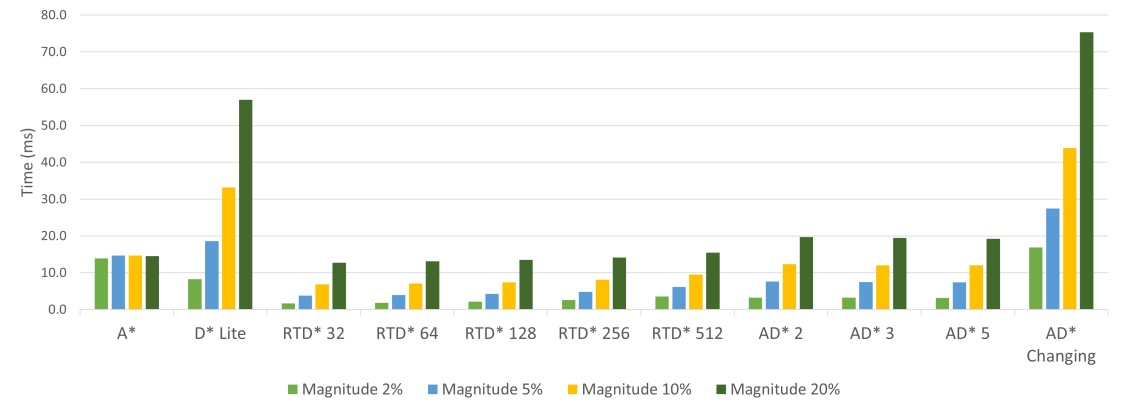


Figure 5.17: Random 512. Average path recompute time.

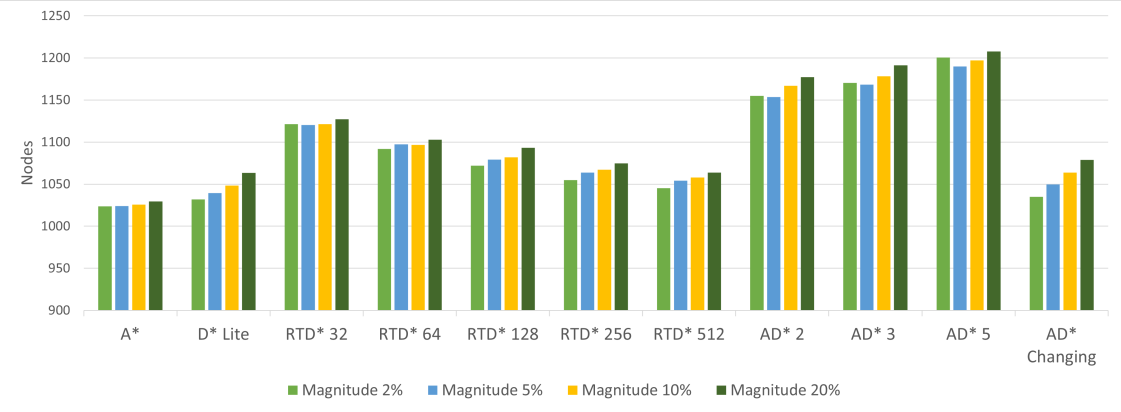


Figure 5.18: Random 512. Traveled path length.

5.2 Algorithm performance analysis

5.2.1 A*

In these tests, A* was mainly used as a benchmark to measure the other tested algorithms against, as A* does not have any meaningful differences in internal functionality on dynamic versus static environments. Still, in this section, some notes and analysis are presented based on the results it achieved and how it fared against the other algorithms that have elements to help them navigate dynamic environments.

The main effect increasing the obstacle change magnitude had on the performance of A* was the increase in path lengths, especially on the room grid. This is because more changes in the grid mean that it is more likely for the path currently traveled to become untraversable. In room maps, this also matters more, since while in random grids this often means just navigating around a new obstacle, in room grids, this can cause serious detours due to the labyrinthine structure those grids often adopt.

There is also a general trend observed in the total time and average recompute time results for A* where the time taken increases with obstacle change magnitude

on a given grid. This is most likely due to the increased path length requiring to do more searches since a new path is computed every time there is a change in the grid, which in these tests happened every 10 steps.

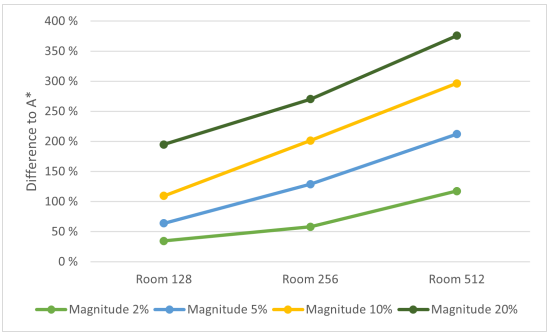
A* consistently had the shortest average path length out of the all algorithms in the tests, and in the cases it was beaten by another algorithm, it was only by a couple of nodes, which can be attributed to the random nature of the dynamic grids used in these tests.

In terms of the total time taken and average path recomputation time results, A* was often the slowest or among the slowest algorithms when obstacle change magnitude was 2%, but it also often achieved some of the fastest times on obstacle change magnitude of 20%. This is because, on low change magnitudes, the other algorithms can better make use of their incremental features, while on high change magnitudes, it starts to become detrimental when not much of the old search data is compatible with the newly changed grid.

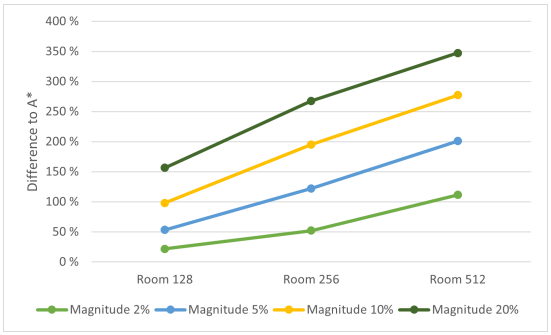
5.2.2 D* Lite

When looking at the results for the time taken and the average recompute time, it can be seen that the performance speed of D* Lite vastly decreases with the magnitude of the changes in the environment. This is logically sound as larger changes mean that D* Lite is less likely to be able to reuse its past calculations effectively. On the lower magnitudes, it performs remarkably well and consistently beats out A* when it comes to speed. The sole exception to this pattern is observed in the room grid type size 512×512 results, seen in Figures 5.7 and 5.8, where A* performs faster, hinting at the potential influence of both map size and type on the comparative performance between D* Lite and A*. This dependency becomes apparent when examining the relative difference of D* Lite compared to A*, as illustrated in Figure 5.19. It can be seen that D* Lite has diminishing performance

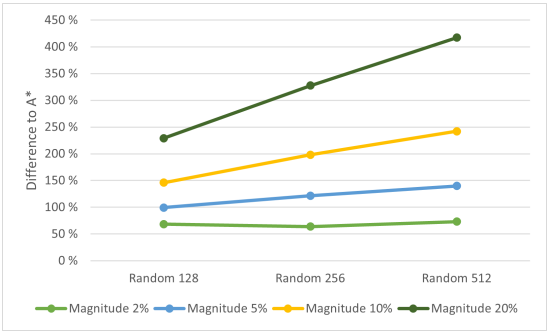
gains on larger maps. In addition, this effect seems to be more pronounced in the results for the room grid type.



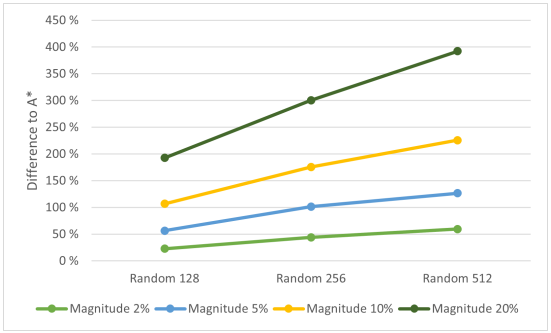
(a) D* Lite relative total time compared to A* on a room grid



(b) D* Lite relative recompute time compared to A* on a room grid



(c) D* Lite relative total time compared to A* on a random grid

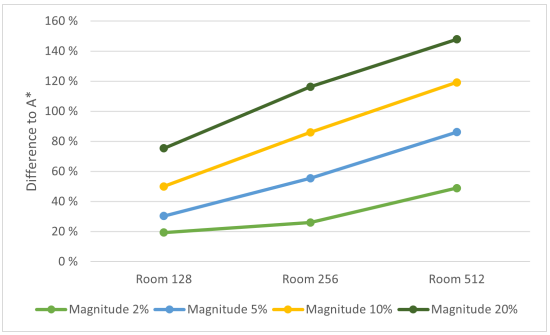


(d) D* Lite relative recompute time compared to A* on a random grid

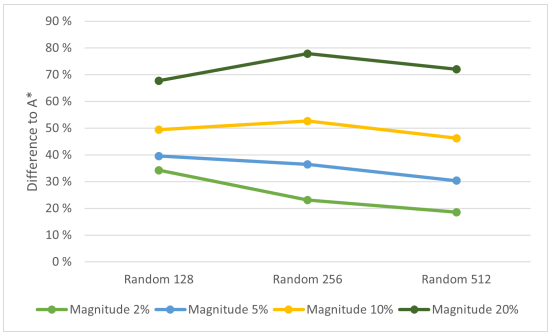
Figure 5.19: D* Lite relative difference to A* in measured time.

On the other hand, Figure 5.20 shows the relative difference of D* Lite to A* when comparing the number of node expansions. On the random-type grids, the relative difference in node expansions generally decreases when maps get larger. The study [4] shows comparative results on the effect of the grid size on random grids. However, the effect seems to be almost the opposite in room grids. The rate of the increase in the number of node expansions is higher on D* Lite than on A* when the map size increases. D* Lite did the best at the lowest tested obstacle change magnitude of 2%. In this D* Lite only performs 30% of the node expansions A* did at the random grid of size 128×128 which drops to just under 20% at the size 512×512. On the room grid, D* Lite has around 20% of A*'s node expansions at

size 128 × 128, which grows to almost 50% at size 512 × 512. This would seem to indicate that, aside from the magnitude of changes, the performance gains D* Lite achieves are dependent on the combination of the map size and its structure, when compared to the performance of A*.



(a) D* Lite relative difference in node expansions compared to A* on a room grid



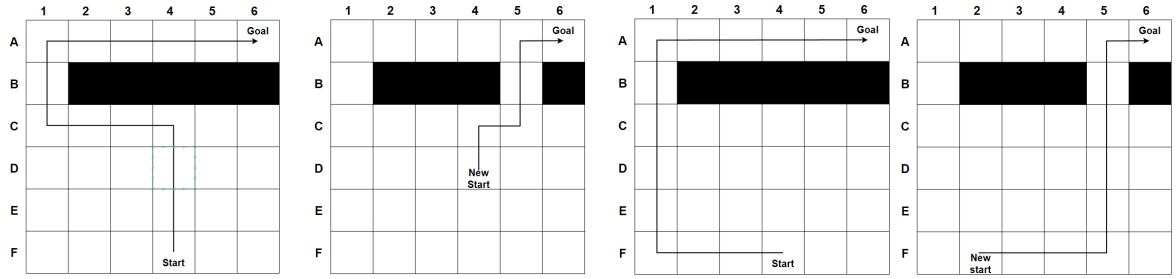
(b) D* Lite relative difference in node expansions compared to A* on a random grid

Figure 5.20: D* Lite relative difference to A* in node expansions.

Regarding the length of the traveled path, D* Lite was expected to perform the best, equally with A* as both always provide the optimal path. However, the results show that on average D* Lite has a slightly higher path length compared to A* in most of the test cases. It only beats or equals A* in 3 out of 24 test cases, those being the room grid type of size 128 × 128 with obstacle change magnitudes of 2% and 5%, seen in Figures 5.1 and 5.2, and the random grid type of size 128 × 128 with a change magnitude of 2%, seen in Figures 5.10 and 5.11. The reason for different path lengths is due to the nature of dynamic environments and the fact that while both of the algorithms do provide an optimal path they might not provide the same path but different paths of the same length. Whenever a dynamic change happens in the environment it can favor one path more than the other.

In the tests, the goal node was always at the top-right corner, which means that any unnecessary movement to the left or down would add to the distance compared to the optimal path which would only consist of steps to the right or up. It was found that when obstacle placements necessitate movement to left or down D* Lite

will more likely take such movement early on in its path whereas A* will take such movement only upon encountering the obstacle. If the obstacle then opens, allowing an agent to travel through, before an agent using A* has executed the steps to avoid the obstacle then it will end up with a shorter path to the goal than an agent using D* Lite since the D* Lite agent has already executed the movement to avoid the obstacle resulting in unnecessary steps. Figure 5.21 shows an example of this. Some tie-breaking criteria [25] can help alleviate this, such as choosing the node with the lowest heuristic cost to travel to in the next step when multiple nodes have the same travel cost. No tie-breaking criteria was used for D* Lite in these tests, however.



(a) A* initial path. (b) A* new path. (c) D* Lite initial path. (d) D* Lite new path.

Figure 5.21: Difference between paths found by A* and D* Lite. A* and D* Lite both compute a path of the optimal length in (a) and (c) respectively. However, due to the direction of the initial steps, once an obstacle opens after two steps in (b) and (d), D* Lite ends up with a longer distance to the goal node.

Based on the overall results of these tests, D* Lite seems to perform excellently when the environment only sees small changes to it and is properly able to reuse its past computations. In an environment with small changes, as is the case in these tests with the change magnitude of 2% and to an extent 5%, it can perform swift searches while keeping the travel distance short. However, in environments with larger changes its search speed increases substantially, making it potentially multiple times slower than A*.

5.2.3 RTD*

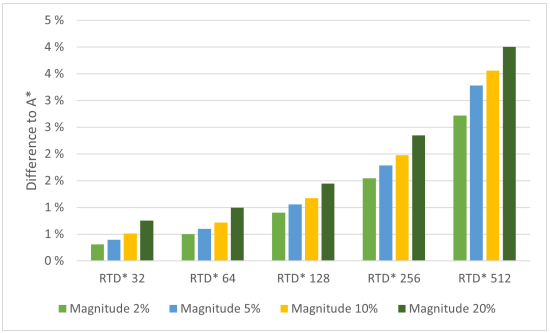
RTD* has two parameters that can be set that affect its performance: the per-step computation limit and the local ratio. In these tests, the local ratio was kept static at 50%, and the computation limit was tested with five different values: 32, 64, 128, 256, and 512. These values direct how many node expansions are done at any computation instance, and the local ratio specifies what ratio of the node expansions are done using the local search algorithm if the global search has not yet found a valid path. At lower computation limit values, the local search will mostly attempt to travel in the general direction of the goal node, while at larger values it can better travel around bigger obstacles or obstacle groups.

In the tests, RTD* did well when looking at the results for the average time taken for path recomputation. In a real-life scenario, this would mean that the agent would not have to sit still waiting for a new path to follow for long. RTD* has the fastest recompute times out of the tested algorithms on the room type grid and on the random type grid it still generally has the fastest recompute time, although RTD* with higher expansion limits is in some cases beaten by AD*. On the room grid, RTD* had orders of magnitude faster recompute times than the other algorithms, while on the random grid, the results were a decent bit closer. This is potentially due to the complex maze-like structure that this grid type can have. Aside from having to update the vertices after changes happen in the environment, RTD* has a set computation amount for each step, that it can perform each time. This means that it has a fairly predictable computation time for each step. In more computationally challenging environments it can continue working on finding the path to goal while trying to move toward the general direction of the goal using its local search method.

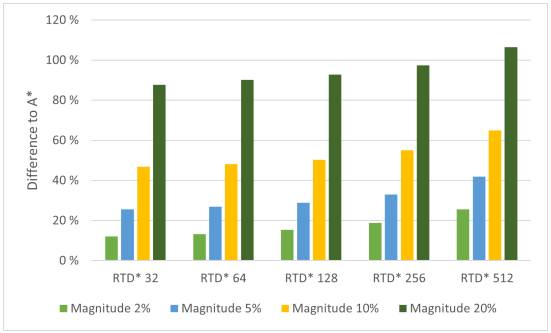
The room grid also has only a few dynamic nodes that can change their state into an empty or an obstacle node, while on the random grid, every obstacle can have state changes. Since the biggest variable for the recomputation speed for RTD* is the

amount of changed edge costs between nodes, it should mean that recomputation on random maps is slower. And when looking at the results, that appears to be the case. The average recomputation times are much lower on the room result in comparison to the other algorithms, especially on the maps of size 256×256 and 512×512. There does appear to be some variance in the difference magnitude between the per-step computation limits used. Lower limit values have the largest differences between the two grid types, while the larger limits can get a bit closer in performance, although still generally being multiple times slower on a random grid.

Figure 5.22 shows the recompute time differences to A* with the different expansion limit values tested with RTD* on room and random grids of size 512×512. From this, we can not only see that RTD* has in general much faster recompute times on the room grid when measuring against A* than on the random grid but also that the expansion limit dictates much more how much time each recompute needs on room grids. On the room grid, the driving factor towards a recompute time seems to be the expansion limit used, while on the random grid, it seems to be the change magnitude, as in the number of nodes that have changed their state.



(a) Recompute differences against A* on the room 512 grid.



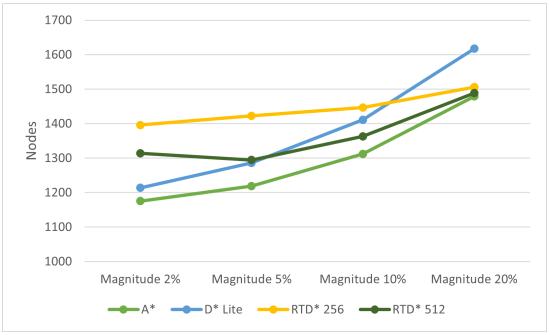
(b) Recompute differences against A* on the random 512 grid.

Figure 5.22: Differences in recompute time with different expansion limit values on RTD* when measuring against A*

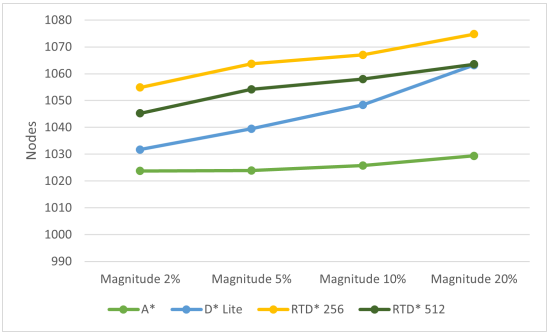
The total time measurements for RTD* include the global and local search that is done after each step if a path has not yet been found. The recompute measurements

only include the time it takes to compute the next step after a change has happened in the environment. This is why the total time results for RTD* appear higher than what the recompute results would lead to think. However, the total time can be slightly misleading of the actual performance. Since in a real-world situation, the agent using RTD* most likely will not be able to instantly move to the next node like in these tests, RTD* would ideally be given such local ratio and expansion limit parameters that it would be able to perform the computations for each step within the amount of time it takes the agent to complete its current step.

The variance in the path lengths the expansion limit values give on the room maps is also a lot higher than on random maps. This is also likely due to rooms generating more dead-ends and misleading paths where going towards the goal is harmful. Since at lower expansion limit values, the local search cannot see too far ahead and often gets caught in these. Also, unlike A* and D* Lite, on the lower expansion limit values, RTD* has the longest paths at the lowest obstacle change magnitude in room grids. Figure 5.3 shows this happening on 128×128 sized grid with expansion limit values of 32 and 64, as well as on 256×256 sized grid in Figure 5.6. With the grid size of 512×512, seen in Figure 5.9, the expansion limit value of 128 also seems to exhibit the same behavior where it has the longest path with the lowest obstacle change magnitude and shortest path with the highest obstacle change magnitude. On random maps, however, it does have the lowest path length at the lowest change magnitudes. The reason for this seemingly abnormal behavior in room grid path lengths could be due to the previously mentioned effect of the dynamic room maps, where going directly toward the goal is often detrimental. With higher obstacle change magnitudes however, it might not be as disadvantageous, since the environment is less predictable and new paths are more likely to open, making moves that would previously be considered negative to be more likely to end up leading toward the goal.



(a) Path length comparisons on the room 512 grid.



(b) Path length comparisons on the random 512 grid.

Figure 5.23: Comparing RTD* path lengths with A* and D* Lite.

Fast and predictable computation time is one of the strengths of RTD*. This is however traded with longer path lengths. How large this trade-off is depends on the parameters used. In these tests, it became clear that the differences between low and high expansion limit parameter variables can be quite large, especially on more complex maps like the room grid. On random maps, RTD* generally outperformed A* in average recompute time even on high obstacle change magnitudes where incremental algorithms, such as D* Lite in these tests, perform poorly. Path lengths it achieved were also fairly competitive with the optimal path algorithms, A* and D* Lite, on random grids where the punishment for going in the "wrong" direction is not usually harsh. Path lengths of RTD* with expansion limits of 256 and 512 are compared with A* and D* Lite in Figure 5.23 on 512×512 sized grids. While RTD* performs worse than A* the difference is not too big. RTD* does also beat D* Lite in path length when using high expansion limit values on room grids with high obstacle change magnitude. Also in the case of the room grid of size 256×256, found in Figure 5.6, with obstacle change magnitude of 20%, RTD* with expansion limits of 256 and 512 beats both A* and D* Lite in traveled path length by around 3%, which is a fairly significant difference considering both A* and D* Lite are optimal algorithms. This is however the only test scenario where RTD* achieved shorter path lengths than A*, and it would require further testing to see

whether these results can be replicated in similar circumstances or if this was just a freak occurrence, where the biases of the test environment favored it.

5.2.4 AD*

AD* is an anytime algorithm, meaning that it was designed to provide a solution at any point during its execution, improving the solution the more time it has. This is controlled by an inflation factor ϵ, which directs the optimality of the algorithm. At ϵ-value of 1, AD* performs optimally. Increasing the value also increases the suboptimality of the solution, giving faster computation times but also longer path lengths. AD* was tested with static ϵ-values of 2, 3, and 5 and with a dynamic ϵ-value where it decreases by 0.1 each step and increases by 0.5 whenever any changes in the grid happen. With the dynamic ϵ-value, an improved path was computed at every step if the current and previous ϵ-values were not 1. The time taken for computing the improved paths was included in the total time, but excluded for recompute time results. Due to the limitations of the testing environment and due to wanting to keep the tests as equal as possible across all the algorithms, AD* was not able to be tested at its possibly most ideal scenario, where instead of the agent waiting for the algorithm to give a next node to travel to, letting the algorithm to compute a path for some time before asking for the next node which would be the case for a moving agent wanting to know the next node to move to right after its finished moving to the current node.

AD* with static ϵ-values performed well in terms of recompute times. On lower change magnitudes it consistently achieved faster times than A*. On random grids it had faster recompute times than D* Lite in all cases and on room grids it was faster than D* Lite in most cases. The only time D* Lite was faster on room grids was at the smallest tested grid size of 128×128, seen in Figure 5.2. In this D* Lite was faster than all the tested AD* variants with static ϵ-values on change magnitude

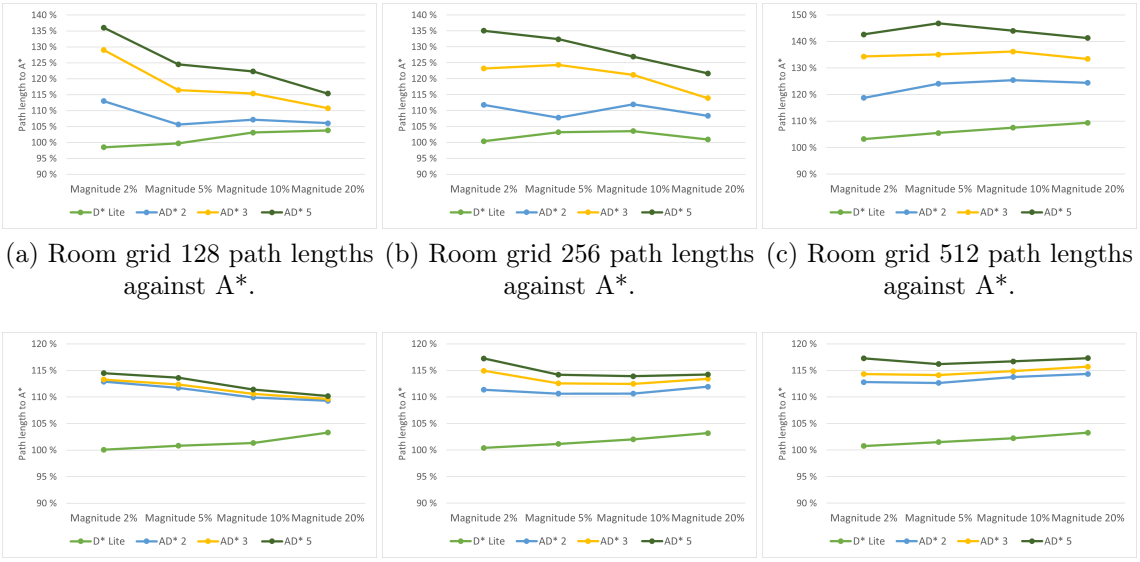
of 2%. However, at the higher change magnitudes AD* with ϵ-values of 3 and 5 are achieving faster recompute times than D* Lite. At grid sizes of 256×256 and 512×512, seen in Figures 5.5 and 5.8, AD* with static ϵ-values was generally always faster than D* Lite.

The AD* version with a changing ϵ-value had similar results for average recompute times as D* Lite on the room grids. With grid sizes of 128×128 and 256×256, no clear difference can be seen, although on the grid size of 512×512 AD* with changing a ϵ-value was a bit faster on all obstacle change magnitudes. On random grids, however, this version of AD* was consistently slower in recompute times than D* Lite in all test cases.

On the room grid type, the recompute times of AD* seem to decrease when increasing the ϵ-value. However, on the random grid type, the recompute times for all the ϵ-values of 2, 3, and 5 were very similar on every grid size and change magnitude combination. Since ϵ-values lower than 2 or higher than 5 were not tested, it cannot be verified whether decreasing or increasing the ϵ-value past these values would show more of a difference. However at ϵ-value of 1 AD* should perform similarly to D* Lite.

The performance of AD* with static ϵ-values in terms of the traveled path length is shown in Figure 5.24 measured against A* with D* Lite also shown as a reference performance. As expected, AD* with ϵ-values of 2, 3, and 5 never beat the optimal algorithms of A* and D* Lite in terms of path length. Also as expected, a lower ϵ-value never results in a longer path than a higher ϵ-value. The difference in path lengths to A* ranged from about 106% to 125% for ϵ-value of 2, 110% to 136% for ϵ-value of 3, and 110% to 147% for ϵ-value of 5.

The path lengths also were much closer to each other on the random grids compared to the fairly spread-out results of the room grids. Especially on the smallest room grid, sized 128×128, the differences between the performance of the AD*



(a) Room grid 128 path lengths against A*.

(b) Room grid 256 path lengths against A*.

(c) Room grid 512 path lengths against A*.

(d) Random grid 128 path lengths against A*.

(e) Random grid 256 path lengths against A*.

(f) Random grid 512 path lengths against A*.

Figure 5.24: Path distances of AD* with static ϵ-values measured against A*, with the performance of D* Lite also show as reference.

variants are within a few percentage points of each other. AD* with ϵ-value of 2 comes close to the path lengths achieved by D* Lite occasionally on the room grids, while the perceived difference is larger between the two on the random grids. The difference between the performance of AD* in these two grid types potentially comes from the fact that on same-sized grids, the agent would have to travel through more nodes on the room grid than on the random grid due to the nature of these grids. On the random grid, the agent can often travel in the general direction of the goal, whereas the room grid has more dead-ends and a more complex structure meaning travel would have to be more informed in this type of setting. This would mean that high enough ϵ-values on a random grid will not have much difference to each other compared to the same ϵ-values on a room grid. However the performance of D* Lite should closely match the performance of AD* with ϵ-value of 1. When comparing the traveled path length results of D* Lite to AD* it can assumed that the path lengths AD* with ϵ-values between 1 and 2 would fall between the results of D* Lite and AD* with ϵ-value of 2.

In these tests, AD* with changing a ϵ-value was expected to perform close to the best parts of D* Lite and AD* with a ϵ-value higher than 1. The traveled path length was expected to have been close to D* Lite, but with faster path recompute times like with AD* with static ϵ-values in these tests. Path length-wise it does achieve this and has results close to what D* Lite managed, while also occasionally beating D* Lite in some grid size and obstacle change magnitude combinations on the room grid. However, the results were mixed when focusing on the recompute times. This version of AD* only achieved faster times on room grids of size 256×256 and 512×512, while being slower on all random grids. It was also slower on the room grid sized 128×128 but with the results looking to be closer than on the random grid. It would seem that this version of AD* is more favored on these room grids and that the effect that the map type it is used in does have somewhat significant results on its performance. It could also be the result of the random grids having a lot more nodes that can change their states to obstacles and back, which D* Lite would be able to process quicker.

6 Conclusions

In this thesis, four different pathfinding algorithms were tested in a dynamic environment, and their performance was measured and analyzed. The dynamism of the environment was achieved by changing the state of a portion of the predefined dynamic nodes on a grid to a free or an obstacle node. This change happened every 10 steps of the agent that used the algorithm to navigate the environment. The tests were run on two types of grids with both having 3 different sizes. The effect of the magnitude of change in the environment was also tested by having four different values indicating the portion of changing dynamic nodes for each grid. This totals to 24 different scenarios for each algorithm that were tested. The measurements taken were total time, average recompute time, traveled path length, and number of node expansions. The results are shown as an average of 100 runs for each test case.

The research questions aimed to be answered were:

- How do the selected pathfinding algorithms compare in dynamic environments?
- How different environmental variables affect the performance of these algorithms in dynamic environments?

The rest of this chapter discusses and summarises the findings of the thesis to answer these research questions.

A* was selected to be tested due to its widespread usage. It was also used as a comparison for the other algorithms that were developed for dynamic environments. In terms of the traveled path lengths it performed the best and even D* Lite which is also an optimal algorithm, had to often travel longer distances to reach the goal, when no tie-breaking criteria when choosing the next step is used. In terms of total time and the average recompute time D* Lite was generally quite a bit faster than A* when environment changes were small. However, the execution time rose rapidly when larger changes were introduced whereas the performance of A* stayed relatively consistent. An effect of the grid size used was also noted, where the performance of D* Lite, measured against A*, decreased on larger grids.

RTD* has two parameters that affect its performance the per-step computation limit and the local ratio. In these tests, five different computation limits were set and tested while the local ratio was kept the same. RTD* achieved fast recomputation times, especially on lower computation limit values. Like D* Lite its performance was heavily affected by the increase in the amount of changing obstacles. However, even in test cases with large changes, it still often achieved competitive times. RTD* was found particularly fast on the room grid type where only the doors between rooms can open or close resulting in a low number of nodes that can change their state.

Path lengths, however, were on average longer than with A* or D* Lite as RTD* trades optimality for faster computation time. Still, there were cases where the path lengths achieved by RTD* were close to or even shorter than what was achieved by D* Lite, such as in large room grids. The effects of the computation limit parameter were found to be quite significant, with smaller values giving faster times but longer path lengths than higher values, and it should be selected to be used case-by-case depending on the environment and desired performance trade-off between execution time and path lengths.

AD* uses an inflation factor ϵ to control the optimality of the algorithm. AD* was tested with four versions of static ϵ-values and a version where the ϵ-value was lowered after every search and increased when changes happened in the environment. The versions with static ϵ-values generally performed well in terms of recomputation times, especially on the random grids where obstacles are generated randomly with no structure to them. However, the path lengths on the random grids were the longest, on the room grid the path lengths were still noticeably longer than A* or D* Lite.

The AD* version with changing ϵ-value achieved far shorter path lengths than the versions with static values, almost equaling D* Lite. However, this version only managed faster recomputation times compared to D* Lite in the room grids. On the random grids, it was the slowest algorithm.

The tests done in this thesis showed that the different elements of the environment play a large part in the difference in the performance of pathfinding algorithms, be it the size or the structure of the environment or the size of changes in obstacles. Other environment variances that were not tested but would also likely affect the performance of the algorithms could be e.g. the density of the obstacles and the frequency of the changes.

References

[1] Z. Abd 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. 1–11, Apr. 2015. DOI: 10.1155/2015/736138.

[2] S. L. Pardede, F. R. Athallah, Y. N. Huda, and F. D. Zain, "A review of pathfinding in game development", *CEPAT] Journal of Computer Engineering: Progress, Application and Technology*, vol. 1, no. 01, p. 47, 2022.

[3] V. Rahmani and N. Pelechano, "Towards a human-like approach to path finding", *Computers Graphics*, vol. 102, pp. 164–174, 2022, ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2021.08.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849321001849.

[4] D. Mackay, "Path planning with D*-Lite", *DRDC Suffield TM*, vol. 242, 2005.

[5] A. Botea, B. Bouzy, M. Buro, C. Bauckhage, and D. Nau, "Pathfinding in Games", in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds., vol. 6, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013, pp. 21–31, ISBN: 978-3-939897-62-0. DOI: 10.4230/DFU.Vol6.12191.21. [Online]. Available: https://drops-dev.dagstuhl.de/entities/document/10.4230/DFU.Vol6.12191.21.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

[7] S. Koenig and M. Likhachev, "D*lite", in *Eighteenth National Conference on Artificial Intelligence*, Edmonton, Alberta, Canada: American Association for Artificial Intelligence, 2002, pp. 476–483, ISBN: 0262511290.

[8] D. Bond, N. Widger, R. Wheeler, and X. Sun, "Real-time search in dynamic worlds", *Third Annual Symposium on Combinatorial Search*, vol. 1, no. 1, pp. 16–22, 2010. DOI: 10.1609/socs.v1i1.18174.

[9] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic A*: An anytime, replanning algorithm", ser. ICAPS'05, Monterey, California, USA: AAAI Press, 2005, pp. 262–271, ISBN: 1577352203.

[10] K. Cai, C. Wang, J. Cheng, C. W. de Silva, and M. Q. Meng, "Mobile robot path planning in dynamic environments: A survey", *CoRR*, vol. abs/2006.14195, 2020. arXiv: 2006.14195. [Online]. Available: https://arxiv.org/abs/2006.14195.

[11] S. Lawande, G. Jasmine, L. Anbarasi, and L. Izhar, "A systematic review and analysis of intelligence-based pathfinding algorithms in the field of video games", *Applied Sciences*, vol. 12, p. 5499, May 2022. DOI: 10.3390/app12115499.

[12] E. W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. [Online]. Available: https://api.semanticscholar.org/CorpusID:123284777.

[13] J. Smed and H. Hakonen, "Path finding", in *Algorithms and Networking for Computer Games*. John Wiley Sons, Ltd, 2017, ch. 7, pp. 159–174, ISBN: 9781119259770. DOI: https://doi.org/10.1002/9781119259770.ch7.

eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119259770.ch7. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119259770.ch7.

[14] S. K. Ghosh, "Visibility algorithms in the plane", 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:59646865.

[15] N. M. Wardhana, H. Johan, and H. Seah, "Enhanced waypoint graph for surface and volumetric path planning in virtual worlds", *The Visual Computer*, vol. 29, Oct. 2013. DOI: 10.1007/s00371-013-0837-x.

[16] E. Volna and M. Kotyrba, "Pathfinding in a dynamically changing environment", in *Intelligent Information and Database Systems: 10th Asian Conference, ACIIDS 2018, Dong Hoi City, Vietnam, March 19-21, 2018, Proceedings, Part II 10*, Springer, 2018, pp. 265–274.

[17] A. Le and T. Le, "Search-based planning and replanning in robotics and autonomous systems", in Sep. 2018, ISBN: 978-1-78923-578-4. DOI: 10.5772/intechopen.71663.

[18] A. Vemula, K. Mülling, and J. Oh, "Path planning in dynamic environments with adaptive dimensionality", *CoRR*, vol. abs/1605.06853, 2016. arXiv: 1605.06853. [Online]. Available: http://arxiv.org/abs/1605.06853.

[19] P. Raja and S. Pugazhenthi, "Path planning for a mobile robot in dynamic environments", *International Journal of Physical Sciences*, vol. 6, Sep. 2011.

[20] R. Smierzchalski and Z. Michalewicz, "Path planning in dynamic environments", in *Innovations in Robot Mobility and Control*, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:10954221.

[21] A. Patel. "Game programming - heuristics". https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. (1997, last modified: 04 Nov 2023).

[22] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. Prentice Hall, 2010.

[23] A. Stentz, *Optimal and Efficient Path Planning for Unknown and Dynamic Environments*, eng. 1993.

[24] S. Koenig and M. Likhachev, "Incremental A*", in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS'01, Vancouver, British Columbia, Canada: MIT Press, 2001, pp. 1539–1546.

[25] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain", *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005. DOI: 10.1109/TRO.2004.838026.

[26] R. E. Korf, "Real-time heuristic search", *Artificial Intelligence*, vol. 42, no. 2, pp. 189–211, 1990, ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(90)90054-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370290900544.

[27] S. Koenig and X. Sun, "Comparing real-time and incremental heuristic search for real-time situated agents", eng, *Autonomous agents and multi-agent systems*, vol. 18, no. 3, pp. 313–341, 2009, ISSN: 1387-2532.

[28] L. H. O. Rios and L. Chaimowicz, "A survey and classification of A* based best-first heuristic search algorithms", in *Advances in Artificial Intelligence – SBIA 2010*, A. C. da Rocha Costa, R. M. Vicari, and F. Tonidandel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 253–262, ISBN: 978-3-642-16138-4.

[29] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality", in *Proceedings of the 16th International Conference*

on Neural Information Processing Systems, ser. NIPS'03, Whistler, British Columbia, Canada: MIT Press, 2003, pp. 767–774.

[30] H. Müller-Merbach, "Heuristics: Intelligent search strategies for computer problem solving: Judea PEARL Addison-Wesley, Reading, 1984", *European Journal of Operational Research*, vol. 21, no. 2, pp. 278–279, 1985, ISSN: 0377-2217. DOI: https://doi.org/10.1016/0377-2217(85)90047-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377221785900475.

[31] T. Cazenave, "Optimizations of data structures, heuristics and algorithms for path-finding on maps", in *2006 IEEE Symposium on Computational Intelligence and Games*, 2006, pp. 27–33. DOI: 10.1109/CIG.2006.311677.

[32] N. R. Sturtevant, "Benchmarks for grid-based pathfinding", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144–148, 2012. DOI: 10.1109/TCIAIG.2012.2197681.

Appendix A Results data

This appendix covers all the results of the test cases. The value next to RTD* in the tables refers to the expansion limit used. The value next to AD* refers to the ϵ value used. AD* changing refers to the ϵ value changing between searches. All the results are obtained as the average of 100 runs.

Table A.1: Room 128. 2% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	71	2.333	301	62910
D* Lite	25	0.509	297	12170
RTD* 32	38	0.074	603	8086
RTD* 64	56	0.118	512	11938
RTD* 128	76	0.189	421	15420
RTD* 256	73	0.243	334	13985
RTD* 512	63	0.288	307	11743
AD* 2	42	0.988	341	11021
AD* 3	37	0.766	389	10008
AD* 5	29	0.589	410	7709
AD* changing	88	0.601	299	22161

Table A.2: Room 128. 5% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	80	2.464	322	74520
D* Lite	51	1.314	321	22589
RTD* 32	34	0.072	549	7332
RTD* 64	54	0.123	478	11201
RTD* 128	82	0.217	418	16478
RTD* 256	107	0.353	362	20523
RTD* 512	122	0.536	335	22454
AD* 2	58	1.580	340	17200
AD* 3	50	1.253	375	14301
AD* 5	42	0.982	401	11957
AD* changing	137	1.816	322	33784

Table A.3: Room 128. 10% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	91	2.642	345	85707
D* Lite	100	2.589	355	42850
RTD* 32	33	0.073	537	7153
RTD* 64	58	0.138	486	12008
RTD* 128	93	0.258	425	18368
RTD* 256	132	0.450	373	24413
RTD* 512	183	0.763	360	33236
AD* 2	114	2.921	369	35646
AD* 3	96	2.258	398	29683
AD* 5	75	1.699	421	22841
AD* changing	205	2.513	346	63459

Table A.4: Room 128. 20% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	106	2.709	391	99203
D* Lite	178	4.241	406	74713
RTD* 32	34	0.080	537	7316
RTD* 64	58	0.152	468	11720
RTD* 128	104	0.295	437	20254
RTD* 256	159	0.539	394	29321
RTD* 512	267	0.985	411	48250
AD* 2	187	4.285	415	59132
AD* 3	146	3.292	433	45490
AD* 5	104	2.206	451	32136
AD* changing	323	4.315	394	105331

Table A.5: Room 256. 2% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	647	10.778	626	539873
D* Lite	375	5.611	628	140604
RTD* 32	81	0.088	1125	16298
RTD* 64	127	0.164	941	24795
RTD* 128	212	0.289	822	39217
RTD* 256	335	0.537	721	58822
RTD* 512	509	0.915	677	86787
AD* 2	395	5.714	699	110492
AD* 3	318	4.185	771	86826
AD* 5	252	3.016	845	68205
AD* changing	900	4.642	635	242681

Table A.6: Room 256. 5% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	670	10.556	669	559229
D* Lite	864	12.896	691	310169
RTD* 32	80	0.101	1105	16066
RTD* 64	131	0.178	939	24952
RTD* 128	226	0.333	824	41440
RTD* 256	374	0.619	720	64439
RTD* 512	860	1.396	767	142080
AD* 2	709	10.356	721	202143
AD* 3	663	8.377	832	188141
AD* 5	493	5.891	886	139151
AD* changing	1626	10.269	683	474271

Table A.7: Room 256. 10% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	774	11.574	707	639216
D* Lite	1559	22.567	732	550125
RTD* 32	78	0.113	1049	15246
RTD* 64	128	0.202	888	23626
RTD* 128	229	0.364	797	41105
RTD* 256	420	0.682	743	70453
RTD* 512	755	1.288	715	123649
AD* 2	1154	15.589	791	333105
AD* 3	912	11.502	857	262252
AD* 5	706	8.444	898	202187
AD* changing	3296	21.024	735	1012450

Table A.8: Room 256. 20% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	946	12.749	792	775388
D* Lite	2557	34.139	799	901818
RTD* 32	80	0.132	1034	15095
RTD* 64	135	0.220	907	24561
RTD* 128	242	0.400	810	42970
RTD* 256	453	0.738	771	76432
RTD* 512	860	1.396	767	142080
AD* 2	1784	22.458	857	515567
AD* 3	1376	16.552	902	396935
AD* 5	1046	11.655	963	301610
AD* changing	6197	37.332	806	1945127

Table A.9: Room 512. 2% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	5041	45.958	1176	3653122
D* Lite	5914	51.270	1214	1785302
RTD* 32	212	0.141	2495	38269
RTD* 64	335	0.232	2087	60139
RTD* 128	523	0.416	1614	87971
RTD* 256	804	0.710	1396	126735
RTD* 512	1353	1.250	1314	208062
AD* 2	2457	18.818	1396	625645
AD* 3	1906	12.920	1579	483396
AD* 5	1538	9.795	1677	388050
AD* changing	11601	35.672	1216	2881454

Table A.10: Room 512. 5% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	5044	44.646	1219	3678180
D* Lite	10697	89.756	1286	3167228
RTD* 32	205	0.178	2338	35757
RTD* 64	340	0.268	2084	60907
RTD* 128	536	0.471	1579	88526
RTD* 256	879	0.798	1422	137982
RTD* 512	1456	1.464	1295	221097
AD* 2	4088	29.263	1512	1065771
AD* 3	3258	21.279	1646	850201
AD* 5	2715	16.392	1789	707477
AD* changing	21738	69.829	1292	5700029

Table A.11: Room 512. 10% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	5611	46.325	1313	4070216
D* Lite	16640	128.477	1412	4852433
RTD* 32	212	0.237	2293	35163
RTD* 64	354	0.331	2071	60771
RTD* 128	554	0.544	1578	90592
RTD* 256	963	0.915	1447	150287
RTD* 512	1677	1.648	1364	256991
AD* 2	6082	40.425	1646	1613898
AD* 3	4832	29.454	1788	1275526
AD* 5	3910	22.542	1890	1031184
AD* changing	39005	114.963	1435	10456869

Table A.12: Room 512. 20% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	6236	45.710	1479	4543551
D* Lite	23434	158.723	1618	6716797
RTD* 32	224	0.346	2246	34529
RTD* 64	366	0.455	2044	60432
RTD* 128	562	0.661	1547	90358
RTD* 256	1077	1.073	1506	164323
RTD* 512	2011	1.831	1489	309174
AD* 2	8592	50.590	1841	2282869
AD* 3	6669	36.815	1973	1769471
AD* 5	5462	28.528	2090	1441077
AD* changing	58000	155.508	1599	15632080

Table A.13: Random 128. 2% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	26	1.027	254	24031
D* Lite	18	0.236	254	8226
RTD* 32	16	0.156	257	2413
RTD* 64	23	0.202	257	3664
RTD* 128	31	0.272	254	4992
RTD* 256	38	0.304	254	6218
RTD* 512	45	0.388	254	7096
AD* 2	7	0.219	287	507
AD* 3	7	0.214	288	473
AD* 5	7	0.213	291	459
AD* changing	66	0.647	254	8983

Table A.14: Random 128. 5% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	26	1.059	254	25863
D* Lite	26	0.597	256	10236
RTD* 32	21	0.297	263	2848
RTD* 64	30	0.372	259	4403
RTD* 128	42	0.490	257	6254
RTD* 256	52	0.659	257	7829
RTD* 512	57	0.750	256	8937
AD* 2	13	0.447	284	1070
AD* 3	12	0.434	286	990
AD* 5	12	0.430	289	973
AD* changing	81	0.992	257	10802

Table A.15: Random 128. 10% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	28	1.124	255	26998
D* Lite	41	1.201	258	13335
RTD* 32	29	0.493	271	3323
RTD* 64	41	0.592	264	5279
RTD* 128	57	0.775	260	7731
RTD* 256	73	1.053	259	10120
RTD* 512	80	1.385	258	11608
AD* 2	22	0.854	280	2426
AD* 3	22	0.827	282	2277
AD* 5	22	0.830	284	2311
AD* changing	104	1.626	259	13610

Table A.16: Random 128. 20% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	30	1.183	255	27910
D* Lite	68	2.281	263	18912
RTD* 32	40	0.841	276	3666
RTD* 64	57	0.978	270	6292
RTD* 128	78	1.211	266	9736
RTD* 256	106	1.634	263	13397
RTD* 512	123	2.187	263	16342
AD* 2	37	1.461	278	3888
AD* 3	37	1.437	279	3773
AD* 5	37	1.431	281	3778
AD* changing	150	2.918	263	19546

Table A.17: Random 256. 2% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	175	3.673	510	160376
D* Lite	112	1.627	512	37067
RTD* 32	58	0.450	533	6838
RTD* 64	95	0.555	526	11656
RTD* 128	137	0.727	519	18294
RTD* 256	192	1.009	516	25455
RTD* 512	228	1.330	514	31520
AD* 2	43	0.777	568	2418
AD* 3	45	0.793	586	2493
AD* 5	45	0.790	598	2556
AD* changing	487	2.874	514	39374

Table A.18: Random 256. 5% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	183	3.826	511	163676
D* Lite	222	3.881	517	59731
RTD* 32	93	0.998	549	7587
RTD* 64	128	1.119	536	13595
RTD* 128	202	1.370	528	22977
RTD* 256	285	1.790	521	35346
RTD* 512	377	2.475	519	47121
AD* 2	98	1.889	565	8846
AD* 3	98	1.851	575	9000
AD* 5	99	1.838	583	8984
AD* changing	704	5.214	520	64679

Table A.19: Random 256. 10% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	192	4.044	512	170366
D* Lite	380	7.095	522	89712
RTD* 32	132	1.796	553	7875
RTD* 64	186	1.977	544	14723
RTD* 128	261	2.255	536	26652
RTD* 256	385	2.774	527	43993
RTD* 512	557	3.764	525	64504
AD* 2	180	3.496	566	17502
AD* 3	186	3.557	576	17247
AD* 5	190	3.594	583	17353
AD* changing	1000	9.168	528	105960

Table A.20: Random 256. 20% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	200	4.173	513	174809
D* Lite	656	12.547	530	136167
RTD* 32	217	3.391	555	8070
RTD* 64	255	3.518	546	15249
RTD* 128	348	3.836	539	28284
RTD* 256	500	4.394	532	49671
RTD* 512	767	5.698	529	80281
AD* 2	305	5.887	574	23217
AD* 3	303	5.771	582	22976
AD* 5	300	5.680	586	22608
AD* changing	1509	16.402	536	186514

Table A.21: Random 512. 2% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	1305	13.886	1024	1174746
D* Lite	953	8.293	1032	217966
RTD* 32	262	1.671	1121	16642
RTD* 64	359	1.829	1092	30567
RTD* 128	553	2.133	1072	54219
RTD* 256	804	2.621	1055	88803
RTD* 512	1173	3.563	1045	134909
AD* 2	346	3.251	1155	27700
AD* 3	348	3.220	1170	26070
AD* 5	353	3.199	1201	26568
AD* changing	5009	16.905	1035	236242

Table A.22: Random 512. 5% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	1382	14.698	1024	1244540
D* Lite	1934	18.608	1039	377591
RTD* 32	473	3.766	1120	16915
RTD* 64	578	3.948	1097	31784
RTD* 128	776	4.234	1079	59527
RTD* 256	1133	4.842	1064	107103
RTD* 512	1735	6.165	1054	178178
AD* 2	797	7.622	1153	75166
AD* 3	792	7.487	1168	71616
AD* 5	801	7.436	1190	70894
AD* changing	7266	27.459	1050	441818

Table A.23: Random 512. 10% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	1386	14.690	1026	1252571
D* Lite	3354	33.158	1048	579126
RTD* 32	783	6.882	1121	17060
RTD* 64	874	7.074	1097	32387
RTD* 128	1069	7.401	1082	61654
RTD* 256	1447	8.094	1067	115038
RTD* 512	2169	9.538	1058	202847
AD* 2	1293	12.302	1167	100539
AD* 3	1279	12.033	1178	95659
AD* 5	1295	12.013	1197	94843
AD* changing	10071	43.937	1064	759609

Table A.24: Random 512. 20% magnitude.

Algorithm	**Total (ms)**	**Recompute (ms)**	**Length**	**Nodes**
A*	1376	14.530	1029	1242035
D* Lite	5740	56.991	1063	895093
RTD* 32	1375	12.745	1127	17273
RTD* 64	1468	13.092	1103	33001
RTD* 128	1671	13.477	1093	63487
RTD* 256	2054	14.164	1075	119739
RTD* 512	2830	15.472	1064	218397
AD* 2	2085	19.697	1177	109285
AD* 3	2090	19.504	1191	105125
AD* 5	2093	19.269	1208	103140
AD* changing	14738	75.336	1079	1398342