Acknowledgements

First and foremost, I would like to thank my advisor, Sven Koenig, for his guidance and strong support throughout this journey. I would also like to thank my dissertation committee and dissertation proposal committee, Maxim Likhachev, Raghu Raghavendra, Michael Zyda, Aiichiro Nakano and Cyrus Shahabi for their helpful comments and suggestions. I would also like to thank my colleagues and collaborators Xiaoming Zheng, Po-An Chen, Kenny Daniel, Alex Nash, Janusz Marecki, Changhe Yuan, Jason Tsai, Carlos Hernandez, Pedro Meseguer, Wheeler Ruml, Marek Druzdzel, Pradeep Varakantham, David Bond, Tansel Uras, and Niels Widger for all the stimulating discussions we have had. I thank Nathan Sturtevant for providing the game maps used in the chapter on Experimental Evaluation to measure the runtimes of the developed incremental search algorithms. I would like to separately acknowledge Maxim Likhachev, the creator of SBPL, for providing me with the source code of SBPL and explaining to me how to use it, and William Yeoh for our fruitful collaborations during the past seven years that directly influenced my dissertation. Last but not least, I would like to thank my wife for being so understanding and patient with me.
**Table of Contents**

Acknowledgements ii

List of Tables vi

List of Figures vii

Abstract xi

Chapter 1: Introduction 1

1.1 Problem ........................................... 4
1.1.1 Approaches .................................... 5
1.1.2 Terrains ...................................... 7
1.2 Hypothesis ...................................... 9
1.3 Contributions .................................. 12
1.4 Dissertation Structure .......................... 15

Chapter 2: Background 16

2.1 Notation ........................................ 16
2.2 State Space Representation ....................... 17
2.2.1 Grids ....................................... 18
2.2.2 State Lattices ................................ 19
2.3 A* ............................................... 21
2.3.1 Values ....................................... 21
2.3.2 OPEN and CLOSED Lists ....................... 22
2.3.3 Operations ................................... 24
2.3.4 Properties ................................... 25
2.3.5 Example ..................................... 27
2.3.6 Applying A* to Moving Target Search ........ 29
2.3.6.1 Search Directions ......................... 29
2.3.6.2 Repeated A* .............................. 30
2.4 Repeated A*-based Path Planning Algorithms for Agent Navigation 34
2.4.1 Real-time Search .............................. 34
2.4.2 Incremental Search ............................ 37
2.4.2.1 Adaptive A* .............................. 40
2.4.2.2 D* Lite ................................. 52
### Chapter 3: Incremental Search by Learning $h$-values

3.1 Review of Moving Target Adaptive A* .......................... 67
   3.1.1 Eager Moving Target Adaptive A*
      3.1.1.1 Applicability ........................................ 68
      3.1.1.2 Principle ........................................ 68
      3.1.1.3 Operations ........................................ 69
      3.1.1.4 Example ........................................ 70
   3.1.2 Lazy Moving Target Adaptive A*
      3.1.2.1 Applicability ........................................ 74
      3.1.2.2 Principle ........................................ 75
      3.1.2.3 Operations ........................................ 76
      3.1.2.4 Example ........................................ 82
3.2 Generalized Adaptive A* ........................................ 85
   3.2.1 Applicability ........................................ 85
   3.2.2 Principle ........................................ 86
   3.2.3 Operations ........................................ 86
   3.2.4 Example ........................................ 93
3.3 Summary .................................................... 96

### Chapter 4: Incremental Search by Reusing Search Trees

4.1 Generalized Fringe-Retrieving A* and its Optimization ..... 100
   4.1.1 Generalized Fringe-Retrieving A*
      4.1.1.1 Applicability ........................................ 100
      4.1.1.2 Principle ........................................ 101
      4.1.1.3 Operations ........................................ 101
      4.1.1.4 Example ........................................ 111
   4.1.2 Fringe-Retrieving A* .................................... 113
      4.1.2.1 Applicability ........................................ 113
      4.1.2.2 Principle ........................................ 114
      4.1.2.3 Operations ........................................ 115
      4.1.2.4 Example ........................................ 123
4.2 Moving Target D* Lite ......................................... 126
   4.2.1 Basic Moving Target D* Lite ............................ 127
      4.2.1.1 Applicability ........................................ 127
      4.2.1.2 Principle ........................................ 127
      4.2.1.3 Operations ........................................ 129
      4.2.1.4 Example ........................................ 133
   4.2.2 Moving Target D* Lite .................................. 135
      4.2.2.1 Applicability ........................................ 135
      4.2.2.2 Principle ........................................ 136
      4.2.2.3 Operations ........................................ 136
Chapter 5: Experimental Evaluation

5.1 Objective .................................................. 144
5.2 Experimental Setups ........................................ 146
  5.2.1 Algorithms .............................................. 146
  5.2.2 State Spaces ............................................. 147
  5.2.3 Terrains ................................................ 148
  5.2.4 Target Movement Strategies ............................ 149
  5.2.5 Measures ............................................... 150
5.3 Experimental Results ..................................... 152
  5.3.1 Experiments in Known Static Terrains ................ 152
    5.3.1.1 Target Movement Strategy: Random Waypoint .... 152
    5.3.1.2 Target Movement Strategy: TrailMax ............... 161
    5.3.1.3 Summary ............................................ 166
  5.3.2 Experiments in Known Dynamic Terrains .............. 168
    5.3.2.1 Summary ............................................ 176
  5.3.3 Experiments in Unknown Static Terrains .............. 178
    5.3.3.1 Summary ............................................ 181
5.4 Conclusions ................................................. 182

Chapter 6: Applications

6.1 Generalized Adaptive A* Application ...................... 190
  6.1.1 Background ............................................. 191
  6.1.2 State Space Representation ............................ 193
  6.1.3 Overview of the Experimental Results ................ 193
  6.1.4 Discussion .............................................. 194
6.2 Generalized Fringe-Retrieving A* Application ............ 195
  6.2.1 Background ............................................. 196
  6.2.2 State Space Representation ............................ 196
  6.2.3 Overview of the Experimental Results ................ 198
  6.2.4 Discussion .............................................. 199
6.3 Moving Target D* Lite Application ......................... 200
  6.3.1 Background ............................................. 201
  6.3.2 State Space Representation ............................ 201
  6.3.3 Overview of the Experimental Results ................ 202
  6.3.4 Discussion .............................................. 203
6.4 Conclusions ................................................. 204

Chapter 7: Conclusions

Reference List ................................................ 211
Bibliography .................................................. 211
List of Tables

1.1 Search Problems that Existing Incremental Search Algorithms Apply to . 9
1.2 Contributions ............................................................... 11
1.3 Fastest Incremental Search Algorithms for Different Terrains ........... 15
2.1 Operations of A* ........................................................... 29
3.1 Search Problems that Heuristic Learning Incremental Search Algorithms Apply to .............................................................. 65
4.1 Search Problems that Search Tree Transforming Incremental Search Algorithms Apply to .......................................................... 97
5.1 Experimental Results in Known Static Grids (Target Movement Strategy: Random Waypoint) .................................................. 152
5.2 Experimental Results in Known Static Grids (Target Movement Strategy: TrailMax) ............................................................... 161
5.3 Experimental Results in Known Dynamic Grids ............................ 169
5.4 Experimental Results in Unknown Static Grids ............................ 178
5.5 Best Incremental Search Algorithms for Different Terrains ............... 183
## List of Figures

1.1 Contributions ............................................................. 12
2.1 Grids ................................................................. 17
2.2 Example Representation .............................................. 19
2.3 Example Feasible Motion Primitive ................................. 20
2.4 Pseudo Code of A* ................................................... 23
2.5 Legend of Figure 2.6 .................................................. 27
2.6 Example Trace of A* .................................................. 28
2.7 Pseudo Code of Forward Repeated A* ............................... 31
2.8 Legend of Figure 2.9 .................................................. 33
2.9 Example Trace of Forward Repeated A* ............................. 33
2.10 Pseudo Code of Eager Adaptive A* ................................. 42
2.11 Legend of Figure 2.12 ............................................... 46
2.12 Example Trace of Forward Repeated A* ............................. 47
2.13 Example Trace of Eager Adaptive A* ............................... 47
2.14 Pseudo Code of Lazy Adaptive A* .................................. 49
2.15 Legend of Figure 2.16 ............................................... 51
2.16 Example Trace of Lazy Adaptive A* ............................... 52
Abstract

In this dissertation, I demonstrate how to speed up path planning for moving target search, which is a problem where an agent needs to move to a target and the target can move over time. It is assumed that the current locations of the agent and the target are known to the agent at all times. The information about the terrain that an agent has is the action costs of the agent in any particular location of the terrain. The information about the terrain that the agent has can change over time depending on different applications: For example, when a robot is deployed in a new terrain without any map a priori, the robot initially does not have any information about the terrain and it has to acquire information about the terrain during navigation. However, a character in a computer game may have complete information about the terrain that remains unchanged over time given that the whole game map is loaded in memory and is available to the character. I use the following movement strategy for the agent that is based on assumptive planning: The agent first finds a cost-minimal path to the target with the information about the terrain that is currently available to it. The agent then starts moving along the path. Whenever new information about the terrain is acquired or the target moves off the path, the agent performs a new search to find a new cost-minimal path from the agent to the target. The agent uses this movement strategy until either the target is caught or the agent finds that
there does not exist any path from the agent to the target after a search (and in any future searches), upon which the agent stops navigation. Since the agent’s information about the terrain can change and the target can move over time, the agent needs to repeatedly perform searches to find new cost-minimal paths to the target. Path planning for moving target search by using this movement strategy is thus often a repeated search process. Additionally, agents need to find new cost-minimal paths as fast as possible, such that they move smoothly and without delay.

Many path planning algorithms have been developed, among which incremental search algorithms reuse information from previous searches to speed up the current search and are thus often able to find cost-minimal paths for series of similar search problems faster than by solving each search problem from scratch. Incremental search algorithms have been demonstrated to be very successful in path planning for many important applications in robotics. However, it is assumed that the target does not move over time during navigation for most incremental search algorithms, and they are either inapplicable or run more slowly than A* to solve moving target search. Thus, I demonstrate how to speed up search-based path planning for moving target search by developing new incremental search algorithms.

In my dissertation, I make the following contributions: (1) I develop Generalized Adaptive A* (GAA*), that learns h-values (= heuristic values) to make them more informed for moving target search. GAA* applies to moving target search in terrains where the action costs of the agent can change between searches. (2) I develop Generalized Fringe-Retrieving A* (G-FRA*), that transforms the search tree of the previous search to the search tree of the current search for moving target search. Though G-FRA*
applies only to moving target search in terrains where the action costs of the agent do not change between searches, it creates a new way of transforming the search tree of the previous search to the search tree of the current search. (3) I develop Moving Target D* Lite (MT-D* Lite), that transforms the search tree of the previous search to the search tree of the current search for moving target search. MT-D* Lite combines the principles of G-FRA* and D* Lite, an existing incremental search algorithm. MT-D* Lite applies to moving target search in terrains where the action costs of the agent can change between searches. (4) I compare the new incremental search algorithms, discuss their strengths and weaknesses and provide guidelines for when to choose a particular algorithm over another. Simulation results show that the developed incremental search algorithms run up to one order of magnitude faster than A* for moving target search.
Chapter 1: Introduction

An agent often needs to move from its start location to a target in a terrain, which is the vertical and horizontal dimension of land surface, known as agent navigation (Thrun, 1998; Goto & Stentz, 1987; Hrabar & Sukhatme, 2009). Although in different applications the action costs of the agent can be affected by many things such as the speed of the wind or the battery level of the agent, for ease of description of the developed algorithms, I assume that the action costs of the agent can only be affected by the terrain. This assumption can easily be extended to cover more things that can affect the action costs of the agent in different applications while no changes to the developed algorithms are necessary, because the pseudo codes of the developed algorithms are independent of the action costs of the agent. Agent navigation has a wide range of applications in artificial intelligence, such as in robotics (Garrido, Moreno, Abderrahim, & Monar, 2006; Kushleyev & Likhachev, 2009; Vandapel, Donamukkala, & Hebert, 2006; Ferguson & Stentz, 2005a) and real-time computer games (Rabin, 2002; Yap, Burch, Holte, & Schaeffer, 2011). In such situations, an agent has to take into account new information about the terrain during navigation and respond quickly. For example, a character in real-time computer
games often has to deal with newly observed obstacles that block its path during navigation and repeatedly searches for new paths to catch its target. Moreover, an agent usually has a limited amount of time for planning its path. For example, a character in real-time computer games needs to plan its path quickly in order to move smoothly and without delay. The computer game company Bioware recently imposed a limit of 1-3 ms on the search time (Bulitko, Bjornsson, Luvstrek, Schaeffer, & Sigmundarson, 2007) for game maps of approximately $500 \times 500$ cells.\footnote{I thank Vadim Bulitko for providing the information about the size of the game maps.} Planning a new path from scratch can be computationally expensive or may not even be possible if an agent frequently receives new information (Ferguson, 2006).

Agent navigation generally falls into two classes: Stationary target search, where the target does not move over time (Stentz, 1995; Koenig & Likhachev, 2002; Likhachev, 2005; Ferguson, 2006), and moving target search, where the target can move over time (Ishida & Korf, 1991; Moldenhauer & Sturtevant, 2009b, 2009a). Lots of research has focused on developing path planning algorithms for stationary target search. For example, incremental search algorithms reuse information from previous searches to speed up the current search and are thus often able to find cost-minimal paths for series of similar search problems faster than by solving each search problem from scratch (Koenig, Likhachev, Liu, & Furcy, 2004). Most of the recently developed incremental search algorithms are based on A* (Hart, Nilsson, & Raphael, 1968), a popular search algorithm that uses $h$-values (= heuristic values), estimates of the minimum costs of moving from each state to the goal state, to guide its search. The informedness of an $h$-value of a given state is a measure of how accurate that $h$-value estimates the minimum cost of moving from the given
state to the goal state. More informed $h$-values (= $h$-values that estimate the minimum costs of moving from each state to the goal state more accurately) make A* focus its search better and hence dramatically out-perform a search that does not use $h$-values (= uninformed search) (Pearl, 1985). Incremental search algorithms are well-suited for path planning for stationary target search. For example, Field D* (Ferguson & Stentz, 2005c), an incremental search algorithm extended from D* Lite (Koenig & Likhachev, 2002), is currently implemented as the path planner in several fielded robotic systems, such as Automated E-Gator, GDRS XUV, and Mars rovers of NASA’s Jet Propulsion Laboratory (Ferguson & Stentz, 2005b). Anytime D* (Likhachev, Ferguson, Gordon, Stentz, & Thrun, 2005a, 2005b), an anytime incremental search algorithm, is able to efficiently generate complex paths over large, obstacle-laden terrains. Anytime D* has been applied to efficient path planning for land-based mobile robots, such as the ATRV and Segbot robotic platforms (Likhachev et al., 2005a; Likhachev & Ferguson, 2009), and “BOSS”, the autonomous Chevy Tahoe that won the 2007 Defense Advanced Research Projects Agency (DARPA) Urban Challenge, where unmanned ground vehicles needed to autonomously navigate a 60-mile urban course in moving traffic (Urmson et al., 2008). All these applications of incremental search algorithms make solid progress on fast path planning in robotics.

Thus far, incremental search algorithms, such as D* Lite and its variants, were designed for stationary target search. They are either inapplicable or run more slowly than A* for moving target search. However, many applications in artificial intelligence need to plan and replan paths in response to target movements during navigation (Anderson, 1988; Mehrandezh, Sela, Fenton, & Benhabib, 2000; Riley & Atkeson, 2002; Croft,
For example, a character in computer games often needs to catch other moving characters (Rabin, 2002). Therefore, in this dissertation, I develop new incremental search algorithms for moving target search. I hypothesize that one can develop new incremental search algorithms for moving target search that can speed up repeated A*.

1.1 Problem

In this dissertation, I demonstrate how to speed up path planning for moving target search, which is a problem where an agent needs to move to its target and the target can move over time. It is assumed that the current locations of the agent and the target are known to the agent at all times. This assumption is used in many applications, such as computer games, in which a game character knows its current location and its target’s current location at all times. For example, the target can be another game character that the game character wants to catch. The chase process is triggered when the game player sets another game character as the target. There is no assumption about the movement strategy or speed of the target except that the target can move from a location to another location only when there exists a path between these two locations. I first introduce the approach that I use for moving target search in Section 1.1.1, then I introduce different classes of terrains used in my dissertation in Section 1.1.2.
1.1.1 Approaches

Existing algorithms for moving target search generally fall into two classes:

- The first class is a class of offline path planning algorithms that take into account all possible contingencies (for example, all possible movements of the target and all possible action cost changes in the terrain) to find paths for the agent. Thus, the agent does not perform any path planning during navigation. For example, one version of the minimax search algorithm (Hahn & MacGillivray, 2006) falls into this class. Reverse Minimax A* (RMA*) (Moldenhauer & Sturtevant, 2009b) extends this algorithm by incorporating a heuristic function that estimates the minimum cost of moving from each state to the goal state to speed up the search. However, algorithms of this class are typically slow and do not scale up well due to the large number of contingencies (Moldenhauer & Sturtevant, 2009a).

- The second class is a class of online path planning algorithms that interleave path planning with action execution. Online path planning algorithms do not need to consider all possible contingencies: An agent first finds a path with the information currently available to it. The agent then starts to move along the path found until new information is acquired (for example, the target moved and the path does not lead to the target anymore, or the path is blocked by newly observed obstacles and thus becomes infeasible), upon which the agent performs a new search to find a new path from the agent to the target. The agent uses this approach until either the target is caught successfully or the agent finds that there does not exist any path from the agent to the target after one search (and in future searches), upon which
the agent stops navigation. The solution quality of path planning for moving target search is the total action cost of the agent from the moment it starts navigation until it stops navigation (= the cost of the trajectory of the agent). This approach is known as assumptive planning (Nourbakhsh & Genesereth, 1996). The main advantage of this approach compared to offline algorithms is that it does not need to consider contingencies that the agent has not encountered and hence is faster and more memory-efficient. It has been demonstrated that online path planning algorithms scale up well in the size of the terrain (Likhachev & Ferguson, 2009), and the path of the agent can be calculated reasonably fast to satisfy the runtime requirements of certain applications, such as robotics (Urmson, 2008).

The limitation of assumptive planning is that it is incomplete, that is, an agent using assumptive planning might fail to catch the target if the terrain changes in a way that prevents the agent from catching the target. For example, consider a situation where a robot is trapped in a room. There is a door 10 meters away from the robot to the north (= northern door). The northern door is always open and provides an exit from the room. There is another door 5 meters away from the robot to the south (= southern door). The southern door is controlled by an adversarial agent. The target is right behind the southern door outside of the room. Whenever the robot is within 1 meter from the southern door, the door is shut by the adversarial agent temporarily and reopened when the robot is more than 2 meters away from it. While the robot always has a way out by passing through the northern door, a path planning algorithm using assumptive planning will guide the robot to toggle between the two doors forever. In this situation, the robot
needs to reason explicitly about the changes of the terrain to learn the features of the changes, and identify the southern door as an untraversable area. Despite the fact that path planning algorithms based on assumptive planning are incomplete, agent navigation using assumptive planning has been a great success in many applications of robotics (Stentz, 1997; Koenig & Smirnov, 1996; Likhachev, 2005; Ferguson, 2006). In this dissertation, I develop incremental search algorithms based on assumptive planning for moving target search, and test them by using problems without adversarial agents in them. In this situation, the developed incremental search algorithms are complete and can guide the agent to catch the target in all problems.

1.1.2 Terrains

An agent’s information about the terrain may vary depending on the application. For example, when a robot is deployed in a new terrain without any map a priori, the robot initially does not have any information about the terrain and it has to acquire information about the terrain during navigation. On the other hand, a character in a computer game can have complete information about the terrain given that the whole game map can be loaded into memory and is available to the character. In order to evaluate the runtime efficiency of the developed incremental search algorithms in different terrains, I systematically consider three classes of terrains in this dissertation:

- **Known Static Terrain**: The terrain does not change over time, and the agent has complete information about the terrain at all times. The only information updated
over time for the agent is the current locations of the agent and the target. In this terrain, the action costs of the agent do not change over time.

- **Known Dynamic Terrain:** The terrain can change over time, and the agent has complete information about the terrain at all times. The information updated over time for the agent is thus (1) the current locations of the agent and the target, and (2) the information about the terrain. In this terrain, the action costs of the agent can increase and decrease over time.

- **Unknown Static Terrain:** The terrain does not change over time, but the agent initially has no information about the terrain and thus assumes that the terrain is traversable everywhere (= no obstacles in the terrain).\(^2\) The sensors on-board the agent can typically observe the terrain only within a certain range around its current location and update its information about the terrain if any obstacles are newly observed. The information updated over time for the agent is thus (1) the current locations of the agent and the target, and (2) the information about any newly observed obstacles within the sensor range. In this terrain, the action costs of the agent can increase but not decrease over time.

In this dissertation, most examples use grids for ease of description of the developed algorithms. I use the term “known static grids” to refer to grids in known static terrains. Similarly, the terms “known dynamic” and “unknown static grids” refer to grids in the corresponding terrains.

\(^2\)This is a simple but reasonable assumption since, if it is assumed that a traversable area is untraversable, then there is a chance that no path can be found even if there exists one.
### Table 1.1: Search Problems that Existing Incremental Search Algorithms Apply to

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Action Costs Can Increase Between Searches</th>
<th>Action Costs Can Decrease Between Searches</th>
<th>Start State Can Change Between Searches</th>
<th>Goal State Can Change Between Searches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic Learning</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Incremental Search Algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search Tree Transforming</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Incremental Search Algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.2 Hypothesis

Incremental search algorithms can combine assumptive planning to repeatedly find cost-minimal paths from the agent to the target faster than repeated A* searches. Incremental search algorithms have been demonstrated to be well-suited for path planning for agent navigation where the target does not move over time (Likhachev, 2005; Ferguson, 2006).

Existing incremental search algorithms generally fall into two classes:

- The first class uses information from previous searches to update the \( h \)-values of the current search so that they become more informed and focus the current search better (Principle 1), referred to as “heuristic learning incremental search algorithms”.

- The second class transforms the search tree of the previous search to the search tree of the current search and hence the current search starts with the transformed search tree instead of from scratch (Principle 2), referred to as “search tree transforming incremental search algorithms”.

Table 1.1 lists classes of search problems that existing incremental search algorithms apply to. It shows “Yes” if at least one algorithm is known in that class that applies to the problems and runs faster than repeated A*, and it shows “No” otherwise. Note that, the “start state” and “goal state” in Table 1.1 as well as the discussions in the following
two paragraphs refer to the start state and goal state of each search, respectively, rather than the current states of the agent and the target. Table 1.1 shows that, although existing incremental search algorithms have been demonstrated to be able to speed up path planning for stationary target search, they are either inapplicable or run more slowly than repeated A* for moving target search where both the agent and the target can move and the action costs of the agent can change over time:

- Heuristic learning incremental search algorithms, such as Adaptive A* (Koenig & Likhachev, 2005) and its extension Moving Target Adaptive A* (MT-Adaptive A*) (Koenig, Likhachev, & Sun, 2007), need consistent $h$-values with respect to the goal state of each search, that is, $h$-values that satisfy the “triangle inequality” (Pearl, 1985), to guide their searches. Adaptive A* and MT-Adaptive A* basically transform consistent $h$-values into more informed consistent $h$-values. Adaptive A* is not guaranteed to find cost-minimal paths in terrains where (1) the action costs of the agent can decrease over time or (2) the goal state can change between searches, because consistent $h$-values do not necessarily remain consistent with respect to the goal state of each search in these two situations. MT-Adaptive A* extends Adaptive A* to maintain the consistency of the $h$-values with respect to the goal state after the goal state changed between searches. However, MT-Adaptive A* cannot maintain the consistency of the $h$-values with respect to the goal state of each search when the action costs of the agent decrease between searches. Thus, it is necessary to maintain the consistency of the $h$-values of MT-Adaptive A* with respect to the goal state in terrains where the action costs of the agent can decrease.
between searches, so that it applies to moving target search in all three classes of terrains (see Section 1.1.2).

- Search tree transforming incremental search algorithms, such as D* Lite and its variants, are efficient only if the start state does not change between searches. For stationary target search, D* Lite performs backward searches from the target to the agent by assigning the current states of the agent and target to the goal and start states of each search, respectively. However, it has not been investigated how to apply D* Lite and its variants to moving target search and make it run faster than A* where both the start and goal states can change between searches.

Therefore, in this dissertation, I demonstrate how to speed up path planning for moving target search by developing new incremental search algorithms. My hypothesis is as follows:

*One can develop incremental search algorithms that can find cost-minimal paths faster than repeated A* for moving target search.*
1.3 Contributions

To validate my hypothesis, I developed new incremental search algorithms for moving target search. The newly developed algorithms thus extend the applicability of incremental search algorithms. Table 1.2 lists types of path planning problems that the newly developed incremental search algorithms apply to. In detail, my contributions are shown in Figure 1.1:

- For heuristic learning incremental search algorithms, I develop Generalized Adaptive A* (GAA*) (Sun, Koenig, & Yeoh, 2008), an incremental A* variant for moving target search that extends MT-Adaptive A* (Koenig et al., 2007). GAA* extends

<table>
<thead>
<tr>
<th>Existing Work</th>
<th>My Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic Learning</td>
<td>Adaptive A*</td>
</tr>
<tr>
<td>Incremental Search</td>
<td>MT-Adaptive A*</td>
</tr>
<tr>
<td>Algorithms</td>
<td>GAA*</td>
</tr>
<tr>
<td>Search Tree</td>
<td>G-FRA*</td>
</tr>
<tr>
<td>Transforming</td>
<td>FRA*</td>
</tr>
<tr>
<td>Incremental Search</td>
<td>MT-D* Lite</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Basic MT-D* Lite</td>
</tr>
<tr>
<td>D* Lite</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: Contributions
MT-Adaptive A* to apply to moving target search in terrains where the action costs of the agent can both increase and decrease between searches. Thus, GAA* applies to all three classes of terrains (see Section 1.1.2). GAA* is currently the fastest incremental search algorithm for moving target search in unknown static terrains.

- For search tree transforming incremental search algorithms, I make the following contributions:

  – First, I develop Generalized Fringe-Retrieving A* (G-FRA*), an incremental A* variant for moving target search in known static terrains (see Section 1.1.2). G-FRA* runs up to one order of magnitude faster than A* when applied to moving target search in known static terrains.

  – Second, I develop Fringe-Retrieving A* (FRA*) (Sun, Yeoh, & Koenig, 2009), an incremental A* variant for moving target search in known static grids (see Section 1.1.2). FRA* optimizes G-FRA* to apply to moving target search in known static grids only. FRA* runs up to one order of magnitude faster than A* when applied to moving target search in known static grids.

  – Finally, I develop Moving Target D* Lite (MT-D* Lite) by combining the principles of G-FRA* and D* Lite. MT-D* Lite applies to moving target search in terrains where the action costs of the agent can increase and decrease between searches. Thus, MT-D* Lite applies to all three classes of terrains (see Section 1.1.2). MT-D* Lite runs faster than A* and GAA* by up to a factor of four for moving target search in known dynamic terrains.
I compare the runtime of the developed incremental search algorithms, discuss their strengths and weaknesses and provide guidelines for when to choose a particular algorithm over another. Table 1.3 provides a summary of the best algorithms in terms of runtime per search in all three classes of terrains (see Section 1.1.2): A* and GAA* can perform repeated searches either from the current state of the agent to the current state of the target, referred to as Forward Repeated A* and Forward GAA*, respectively, or from the current state of the target to the current state of the agent, referred to as Backward Repeated A* and Backward GAA*, respectively. FRA*, G-FRA* and MT-D* Lite can only perform forward but not backward searches. The numbers in square brackets below each incremental search algorithm are the smallest ratios of the runtime per search of repeated A* and the runtime per search of the incremental search algorithm. The runtime per search of repeated A* is the smaller runtime per search between Forward Repeated A* and Backward Repeated A*.

– For known static terrains, FRA* is the fastest algorithm on grids. G-FRA* runs slightly more slowly than FRA* on known static grids, and it is the second fastest algorithm. However, G-FRA* applies to arbitrary graphs and FRA* applies only to known static grids.

– For known dynamic terrains, if the h-values are well-informed, MT-D* Lite is the fastest algorithm. If the h-values are ill-informed:

  ◦ Backward GAA* is the fastest algorithm if the number of actions whose costs change is small.
Table 1.3: Fastest Incremental Search Algorithms for Different Terrains

- MT-D* Lite is the fastest algorithm if the number of actions whose costs change is medium.
- Forward Repeated A* and Backward Repeated A* are the fastest algorithms if the number of actions whose costs change is large.
  - For unknown static terrains, Forward GAA* is the fastest algorithm.

1.4 Dissertation Structure

This dissertation is structured as follows: In Chapter 2, I introduce background knowledge on path planning and incremental search algorithms. In Chapter 3, I introduce GAA*, a new heuristic learning incremental search algorithm. In Chapter 4, I introduce G-FRA*, FRA*, Basic MT-D* Lite and MT-D* Lite, new search tree transforming incremental search algorithms. In Chapter 5, I compare the new incremental search algorithms for moving target search to evaluate their runtime. In Chapter 6, I introduce the applications of the new incremental search algorithms. In Chapter 7, I provide a summary of my work.
Chapter 2: Background

In this chapter, I introduce background knowledge on search-based path planning for moving target search. The structure of this chapter is as follows: In Section 2.1, I introduce notation used in the rest of my dissertation. In Section 2.2, I describe representations of path planning problems that allow them to be solved with search algorithms. In Section 2.3, I describe A*, a popular search algorithm in artificial intelligence, based on which new incremental search algorithms for moving target search are developed. In Section 2.4, I describe existing search-based path planning algorithms. In Section 2.5, I describe evasion algorithms for the target in moving target search. In Section 2.6, I provide a summary of this chapter.

2.1 Notation

I use the following notation throughout this dissertation: $S$ denotes the finite set of states. $s_{start} \in S$ denotes the start state of a search, and $s_{goal} \in S$ denotes the goal state of a search. $\text{Succ}(s)$ denotes the finite set of successor states of $s \in S$. $\text{Pred}(s)$ denotes the finite set of predecessor states of $s \in S$. $A(s)$ denotes the finite set of actions that can be executed in $s \in S$. Executing $a \in A(s)$ in $s \in S$ results in a transition to
$\text{succ}(s,a) \in \text{Succ}(s)$. It is assumed that, for all $s \in S$, there exists only a unique action $a \in A(s)$ that transitions from $s \in S$ to $s' \in \text{Succ}(s)$, and $c(s,s') > 0$ denotes the action cost that transitions from $s \in S$ to $s' \in \text{Succ}(s)$.\footnote{It is trivial to extend the assumption to the situation where there exists more than one action that is able to transition from $s \in S$ to $s' \in \text{Succ}(s)$. In that situation, for each pair of $s \in S$ and $s' \in \text{Succ}(s)$, one can only use the action with the minimum cost that transitions from $s \in S$ to $s' \in \text{Succ}(s)$ at any point in time.} $\text{dist}^*(s_1,s_2)$ denotes the minimum cost of moving from $s_1 \in S$ to $s_2 \in S$. The state space consists of $S$ and all actions that can be executed in each $s \in S$. A path from $s_{\text{start}}$ to $s_{\text{goal}}$ consists of the sequence of actions $\{a_1,a_2,a_3,\ldots,a_{n-1}\}$ and the sequence of states $\{s_1 = s_{\text{start}}, s_2, s_3,\ldots, s_n = s_{\text{goal}}\}$ that satisfies $s_{i+1} = \text{succ}(s_i,a_i)$ ($1 \leq i < n$). The cost of the path is $\sum_{i=1}^{n-1} c(s_i, \text{succ}(s_i,a_i))$. A cost-minimal path is a path with the minimum cost among all paths from $s_{\text{start}}$ to $s_{\text{goal}}$.

2.2 State Space Representation

Path planning often needs to deal with continuous states and continuous actions (Likhachev, 2005; Ferguson, 2006). A common approach for path planning is to simplify these problems by discretizing these continuous elements.
2.2.1 Grids

Four-neighbor and eight-neighbor grids are commonly used in path planning for agent navigation (Ferguson, 2006; Likhachev, 2005; Koenig & Likhachev, 2002; Ishida & Korf, 1991; Koenig et al., 2007; Sun et al., 2008, 2009; Moldenhauer & Sturtevant, 2009b). Figure 2.1 shows an example of a four-neighbor grid and an eight-neighbor grid. An agent can always move from an unblocked cell to one of the four neighboring cells with action cost one (four-neighbor grid) or one of the eight neighboring cells with action cost one for orthogonal movements or $\sqrt{2}$ for diagonal movements (eight-neighbor grid) provided that the neighboring cell is unblocked. The undirected lines between $s$ and $s_i$ ($1 \leq i \leq 4$ in four-neighbor grids and $1 \leq i \leq 8$ in eight-neighbor grids) indicate that an agent can move from $s$ to $s_i$ and vice versa, if $s_i$ is unblocked. All other action costs are infinite.\footnote{In this dissertation, I regard actions with infinite costs as nonexistent, unless they become finite in known dynamic terrains, since these actions cannot be in any path of finite cost that can be followed by the agent. Similarly, I regard blocked states as nonexistent, unless they become unblocked in known dynamic terrains.}

Figure 2.2 shows an example path planning problem from a given start location to a given goal location. Figure 2.2(a) shows the real terrain: The start location is marked $S$, and the goal location is marked $G$. The black areas represent obstacles, which are blocked. The white area is unblocked (= traversable). I use the same convention in the rest of the dissertation, namely that, black cells are blocked and white cells are unblocked. Figure 2.2(b) shows a uniform discretization of the real terrain. Each cell that is either partially or completely occupied by obstacles is treated as blocked. This results in the (eight-neighbor) grid representation of the terrain shown in Figure 2.2(c). It is the user’s
choice whether the grid representation of the terrain is eight-neighbor or four-neighbor, which depends on the intended application.

The start and goal cells are the cells that contain the start and goal locations in the real terrain, respectively. Figure 2.2(d) shows that a state is assigned to each cell in Figure 2.2(c), with actions connecting each pair of neighboring states, referred to as the state space representation. The start and goal states are the states assigned to the start and goal cells, respectively. Then, one can search for a cost-minimal path from the start state to the goal state by applying search algorithms, such as A*, to the state space.

### 2.2.2 State Lattices

State lattices are extensions of grids that can model motion constraints (Pivtoraiko & Kelly, 2005b; Likhachev & Ferguson, 2009; Kushleyev & Likhachev, 2009) and are therefore well suited to path planning for non-holonomic and highly constrained robotic systems with limited maneuverability, such as unmanned ground vehicles (UGVs) (Likhachev
Figure 2.3: Example Feasible Motion Primitive

& Ferguson, 2009; Urmson et al., 2008). A state lattice is constructed by discretizing
the terrain into a multi-dimensional grid and connecting the cells of the grid with motion
primitives, which are the building blocks for more complicated motions. In this disserta-
tion, I use state lattices that are constructed from three-dimensional grids: A state in a
state lattice is a tuple \((x, y, \theta)\), where \(x\) and \(y\) together define the location of the center
of the UGV and \(\theta\) is its orientation. A motion primitive is feasible in a state iff the UGV
does not collide with obstacles when executing it in that state. Ideally, an action from
state \(u\) to state \(v\) in a state lattice exists iff there is a feasible motion primitive in \(u\) whose
execution results in \(v\). State lattices often include only a subset of actions to make path
planning fast (Howard & Kelly, 2007; Likhachev & Ferguson, 2009). Figure 2.3 shows a
feasible motion primitive in state \((2, D, 90^\circ)\), whose execution results in state \((10, C, 90^\circ)\),
where \(90^\circ\) indicates an east-facing orientation.

The black curve represents the path of the center of the UGV, and the blue rectangles
are the perimeters of the UGV as it executes the motion primitive.
2.3 A*

A* (Hart et al., 1968) is probably the most popular search algorithm in artificial intelligence. The objective of each search of A* is to find a cost-minimal path from the start state to the goal state.

2.3.1 Values

A* maintains four values for every state \( s \in S \):

- The \( h \)-value (heuristic value) \( h(s) \) of \( s \) estimates \( \text{dist}^*(s, s_{\text{goal}}) \), the minimum cost of moving from \( s \) to \( s_{\text{goal}} \) of a search. The \( h \)-values are \textit{admissible} with respect to \( s_{\text{goal}} \) iff \( 0 \leq h(s) \leq \text{dist}^*(s, s_{\text{goal}}) \) for all \( s \in S \). The \( h \)-values are \textit{consistent} with respect to \( s_{\text{goal}} \) iff \( h(s) \) satisfies the triangle inequality, namely \( h(s_{\text{goal}}) = 0 \) and \( 0 \leq h(s) \leq c(s, s') + h(s') \) for all \( s \in S \) and \( s' \in \text{Succ}(s) \) (Pearl, 1985). In this dissertation, I use a function \( H(s, s_{\text{goal}}) \) to calculate the user-provided \( h \)-value of \( s \) with respect to \( s_{\text{goal}} \). The user-provided \( h \)-values \( H(s, s_{\text{goal}}) \) of all states \( s \in S \) need to be consistent with respect to \( s_{\text{goal}} \), where \( s_{\text{goal}} \) can be any state in \( S \). The user-provided \( h \)-values \( H(s, s_{\text{goal}}) \) also need to satisfy \( H(s, s'') \leq H(s, s') + H(s', s'') \) for all states \( s, s', s'' \in S \), referred to as “H-value triangle inequality”.

In this dissertation, I use the Manhattan distances and the Octile distances (Bulitko & Lee, 2006) to the goal state as the user-provided \( h \)-values in four-neighbor and eight-neighbor grids, respectively. The Manhattan distance between a pair of states \((x_1, y_1)\) and \((x_2, y_2)\) is equal to \(|x_1 - x_2| + |y_1 - y_2|\), and the Octile distance between a pair of states \((x_1, y_1)\) and \((x_2, y_2)\) is equal to \( \sqrt{2} \times \min(|x_1 - x_2|, |y_1 - y_2|) + \).
\[||x_1 - x_2| - |y_1 - y_2||\] (Zhang, Sturtevant, Holte, Schaeffer, & Felner, 2009), which correspond to the minimum cost of moving from \((x_1, y_1)\) to \((x_2, y_2)\) when all cells are unblocked in four-neighbor and eight-neighbor grids, respectively. The Manhattan distances and the Octile distances are not only consistent with respect to any \(s_{goal}\) but also satisfy the H-value triangle inequality.

A* is an informed search algorithm, that is, A* uses the \(h\)-values to guide its search. For two consistent \(h\)-values \(h_1\) and \(h_2\) of A*, if \(h_1(s) \leq h_2(s)\) for all \(s \in S\), then \(h_2\) is no less informed than \(h_1\) (Pearl, 1985). \(h\)-values are called “well-informed” if they are close to \(dist^*(s, s_{goal})\), that is, are accurate estimates of \(dist^*(s, s_{goal})\), and “ill-informed” otherwise.

- The \(g\)-value \(g(s)\) is the minimum cost of moving from the start state to \(s\) found so far.

- The \(f\)-value \(f(s) := g(s) + h(s)\) is an estimate of the minimum cost of moving from the start state via \(s\) to the goal state.

- The parent pointer \(parent(s)\) points to one of the predecessor states \(s'\) of \(s\). \(s'\) is called the parent of \(s\). The parent pointers are used to extract the path after the search terminates.

### 2.3.2 OPEN and CLOSED Lists

A* maintains two data structures:

- The \textit{OPEN} list is a priority queue that contains all states to be considered for expansion. Initially, it contains the start state only.
function CalculateKey(s)
    return g(s) + h(s);

procedure InitializeState(s)
    if search(s) ≠ counter
        g(s) := ∞;
        h(s) := H(s, s_goal);
        search(s) := counter;
    end if

procedure UpdateState(s)
    if s ∈ OPEN
        OPEN.Update(s, CalculateKey(s));
    else
        OPEN.Insert(s, CalculateKey(s));
    end if

function ComputePath()
    while OPEN.TopKey() < CalculateKey(s_goal)
        s := OPEN.Pop();
        for all s′ ∈ Succ(s)
            InitializeState(s′);
            if g(s′) > g(s) + c(s, s′)
                g(s′) := g(s) + c(s, s′);
                parent(s′) := s;
                UpdateState(s′);
            end if
        end for
    end while
    return false;
end function

function Main()
    counter := 0;
    for all s ∈ S
        search(s) := 0;
        InitializeState(s);
    end for
    g(s_start) := 0;
    OPEN := {};
    OPEN.Insert(s_start, CalculateKey(s_start));
    if ComputePath() = false
        return false; /* No path found */
    end if
    return true;
end function

Figure 2.4: Pseudo Code of A*

- The CLOSED list is a set that contains all states that have been deleted from the OPEN list. Initially, it is empty.3

3Although the pseudocode of A* in Figure 2.4 does not maintain a CLOSED list, we can add “CLOSED := ∅” between Line 33 and Line 34 to initialize the CLOSED list, and add “CLOSED.Insert(s)” between Line 15 and Line 16 to insert s into the CLOSED list, in order to maintain a CLOSED list explicitly.
2.3.3 Operations

Figure 2.4 gives the pseudo code of a version of A* for only one search. For ease of description of how A* can be applied to moving target search later in this dissertation where multiple searches might be performed, I use a variable \( \text{counter} \) to indicate how many searches have been performed so far, including the current search. Since this is the first search, \( \text{counter} \) is set to one (Lines 26 and 29). I use a variable \( \text{search}(s) \) for all \( s \in S \) to indicate whether \( g(s) \) and \( h(s) \) have been initialized in the \( \text{counter} \)-th search. \( g(s) \) and \( h(s) \) have been initialized in the \( \text{counter} \)-th search iff \( \text{counter} = \text{search}(s) \). Initially, \( \text{search}(s) = 0 \) for all \( s \in S \), which indicates that none of the states’ \( g \)- and \( h \)-values have been initialized (Lines 27 - 28). A* then initializes the \( g \)- and \( h \)-values of \( s_{\text{start}} \) and \( s_{\text{goal}} \) (Lines 30 - 31) and initializes the OPEN list to contain \( s_{\text{start}} \) only (Lines 33 - 34). The main routine of A* is performed in ComputePath() (Lines 13 - 24). One execution of ComputePath() is called one search.

A* repeats the following procedure in ComputePath(): It deletes a state \( s \) with the smallest \( f \)-value from the OPEN list (Line 15) and expands \( s \) by performing the following operations for each successor state \( s' \in \text{Succ}(s) \) (Lines 17 - 21). If \( g(s') \) and \( h(s') \) has not yet been initialized (\( \text{search}(s') \neq \text{counter} \)), A* initializes them in InitializeState(\( s' \)) (Line 17) by assigning \( g(s') := \infty \) and calculating the user-provided \( h \)-value \( h(s') := H(s', s_{\text{goal}}) \) (Lines 05 - 06). A* assigns \( \text{search}(s') := \text{counter} \) (Line 07) each time InitializeState(\( s' \))

---

\(^4\)In this dissertation, all pseudo codes use the following functions to manage the OPEN list unless otherwise specified: The keys of all states in the OPEN list are their \( f \)-values. CalculateKey(\( s \)) calculates the key of \( s \). OPEN.Top() returns a state with the smallest key of all states in the OPEN list. OPEN.Pop() returns a state with the smallest key of all states in the OPEN list and removes the state from the OPEN list. OPEN.TopKey() returns the smallest key of all states in the OPEN list. If the OPEN list is empty, then OPEN.TopKey() returns \( \infty \). OPEN.Insert(\( s, k \)) inserts state \( s \) into the OPEN list with key \( k \). OPEN.Update(\( s, k \)) changes the key of state \( s \) in the OPEN list to \( k \). OPEN.Delete(\( s \)) deletes state \( s \) from the OPEN list.
is called to indicate that $g(s')$ and $h(s')$ have been initialized in the counter-th search. If $g(s')$ is larger than $g(s) + c(s, s')$, then A* generates $s'$ by performing the following three steps: (1) Assigning $g(s') := g(s) + c(s, s')$ (Line 19); (2) Setting the parent pointer of state $s'$ to state $s$ (Line 20); and (3) Updating the key of $s'$ to reflect its current $f$-value if $s'$ is in the OPEN list or inserting $s'$ into the OPEN list with its current $f$-value as its key (Line 21). A* terminates when its OPEN list is empty (Lines 22 - 23) or when the smallest $f$-value of all states in the OPEN list is no smaller than the $f$-value of $s_{goal}$ (Line 24). The former condition indicates that no path exists from $s_{start}$ to $s_{goal}$, and the latter condition indicates that A* found a cost-minimal path from $s_{start}$ to $s_{goal}$.

The principle behind A* is that, when examining whether a state $s$ should be expanded next, A* considers not only the cost of the path from $s_{start}$ to $s$ found so far ($= g(s)$), but also the estimated cost of the cost-minimal path from $s$ to $s_{goal}$ ($= h(s)$). The sum of $g(s)$ and $h(s)$ ($= f(s)$) approximates the cost of a cost-minimal path from $s_{start}$ to $s_{goal}$ via $s$, which provides more accurate guidance than only considering $g(s)$ when A* picks the most promising state to expand.

### 2.3.4 Properties

A* has the following properties when using consistent $h$-values (Pearl, 1985), that will be used in this dissertation:

- **A* Property 1**: During a search, every state can be expanded at most once.

- **A* Property 2**: During a search, A* expands no more states than an otherwise identical version of A* (with the same tie-breaking strategy) for the same search
problem if the $h$-values used by the first A* search are no less informed than the $h$-values used by the latter A* search.

- **A* Property 3:** During a search, the $f$-values of the series of expanded states over time are monotonically non-decreasing. Thus, $f(s) \leq f(s_{\text{goal}})$ for all states $s \in S$ that were expanded during an A* search, and $f(s_{\text{goal}}) \leq f(s)$ for all $s \in S$ that were generated but remained unexpanded (= remained in the OPEN list) when the A* search terminates.

- **A* Property 4:** During a search, every expanded state $s$ (= every state in the CLOSED list) satisfies the following conditions: (a) If $s \neq s_{\text{start}}$, then the parent of $s$ has also been expanded and $g(s) = g(\text{parent}(s)) + c(\text{parent}(s), s)$. (b) The $g$-value of $s$ satisfies $g(s) = g(s_{\text{start}}) + \text{dist}^*(s_{\text{start}}, s)$. Thus, the difference between $g(s)$ of every expanded state $s$ and $g(s_{\text{start}})$ is equal to the cost of a cost-minimal path from $s_{\text{start}}$ to $s$.

- **A* Property 5:** During a search, a path from $s_{\text{start}}$ to any generated state $s \in S$ can be identified in reverse by repeatedly following the parent pointers from $s$ to $s_{\text{start}}$. This path is a cost-minimal path from $s$ to $s_{\text{start}}$ if $s$ has been expanded in that search. The search tree of an A* search consists of $s_{\text{start}}$ and all generated states with their parent pointers pointing to their parents. The root of the search tree is $s_{\text{start}}$. The subtree (of the search tree of an A* search) rooted in $s$ consists of (1) $s$, (2) every generated state $s' \in S$ ($s' \neq s$) with a path to $s$, which can be identified in reverse by repeatedly following the parent pointers from $s'$ to $s$, and
(3) the parent pointer of state $s'$. A* Property 5 implies that the states in the CLOSED list form a contiguous area in grids (Sun & Koenig, 2007).

- **A* Property 6:** During a search, the OPEN list contains the following states: If the CLOSED list is empty, then the OPEN list contains only $s_{start}$. Otherwise, the OPEN list contains exactly all states that are not in the CLOSED list but have at least one predecessor state in the CLOSED list. Every state $s$ ($s \neq s_{start}$) in the OPEN list satisfies the following conditions: (a) The parent of $s$ is the state $s'$ in the CLOSED list that minimizes $g(s') + c(s', s)$. (b) The $g$-value of $s$ satisfies $g(s) = g(parent(s)) + c(parent(s), s)$.

- **A* Property 7:** A* terminates. If A* terminates because its OPEN list is empty, then no path exists from the start state to the goal state. Otherwise, A* terminates when the smallest $f$-value of all states in the OPEN list is no smaller than the $f$-value of the goal state. One can then identify a cost-minimal path from the start state to the goal state in reverse by repeatedly following the parent pointers from the goal state to the start state.

2.3.5 Example

```
g
f
```

```
h search
```

Figure 2.5: Legend of Figure 2.6
Figure 2.6: Example Trace of A*

Figure 2.6 gives an example of A* in a four-neighbor grid. In this example, A* only performs one search. The start state is D2 (marked S) and the goal state is C4 (marked G). All states have their search-value in the lower right corner. States that have been initialized in InitializeState() also have their g-values in the upper left corner, h-values in the lower left corner, and f-values in the upper right corner (see Figure 2.5). The outgoing arrow from a state is the parent pointer of the state and points to its parent. A time step is one execution of Lines 15 - 21 in Figure 2.4. The state that is being expanded in each
<table>
<thead>
<tr>
<th>Time Step</th>
<th>Expanded States</th>
<th>States in the OPEN List</th>
<th>State being Expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Step 0</td>
<td>∅</td>
<td>D2</td>
<td>NULL</td>
</tr>
<tr>
<td>Time Step 1</td>
<td>∅</td>
<td>D1</td>
<td>D2</td>
</tr>
<tr>
<td>Time Step 2</td>
<td>D2</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>Time Step 3</td>
<td>D1,D2</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>Time Step 4</td>
<td>D1,D2,C1</td>
<td>A1,B2</td>
<td>B1</td>
</tr>
<tr>
<td>Time Step 5</td>
<td>D1,D2,C1,B1</td>
<td>A1,A2,B3</td>
<td>B2</td>
</tr>
<tr>
<td>Time Step 6</td>
<td>D1,D2,C1,B1,B2</td>
<td>A1,A2,A3,B4,C3</td>
<td>B3</td>
</tr>
<tr>
<td>Time Step 7</td>
<td>D1,D2,C1,B1,B2,B3</td>
<td>A1,A2,A3,A4,C3,C4</td>
<td>B4</td>
</tr>
<tr>
<td>Time Step 8</td>
<td>D1,D2,C1,B1,B2,B3,B4</td>
<td>A1,A2,A3,A4,C3,C4</td>
<td>Terminate</td>
</tr>
</tbody>
</table>

Table 2.1: Operations of A*

time step is highlighted with a bold frame. The states that have been expanded before each time step are shaded grey. A* breaks ties among states with the same $f$-values in favor of states with larger $g$-values, which is known to be a good tie-breaking strategy.

The operations of A* and the corresponding states in the OPEN list, expanded states, and the state being expanded in each time step are given in Table 2.1. After the search, one can identify a cost-minimal path from the start state to the goal state in reverse by repeatedly following the parent pointers from C4 to D2.

2.3.6 Applying A* to Moving Target Search

A* is able to find a cost-minimal path from the start state to the goal state in a search. In this section, I introduce how to apply A* to moving target search, where multiple searches can be performed.

2.3.6.1 Search Directions

When applying A* to moving target search, A* can perform a search in two different directions:
• A* performs a search from the agent to the target by assigning the current states of the agent and target to the start and goal states of the search, respectively, referred to as Forward A*.

• A* performs a search from the target to the agent by assigning the current states of the agent and target to the goal and start states of the search, respectively, referred to as Backward A*.

In this dissertation, “forward search” refers to a search that assigns the current states of the agent and target to the start and goal states, respectively, and “backward search” refers to a search that assigns the current states of the agent and target to the goal and start states, respectively.

### 2.3.6.2 Repeated A*

One can apply Forward A* (by using assumptive planning introduced in Section 1.1.1) to moving target search by assigning the current states of the agent and target to the start and goal states for each search, respectively, and repeatedly perform searches to find paths for the agent, referred to as Forward Repeated A*. One can also apply Backward A* to moving target search by assigning the current states of the agent and the target to the goal and start states for each search, respectively, and repeatedly perform searches to find paths for the agent, referred to as Backward Repeated A*. In this section, I only introduce Forward Repeated A*, since it is trivial to switch the search direction to backward search, resulting in Backward Repeated A*. Forward Repeated A* is guaranteed to find a cost-minimal path from the start state to the goal state for each search no matter how the
function CalculateKey(s):
    return g(s) + h(s);

procedure InitializeState(s):
    if search(s) ≠ counter
        g(s) := ∞;
        h(s) := H(s, sgoal);
        search(s) := counter;
    
procedure UpdateState(s):
    if s ∈ OPEN
        OPEN.Update(s, CalculateKey(s));
    else
        OPEN.Insert(s, CalculateKey(s));

function ComputePath():
    while OPEN.TopKey() < CalculateKey(sgoal)
        s := OPEN.Pop();
        for all s' ∈ Succ(s)
            InitializeState(s');
            if g(s') > g(s) + c(s, s')
                g(s') := g(s) + c(s, s');
                parent(s') := s;
                UpdateState(s');
        if OPEN = ∅
            return false;
    return true;

function Main():
    counter := 0;
    sstart := the current state of the agent;
    sgoal := the current state of the target;
    for all s ∈ S
        search(s) := 0;
    counter := counter + 1;
    InitializeState(sstart);
    InitializeState(sgoal);
    g(sstart) := 0;
    OPEN := ∅;
    OPEN.Insert(sstart, CalculateKey(sstart));
    if ComputePath() = true
        while target not caught AND target on path from sstart to sgoal AND action costs do not change
            agent follows the path from sstart to sgoal;
            if agent caught target
                return true;
            else
                wait until some action costs decrease;
                sgoal := the current state of the target;
                update the action costs (if any);
        
Figure 2.7: Pseudo Code of Forward Repeated A*

The action costs of the agent change and the target moves between searches, since it performs Forward A* for each search from scratch.

Figure 2.7 gives the pseudo code of one version of Forward Repeated A* that extends the pseudo code of Forward A* in Figure 2.4, which applies to all terrains introduced in
Section 1.1.2. This pseudo code can be optimized when applied to unknown static terrains, where the action costs are non-decreasing between searches: We can replace Line 39 with “while target not caught AND target on path from $s_{\text{start}}$ to $s_{\text{goal}}$ AND action costs do not increase between the agent and the target along the path”, and replace Line 45 with “return false;”. For ease of description of the algorithms, I model the problem in a way that the agent and the target take turns moving at most one step at a time rather than moving simultaneously.

The main routine of Forward Repeated A* is performed in ComputePath() (Lines 13 - 24), which remains unchanged. The main change of Forward Repeated A* over Forward A* is that multiple searches can be performed in the while loop on Line 31 in order to repeatedly find new cost-minimal paths from the current state of the agent to the current state of the target: After each search, if a path is found (Line 38), then the agent moves along the path to catch the target until either the target is caught or the current state of the target is no longer on the path or the action costs of the agent change (Lines 39 - 43). If no path exists from the current state of the agent to the current state of the target, the agent does not move, but waits until some action costs decrease upon which there might exist a path from the current state of the agent to the current state of the target (Lines 44 - 45).\(^5\) The agent then updates the current states of the agent and target (Lines 43 and 46) and the action costs of the agent (if they changed) (Line 47). The agent then performs the next search to find a path from the current state of the agent to the current state of the target given the current action costs of the agent. Forward Repeated A* terminates when the agent caught the target (Line 42).

\(^5\)In unknown static terrains, where the action costs are non-decreasing, Forward Repeated A* can terminate immediately and determine that the target cannot be caught if no path is found after a search.
I use an example problem of moving target search in unknown static terrains to
demonstrate the operations of Forward Repeated A*. In the rest of the dissertation, all
example problems on grids are given on four-neighbor grids. Figure 2.9(a) illustrates
the real terrain: The current states of the agent and the target are D2 (marked S) and
C4 (marked G), respectively. The sensor onboard the agent can observe only the four
neighboring states of the current state of the agent. Thus, initially, the agent only knows
that C2 and D3 are blocked, and that D1 and D2 are unblocked.

Figures 2.9(b) - (d) illustrate the agent navigation problem solved with Forward Repeated A*. All states have their search-value in the lower right corner. States that have
been initialized in InitializeState() have their h-values in the lower left corner, g-values
in the upper left corner and f-values in the upper right corner (shown in Figure 2.8).
Expanded states during each search are shaded grey. Figure 2.9(b) illustrates the first
search. The agent then moves along the path found by the first search. Figure 2.9(c)

![Legend of Figure 2.9](image)

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>9</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 2.8: Legend of Figure 2.9

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 2.9: Example Trace of Forward Repeated A*

Expanded states during each search are shaded grey. Figure 2.9(b) illustrates the first
search. The agent then moves along the path found by the first search. Figure 2.9(c)
illustrates that, after the first search, the agent moves along the path to B1 where it observes that B2 blocks its path. In the meantime, the target moves to C3 which is off the path found by the first search. The agent then performs the second search to plan a new path from its current state B1 to the current state of the target C3. Figure 2.9(d) illustrates the second search. The agent then moves along this path until it arrives at C3 and catches the target.

Forward and Backward Repeated A* can be applied to moving target search. However, they may not be fast enough to satisfy the runtime requirements of certain applications. For example, Repeated A* is not fast enough to satisfy the runtime requirements of the ROMAN Tutoring System (Belghith et al., 2010), an automatic intelligent tutoring system to train the astronauts to operate the Space Station Remote Manipulator System.

2.4 Repeated A*-based Path Planning Algorithms for Agent Navigation

Repeated A* has been extended in various directions to speed up its search such that it can be applied to different applications of agent navigation (Korf, 1990; Koenig & Likhachev, 2002; Stentz, 1995; Ferguson & Stentz, 2005a; Likhachev, Ferguson, Gordon, Stentz, & Thrun, 2007; Hernández, Meseguer, Sun, & Koenig, 2009).

2.4.1 Real-time Search

Real-time search algorithms (Korf, 1990; Hernández & Meseguer, 2005; Koenig & Likhachev, 2006; Bulitko & Lee, 2006; Furcy & Koenig, 2000; Bond, Widger, Ruml,
& Sun, 2010; Sturtevant & Bulitko, 2011; Hernández & Meseguer, 2005, 2007b, 2007a; Furcy & Koenig, 2000; Shue & Zamani, 1993) have been used for path planning for agent navigation in terrains where the action costs of the agent do not decrease between searches (by using assumptive planning introduced in Section 1.1.1). Real-time search algorithms guarantee a constant search time per action execution (referred to as the “real-time property”).

LRTA* (Korf, 1990) is a real-time search algorithm designed for stationary target search. LRTA* interleaves searches with action executions, potentially at the expense of increasing the number of action executions. LRTA* finds the beginning (= prefix) of a complete path from the current state of the agent to the current state of the target by restricting the search to the states around the current state of the agent with the current information about the terrains, and updates the $h$-value of the current state of the agent to make it more informed after each step. The agent then moves along that path until it reaches the end of the path or observes that the action costs of the agent along the path have increased (for example, the path is blocked by newly observed blocked states). If the current state of the agent is different from the current state of the target, then LRTA* repeats the process. Otherwise, LRTA* terminates successfully. MTS (Ishida & Korf, 1991; Ishida, 1992) extends LRTA* to moving target search. MTS updates $h$-values from the current state of the agent to the current state of the target after each search, and guarantees a constant search time per action execution. Trailblazer (Chimura & Tokoro, 1994) extends MTS to reduce the number of searches needed to catch the target by sacrificing MTS’s real-time property in some situations: Trailblazer stores the trajectories of both the agent and the target in a graph $G$. Before their trajectories
intersect, Trailblazer performs the search in the same way as MTS. When their trajectories intersect, MTS runs Dijkstra’s algorithm (Dijkstra, 1959) on graph $G$ to find a path from the current state of the agent to the current state of the target. Since Dijkstra’s algorithm cannot guarantee a constant search time per action execution, Trailblazer loses its real-time property in this situation. Updating Shortest Paths (USP) (Edelkamp, 1998) is an incremental version of Trailblazer. USP corrects only those $g$-values of the states in $G$ that are not the minimum costs moving from the start state to these states after each search. Thus, USP can run faster than Trailblazer that runs Dijkstra’s algorithm from scratch on graph $G$.\(^6\) A longer overview of real-time search algorithms is given in (Ishida, 1997).

The advantage of real-time search algorithms is that they can guarantee a constant search time per action execution. Unfortunately, the quality of the solution of real-time search algorithms, that is, the total action cost of the agent from the moment it starts navigation until it stops navigation can be quite poor due to the fact that they do not find a complete path from the current state of the agent to the current state of the target in each search and often move back and forth in order to be able to learn the $h$-values necessary to move to the current state of the target. Thus, in order to address the disadvantages of real-time search algorithms, I develop new incremental search algorithms in this dissertation to speed up search-based path planning algorithms that find complete paths from the current state of agent to the current state of the target in each search. I

\(^6\)Trailblazer and USP cannot guarantee a constant search time per action execution in all situations (for example, when the trajectory of the agent intersects the trajectory of the target). However, I still list them as (partial) real-time search algorithms as they are based on the principle of MTS and can provide the real-time property in some situations.
demonstrate that the newly developed algorithms can satisfy the runtime requirements of some applications of moving target search, such as computer games.

### 2.4.2 Incremental Search

Path planning with assumptive planning is often a repeated search process (as discussed in Section 1.1.1) where an agent needs to repeatedly find new paths for a series of similar search problems as the action costs of the agent change between searches. Fortunately, in many path planning applications, the number of actions whose costs change between searches is small (Koenig, Likhachev, & Furcy, 2004a). This suggests that complete recomputations of the paths from scratch can be wasteful and inefficient. Incremental search algorithms reuse information from previous searches to speed up the current search and hence are often able to find cost-minimal paths for series of similar search problems faster than is possible by solving each search problem from scratch. Incremental search algorithms are particularly well-suited for large terrains with slight changes over time, as there is usually a large amount of information that can be reused between searches in such cases (Koenig, Likhachev, Liu, & Furcy, 2004b).

The idea of incremental search was introduced a few decades ago. For example, Deo and Pang wrote an overview article on cost-minimal path algorithms (Deo & Pang, 1984), which already cited several incremental search algorithms, including several ones published in the late 1960s (Koenig et al., 2004b). Since then, additional incremental search algorithms have been introduced in the literature (Stentz, 1994, 1995; Barbehenn & Hutchinson, 1995; Ramalingam & Reps, 1996; Frigioni, Marchetti-Spaccamela, & Nanni, 1998). Recent developments of incremental search algorithms have focused on replanning
with A*, referred to as incremental heuristic search algorithms, which have been used as path planning algorithms for a wide range of robotic systems (Likhachev et al., 2005a; Urmson et al., 2008; Likhachev & Ferguson, 2009; Likhachev et al., 2007, 2005b; Ferguson & Stentz, 2005b, 2005a). Existing incremental heuristic search algorithms generally fall into two classes:

- **Class 1:** Heuristic learning incremental search algorithms. The first class of algorithms updates the $h$-values for the current search by using information from previous searches to make them more informed and thus future searches more focused, referred to as heuristic learning incremental search algorithms. For example, Adaptive A* (Koenig & Likhachev, 2005) and Moving Target Adaptive A* (MT-Adaptive A*) (Koenig et al., 2007) belong to this class. Adaptive A* is simple to understand and easy to implement. However, Adaptive A* has restrictions on its applicability: Adaptive A* finds cost-minimal paths in terrains where (1) the goal state does not change between searches, and (2) the action costs of the agent do not decrease between searches. It cannot guarantee to find cost-minimal paths in other situations. MT-Adaptive A* extends Adaptive A* to find cost-minimal paths in terrains where both the start and goal states can change between searches. However, MT-Adaptive A* still cannot guarantee to find cost-minimal paths in terrains where the action costs of the agent decrease between searches.

- **Class 2:** Search tree transforming incremental search algorithms. The second class of algorithms transforms the search tree of the previous search to the search tree of the current search such that the current search can start with the transformed
search tree instead of starting from scratch, referred to as search tree transforming incremental search algorithms. For example, DynamicSWSF-FP (Ramalingam & Reps, 1996), Differential A* (Trovato, 1990), Dynamic A* (D*) (Stentz, 1995), Lifelong Planning A* (LPA*) (Koenig et al., 2004a) and its generalized version D* Lite (Koenig & Likhachev, 2002) belong to this class. D* Lite is one of the state-of-the-art incremental search algorithms that has been extended and applied to many applications (Ferguson & Stentz, 2005b; Likhachev et al., 2005a, 2005b). However, D* Lite also has restrictions on its applicability: D* Lite can find cost-minimal paths in terrains where the goal state can change between searches but the start state does not change between searches. It is still unknown how to apply D* Lite to search problems where both the start and goal states can change between searches and make it run faster than A*.

Incremental search algorithms have the following properties:

- First, like Repeated A*, incremental search algorithms find paths with a memory requirement that is in linear in the number of states in the state space, which can be satisfied by many applications. For example, Repeated A* variants have been implemented in computer games (Rabin, 2002), and incremental search algorithms have been implemented in robotic systems (Likhachev, 2005; Ferguson & Stentz, 2005c). It has been demonstrated that incremental search algorithms can solve agent navigation problems in large state spaces (Likhachev & Ferguson, 2009). Thus, in this dissertation, I focus mainly on the runtimes of the developed incremental search algorithms.
Second, incremental search algorithms are runtime-efficient. Incremental search algorithms can reuse information from previous searches to speed up the current search, and can find cost-minimal paths for a series of similar search problems faster than by solving each search problem from scratch (Koenig et al., 2004b). For example, D* Lite can achieve a speedup of one to two orders of magnitude over repeated A* searches for stationary target search (Koenig & Likhachev, 2002).

Third, like Repeated A*, incremental search algorithms guarantee that the path returned by each search is a cost-minimal path based on the information currently available to the agent (Koenig et al., 2004a; Koenig & Likhachev, 2005; Sun et al., 2009), no matter how the state space changes.

I now give an overview of three popular incremental search algorithms: namely Adaptive A* (Koenig & Likhachev, 2005), D* Lite (Koenig & Likhachev, 2002) and Differential A* (Trovato, 1990). I give more details on Adaptive A* and D* Lite than on Differential A* because I develop new incremental search algorithms for moving target search based on Adaptive A* and D* Lite. A longer overview of incremental search has been given in (Koenig et al., 2004b).

2.4.2.1 Adaptive A*

Adaptive A* (Koenig & Likhachev, 2005) is a heuristic learning incremental search algorithm designed for stationary target search in unknown static terrains. It needs consistent h-values with respect to the goal state of each search to guide its search. Adaptive A* performs Forward Repeated A* to repeatedly find cost-minimal paths from the current
state of the agent to the current state of the target by assigning the current states of
the agent and the target to the start and goal states of each search, respectively. Similar
to Forward Repeated A*, Adaptive A* constructs its search tree for each search from
scratch. Different from Forward Repeated A*, Adaptive A* updates its $h$-values between
searches to make them more informed and thus future searches more focused.

Adaptive A* can only perform forward (but not backward) searches to find cost-
minimal paths between the current states of the agent and the target. This is so because
the $h$-value of a state is an estimate of the minimum cost of moving from the state
to the goal state of a search. After each search, the $h$-values of all expanded states
are updated with respect to the goal state of the previous search (= the $h$-values are
updated to estimate the minimum cost of moving from the state to the goal state of the
previous search). Since the target is stationary, the goal state does not change between
searches if forward searches are performed. Thus, the updated $h$-values remain consistent
with respect to the goal state. However, the goal state is the current state of the agent
if backward searches are performed. Thus, the $h$-values can become inadmissible and
inconsistent with respect to the goal state when the agent moves. Therefore, Adaptive
A* can have different start states but has to have the same goal state. In addition,
Adaptive A* is guaranteed to find cost-minimal paths only in terrains where the action
costs of the agent do not decrease between searches. Thus, Adaptive A* only applies to
known static terrains and unknown static terrains (see Section 1.1.2).

I first review an eager version of Adaptive A* that is easier to understand, referred
to as Eager Adaptive A* (Koenig & Likhachev, 2005). I then review a more complicated
function CalculateKey(s)
return g(s) + h(s);

procedure InitializeState(s)
if search(s) = 0
    g(s) := ∞;
    h(s) := H(s, s_{goal});
else if search(s) ≠ counter
    g(s) := ∞;
    search(s) := counter;

procedure UpdateState(s)
if s ∈ OPEN
    OPEN.Update(s, CalculateKey(s));
else
    OPEN.Insert(s, CalculateKey(s));

function ComputePath()
while OPEN.TopKey() < CalculateKey(s_{goal})
    s := OPEN.Pop();
    CLOSED.Insert(s);
    for all s' ∈ Succ(s)
        InitializeState(s');
        if g(s') > g(s) + c(s, s')
            g(s') := g(s) + c(s, s');
            parent(s') := s;
            UpdateState(s');
    if OPEN = ∅
        return false;
    return true;

function Main()
counter := 0;
s_{start} := the current state of the agent;
s_{goal} := the current state of the target;
for all s ∈ S
    search(s) := 0;
while s_{start} ≠ s_{goal}
    counter := counter + 1;
    InitializeState(s_{start});
    InitializeState(s_{goal});
    g(s_{start}) := 0;
    OPEN := CLOSED := ∅;
    OPEN.Insert(s_{start}, CalculateKey(s_{start}));
    if ComputePath() = false
        return false; /* Target cannot be caught*/
    for all s ∈ CLOSED
        h(s) := g(s_{goal}) - g(s);
    while target not caught AND action costs on path do not increase
        agent follows the path from s_{start} to s_{goal};
    if agent caught target
        return true;
    s_{start} := the current state of the agent;
    update the increased action costs (if any);
return true;

Figure 2.10: Pseudo Code of Eager Adaptive A*
version, referred to as Lazy Adaptive A* (Koenig & Likhachev, 2005), that is more efficient than the eager version.
**Eager Adaptive A**

As discussed earlier, Eager Adaptive A* can only perform forward searches, and it only applies to stationary target search in known static terrains and unknown static terrains (see Section 1.1.2), where the action costs of the agent do not decrease between searches.

The principle behind Eager Adaptive A* is the following: Let \( s \) denote any expanded state by an A* search. Then, \( g(s) \) is equal to \( \text{dist}^*(s_{\text{start}}, s) \) since \( s \) was expanded by the search (see A* Property 4 in Section 2.3.4). Similarly, \( g(s_{\text{goal}}) \) is equal to \( \text{dist}^*(s_{\text{start}}, s_{\text{goal}}) \). \( \text{dist}^*(s_{\text{start}}, s), \text{dist}^*(s, s_{\text{goal}}) \) and \( \text{dist}^*(s_{\text{start}}, s_{\text{goal}}) \) satisfy the triangle inequality:

\[
\text{dist}^*(s_{\text{start}}, s_{\text{goal}}) \leq \text{dist}^*(s_{\text{start}}, s) + \text{dist}^*(s, s_{\text{goal}}) \\
\text{dist}^*(s_{\text{start}}, s_{\text{goal}}) - \text{dist}^*(s_{\text{start}}, s) \leq \text{dist}^*(s, s_{\text{goal}}) \\
g(s_{\text{goal}}) - g(s) \leq \text{dist}^*(s, s_{\text{goal}}). 
\]

Thus, \( g(s_{\text{goal}}) - g(s) \) is an admissible estimate of the minimum cost of moving from \( s \) to \( s_{\text{goal}} \) that can be calculated quickly. It can thus be used as a new admissible \( h \)-value of \( s \). Adaptive A* therefore updates the \( h \)-values by assigning

\[
h(s) := g(s_{\text{goal}}) - g(s) 
\tag{2.1}
\]

for all expanded states \( s \) by the search. Let \( h'(s) \) denote the \( h \)-values after the updates. The updated \( h \)-values \( h'(s) \) are not only admissible but also consistent with respect to
s_{goal} (Koenig & Likhachev, 2005). Furthermore, since s was expanded by the search, it holds that \( f(s) \leq f(s_{goal}) \) (see A* Property 3 in Section 2.3.4), and \( f(s_{goal}) = g(s_{goal}) \). Thus,

\[
\begin{align*}
g(s) + h(s) & \leq g(s_{goal}) \\
h(s) & \leq g(s_{goal}) - g(s) \\
h(s) & \leq h'(s)
\end{align*}
\]

Thus, the updated \( h \)-values \( h'(s) \) of all expanded states \( s \) are no smaller than the immediately preceding \( h \)-values \( h(s) \). This property implies that the \( h \)-values \( h'(s) \) are no less informed than the immediately preceding \( h \)-values \( h(s) \) and thus also all previous \( h \)-values, including the user-provided \( h \)-values. Consequently, a search with the \( h \)-values \( h'(s) \) cannot expand more states than an A* search with the user-provided \( h \)-values with the same tie-breaking strategy (see A* Property 2 in Section 2.3.4). It therefore cannot be slower (except possibly for the small amount of runtime needed by the bookkeeping and \( h \)-value update operations) than an A* search with the user-provided \( h \)-values. In this sense, Eager Adaptive A* is guaranteed to reduce its runtime per search more and more over time, and its runtime per search cannot be larger than that of a search that does not modify the user-provided \( h \)-values. This principle was used in (Holte, Mkadmi, & Macdonald, 1996) and later resulted in the independent development of Eager Adaptive A* (Koenig & Likhachev, 2005). Eager Adaptive A* does not update the \( h \)-values of the states \( s \) that remained unexpanded, because \( f(s) \leq f(s_{goal}) \) holds for all expanded states.
(see A* Property 3 in Section 2.3.4) but not necessarily for an unexpanded state. Thus, Inequality (2.2) is not guaranteed to hold for unexpanded states \( s \), and \( g(s_{goal}) - g(s) \) can be smaller than the immediately preceding \( h \)-value \( h(s) \). Thus, updating these \( h \)-values could decrease them, and the argument above thus does not hold for them.

Figure 2.10 gives the pseudo code of Eager Adaptive A*,\(^7\) which extends the pseudo code of Forward Repeated A* in Figure 2.7. Eager Adaptive A* applies only to terrains where the action costs of the agent do not decrease between searches. Thus, Eager Adaptive A* can terminate when it finds that no path exists between the start state and the goal state after a search (Line 42), since it is guaranteed that Eager Adaptive A* will not be able to find any path in future searches. Similar to Forward Repeated A*, Eager Adaptive A* repeatedly performs Forward A* searches in ComputePath() (Lines 15 - 27). InitializeState(\( s \)) is executed when (1) the \( g \)-value and \( h \)-value of \( s \) are needed (Line 20) during each search and (2) \( s_{start} \) and \( s_{goal} \) are initialized before each search (Lines 36 - 37). It initializes the \( h \)-value of \( s \) to its user-provided \( h \)-value iff \( s \) has not yet been initialized by InitializeState(\( s \)) in any search \( (search(s) = 0) \) (Lines 04 and 06) and initializes the \( g \)-value of \( s \) to infinity iff \( s \) has not been initialized by InitializeState(\( s \)) during the \( counter \)-th search \( (= \) the current search) (Lines 05 and 08). \( search(s) \) is set to \( counter \) each time InitializeState(\( s \)) is executed to indicate that the \( g \)-values of \( s \) have been initialized in the \( counter \)-th search and should not be re-initialized in the same search (Line 09).

\(^7\)In this dissertation, all pseudo codes use the following functions to manage the \texttt{CLOSED} list unless otherwise specified: \texttt{CLOSED.Insert(}\( s \)\texttt{)} inserts state \( s \) into the \texttt{CLOSED} list. \texttt{CLOSED.Delete(}\( s \)\texttt{)} deletes state \( s \) from the \texttt{CLOSED} list.
The main difference between Forward Repeated A* and Eager Adaptive A* is as follows: After each search, Eager Adaptive A* updates the $h$-values of all expanded states in that search by executing Assignment (2.1) (Lines 43 - 44). Eager Adaptive A* has to update the $h$-values after the search rather than during the search because it has to know the value of $g(s_{\text{goal}})$ after the search. In order to store all expanded states in each search so that their $h$-values can be updated after the search, Eager Adaptive A* maintains a $CLOSED$ list that stores all expanded states in each search (Lines 18 and 39).

I use the same example problem of agent navigation in unknown static terrains shown in Figure 2.9, except that the target does not move between searches, to demonstrate the operations of Eager Adaptive A*. Figures 2.12 and 2.13 illustrate the same agent navigation problem solved by Forward Repeated A* and Eager Adaptive A*, respectively. All states have their $search$-value in the lower right corner. States that have been initialized in InitializeState() have their $h$-values in the lower left corner, $g$-values in the upper left corner and $f$-values in the upper right corner (shown in Figure 2.11). Expanded states during each search are shaded grey. All search algorithms break ties among states with the same $f$-values in favor of states with larger $g$-values, which is known to be a good tie-breaking strategy (Koenig & Likhachev, 2005).

![Figure 2.11: Legend of Figure 2.12](image-url)
For the first search, Forward Repeated A* (shown in Figure 2.12(a)) and Eager Adaptive A* (shown in Figure 2.13(a)) perform the same operations, since Eager Adaptive A* and Forward Repeated A* both use the user-provided $h$-values (= Manhattan Distances) to guide the search.

After the first search, Eager Adaptive A* (shown in Figure 2.13(b)) updates the $h$-values of all expanded states during the first search (shaded grey) by executing Assignment (2.1). The expanded states have their updated $h$-values in the lower left corner. In contrast to Eager Adaptive A*, Forward Repeated A* does not update any $h$-values after the first search. The agent then moves along the cost-minimal path returned by the first
search until it reaches B1, where it observes that obstacle B2 blocks its path. Then, the agent performs a second search to find a new path from the current state of the agent B1 to the current state of the target C4.

For the second search, Forward Repeated A* still uses the user provided $h$-values, while Eager Adaptive A* uses updated $h$-values for all states that were expanded during the first search, which can be more informed. Figure 2.12(c) shows that Forward Repeated A* expands D1 and D2 because the user-provided $h$-values are misleading and form a local minimum. Eager Adaptive A* avoids expanding D1 and D2 during the second search (shown in Figure 2.13(c)) resulting in a smaller number of state expansions.

**Lazy Adaptive A*** Eager Adaptive A* first performs a Forward A* search and then updates the $h$-values of the states that were expanded during the search. A disadvantage of updating the $h$-values of all states that were expanded during the search is that one potentially updates the $h$-values of states that are not needed by future searches. Lazy Adaptive A* addresses this problem by postponing updating the $h$-values of the expanded states after one search and updates the $h$-values only when needed during a future search.

Like Eager Adaptive A*, Lazy Adaptive A* can only perform forward searches, and it applies only to agent navigation where the target does not move between searches in known static terrains and unknown static terrains (see Section 1.1.2), where the action costs of the agent do not decrease between searches.

Eager Adaptive A* and Lazy Adaptive A* share the same principle, and they use the same $h$-values for all states when they are needed during each search. Thus, they find
function CalculateKey(s)
    return g(s) + h(s);

procedure InitializeState(s)
    if search(s) = 0
        g(s) := ∞;
        h(s) := H(s, s\text{goal});
    else if search(s) ≠ counter
        if g(s) + h(s) < pathcost(search(s))
            h(s) := pathcost(search(s)) - g(s);
        g(s) := ∞;
        search(s) := counter;

procedure UpdateState(s)
    if s ∈ OPEN
        OPEN.Update(s, CalculateKey(s));
    else
        OPEN.Insert(s, CalculateKey(s));

function ComputePath()
    while OPEN.TopKey() < CalculateKey(s\text{goal})
        s := OPEN.Pop();
        for all s′ ∈ Succ(s)
            InitializeState(s′);
            if g(s′) > g(s) + c(s, s′)
                g(s′) := g(s) + c(s, s′);
                parent(s′) := s;
                UpdateState(s′);
                if OPEN = ∅
                    return false;
        return true;

function Main()
    counter := 0;
    s\text{start} := the current state of the agent;
    s\text{goal} := the current state of the target;
    for all s ∈ S
        search(s) := 0;
    while s\text{start} ≠ s\text{goal}
        counter := counter + 1;
        InitializeState(s\text{start});
        InitializeState(s\text{goal});
        g(s\text{start}) := 0;
        OPEN := ∅;
        OPEN.Insert(s\text{start}, CalculateKey(s\text{start}));
        if ComputePath() = false
            return false; /* Target cannot be caught*/
        pathcost(counter) := g(s\text{goal});
        while target not caught AND action costs on path do not increase
            agent follows the path from s\text{start} to s\text{goal};
        if agent caught target
            return true;
        s\text{start} := the current state of the agent;
        update the increased action costs (if any);
    return true;

Figure 2.14: Pseudo Code of Lazy Adaptive A*

the same paths and move the agent along the same trajectory when using the same tie-
breaking strategy. The only difference between them is that Lazy Adaptive A* updates
the h-values of the states only when they are needed by future searches.
Figure 2.14 gives the pseudo code of Lazy Adaptive A*. Lazy Adaptive A* remembers some information during each search, such as the g-values (Lines 23 and 39) and search-values of states (Line 11), and some information after each search, such as the g-value of the goal state (Lines 44), and then uses this information to calculate the h-value of a state only when it is needed by a future search. During the counter-th search (= the current search), InitializeState(s) is called to initialize the g- and h-values of s (Lines 21, 37 and 38): If h(s) has not yet been initialized by any search before (search(s) = 0), Lazy Adaptive A* initializes h(s) with the user-provided h-value (Line 06). Otherwise, if search(s) ≠ counter, which indicates that h(s) has not been calculated during the current search (Line 07), Lazy Adaptive A* checks whether s was expanded during the search(s)-th search, where h(s) was last calculated: If g(s) + h(s) < pathcost(search(s)) (Line 08), then the f-value of s (= g(s) + h(s)) is smaller than the g-value of the goal state during the search(s)-th search (Line 44). Since the g-value of the goal state is always identical to the f-value of the goal state (with consistent h-values), the f-value of s is smaller than the f-value of the goal state during the search(s)-th search. Thus, s must have been expanded during the search(s)-th search according to A* Property 3 (see Section 2.3.4). Lazy Adaptive A* updates h(s) by executing h(s) := pathcost(search(s)) – g(s) (Line 09). If g(s) + h(s) > pathcost(search(s)), then h(s) is not updated, since s was not expanded during the search(s)-th search. If g(s) + h(s) = pathcost(search(s)), then h(s) is not updated either, since updating it by executing Line 09 will not change its value. g(s) is initialized to infinity if s has not been initialized during the current search (Lines 05 and 10). Finally, search(s) is set to counter each time InitializeState(s) is executed to
indicate that the \( g \)-value of \( s \) has been initialized, and should not be re-initialized during the same search (Line 11).

<table>
<thead>
<tr>
<th>g</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>search</td>
</tr>
</tbody>
</table>

Figure 2.15: Legend of Figure 2.16

I now use the same example path planning problem for Eager Adaptive A* shown in Figure 2.13 to demonstrate the behavior of Lazy Adaptive A*. Figure 2.16 shows the behavior of Lazy Adaptive A*: All states have their \( search \)-value in the lower right corner. States that have been initialized in InitializeState() also have their \( g \)-values in the upper left corner, \( h \)-values in the lower left corner, and \( f \)-values in the upper right corner (shown in Figure 2.15). Expanded states in each search are shaded grey.

Figure 2.16(a) shows the first search of Lazy Adaptive A*, that searches from the current state of the agent D2 to the current state of the target C4. For the first search, Lazy Adaptive A* expands the same states as Eager Adaptive A* (shown in Figure 2.13(a)), since Eager Adaptive A* and Lazy Adaptive A* both use the user-provided \( h \)-values (= Manhattan Distances) to guide the search.

After the first search, Lazy Adaptive A* returns a cost-minimal path with path cost 7, and thus \( pathcost(1) = 7 \) (Line 44). In contrast to Eager Adaptive A* (shown in Figure 2.13(b)), Lazy Adaptive A* does not update any \( h \)-values directly after the first search (shown in Figure 2.16(b)). The agent then moves along the cost-minimal path returned by the first search until it reaches B1, where it observes that obstacle B2 blocks
Figure 2.16: Example Trace of Lazy Adaptive A*

its path. Then, the agent performs a second search to replan a new path from the current state of the agent B1 to the current state of the target C4 (shown in Figure 2.16(c)).

During the second search, Lazy Adaptive A* updates the $h$-values of some states expanded during the first search to make them more informed. An example is D1: InitializeState(D1) updates its $h$-value from 4 to 6 by using Assignment (2.1) (Lines 07 - 09), which is more informed than the user-provided $h$-value. Lazy Adaptive A* updates the $h$-values of the expanded states of the first search only when they are needed during a search. Thus, Lazy Adaptive A* might update the $h$-values of a smaller number of states than Eager Adaptive A*. For example, the $h$-value of D2 is updated from 3 to 7 by Eager Adaptive A* (shown in Figure 2.13(c)) but is not updated by Lazy Adaptive A*.

2.4.2.2 D* Lite

D* Lite (Koenig & Likhachev, 2002) is a search tree transforming incremental search algorithm designed for stationary target search. D* Lite requires the root of the search
function CalculateKey(s)
return [\min(g(s), rhs(s)) + H(s_{goal}, s) + k_m; \min(g(s), rhs(s))];

procedure Initialize()
OPEN := \emptyset;
k_m := 0;
for all s ∈ S
rhs(s) := g(s) := \infty;
parent(s) := NULL;
s_{start} := the current state of the target;
s_{goal} := the current state of the agent;
rhs(s_{start}) := 0;
OPEN Insert(s_{start}, CalculateKey(s_{start}));

procedure UpdateState(u)
if g(u) \neq rhs(u) AND u ∈ OPEN
OPEN Update(u, CalculateKey(u));
else if g(u) \neq rhs(u) AND u \notin OPEN
OPEN Insert(u, CalculateKey(u));
else if g(u) = rhs(u) AND u ∈ OPEN
OPEN Delete(u);

function ComputePath()
while OPEN.TopKey() < CalculateKey(s_{goal}) OR rhs(s_{goal}) > g(s_{goal})

u := OPEN Top();
k_{old} := OPEN TopKey();
k_{new} := CalculateKey(u);
if k_{old} < k_{new}
OPEN Update(u, k_{new});
else if g(u) > rhs(u)
g(u) := rhs(u);
OPEN Delete(u);
for all s ∈ Pred(u)
if s \neq s_{start} AND rhs(s) > g(u) + c(s, u)
parent(s) := u;
rhs(s) := g(u) + c(s, u);
UpdateState(s);
else
if s ∈ Pred(u) \cup \{u\} AND parent(s) = u
rhs(s) := \min_{s'} ∈ Succ(s) (g(s') + c(s, s'));
if rhs(s) = \infty
parent(s) := NULL;
else
parent(s) := arg \min_{s'} ∈ Succ(s) (g(s') + c(s, s'));
UpdateState(s);
if rhs(s_{goal}) = \infty
return false;
return true;

Figure 2.17: Pseudo Code of D* Lite (Part 1)
tree to remain unchanged. Thus, for stationary target search, D* Lite performs backward searches (and cannot perform forward searches) by assigning the current states of the agent and target to the goal and start states of each search, respectively. Thus, the goal state of each search can change between searches since the agent can move between searches. However, the start state of each search does not change between searches since
function Main()
Initialize();
while \( s_{\text{start}} \neq s_{\text{goal}} \)
\( s_{\text{oldgoal}} := s_{\text{goal}}; \)
if ComputePath() = true
while target not caught AND action costs do not changed
agent follows the path from \( s_{\text{goal}} \) to \( s_{\text{start}} \);
if agent caught target
return true;
\( s_{\text{goal}} := \) the current state of the agent;
\( k_m := k_m + H(s_{\text{oldgoal}}, s_{\text{goal}}); \)
else
wait until some action costs decrease;
for all actions whose costs changed from \( c'(u, v) \) to \( c(u, v) \)
if \( c'(u, v) > c(u, v) \)
\( \text{if } u \neq s_{\text{start}} \text{ AND } \text{rhs}(u) > g(v) + c(u, v) \)
\( \text{parent}(u) := v; \)
\( \text{rhs}(u) := g(v) + c(u, v); \)
\( \text{UpdateState}(u); \)
\( \text{else} \)
\( \text{if } u \neq s_{\text{start}} \text{ AND parent}(u) = v \)
\( \text{rhs}(u) := \min_{s' \in \text{Succ}(u)} (g(s') + c(u, s')); \)
\( \text{if } \text{rhs}(u) = \infty \)
\( \text{parent}(u) := \text{NULL}; \)
\( \text{else} \)
\( \text{parent}(u) := \arg \min_{s' \in \text{Succ}(u)} (g(s') + c(u, s')); \)
\( \text{UpdateState}(u); \)
\( \text{return true; } \)

Figure 2.18: Pseudo Code of D* Lite (Part 2)
the target does not move between searches. D* Lite maintains a cost-minimal path between the start and goal states of each search in terrains where the action costs of the agent can both increase and decrease between searches. Thus, D* Lite applies to agent navigation where the target does not move between searches in all three classes of terrains (see Section 1.1.2). However, it is still unknown how to apply D* Lite to moving target search where both the start and goal states can change between searches and make it run faster than A*.
Figures 2.17 and 2.18 show the pseudo code of D* Lite.\textsuperscript{8} D* Lite maintains an $h$-value, $g$-value and parent pointer for every state $s$, with similar semantics as used by A*, but it also maintain an $rhs$-value. The $rhs$-value of $s$ is defined to be

$$rhs(s) = \begin{cases} 
    c & \text{if } s = s_{start} 
    \text{ (Eq. 1)} \\
    \min_{s' \in \text{Succ}(s)} (g(s') + c(s, s')) & \text{otherwise} 
    \text{ (Eq. 2)}
\end{cases}$$

where $c = 0$ for all searches. Thus, the $rhs$-value is basically a one-step lookahead $g$-value. A state $s$ is called locally inconsistent iff $g(s) \neq rhs(s)$. A locally inconsistent state $s$ is called locally underconsistent iff $g(s) < rhs(s)$, and locally overconsistent iff $g(s) > rhs(s)$. Finally, the parent pointer of state $s$ is defined to be

$$parent(s) = \begin{cases} 
    \text{NULL} & \text{if } s = s_{start} \text{ OR } rhs(s) = \infty 
    \text{ (Eq. 3)} \\
    \arg \min_{s' \in \text{Succ}(s)} (g(s') + c(s, s')) & \text{otherwise}
    \text{ (Eq. 4)}
\end{cases}$$

The main routine of D* Lite is performed in ComputePath(), which calculates a cost-minimal path from $s_{goal}$ to $s_{start}$. The OPEN list is a priority queue that allows A* to always expand a state with the smallest $f$-value in it. D* Lite also maintains an OPEN list with a similar semantics as used by A*. The OPEN list contains exactly

\textsuperscript{*}The key $k(s)$ of a state $s$ in the OPEN list roughly corresponds to its $f$-values used by A* and is the pair: $k(s) = [k_1(s); k_2(s)]$, where $k_1(s) = \min(g(s); rhs(s)) + H(s_{goal}; s) + k_m$ and $k_2(s) = \min(g(s); rhs(s))$ (Line 02). The pseudo code uses the following functions to manage the OPEN list: OPEN.Top() returns a state with the smallest key of all states in the OPEN list. OPEN.TopKey() returns the smallest key of all states in the OPEN list. (If the OPEN list is empty, then OPEN.TopKey() returns $[\infty; \infty]$.) OPEN.Insert($s$, $k$) inserts state $s$ into the OPEN list with key $k$. OPEN.Update($s$, $k$) changes the key of state $s$ in the OPEN list to $k$. OPEN.Delete($s$) deletes state $s$ from the OPEN list.
the locally inconsistent states, whose $g$-values need to be updated to make the states locally consistent. Keys are compared according to a lexicographical order. For example, a key $k(s) = [k_1(s), k_2(s)]$ is less than or equal to a key $k'(s) = [k'_1(s), k'_2(s)]$, denoted by $k(s) \leq k'(s)$, iff either $k_1(s) < k'_1(s)$ or ($k_1(s) = k'_1(s)$ and $k_2(s) \leq k'_2(s)$). The first component of the key $k_1(s)$ corresponds to the $f$-value $f(s) := g(s) + h(s)$ used by A*, because both the $g$-values and rhs-values of D* Lite correspond to the $g$-values of A* and the $h$-values of D* Lite correspond to the $h$-values of A*.

The second component of the key $k_2(s)$ corresponds to the $g$-values of A*. D* Lite always expands a state in the OPEN list (by executing Lines 27 - 44) with the smallest $k_1$-value, which corresponds to the $f$-value of A*, breaking ties in favor of the state with the smallest $k_2$-value, which corresponds to the $g$-value of A*. This is similar to a version of A* that always expands a state in the OPEN list with the smallest $f$-value, breaking ties towards smallest $g$-values.

Before ComputePath() is called, states can have arbitrary $g$-values but their rhs-values and parent pointers have to satisfy Eqs. 1-4 and the OPEN list has to contain

---

9Since D* Lite performs backward search, it calculates the user-provided h-values by using $H(s_{goal}, s)$, which estimates $dist^*(s_{goal}, s)$, the minimum cost of moving from $s_{goal}$ (= the current state of the agent) to $s$ of a search.

10Every time ComputePath() is performed to find a cost-minimal path, D* Lite could recalculate the keys of the states in the OPEN list when the agent observes a change in action costs after it has moved. Since the goal state changes between searches after the agent has moved and the $h$-values of all states in the OPEN list were calculated with respect to the goal state of the previous search, the $h$-values need to be updated with respect to the current goal state. However, the repeated reordering of the OPEN list can be computationally expensive since the OPEN list often contains a large number of states. D* Lite therefore uses a method derived from focussed D* (Stentz, 1995) to avoid having to reorder the keys of all states in the OPEN list, namely keys that are lower bounds on the keys that D* Lite uses for the corresponding states: After the agent has moved from the goal state of the previous search $s_{oldgoal}$ to the current goal state $s_{goal}$, where it observes changes in action costs, the first component of the keys can have decreased by at most $H(s_{oldgoal}, s_{goal})$. (The second component does not depend on the $h$-values and thus remains unchanged.) Thus, in order to maintain lower bounds, D* Lite needs to subtract $H(s_{oldgoal}, s_{goal})$ from the first component of the keys of all states in the OPEN list. However, since $H(s_{oldgoal}, s_{goal})$ is the same for all states in the OPEN list, the order of the states in the OPEN list does not change if the subtraction is not performed. Then, when new keys are computed, their first components are by $H(s_{oldgoal}, s_{goal})$ too small relative to the keys in the OPEN list. Thus, $H(s_{oldgoal}, s_{goal})$ has to be added to their first components every time some action costs change. If the agent moves again and then observes action cost changes again, then the constants need to get added up. D* Lite maintains a variable $k_m$ for this method (Lines 02, 05 and 58) (Koenig & Likhachev, 2002).
exactly all locally inconsistent states. ComputePath() terminates when $s_{goal}$ is not locally underconsistent and its key is less than or equal to $OPEN$.TopKey() (Line 21). If $rhs(s_{goal}) = \infty$, then D* Lite terminates with no path found (Line 46). Otherwise, D* Lite finds a cost-minimal path from $s_{start}$ to $s_{goal}$.

The principle of D* Lite can be summarized as follows: D* Lite initially finds a cost-minimal path from the start state to the goal state. When some action costs change between searches, all states $s$ whose $rhs$-values are affected by these changes have their $rhs$-values updated and are inserted into the $OPEN$ list if they become locally inconsistent and are not in the $OPEN$ list yet, or removed from the $OPEN$ list if they become locally consistent and were in the $OPEN$ list before. D* Lite then propagates the effects of these $rhs$-value changes to the rest of the state space until ComputePath() terminates, upon which all states $s$ whose keys are smaller than the key of $s_{goal}$ are locally consistent, and $rhs(s)$ and $g(s)$ are equal to $dist^*(s, s_{start})$.

![g rhs](#)

Figure 2.19: Legend of Figure 2.20

Figure 2.20 gives an example stationary target search problem using D* Lite in known dynamic grids. For ease of illustration, I use $H(s_{goal}, s)$ (= user-provided h-values for D* Lite) that are zero for all states and do not show them in Figure 2.20. All states have their $g$-values in the upper left corner and their $rhs$-values in the upper right corner (shown in Figure 2.19). Expanded states are shaded grey. The state of the target is B1 (marked S), and the current state of the agent is D5 (marked G). The first search of D* Lite runs
with B1 and D5 as the start and goal states, respectively (shown in Figure 2.20(a)). The cost-minimal path is B1, B2, B3, A3, A4, A5, B5, C5 and D5. The agent then moves along this path to C5, at which point in time B4 becomes unblocked, which changes the costs $c(B3,B4)$, $c(A4,B4)$, $c(B4,B5)$, $c(B4,B3)$, $c(B4,A4)$ and $c(B5,B4)$ from infinity to one (shown in Figure 2.20(b)). D* Lite then updates the $rhs$-values and parent pointers of the states affected by the action cost changes by using Eqs. 1-4: D* Lite updates the $rhs$-value and parent pointer of B4 to 3 and B3 respectively, and inserts B4 into the OPEN list. The OPEN list then contains all locally inconsistent states, which provides a starting point for the second search. With all locally inconsistent states in the OPEN list, D* Lite then propagates the $rhs$-value changes to the states whose $rhs$-values need to be updated by executing ComputePath() (Lines 20 - 47). The second search of D* Lite then runs with B1 and C5 as its start and goal states, respectively, and performs only two state expansions B4 (shown in Figure 2.20(c)) and B5 (shown in Figure 2.20(d)). The cost-minimal path is B1, B2, B3, B4, B5 and C5. The agent then moves along this path until it reaches B1.

D* Lite is efficient for path planning for stationary target search in unknown static terrain because an agent can usually observe action cost changes only within its sensor range. Thus, the action cost changes are commonly close to the agent. Since D* Lite
performs backward searches by assigning the current states of the agent and target to the goal and start states of each search, respectively, these changes are usually close to the goal state and far away from the start state (= the root of the search tree). Thus, D* Lite only needs to propagate the \textit{rhs}-value changes to a small number of states. It is possible for D* Lite and its variants to be slower than A* for some easy navigation problems, where the agent catches the target with only a small number of searches (Hernández, Baier, Uras, & Koenig, 2012) or if the changes are close to the root of the search tree, in which case D* Lite can even expand more states than an A* search that searches from scratch. It has been demonstrated that D* Lite can achieve a speedup of one to two orders of magnitude over repeated A* searches for agent navigation in unknown static terrains (Koenig & Likhachev, 2002). However, it is still unknown how to adapt D* Lite to moving target search, where both the start and goal states can change between searches, and make it run faster than A*.

\subsection{2.4.2.3 Differential A*}

Differential A* (Trovato, 1990) is a search tree transforming incremental search algorithm which has been designed for agent navigation in terrains where the action costs of the agent can both increase and decrease between searches. Differential A* can either perform forward searches or backward searches since it applies to search problems where both the start and goal states can change between searches. Thus, it applies to moving target search in all three classes of terrains (see Section 1.1.2).

The principle of Differential A* is as follows: Differential A* defines an “active transition” for each \( s \in S \) in the search tree, which is the transition among all transitions
from $s' \in \text{Pred}(s)$ to $s$ that minimizes $g(s') + c(s', s)$. When the start state or action costs change between searches, Differential A* uses a “Difference Engine” to perform a case-by-case analysis (with 14 cases) to determine the potentially affected active transitions and the states involved. All potentially affected active transitions and the states are deleted from the search tree of the previous search. Differential A* then starts a new search with the remaining search tree of the previous search and thus does not have to construct the search tree for the new search from scratch. Differential A* is efficient when the number of potentially affected active transitions is small and hence a large portion of the search tree of the previous search can be reused for the current search. When the start state changes between searches, Differential A* deletes the previous start state and creates the new start state. Deleting the previous start state potentially affects the active transitions of all states in the search tree of the previous search, and Differential A* has to recursively delete the search tree of the previous search. Thus, Differential A* cannot reuse any part of the search tree from the previous search in this case (but has to spend effort deleting the search tree from the previous search) and hence can be slower than A* for moving target search where both the start and goal states can change over time.

### 2.5 Evasion Algorithms for the Targets

Although most research on moving target search has focused on developing efficient path planning algorithms for the agent to catch the target, there are several path planning algorithms for the target to avoid the agent, referred to as “evasion algorithms for the targets.” Evasion algorithms for the targets generally fall into two classes:
• The first class is a class of offline algorithms. Offline algorithms, such as mini-
max search, can be used for the target to avoid the agent for as long as possible. 
Unfortunately, even if well optimized, offline algorithms do not scale to large ter-
rains (Moldenhauer & Sturtevant, 2009a).

• The second class is a class of online algorithms. Online algorithms, such as 
MTS (Ishida & Korf, 1991), have been used to guide the target to a state as 
far away from the current state of the agent as possible (see the “Experimental 
Evaluation Section” in (Ishida & Korf, 1991)). Four simple algorithms are used 
in (Goldenberg, Kovarsky, Wu, & Schaeffer, 2003) to guide the target: (1) moving 
to a state that maximizes the minimum cost of moving from the current state of 
the agent to the state, (2) moving to a state that maximizes the move choices of the 
target and thus its mobility, (3) moving to a state that is not in line of sight of the 
agent, and (4) moving randomly. The cover heuristic algorithm (Isaza, Lu, Bulitko, 
& Greiner, 2008) is a path planning algorithm for the target to avoid the agent: It 
computes the number of states in the state space that the target can move to before 
the agent from each successor state of the current state of the target, referred to as 
the “target’s cover” of that successor state. The algorithm then moves the target 
to the successor state with the largest target’s cover. TrailMax (Moldenhauer & 
Sturtevant, 2009a) is a recent evasion algorithm for the target to avoid the agent. 
TrailMax generates paths for the target that maximize the minimum cost of the 
trajectory of the agent to catch the target in known static terrains. TrailMax in-
terleaves two independent searches by using Dijkstra’s algorithm starting from the
current state of agent, referred to as the agent’s search, and the current state of
the target, referred to as the target’s search, respectively. Two priority queues are
maintained, one for the agent’s search and one for the target’s search. The speeds
of the agent and the target are $v_a$ and $v_t$, respectively. Each state $s$ maintains two
versions of $g$-values: (1) $g_a(s)$ is the $g$-value of $s$ in the agent’s search, which is the
minimum cost of moving from the current state of the agent to $s$. (2) $g_t(s)$ is the
$g$-value of $s$ in the target’s search, which is the minimum cost of moving from the
current state of the target to $s$. Initially, the $g_a$-value of the current state of the
agent and the $g_t$-value of the current state of the target are zero, and the $g_a$- and
$g_t$-values of all other states are infinity. TrailMax repeatedly performs the follow-
ing procedure: The agent’s search and the target’s search take turns to expand all
states with a given $g$-value (starting from 0), starting with the agent’s search. In
the agent’s search, states are expanded as in Dijkstra’s algorithm. In the target’s
search, whenever a state $s$ is deleted from the priority queue for expansion, TrailMax
checks whether $g_t(s)/v_t \geq g_a(s)/v_a$. If this is the case, then $s$ is discarded and not
expanded in the target’s search, because the agent can move to $s$ before the target
and catch the target at $s$. Otherwise, $s$ is expanded as in Dijkstra’s algorithm. If
all states that have been expanded in the target’s search have been expanded in the
agent’s search, then TrailMax terminates and the last state that was expanded in
the agent’s search is the state that the target moves to. Otherwise, TrailMax sets
the $g$-value to the smallest value among the $g_a$- and $g_t$-values of all states in the
two priority queues and repeats the procedure.
In this dissertation, I will evaluate the runtime of the developed incremental search algorithms by applying them to moving target search with different movement strategies of the target:

- The target repeatedly moves to a randomly chosen state with the restriction that there exists a path from the current state of the target to the randomly chosen state.

- The target uses TrailMax to avoid the agent.

By doing so, I demonstrate the runtime advantage of the developed incremental search algorithms over Repeated A* and all competing incremental search algorithms with respect to different movement strategies of the target.

2.6 Summary

In this chapter, I introduced background knowledge of path planning for agent navigation. In Section 2.1, I introduced the notation used in this dissertation. In Section 2.2, I described a way of representing path planning problems by using discretization, so that they can be solved with search algorithms. In Section 2.3, I described A*, a popular search algorithm in artificial intelligence. All incremental search algorithms developed in this dissertation are based on A*. In Section 2.4, I described existing Repeated A*-based path planning algorithms, including real-time and incremental search algorithms, and described their advantages and disadvantages. Incremental search algorithms have been a great success for many applications of stationary target search. However, there does not exist any incremental search algorithm that applies to moving target search and
runs faster than Repeated A* in terrains where the action costs of the agent can change over time. Thus, the objective of my dissertation is to develop new incremental search algorithms for moving target search that run faster than Repeated A*. In Section 2.5, I described evasion algorithms for the targets to avoid the agent in moving target search. I will evaluate the runtime of the developed incremental search algorithms by applying them to moving target search with different movement strategies of the target.
Chapter 3: Incremental Search by Learning $h$-values

In this chapter, I introduce heuristic learning incremental search algorithms for moving target search.

In Section 2.4.2.1, I reviewed Adaptive A* (Koenig & Likhachev, 2005) designed for stationary target search in unknown static terrains. Thus far, there does not exist a heuristic learning incremental search algorithm that applies to moving target search in all three classes of terrains (see Section 1.1.2). Thus, I develop GAA* (Sun et al., 2008), a new heuristic learning incremental search algorithm for moving target search. Table 3.1 illustrates the search problems that Adaptive A*, MT-Adaptive A* and GAA* apply to:

- Adaptive A* applies to search problems where the start state can change and the action costs do not decrease between searches. It does not apply to search problems where the goal state can change or the action costs can decrease between searches.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Action Costs Can Increase Between Searches</th>
<th>Action Costs Can Decrease Between Searches</th>
<th>Start State Can Change Between Searches</th>
<th>Goal State Can Change Between Searches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive A*</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MT-Adaptive A*</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GAA*</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.1: Search Problems that Heuristic Learning Incremental Search Algorithms Apply to
Thus, Adaptive A* can only perform forward (but not backward) searches by assigning the current states of the agent and the target to the start and goal states of each search, respectively, for stationary target search. Adaptive A* applies only to stationary target search in known static terrains and unknown static terrains (see Section 1.1.2).

- MT-Adaptive A* extends Adaptive A* to search problems where both the start and goal states can change between searches. Thus, MT-Adaptive A* can either perform forward searches by assigning the current states of the agent and the target to the start and goal states of each search, respectively, referred to as Forward Moving Target Adaptive A* (Forward MT-Adaptive A*), or perform backward searches by assigning the current states of the agent and the target to the goal and start states of each search, respectively, referred to as Backward Moving Target Adaptive A* (Backward MT-Adaptive A*). However, MT-Adaptive A* still does not apply to search problems where the action costs can decrease between searches. Thus, MT-Adaptive A* applies only to moving target search in known static terrains and unknown static terrains (see Section 1.1.2).

- GAA* extends MT-Adaptive A* to search problems where both the start and goal states can change and the action costs can both increase and decrease between searches. Thus, GAA* can either perform forward searches by assigning the current states of the agent and the target to the start and goal states of each search, respectively, referred to as Forward Generalized Adaptive A* (Forward GAA*), or perform backward searches by assigning the current states of the agent and
the target to the goal and start states of each search, respectively, referred to as Backward Generalized Adaptive A* (Backward GAA*). GAA* applies to moving target search in all three classes of terrains (see Section 1.1.2).

The structure of this chapter is as follows: In Section 3.1, I review MT-Adaptive A* (Koenig et al., 2007) that extends Adaptive A* to moving target search by maintaining the consistency of the $h$-values with respect to the goal state after it changed between searches. In Section 3.2, I introduce GAA* (Sun et al., 2008) that extends MT-Adaptive A* to moving target search in terrains where the action cost can both increase and decrease between searches. In Section 3.3, I provide a summary of the heuristic learning incremental search algorithms.

### 3.1 Review of Moving Target Adaptive A*

MT-Adaptive A* (Koenig et al., 2007) extends Adaptive A* to moving target search by maintaining the consistency of the $h$-values with respect to the goal state after it changed between searches. In Section 3.1.1, I first review an eager version of MT-Adaptive A*, referred to as Eager Moving Target Adaptive A* (Eager MT-Adaptive A*) (Koenig et al., 2007). In Section 3.1.2, I then review a more complicated version, referred to as Lazy Moving Target Adaptive A* (Lazy MT-Adaptive A*) (Koenig et al., 2007).

#### 3.1.1 Eager Moving Target Adaptive A*

In this section, I review Eager MT-Adaptive A* (Koenig et al., 2007), that extends Eager Adaptive A* to moving target search.
3.1.1.1 Applicability

Eager MT-Adaptive A* can perform either forward or backward searches, and it applies only to moving target search in known static terrains and unknown static terrains (see Section 1.1.2). In this section, I review only Forward Eager MT-Adaptive A*, since it is trivial to switch the search direction, resulting in Backward Eager MT-Adaptive A*.

3.1.1.2 Principle

It is not straight-forward to extend Eager Adaptive A* to moving target search since the $h$-values need to be consistent with respect to the goal state. In moving target search, where the goal state can change between searches, the $h$-values have to be corrected in order to maintain the consistency of the $h$-values with respect to the goal state after it changed between searches, which is the issue addressed by Forward Eager MT-Adaptive A*. The principle behind Forward Eager MT-Adaptive A* is as follows: Assume that the goal state changes from $s_{goal}$ to $s'_{goal}$ with $s'_{goal} \neq s_{goal}$ after a search. For all $s \in S$, $h(s)$ is the $h$-value of $s$ immediately after the previous search. Assume that the $h$-values of all $s \in S$ are consistent with respect to $s_{goal}$. Forward Eager MT-Adaptive A* then corrects the $h$-values of all $s \in S$ to make them consistent with respect to $s'_{goal}$ by assigning

$$h(s) := \max(H(s, s'_{goal}), h(s) - h(s'_{goal})). \quad (3.1)$$

It has been shown that the $h$-values of all $s \in S$ are consistent with respect to $s'_{goal}$ after the assignment (Koenig et al., 2007). Moreover, taking the maximum of the
user-provided $h$-value $H(s, s'_{goal})$ and $h(s) - h(s'_{goal})$ ensures that the $h$-values used by Forward Eager MT-Adaptive A* are at least as informed as the user-provided $h$-values with respect to $s'_{goal}$. Thus, a search of Forward Eager MT-Adaptive A* with the new $h$-values cannot expand more states than a search of A* with the user-provided $h$-values with respect to $s'_{goal}$ (with the same tie-breaking strategy). However, it is not the case that Forward Eager MT-Adaptive A* is guaranteed to reduce its runtime per search more and more over time since the $h$-values are not guaranteed to become more and more informed. Updating them makes them potentially more informed while correcting them makes them potentially less informed. In the following, I refer to “updating the $h$-values” and “correcting the $h$-values” to distinguish the two steps.

3.1.1.3 Operations

Figure 3.1 gives the pseudo code of Forward Eager MT-Adaptive A*. Since Forward Eager MT-Adaptive A* applies only to terrains where the action costs of the agent do not decrease between searches, Forward Eager MT-Adaptive A* can terminate when no path exists between the start state and the goal state after a search (Line 42), because it is then guaranteed that it will not be able to find any path in future searches. Forward Eager MT-Adaptive A* performs each search by executing ComputePath() (Lines 14 - 26). After each search, it updates the $h$-values of all states that have been expanded by that search by executing Assignment (2.1) (Lines 43 - 44). The updated $h$-values are consistent with respect to the goal state of that search. The agent then moves along the path found by the search. If the goal state changed and it is now off the path because the
target moved after the previous search, then Forward Eager MT-Adaptive $A^*$ corrects the $h$-values of all states in the state space by executing Assignment (3.1) (Lines 53 - 54).

There are only one addition and two modifications to the pseudo code of Eager Adaptive $A^*$ (shown in Figure 2.10). The only addition is Lines 50 - 54, where the $h$-values of all states are corrected if the goal state changes between searches. One modification is Line 32, where the $h$-values are now initialized with the user-provided $h$-values with respect to the goal state of the first search up front since all $h$-values are corrected after the goal state changes between searches and thus need to have been initialized at this point in time. The second modification is Line 45, that now takes into account that the target can move between searches.

### 3.1.1.4 Example

I now use the example problem of moving target search introduced in Figure 2.9 to demonstrate the advantage of Forward Eager MT-Adaptive $A^*$ over Forward Repeated $A^*$. The terrain and the current states of the agent and target are exactly the same as those in Figure 2.9(a) before the first search. After the first search, the target moves from C4 to C3.

Figures 3.3 and 3.4 illustrate the same path planning problem solved with Forward Repeated $A^*$ and Forward Eager MT-Adaptive $A^*$, respectively. All states have their search-value in the lower right corner. States that have been initialized in InitializeState() also have their $g$-values in the upper left corner, $h$-values in the lower left corner, and $f$-values in the upper right corner (shown in Figure 3.2). For Forward Eager MT-Adaptive $A^*$, I also show the $h$-values of the states that have not been initialized in InitializeState()
function CalculateKey(s)
return g(s) + h(s);

procedure InitializeState(s)
if search(s) = 0
    g(s) := ∞;
else if search(s) ≠ counter
    g(s) := ∞;
else
    search(s) := counter;
end if

procedure UpdateState(s)
if s ∈ OPEN
    OPEN.Update(s, CalculateKey(s));
else
    OPEN.Insert(s, CalculateKey(s));
end if

function ComputePath()
while OPEN.TopKey() < CalculateKey(s_goal)
s := OPEN.Pop();
CLOSED.Insert(s);
for all s' ∈ Succ(s)
    InitializeState(s');
g(s') := g(s) + c(s, s');
parent(s') := s;
UpdateState(s');
end for
if OPEN = ∅
    return false;
else
    return true;
end if

function Main()
counter := 0;
s_start := the current state of the agent;
s_goal := the current state of the target;
for all s ∈ S
    h(s) := H(s, s_goal);
    search(s) := 0;
end for
while s_start ≠ s_goal
    counter := counter + 1;
    InitializeState(s_start);
    InitializeState(s_goal);
g(s_start) := 0;
OPEN := CLOSED := ∅;
OPEN.Insert(s_start, CalculateKey(s_start));
if ComputePath() = false
    return false; /* Target cannot be caught */
end if
for all s ∈ CLOSED
    h(s) := g(s_goal) − g(s);
end for
while target not caught AND action costs on path do not increase AND target on path from s_start to s_goal
    agent follows the path from s_start to s_goal;
    if agent caught target
        return true;
    else
        s_start := the current state of the agent;
s_newgoal := the current state of the target;
        if s_goal ≠ s_newgoal
            old := h(s_newgoal);
            for all s ∈ S
                h(s) := max(H(s, s_newgoal), h(s) − old);
            end for
            s_goal := s_newgoal;
            update the increased action costs (if any);
        end if
    end if
end while
return true;

Figure 3.1: Pseudo Code of Forward Eager Moving Target Adaptive A*
since they have been initialized before the first search (Lines 31 - 32). By doing so, I demonstrate how their $h$-values are corrected after the target moved. All search methods break ties between states with the same $f$-values in favor of states with larger $g$-values.

For the first search, Forward Repeated A* (shown in Figure 3.3(a)) and Forward Eager MT-Adaptive A* (shown in Figure 3.4(a)) expand exactly the same states as they both use the user-provided $h$-values to guide the search. The expanded states are shaded grey.

After the first search, Forward Eager MT-Adaptive A* (shown in Figure 3.4(b)) updates the $h$-values of all states that have been expanded in the first search (shaded grey) by...
executing Assignment (2.1). The expanded states have their updated $h$-values in the lower left corner. In contrast to Forward Eager MT-Adaptive A*, Forward Repeated A* does not update any $h$-values after the first search. The agent then moves along the cost-minimal path returned by the first search until it reaches B1, where it observes that B2 is blocked and the target moves from C4 to C3.

After the target moved to C3, Forward Eager MT-Adaptive A* (shown in Figure 3.4(c)) corrects the $h$-values of all states (shaded grey) by executing Assignment (3.1). All states have their corrected $h$-values in the lower left corner. In contrast to Forward Eager MT-Adaptive A*, Forward Repeated A* does not correct the $h$-values.

The agent then performs the second search to replan a new path from its current state B1 to the current state of the target C3. For the second search, Forward Repeated A* still uses the user-provided $h$-values, while Forward Eager MT-Adaptive A* can use the updated and corrected $h$-values, which can be more informed. The expanded states are shaded grey. Figure 3.3(d) shows that Forward Repeated A* expands D1 and D2 because the user-provided $h$-values are misleading. Forward Eager MT-Adaptive A* avoids expanding D1 and D2 during the second search (shown in Figure 3.4(d)) resulting in a smaller number of state expansions.

### 3.1.2 Lazy Moving Target Adaptive A*

Eager MT-Adaptive A* can update and correct the $h$-values of states that are not needed by future A* searches which can be time-consuming if the number of expanded states are large or the goal state changes (Koenig et al., 2007). In this section, I review Lazy Moving
01 function CalculateKey(s)
02 return g(s) + h(s);

03 procedure InitializeState(s)
04 if search(s) = 0
05 g(s) := \infty;
06 h(s) := H(s, s_{goal});
07 else if search(s) \neq counter
08 if g(s) + h(s) < pathcost(search(s))
09 h(s) := pathcost(search(s)) - g(s);
10 h(s) := h(s) - (deltaH(counter) - deltaH(search(s)));
11 h(s) := max(h(s), H(s, s_{goal}));
12 g(s) := \infty;
13 search(s) := counter;

14 procedure UpdateState(s)
15 if s \in OPEN
16 OPEN.Update(s, CalculateKey(s));
17 else
18 OPEN.Insert(s, CalculateKey(s));

19 function ComputePath()
20 while OPEN.TopKey() < CalculateKey(s_{goal})
21 s := OPEN.Pop();
22 for all s’ \in Succ(s)
23 InitializeState(s’);
24 if g(s’) > g(s) + c(s, s’)
25 g(s’) := g(s) + c(s, s’);
26 parent(s’) := s;
27 UpdateState(s’);
28 if OPEN = \emptyset
29 return false;
30 return true;

Figure 3.5: Pseudo Code of Forward Lazy Moving Target Adaptive A* (Part 1)

Target Adaptive A* (Lazy MT-Adaptive A*) (Koenig et al., 2007), that calculates the
h-value of a state only when it is needed by a future A* search.

3.1.2.1 Applicability

Like Eager MT-Adaptive A*, Lazy MT-Adaptive A* can perform either forward or back-
ward searches, and it applies only to moving target search in known static terrains and
unknown static terrains (see Section 1.1.2). In this section, I review only Forward Lazy
MT-Adaptive A*, since it is trivial to switch the search direction, resulting in Backward
Lazy MT-Adaptive A*.
31 function Main()
32 counter := 0;
33 s_{start} := the current state of the agent;
34 s_{goal} := the current state of the target;
35 deltah(1) := 0;
36 for all s \in S
37 search(s) := 0;
38 while s_{start} \neq s_{goal}
39 counter := counter + 1;
40 InitializeState(s_{start});
41 InitializeState(s_{goal});
42 g(s_{start}) := 0;
43 OPEN := \emptyset;
44 OPEN.Insert(s_{start}, CalculateKey(s_{start}));
45 if ComputePath() = false
46 return false; /* Target cannot be caught */
47 pathcost(counter) := g(s_{goal});
48 while target not caught AND action costs on path do not increase AND target on path from s_{start} to s_{goal}
49 agent follows the path from s_{start} to s_{goal};
50 if agent caught target
51 return true;
52 s_{start} := the current state of the agent;
53 s_{newgoal} := the current state of the target;
54 if s_{goal} \neq s_{newgoal}
55 InitializeState(s_{newgoal});
56 if g(s_{newgoal}) + h(s_{newgoal}) < pathcost(counter)
57 h(s_{newgoal}) := pathcost(counter) - g(s_{newgoal});
58 deltah(counter + 1) := deltah(counter) + h(s_{newgoal});
59 s_{goal} := s_{newgoal};
60 else
61 deltah(counter + 1) := deltah(counter);
62 update the increased action costs (if any);
63 return true;

Figure 3.6: Pseudo Code of Forward Lazy Moving Target Adaptive A* (Part 2)

3.1.2.2 Principle

Forward Eager MT-Adaptive A* and Forward Lazy MT-Adaptive A* share the same principle and use the same \( h \)-values for all states when they are needed during each search. Thus, they find the same paths and move the agent along the same trajectory. They differ in that Forward Lazy MT-Adaptive A* updates and corrects the \( h \)-values of the states only when they are needed by future searches, while Forward Eager MT-Adaptive A* updates the \( h \)-values of all expanded states after each search and corrects the \( h \)-values of all states after each search when the goal state changed, no matter whether these \( h \)-values will be needed by future searches or not.
3.1.2.3 Operations

Figures 3.5 and 3.6 give the pseudo code of Forward Lazy MT-Adaptive A*. InitializeState(s) initializes $h(s)$ with the user-provided $h$-value with respect to the goal state of the current search if $h(s)$ has not been initialized by any previous search (Lines 04 and 06).\footnote{This is similar to both Eager Adaptive A* and Lazy Adaptive A* but different from Forward Eager MT-Adaptive A* that initializes the $h$-value up front.} If $h(s)$ has been initialized by one of the previous searches but not the current one (Line 07), then InitializeState(s) first updates $h(s)$ if $s$ has been expanded before (Lines 08 - 09), and then corrects $h(s)$ with respect to the goal state of the current search in case the target has moved since the last time $s$ was initialized by InitializeState(s) (Lines 10 - 11). I now explain in detail how Lazy MT-Adaptive A* corrects the $h$-value of $s$: After each search, if the target moved and hence the goal state changed, the correction for the goal state of the current search decreases the $h$-values of all states by the $h$-value of the goal state of the current search. This $h$-value is first calculated (Lines 55 - 57) and then added to a running sum of all corrections (Line 58). In particular, the value of $deltah(x)$ during the $x$-th search is the running sum of all corrections up to the beginning of the $x$-th search. If $h(s)$ was initialized (by InitializeState(s)) in a previous search but not yet initialized (by InitializeState(s)) in the current search ($search(s) \neq counter$), then InitializeState(s) corrects its $h$-value by the sum of all corrections between the search when $s$ was initialized last and the current search, which is the same as the difference of the value of $deltah$ during the current search ($deltah(counter)$) and the search where $s$ was initialized last ($deltah(search(s))$) (Line 10). It then takes the maximum of this value and the user-provided $h$-value with respect to the goal state of the current search (Line
In this way, Forward Lazy MT-Adaptive A* updates and corrects the $h$-values of all states only when they are needed by future searches, which is faster than updating the $h$-values of all expanded states after each search and correcting the $h$-values of all states after each search when the goal state changed as done by Eager MT-Adaptive A*.

It has been shown that Forward Eager MT-Adaptive A* uses consistent $h$-values with respect to the goal state during every search (Koenig et al., 2007). I now show that during every search Forward Lazy MT-Adaptive A* uses the same $h$-values for all $s \in S$ as Forward Eager MT-Adaptive A* when they use the same tie-breaking strategy. Thus, its $h$-values remain consistent with respect to the goal state.

**Theorem 1** During every search, whenever the $h$-value of any state $s$ is calculated by $\text{InitializeState}(s)$ for Forward Lazy MT-Adaptive A*, it is identical to the $h$-value used by Forward Eager MT-Adaptive A* (with the same tie-breaking strategy) in the same search.

**Proof:** The values at the end of the $i$-th search are indicated via superscript $i$. The $h$-values at the end of the $i$-th search are the same as those used during the $i$-th search since Forward Eager MT-Adaptive A* does not update any $h$-value during a search and Forward Lazy MT-Adaptive A* calculates any $h$-value the first time it is needed by a search and then returns its $h$-value whenever it is needed again by the same search. The values of Forward Eager MT-Adaptive A* are not overlined, while the values of Forward Lazy MT-Adaptive A* are overlined. We do not make this distinction for $s_{\text{goal}}^i$ since $s_{\text{goal}}^i = s_{\text{goal}}^i$ per construction.

We define a new variable $z^i(s) = h^i(s)$ if $s$ was not expanded by Eager MT-Adaptive A* during the $i$-th search and $z^i(s) = g^i(s_{\text{goal}}^i) - g^i(s)$ otherwise. Similarly, we define
a new variable \(\overline{z}^i(s) = \overline{h}^i(s)\) if \(s\) was not expanded by Forward Lazy MT-Adaptive A* during the \(i\)-th search and \(\overline{z}^i(s) = g^i(s_{goal}) - g^i(s)\) otherwise.\(^2\) \(z^{i+1}(s_{goal})\) is equal to \(h(s_{newgoal})\) calculated by Forward Eager MT-Adaptive A* on Line 52, and \(z^{i+1}(s_{goal})\) is equal to \(\overline{h}(s_{newgoal})\) calculated by Forward Lazy MT-Adaptive A* on Line 57. (In this proof, the line numbers that are not overlined refer to the pseudo code of Forward Eager MT-Adaptive A* in Figure 3.1, and the line numbers that are overlined refer to the pseudo code of Forward Lazy MT-Adaptive A* in Figures 3.5 and 3.6, respectively.)

Forward Eager and Lazy MT-Adaptive A* generate and expand exactly the same states, find exactly the same paths for every search when they use the same \(h\)-values and tie-breaking strategy and thus also calculate the same \(g\)- and \(z\)-values (**Property 1**).

We prove the theorem by induction on the number of times Forward Lazy MT-Adaptive A* calls InitializeState. Assume that Forward Lazy MT-Adaptive A* calls InitializeState\((s)\) during the \(j\)-th search. Let \(x\) be equal to \(search(s)\) at that point in time. These \(s, j\) and \(x\) are used in the remainder of the proof.

**Lemma 1** \(z^i(s) \geq h^i(s)\) for all \(i\) with \(i \geq 1\).

**Proof:** After each search \(i\) with \(i \geq 1\), Forward Eager MT-Adaptive A* calculates \(z^i(s)\) for all \(s \in CLOSED\) (Lines 43 - 44). Since Forward Eager MT-Adaptive A* uses consistent \(h\)-values, \(g^i(s) + h^i(s) \leq g^i(s_{goal})\) (A* Property 3), and \(h^i(s) \leq g^i(s_{goal}) - g^i(s) = z^i(s)\).

For all \(s \notin CLOSED\), \(h^i(s) = z^i(s)\) by definition. \(\blacksquare\)

**Lemma 2** \(h^i(s) \geq H(s, s^i_{goal})\) for all \(i\) with \(i \geq 1\).

\(^2\)If Forward Lazy MT-Adaptive A* expands a state \(s\) with \(\overline{z}^i(s) + \overline{h}^i(s) = g^i(s_{goal})\), then it actually sets \(\overline{z}^i(s) = \overline{h}^i(s)\) but this does not cause a problem for our definition since \(\overline{z}^i(s) = \overline{h}^i(s) = g^i(s) + \overline{h}^i(s) - g^i(s) = g^i(s_{goal}) - g^i(s)\).
Proof: The lemma holds trivially during the first search, since \( h^1(s) = H(s, s^1_{\text{goal}}) \) (Lines 31 - 33), and Forward Eager MT-Adaptive A* does not update any \( h \)-value during a search. After each search, there are only two places where \( h \)-values can change in Forward Eager MT-Adaptive A*: (1) \( h \)-values of all states \( s \in \text{CLOSED} \) are updated on Line 44, and we have shown in Lemma 1 that the \( h \)-values cannot be decreased on Line 44. (2) \( h \)-values of all states \( s \in S \) are corrected on Line 54 iff the goal state changed, and it is trivial to see that \( h^{i+1}(s) \geq H(s, s^{i+1}_{\text{goal}}) \) after Line 54 is executed. If the goal state does not change between searches, then only (1) is executed. Otherwise, both (1) and (2) are executed. Combing (1) and (2), the lemma holds no matter whether the goal state changed or not between searches.

Lemma 3 \( z^i(s) \geq h^i(s) \geq H(s, s^i_{\text{goal}}) \) for all \( i \) with \( i \geq 1 \) (Combining Lemma 1 and Lemma 2).

Lemma 4 If \( h^{k+1}(s) = H(s, s^{k+1}_{\text{goal}}) \) for at least one \( k \) with \( 0 \leq x \leq k < j \), then \( h^j(s) = H(s, s^j_{\text{goal}}) \).

Proof: The lemma trivially holds if \( k = j - 1 \). Otherwise, we show that \( h^{l+2}(s) = H(s, s^{l+2}_{\text{goal}}) \) if \( h^{l+1}(s) = H(s, s^{l+1}_{\text{goal}}) \) for \( k \leq l < j \), which implies the lemma. InitializeState\((s)\) was called last during the \( x \)-th search (or has not been called before iff \( x = 0 \)). Thus, \( s \) was expanded last during or before the \( x \)-th search (or has not been expanded yet iff \( x = 0 \)) by Forward Lazy MT-Adaptive A*, and thus also by Forward Eager MT-Adaptive A* (Property 1).

\[
h^{l+2}(s) = \max(h^{l+1}(s) - z^{l+1}(s^{l+2}_{\text{goal}}), H(s, s^{l+2}_{\text{goal}}))
\]

\[
= \max(H(s, s^{l+1}_{\text{goal}}) - z^{l+1}(s^{l+2}_{\text{goal}}), H(s, s^{l+2}_{\text{goal}}))
\]
\[
\leq \max(H(s, s^{l+1}_\text{goal}) - H(s^{l+2}_\text{goal}, s^{l+1}_\text{goal}), H(s, s^{l+2}_\text{goal})) \quad \text{(Lemma 3)}
\]
\[
\leq \max(H(s, s^{l+2}_\text{goal}), H(s, s^{l+2}_\text{goal})) \quad \text{(H-value triangle inequality)}
\]
\[
= H(s, s^{l+2}_\text{goal})
\]

In the meantime, Lemma 2 guarantees that \(h^{l+2}(s) \geq H(s, s^{l+2}_\text{goal})\). Thus, \(h^{l+2}(s) = H(s, s^{l+2}_\text{goal})\). 

Lemma 5 \(\delta_l = \sum_{i=1}^{l-1} z^i(s^{i+1}_\text{goal})\) for all \(1 \leq l \leq j\).

Proof: The lemma holds trivially when \(l = 1\) since \(\delta_1 = 0\) initially (Line 35).

Otherwise, when \(1 < l \leq j\), assume that the lemma holds until the \((l-1)\)-st search, that is, \(\delta_{l-1} = \sum_{i=1}^{l-2} z^i(s^{i+1}_\text{goal})\). After the \((l-1)\)-st search, Forward Lazy MT-Adaptive A* calculates \(\delta_l\) in two cases:

- **Case 1:** If \(s^{l-1}_\text{goal} \neq s^l_{\text{goal}}\) (Lines 54 - 59),
  \[
  \delta_l = \delta_{l-1} + z^{l-1}(s^l_{\text{goal}}) \quad \text{(Line 58)}
  \]
  \[
  = \sum_{i=1}^{l-2} z^i(s^{i+1}_\text{goal}) + z^{l-1}(s^l_{\text{goal}})
  = \sum_{i=1}^{l-1} z^i(s^{i+1}_\text{goal}).
  \]

- **Case 2:** If \(s^{l-1}_\text{goal} = s^l_{\text{goal}}\) (Lines 60 - 61),
  \[
  h^{l-1}(s^{l-1}_\text{goal}) = h^{l-1}(s^l_{\text{goal}}) = 0 \quad \text{since Forward Lazy MT-Adaptive A* uses consistent}
  \]
  \[
  h\text{-values during the \((l-1)\)-st search. Moreover, since } s^{l-1}_\text{goal} \text{ cannot be expanded during the \((l-1)\)-st search due to the condition on Line 20, and } s^{l-1}_\text{goal} = s^l_{\text{goal}} \text{ in this}
  \]
  \[
  \text{case, } s^l_{\text{goal}} \text{ cannot be expanded during the \((l-1)\)-st search either. Thus, } z^{l-1}(s^l_{\text{goal}}) =
  \]
  \[
  h^{l-1}(s^l_{\text{goal}}) \text{ according to our definition. Thus, } z^{l-1}(s^l_{\text{goal}}) = h^{l-1}(s^l_{\text{goal}}) = 0.
  \]
Thus, \( \delta_{L}(l) = \delta_{L}(l-1) \) (Line 61)

\[
\begin{align*}
\delta_{L}(l) &= \delta_{L}(l-1) + 0 \\
&= \delta_{L}(l-1) + \sum_{i=1}^{l-2} \Delta_{i}(s_{goal}^{i+1}) \\
&= \sum_{i=1}^{l-1} \Delta_{i}(s_{goal}^{i+1}).
\end{align*}
\]

Combining Case 1 and Case 2, the lemma holds.

If \( x = j \), then InitializeState(\( s \)) does not change \( \overline{h}(s) \). It was called last during the \( x \)-th search, that is, the current search. It continues to hold that \( h^{i}(s) = \overline{h}^{i}(s) \) according to the induction hypothesis. Otherwise, \( 0 \leq x < j \). We distinguish two cases:

- **Case 1:** Assume that \( x = 0 \) (induction basis). Then, \( h^{i}(s) = H(s, s_{goal}^{j}) = \overline{h}^{i}(s) \) since \( h^{1}(s) = H(s, s_{goal}^{1}) \) and thus \( h^{i}(s) = H(s, s_{goal}^{j}) \) according to Lemma 4.

- **Case 2:** Otherwise, \( x > 0 \). Assume that Forward Eager and Lazy MT-Adaptive A* used the same \( h \)-values every time Forward Lazy MT-Adaptive A* called InitializeState so far. \( s \) was expanded last during or before the \( x \)-th search by Forward Lazy MT-Adaptive A* and thus also by Forward Eager MT-Adaptive A* according to the induction hypothesis since they expand the same states during the same search where they use the same \( h \)-values and tie-breaking strategy. We distinguish two cases:

  - **Case a:** Assume that \( h^{k+1}(s) = H(s, s_{goal}^{k+1}) \) for at least one \( k \) with \( x \leq k < j \). Then \( h^{i}(s) = H(s, s_{goal}^{j}) \) according to Lemma 4. It holds that \( z^{x}(s) - \sum_{l=x}^{j-1} z^{l}(s_{goal}^{l+1}) \leq h^{i}(s) \) due to the monotonicity of the max operator.
used repeatedly in the calculation of \( h^j(s) \) \textbf{(Property 2)} . Thus,

\[
\overline{h}^j(s) = \max(\overline{z}^x(s) - (\text{deltah}(j) - \text{deltah}(x)), H(s, s_{goal}^j)) \quad \text{(Lines 07 - 11)}
\]

\[
= \max(\overline{z}^x(s) - \sum_{l=x}^{j-1} z^l(s_{goal}^{l+1}), H(s, s_{goal}^j)) \quad \text{(Lemma 5)}
\]

\[
= \max(z^x(s) - \sum_{l=x}^{j-1} z^l(s_{goal}^{l+1}), H(s, s_{goal}^j)) \quad \text{(Induction hypothesis)}
\]

\[
\leq \max(h^j(s), H(s, s_{goal}^j)) \quad \text{(Property 2)}
\]

\[
= \max(H(s, s_{goal}^j), H(s, s_{goal}^j))
\]

\[
= H(s, s_{goal}^j)
\]

Since \( x > 0 \), \( \overline{h}^j(s) \geq H(s, s_{goal}^j) \) \textbf{(Line \Pi)}.

Thus, \( \overline{h}^j(s) = H(s, s_{goal}^j) = h^j(s) \).

- Case b: Otherwise, \( h^{x+1}(s) = z^x(s) - z^x(s_{goal}^{x+1}) \) and \( h^{k+1} = h^k(s) - z^k(s_{goal}^{k+1}) \)

for all \( k \) with \( x < k < j \) since \( h^{x+1}(s) = \max(z^x(s) - z^x(s_{goal}^{x+1}), H(s, s_{goal}^{x+1})) \neq H(s, s_{goal}^{x+1}) \) and \( h^{k+1}(s) = \max(h^k(s) - z^k(s_{goal}^{k+1}), H(s, s_{goal}^{k+1})) \neq H(s, s_{goal}^{k+1}) \) for all \( x < k < j \).

Then, \( h^j(s) = \max(h^j(s), H(s, s_{goal}^j)) \) \textbf{(Lemma 2)}

\[
= \max(z^x(s) - \sum_{l=x}^{j-1} z^l(s_{goal}^{l+1}), H(s, s_{goal}^j))
\]

\[
= \max(z^x(s) - \sum_{l=x}^{j-1} z^l(s_{goal}^{l+1}), H(s, s_{goal}^j)) \quad \text{(Induction hypothesis)}
\]

\[
= \max(z^x(s) - (\text{deltah}(j) - \text{deltah}(x)), H(s, s_{goal}^j)) \quad \text{(Lemma 5)}
\]

\[
= \overline{h}^j(s)
\]

\[\square\]

3.1.2.4 Example

I now use the example path planning problem for Forward Eager MT-Adaptive A* shown in Figure 3.4 to demonstrate the behavior of Forward Lazy MT-Adaptive A*. Figure 3.8
Figure 3.7: Legend of Figure 3.8

Figure 3.8: Example Trace of Forward Lazy Moving Target Adaptive A*

shows the behavior of Forward Lazy MT-Adaptive A*: All states have their search-value in the lower right corner. States that have been initialized in InitializeState() also have their g-values in the upper left corner, h-values in the lower left corner, and f-values in the upper right corner (shown in Figure 3.7). Forward Lazy MT-Adaptive A* breaks ties among states with the same f-values in favor of states with larger g-values.

Figure 3.8(a) shows the first search of Forward Lazy MT-Adaptive A* that searches from the current state of the agent D2 to the current state of the target C4. For the first search, Forward Lazy MT-Adaptive A* and Forward Eager MT-Adaptive A* (shown in Figure 3.4(a)) expand exactly the same states as they both use the user-provided h-values to guide the search. The expanded states are shaded grey.

After the first search, Forward Lazy MT-Adaptive A* does not update any h-values (shown in Figure 3.8(b)). The agent then moves along the cost-minimal path returned
by the first search until it reaches B1, where it observes that B2 is blocked and the target
moves from C4 to C3. Since the goal state is now off the path, the agent then performs
a second search to replan a new path from its current state B1 to the goal state C3.
After the goal state changed to C3, Forward Lazy MT-Adaptive A* does not correct the
$h$-values of all states. Instead, it only calculates $h(C3) = 1$, the $h$-value of the goal state
of the second search with respect to the goal state of the first search, and adds $h(C3)$ to
the value of $deltah(1)$, yielding $deltah(2) = 1$ (shown in Figure 3.8(c)), which will be used
for correcting the $h$-values of the states that are used by future searches.

Figure 3.8(d) shows the second search of Forward Lazy MT-Adaptive A*. The ex-
expanded states are shaded grey. It uses the same $h$-values that are needed by the second
search for all states as Forward Eager MT-Adaptive A*. Thus, they find the same paths
and move the agent along the same trajectory. They differ in that Forward Lazy MT-
Adaptive A* updates and corrects the $h$-values of the states only when these $h$-values
are needed by future searches, while Forward Eager Adaptive A* updates the $h$-values
of all states that have been expanded after each search and corrects the $h$-values of all
states after the goal state changed between searches no matter whether the $h$-values of
these states will be needed by future searches. One example is D2: The $h$-value of D2 is
updated from 3 to 7, and corrected from 7 to 6 by Forward Eager MT-Adaptive A* after
the first search. However, the $h$-value of D2 is not needed by the second search and thus
not changed by Forward Lazy MT-Adaptive A*. Another example is D4: The $h$-value
of D4 is corrected from 1 to 2 after the first search by Forward Eager MT-Adaptive A*.
However, the $h$-value of D4 is not needed by the second search and thus not changed by
Forward Lazy MT-Adaptive A*. 
3.2 Generalized Adaptive A*

MT-Adaptive A* extends Adaptive A* to moving target search in terrains where the action costs of the agent do not decrease between searches. However, MT-Adaptive A* does not apply to terrains where the action costs of the agent can decrease between searches. Thus, MT-Adaptive A* only applies to known static terrains and unknown static terrains (see Section 1.1.2). In order to extend MT-Adaptive A* to moving target search in terrains where the action costs can both increase and decrease between searches, so that it applies to all three classes of terrains (see Section 1.1.2), I introduce Generalized Adaptive A* (GAA*) in this section. GAA* extends Lazy MT-Adaptive A*. I choose to extend Lazy MT-Adaptive A* but not Eager MT-Adaptive A* because Eager MT-Adaptive A* can update the $h$-values of states that are not needed by future searches, which is time-consuming if the number of expanded states is large (and Eager MT-Adaptive A* then needs to update the $h$-values of all expanded states) or the target moves (and Eager MT-Adaptive A* then needs to correct the $h$-values of all states) (Koenig et al., 2007).

3.2.1 Applicability

GAA* can either perform forward searches by assigning the current states of the agent and target to the start and goal states of each search, respectively, referred to as Forward GAA*, or perform backward searches by assigning the current states of the agent and target to the goal and start states of each search, respectively, referred to as Backward GAA*. It applies to moving target search in all three classes of terrains (see Section 1.1.2).
In this section, I only introduce Forward GAA*, since it is trivial to switch the search direction, resulting in Backward GAA*.

### 3.2.2 Principle

The principle behind Forward GAA* is as follows: If the cost of no action decreases between searches, Forward GAA* operates in the same way as Forward Lazy MT-Adaptive A*. If some actions whose costs decrease after one search, Forward GAA* performs a version of Dijkstra’s algorithm, referred to as the consistency procedure, to eagerly update the $h$-values of all states that are affected by the action cost decrease, so that they remain consistent with respect to the goal state of the current search.

### 3.2.3 Operations

Figures 3.9 and 3.10 give the pseudo code of Forward GAA*, which extends the pseudo code of Forward Lazy MT-Adaptive A* (shown in Figure 3.5). The major changes to the pseudo code of Forward Lazy MT-Adaptive A* are as follows:

- Forward GAA* repeatedly searches for a new path from the current state of the agent to the current state of the target after it found that there does not exist a path after a search, since the action costs can not only increase but also decrease between searches. There might exist a path from the current state of the agent to the current state of the target after the action costs of the agent decrease in the future. Thus, Forward GAA* terminates only when the agent catches the target (Lines 70 - 71). In order to deal with this termination condition, Forward GAA*
uses a variable last_nopath to remember the last search in which no path existed. Initially, last_nopath = 0 (Line 57) since no search has been performed.

- Forward GAA* contains a consistency procedure, referred to as the consistency procedure (Lines 36 - 51), that eagerly updates consistent $h$-values with respect to the goal state with a version of Dijkstra’s algorithm so that they remain consistent with respect to the goal state after action cost increases and decreases.

I first describe the situation when there exists a path between the current states of the agent and target: After each search, the consistency procedure is performed to maintain the consistency of the $h$-values with respect to the goal state. The consistency procedure maintains an OPEN list, which is a priority queue that contains those states whose $h$-values need to be updated in the consistency procedure. Initially, the OPEN list is set to empty (Line 38). The consistency procedure uses the same OPEN list used in procedure ComputePath(), except that the key of each state in the OPEN list is the $h$-value (rather than the $f$-value) during the consistency procedure, since the objective of the consistency procedure is to propagate the $h$-value changes to the states whose $h$-values need to be updated. Then, for all actions whose costs decreased from $c'(s, s')$ to $c(s, s')$ between searches, the consistency procedure performs the following operations: It first calculates the $h$-values of $s$ and $s'$ by executing InitializeState($s$) and InitializeState($s'$) (Lines 40 - 41). If $h(s) > c(s, s') + h(s')$, $h(s)$ is set to $c(s, s') + h(s')$ (Lines 42 - 43) and inserted into the OPEN list (with $h(s)$ as its key) (Line 44). Starting with the OPEN list, the consistency procedure performs a version of Dijkstra’s algorithm by repeatedly executing the following operations: It deletes a state $s'$ with the smallest $h$-value from
function CalculateKey(s)  
return g(s) + h(s);

procedure InitializeState(s)  
if search(s) ≤ last_xopath  
g(s) := ∞;  
h(s) := H(s, sgoal);
else if search(s) ≠ counter  
if g(s) + h(s) < pathcost(search(s))  
h(s) := pathcost(search(s)) − g(s);  
h(s) := h(s) − (deltah(counter) − deltax(search(s)));  
g(s) := ∞;
search(s) := counter;

procedure UpdateState(s)  
if s ∈ OPEN  
OPEN.Update(s, CalculateKey(s));  
else  
OPEN.Insert(s, CalculateKey(s));

procedure UpdateState2(s)  
if s ∈ OPEN  
OPEN.Update(s, h(s));  
else  
OPEN.Insert(s, h(s));

function ComputePath()  
while OPEN.TopKey() < CalculateKey(sgoal)  
s := OPEN.Pop();  
for all s′ ∈ Succ(s)  
InitializeState(s′);  
if g(s′) > g(s) + c(s, s′)  
g(s′) := g(s) + c(s, s′);  
parent(s′) := s;  
UpdateState2(s′);

procedure ConsistencyProcedure()  
update the increased and decreased action costs (if any);  
OPEN := ∅;  
for all actions whose costs decreased from c′(s, s′) to c(s, s′)  
InitializeState(s);  
InitializeState(s′);  
if h(s) > c(s, s′) + h(s′)  
h(s) := c(s, s′) + h(s′);  
UpdateState2(s);  
while OPEN ≠ ∅  
s′ := OPEN.Pop();  
for all s ∈ Pred(s′) \ {sgoal}  
InitializeState(s);  
if h(s) > c(s, s′) + h(s′)  
h(s) := c(s, s′) + h(s′);  
UpdateState2(s);

Figure 3.9: Pseudo Code of Forward Generalized Adaptive A* (Part 1)

the OPEN list (Line 46) and then checks all s ∈ Pred(s′) with s ≠ sgoal (Line 47). If h(s) > c(s, s′) + h(s′), then h(s) is set to c(s, s′) + h(s′) (Lines 49 - 50) and s is inserted into
function Main()
    counter := 1;
    s\_start := the current state of the agent;
    s\_goal := the current state of the target;
    deltah(1) := 0;
    last\_nopath := 0;
for all s \in S
    search(s) := 0;
while s\_start \neq s\_goal
    InitializeState(s\_start);
    InitializeState(s\_goal);
g(s\_start) := 0;
OPEN := \emptyset;
OPEN.Inset(s\_start, CalculateKey(s\_start));
if ComputePath() = true
    pathcost(counter) := g(s\_goal);
    while target not caught AND action costs do not change AND target on path from s\_start to s\_goal
        agent follows the path from s\_start to s\_goal;
        if agent caught target
            return true;
        s\_start := the current state of the agent;
        s\_newgoal := the current state of the target;
    if s\_goal \neq s\_newgoal
        InitializeState(s\_newgoal);
        if g(s\_newgoal) + h(s\_newgoal) < pathcost(counter)
            h(s\_newgoal) := pathcost(counter) - g(s\_newgoal);
            deltah(counter + 1) := deltah(counter) + h(s\_newgoal);
            s\_goal := s\_newgoal;
        else
            deltah(counter + 1) := deltah(counter);
            counter := counter + 1;
            ConsistencyProcedure();
else
    pathcost(counter) := \infty;
    last\_nopath := counter;
    deltah(counter + 1) := 0;
    counter := counter + 1;
wait until some action costs decrease;
update the increased and decreased action costs (if any);
s\_goal := the current state of the target;
return true;

Figure 3.10: Pseudo Code of Forward Generalized Adaptive A* (Part 2)

the OPEN list (with h(s) as its key) (Line 44). The consistency procedure terminates when the OPEN list is empty (Line 45). It is important to point out that Forward GAA* updates and corrects the h-values in a lazy way during each search. However, the consistency procedure currently updates the h-values in an eager way, which means that it is run between searches whenever action costs decrease. The reason is that after some action costs decrease, the consistency procedure decreases the h-values of the states that are directed affected by the action costs decrease (Lines 39 - 44) and inserts the states into
the \textit{OPEN} list (Line 44). These $h$-value decreases are then propagated to the predecessor states of the states in the \textit{OPEN} list by using a version of Dijkstra's algorithm (Lines 45 - 51) if the condition on Line 49 is satisfied. The $h$-values of those predecessor states then decrease (Lines 49 - 50). By executing the consistency procedure, the $h$-values of all states affected by the action costs decrease are updated (by propagating the $h$-value changes), and they cannot be updated by using Assignment (3.1) in a Lazy way. Thus, the consistency procedure can potentially update a large number of $h$-values that are not needed by future searches.

I have also tried to make the consistency procedure update the $h$-values in a lazy way by interleaving the consistency procedure with searches in order to speed up GAA*. However, the results showed that this variant of GAA* did not run faster than the current version of GAA* described in Figures 3.9 and 3.10. Thus, I only describe the main idea and skip the technical details of this variant.

The variant maintains two priority queues (1) the \textit{OPEN} list $\text{OPEN}_1$ for the searches (the key of each state in $\text{OPEN}_1$ is its $f$-value), and (2) the \textit{OPEN} list $\text{OPEN}_2$ for the consistency procedure (the key of each state in $\text{OPEN}_2$ is its $h$-value), since it needs to interleav the consistency procedure with searches. After each search, the variant only executes lines 36 - 44 of the consistency procedure after some action costs decrease and then starts a new search. During the new search, whenever a state $s$ is deleted from $\text{OPEN}_1$ for expansion (Line 26), if $h(s) \leq \text{OPEN}_2.\text{TopKey}()$, the variant expands state $s$ in the same way as GAA* (Lines 27 - 32). The reason is because the consistency procedure can only decrease the $h$-values but not increase them (Lines 42 - 43 and 49 - 50). Since $h(s) \leq \text{OPEN}_2.\text{TopKey}()$ on Line 26, it is guaranteed that $h(s)$ cannot
be decreased by the consistency procedure (Lines 45 - 51). Thus, it is safe for the variant to expand \( s \) with \( h(s) \) (Lines 27 - 32) before the consistency procedure terminates. Otherwise, the variant repeatedly executes lines 45 - 51 of the consistent procedure until \( h(s) \leq \text{OPEN}_2.\text{TopKey}() \) on Line 45. Then, the variant continues to expand state \( s \) (Lines 27 - 32).

I now describe the situation when there does not exist a path between the current states of the agent and target: If there does not exist a path in the \( \text{counter} \)-th search, Forward GAA* executes Lines 84 - 91: The reason Lines 84 - 91 are added is as follows: Assume Forward GAA* executes Lines 67 - 83 to deal with the situation when there does not exist a path after the \( \text{counter} \)-th search. Then, \( \text{pathcost}(\text{counter}) = \infty \) after the \( \text{counter} \)-th search (Line 67). \( h(s_{\text{newgoal}}) \) is then initialized to \( \infty \) since \( \text{pathcost}(\text{counter}) = \infty \) (Line 77). Thus, \( \text{deltah}(\text{counter} + 1) = \infty \) (Line 78), and the \( \text{deltah} \)-values will be \( \infty \) for all future searches since they cannot decrease (Lines 78 and 81). The consequence is that, whenever a state \( s \) with \( \text{search}(s) = \text{counter} \) is initialized in \( \text{InitializeState}(s) \) in a future search, \( \infty - \infty \) can be executed when calculating \( h(s) \) (Line 10), which causes a problem. To address this problem, Forward GAA* uses a variable \( \text{last_nopath} \) to remember the last search in which no path existed. All information before the \( \text{last_nopath} \)-th search (including the \( \text{last_nopath} \)-th search itself) is discarded: Forward GAA* resets the running sum \( \text{deltah}(\text{last_nopath} + 1) = 0 \) (Line 87) and uses the user-provided \( h \)-values for all states whose \( h \)-values have been calculated by \( \text{InitializeState}() \) last before and during the \( \text{last_nopath} \)-th search (Lines 04 - 06).

I now show that the \( h \)-values remain consistent with respect to the goal state after the consistency procedure.
Theorem 2 The $h$-values remain consistent with respect to the goal state after action cost increases and decreases if the consistency procedure is run.

Proof: Let $c$ denote the action costs before the changes and $c'$ denote the action costs after the changes. Let $h(s)$ denote the $h$-values before running the consistency procedure and $h'(s)$ denote the $h$-values after its termination. Thus, $h(s_{\text{goal}}) = 0$ and $h(s) \leq c(s, s') + h(s')$ for all $s \in S \setminus \{s_{\text{goal}}\}$ and all $s' \in Succ(s)$ since the $h$-values are consistent with respect to $s_{\text{goal}}$ before action cost changes. The $h$-value of $s_{\text{goal}}$ remains zero since it is never updated. The $h$-value of any $s \in S \setminus \{s_{\text{goal}}\}$ is monotonically non-increasing over time since it only gets updated to an $h$-value that is smaller than its current $h$-value (Lines 42 - 43 and 49 - 50). Thus, I distinguish three cases for any $s \in S \setminus s_{\text{goal}}$ and $s' \in Succ(s)$:

- First, the $h$-value of $s'$ never decreased during the consistency procedure (and thus $h(s') = h'(s')$) and $c(s, s') \leq c'(s, s')$. Then, $h'(s) \leq h(s) \leq c(s, s') + h(s') = c(s, s') + h'(s') \leq c'(s, s') + h'(s')$.

- Second, the $h$-value of $s'$ never decreased during the consistency procedure (and thus $h(s') = h'(s')$) and $c(s, s') > c'(s, s')$. Then, $c(s, s')$ decreased and $s \neq s_{\text{goal}}$ and Lines 39 - 43 were thus executed. Let $\tilde{h}(s)$ be the $h$-value of $s$ after the execution of Lines 39 - 44. Then, $h'(s) \leq \tilde{h}(s) \leq c'(s, s') + h'(s')$.

- Third, the $h$-value of $s'$ decreased during the consistency procedure and thus $h(s') > h'(s')$. Then, $s'$ was inserted into the priority queue and later retrieved on Line 46. Consider the last time that it was retrieved on Line 46. Then, Lines 47 - 51 were
executed. Let $\bar{h}(s)$ be the $h$-value of $s$ after the execution of Lines 47 - 51. Then,

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

Thus, $h'(s) \leq c'(s, s') + h'(s')$ in all three cases, and the $h$-values remain consistent with respect to the goal state.

3.2.4 Example

![Figure 3.11: Legend of Figures 3.12 and 3.13](image)

I now use an example problem of moving target search that extends the problem in Figure 2.9 to demonstrate the behavior of GAA*. Figures 3.12 and 3.13 illustrate the example problem solved with Forward Repeated A* and Forward GAA*, respectively. The real terrain and the current states of the agent and target are exactly the same as those in Figure 2.9: Initially, the current states of the agent and target are D2 and C4, respectively. The only difference is that the terrain now is a known dynamic terrain, where the action costs of the agent can both increase and decrease between searches. After the first search, the target moves from C4 to C3 (shown in Figures 3.12(b) and 3.13(b)), and B2 becomes unblocked, which changes the costs $c(B_1, B_2)$, $c(B_2, B_1)$, $c(A_2, B_2)$, $c(B_2, A_2)$, $c(B_3, B_2)$ and $c(B_2, B_3)$ from infinity to one (shown in Figures 3.12(c) and 3.13(c)).

Forward GAA* updates the $h$-values in a lazy way except for the consistency procedure. All states have their search-values in the lower right corner. States that have been
initialized by InitializeState() also have their $g$-values in the upper left corner, $h$-values in the lower left corner, and $f$-values in the upper right corner (shown in Figure 3.11). Both algorithms break ties among states with the same $f$-values in favor of states with larger $g$-values.

For the first search, Forward Repeated A* (shown in Figure 3.12(a)) and Forward GAA* (shown in Figure 3.13(a)) expand exactly the same states as they both use the user-provided $h$-values to guide the search. The expanded states are shaded grey.

After the first search, the agent moves along the cost-minimal path returned by the first search from D2 to D1, and the target moves from C4 to C3. Forward GAA* (shown in Figure 3.13(b)) calculates the $h$-value of the current state of the target C3 with respect
to the goal state of the first search C4, which is equal to one, and then adds it to a running sum of all corrections by executing $deltah(2) = deltah(1) + 1 = 1$. Forward GAA* then assigns the current state of the target C3 to the goal state of the second search. States initialized by InitializeState() are shaded grey (shown in Figure 3.13(b)). In contrast to Forward GAA*, Forward Repeated A* does not perform any operations to update $h$-values after the target moved (shown in Figure 3.12(b)).

When B2 becomes unblocked, Forward GAA* performs the consistency procedure to maintain the consistency of the $h$-values with respect to the goal state of the current search. States initialized by InitializeState() are shaded grey (shown in Figure 3.13(c)). In contrast to Forward GAA*, Forward Repeated A* does not perform any operations to update $h$-values after B2 becomes unblocked (shown in Figure 3.12(c)).

For the second search, Forward Repeated A* still uses the user-provided $h$-values, while Forward GAA* can use the updated $h$-values, which can be more informed. Figure 3.12(d) shows that Forward Repeated A* expands D2 because the user-provided $h$-values are misleading and form a local minimum. Forward GAA* avoids expanding D2 during the second search (shown in Figure 3.13(d)) resulting in a smaller number of state expansions (but at the cost of updating the $h$-values of states in the consistency procedure to maintain their consistency). Thus, it is important to evaluate the runtime of Forward GAA* compared to Forward Repeated A*. 
3.3 Summary

In this chapter, I reviewed MT-Adaptive A*, that extends Adaptive A* to moving target search: In Section 3.1.1, I first reviewed of an eager version of MT-Adaptive A*, referred to as Eager MT-Adaptive A*. In Section 3.1.2, I then reviewed a more complicated version, referred to as Lazy MT-Adaptive A*. Both versions of MT-Adaptive A* apply to moving target search, where the start and goal states can change between searches. However, they do not apply to moving target search in terrains where the action costs of the agent can decrease between searches. Thus, in Section 3.2, I introduced GAA* that extends Lazy MT-Adaptive A* to moving target search in terrains where the action costs of the agent can both increase and decrease between searches. GAA* is thus the first heuristic learning incremental search algorithm that applies to moving target search in all three classes of terrains (see Section 1.1.2).
Chapter 4: Incremental Search by Reusing Search Trees

In this chapter, I introduce search tree transforming incremental search algorithms for path planning for moving target search.

In Section 2.4.2.2, I reviewed D* Lite (Koenig & Likhachev, 2002), a state-of-the-art search tree transforming incremental search algorithm designed for stationary target search in all three classes of terrains (see Section 1.1.2). However, it is still unknown how to apply D* Lite to moving target search and make it run faster than Forward and Backward Repeated A*. Thus, in order to speed up search tree transforming incremental search algorithms and apply them to moving target search, I develop new search tree transforming incremental search algorithms, including Generalized Fringe-Retrieving A* (G-FRA*) (Sun, Yeoh, & Koenig, 2010a), Fringe-Retrieving A* (FRA*) (Sun et al., 2009), Basic Moving Target D* Lite (Basic MT-D* Lite) and Moving Target D* Lite

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Action Costs Can Increase Between Searches</th>
<th>Action Costs Can Decrease Between Searches</th>
<th>Start State Can Change Between Searches</th>
<th>Goal State Can Change Between Searches</th>
</tr>
</thead>
<tbody>
<tr>
<td>D* Lite</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>G-FRA* &amp; FRA*</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Basic MT-D* Lite &amp; MT-D* Lite</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.1: Search Problems that Search Tree Transforming Incremental Search Algorithms Apply to
(MT-D* Lite) (Sun, Yeoh, & Koenig, 2010b). Table 4.1 illustrates the search problems that D* Lite, G-FRA*, FRA*, Basic MT-D* Lite and MT-D* Lite apply to:

- D* Lite applies to search problems where the goal state can change and the action costs of the agent can change between searches. It does not apply to search problems where the start state can change between searches. For stationary target search, D* Lite can only perform backward (but not forward) searches by assigning the current state of the agent and target to the goal and start state of each search, respectively. D* Lite only applies to stationary target search in all three classes of terrains (see Section 1.1.2).

- G-FRA* applies to search problems where the start and goal states can change between searches in known static terrains, where the action costs of the agent do not change between searches. FRA* optimizes G-FRA* for search problems where the start and goal states can change between searches in known static grids, where the action costs of the agent do not change between searches. Both FRA* and G-FRA* require the start state of the current search to remain in the search tree of the previous search since they transform the search tree of the previous search to the search tree of the current search by reusing the part of search tree of the previous search rooted in the start state of the current search. Thus, they can only perform forward (but not backward) searches by assigning the current states of the agent and the target to the start and goal states of each search, respectively, since the current state of the agent can only move along the path found by the previous search and hence cannot move outside of the search tree of the previous search, while
the current state of the target can move to a state that is outside of the search tree of the previous search. If FRA* and G-FRA* perform backward searches and the target moves outside of the search tree of the previous search, then they cannot reuse any part of the search tree from the previous search. FRA* applies to moving target search in known static grids only, and G-FRA* applies to moving target search in known static terrains (see Section 1.1.2). Though FRA* applies only to moving target search in known static grids and G-FRA* applies only to moving target search in known static terrains, where the action costs of the agent do not change between searches, they provide a new way of transforming the search tree of the previous search to the search tree of the current search, which can be used to develop new incremental search algorithms that apply to moving target search in terrains where the action costs of the agent can change between searches, such as MT-D* Lite. FRA* and G-FRA* run up to one order of magnitude faster than both Repeated A* and GAA* for moving target search in known static terrains.

- Basic MT-D* Lite generalizes D* Lite to search problems where both the start and goal states can change, and the action costs of the agent can change between searches. MT-D* Lite combines the principles of G-FRA* and Basic MT-D* Lite to optimize Basic MT-D* Lite and applies to the same search problems as Basic MT-D* Lite. Both Basic MT-D* Lite and MT-D* Lite can only perform forward (but not backward) searches since they require the start state of the current search to remain in the search tree of the previous search, similar to G-FRA*. Basic MT-D* Lite and MT-D* Lite apply to moving target search in all three classes of terrains.
Basic MT-D* Lite runs faster than Repeated A* and GAA* by up to a factor of four and MT-D* Lite runs faster than Repeated A* and GAA* by up to a factor of eight for moving target search in known dynamic terrains.

The structure of this chapter is as follows: I introduce G-FRA* and its optimization FRA* in Section 4.1: First, I introduce G-FRA* in Section 4.1.1. Then, I introduce FRA* in Section 4.1.2. I introduce Basic MT-D* Lite and its optimization MT-D* Lite in Section 4.2: First, I introduce Basic MT-D* Lite in Section 4.2.1. Then, I introduce MT-D* Lite in Section 4.2.2.

4.1 Generalized Fringe-Retrieving A* and its Optimization

In this section, I introduce G-FRA* for moving target search in known static terrains. I then introduce FRA* that optimizes G-FRA* for moving target search in known static grids.

4.1.1 Generalized Fringe-Retrieving A*

G-FRA* (Sun et al., 2010a) is a new search tree transforming incremental search algorithm for moving target search in known static terrains.

4.1.1.1 Applicability

G-FRA* can only perform forward (but not backward) searches by assigning the current states of the agent and target to the start and goal states of each search, respectively. G-FRA* only applies to moving target search in known static terrains (see Section 1.1.2).
4.1.1.2 Principle

The principle behind G-FRA* is as follows: G-FRA* performs forward searches. After a search, A* Properties 4, 5 and 6 hold since G-FRA* uses A* for the search (= previous search). The agent then moves along the path towards the target. Thus, the current state of the agent must remain in the search tree and cannot move outside of the search tree of the previous search. Whenever the target moves off the path, G-FRA* performs a new search (= current search) by assigning the current states of the agent and target to the start and goal states of the current search again. Instead of performing the current search from scratch, G-FRA* transforms the search tree of the previous search to the search tree of the current search by reusing the part of search tree of the previous search rooted in the start state of the current search. The search tree of the previous search is given by the initial \( \textit{OPEN} \) and \( \textit{CLOSED} \) lists as well as the parent pointers and the \( g \)-values of the states in them. Thus, G-FRA* changes the initial \( \textit{OPEN} \) and \( \textit{CLOSED} \) lists as well as the \( g \)-values and parent pointers of the states in them to guarantee that A* Properties 4, 5 and 6 hold again after the search tree has been transformed, and then starts the current search with these initial \( \textit{OPEN} \) and \( \textit{CLOSED} \) lists to find a cost-minimal path from the start state of the current search to the goal state of the current search.

4.1.1.3 Operations

Figures 4.1 and 4.2 give the pseudo code of G-FRA*.\(^1\) Since G-FRA* applies only to terrains where the action costs of the agent do not change between searches, G-FRA* can terminate when no path exists between the start state and the goal state after a search.

---

\(^1\)In this dissertation, all pseudo codes use the following functions to manage the \( \textit{DELETED} \) list unless otherwise specified: \( \textit{DELETED}.\text{Insert}(s) \) inserts state \( s \) into the \( \textit{DELETED} \) list.
(Line 64), because it is then guaranteed that it will not be able to find any path in future searches. Similar to the pseudo code of A* (shown in Figure 2.4), G-FRA* initializes the $g$- and $h$-values of a state $s$ whenever they are needed during a search by executing InitializeState($s$) (Line 19). Different from the pseudo code of A*, G-FRA* maintains a CLOSED list that contains all states that have been expanded in previous searches that can be reused for the current search, and a DELETED list that contains all states that are deleted from the search tree of the previous search. Both the CLOSED and DELETED lists are needed to transform the search tree of the previous search to the search tree of the current search. The OPEN list is complete iff it satisfies the following properties: If the CLOSED list is empty, then the OPEN list contains only $s_{\text{start}}$. Otherwise, the OPEN list contains exactly all states that are not in the CLOSED list but have at least one predecessor state in the CLOSED list. The OPEN list is incomplete if does not satisfy the properties described above. G-FRA* maintains a variable $\text{open}\_\text{incomplete}$ to indicate whether the initial OPEN list is incomplete (Lines 65, 76, and 77). $\text{open}\_\text{incomplete}$ is set to false each time after ComputePath() has been performed (Line 65) since the OPEN list is complete after each search, and set to true each time after Step 2 (Lines 27 - 33) has been performed (Line 76) since deleting states from the initial OPEN and CLOSED lists can result in an incomplete OPEN list. G-FRA* terminates after the goal state is inserted into the CLOSED list (Line 16) and expanded (Lines 17 - 23).

The first search of G-FRA* is identical to an A* search, and starts from scratch to find a cost-minimal path from the start state to the goal state. The agent then moves along the path until it catches the target or the target moves off the path. In the latter
function CalculateKey(s)
  return g(s) + h(s);

procedure InitializeState(s)
  if search(s) ≠ counter
    g(s) := ∞;
    h(s) := H(s, sgoal);
    search(s) := counter;

procedure UpdateState(s)
  if s ∈ OPEN
    OPEN.Update(s, CalculateKey(s));
  else
    OPEN.Insert(s, CalculateKey(s));

function ComputePath()
  while OPEN ≠ ∅
    s := OPEN.Pop();
    CLOSED.Insert(s);
    for all s' ∈ Succ(s)
      if s' ∉ CLOSED
        InitializeState(s');
        if g(s') > g(s) + c(s, s')
          g(s') := g(s) + c(s, s');
          parent(s') := s;
          UpdateState(s');
    if s = sgoal
      return true;
  return false;

procedure Step2()
  parent(sstart) := NULL;
  for all s ∈ S in the subtree rooted in sstart /* After parent(sstart) := NULL, all states in the subtree rooted in sstart are not in the subtree rooted in sstart anymore. */
    parent(s) := NULL;
    if s ∈ OPEN then OPEN.Delete(s);
    if s ∈ CLOSED then CLOSED.Delete(s);
    DELETED.Insert(s);

procedure Step4()
  for all s ∈ OPEN
    h(s) := H(s, sgoal);
    UpdateState(s);
    search(s) := counter;
  for all s ∈ DELETED
    if ∃s' ∈ Pred(s) : s' ∈ CLOSED
      InitializeState(s);
      for all s' ∈ Pred(s)
        if s' ∈ CLOSED AND g(s) > g(s') + c(s', s)
          g(s) := g(s') + c(s', s);
          parent(s) := s';
          UpdateState(s);
    DELETED := ∅;

procedure Step5()
  for all s ∈ OPEN
    h(s) := H(s, sgoal);
  UpdateState(s);

Figure 4.1: Pseudo Code of Generalized Fringe-Retrieving A* (Part 1)

case, G-FRA* runs an A* search with initial OPEN and CLOSED lists obtained from
the previous search rather than from scratch.
function Main()
    counter := 1;
    s_{\text{start}} := \text{the current state of the agent};
    s_{\text{goal}} := \text{the current state of the target};
    for all \( s \in S \)
        search(s) := 0;
        InitializeState(s_{\text{start}});
        g(s_{\text{start}}) := 0;
        OPEN := CLOSED := DELETED := \emptyset;
        OPEN.Insert(s_{\text{start}}, \text{CalculateKey}(s_{\text{start}}));
        while \( s_{\text{start}} \neq s_{\text{goal}} \)
            if ComputePath() = false /* Step 6 */
                return false; /* Target cannot be caught */
            open_incomplete := false;
            while \( s_{\text{goal}} \in CLOSED \)
                if target not caught AND target on path from \( s_{\text{start}} \) to \( s_{\text{goal}} \)
                    agent follows the path from \( s_{\text{start}} \) to \( s_{\text{goal}} \);
                    return true;
                \( s_{\text{oldstart}} := s_{\text{start}}; \)
                \( s_{\text{start}} := \text{the current state of the agent}; \)
                \( s_{\text{goal}} := \text{the current state of the target}; \)
                if \( s_{\text{start}} \neq s_{\text{oldstart}} \)
                    Step2();
                open_incomplete := true;
                if open_incomplete
                    counter := counter + 1;
                else
                    Step5();
            return true;

Figure 4.2: Pseudo Code of Generalized Fringe-Retrieving A* (Part 2)

Figure 4.3: Operations of Generalized Fringe-Retrieving A*
Figure 4.3(a) visualizes the OPEN and CLOSED lists after the previous search, which are used as the initial OPEN and CLOSED lists that the following six steps manipulate.

\( S' \) and \( S \) represent the start states of the previous and current search, respectively, and
\( G' \) and \( G \) represent the goal states of the previous and current search, respectively.

- **Step 1 (Starting A* Immediately):** G-FRA* executes this step if the agent has not moved since ComputePath() was executed last (Line 63), and hence \( s_{\text{start}} = s_{\text{oldstart}} \), where \( s_{\text{oldstart}} \) is the start state when ComputePath() was executed last.
When Step 1 is executed, Step 2 (Deleting States) must have not been executed since ComputePath() was executed last, because $s_{\text{start}} = s_{\text{oldstart}}$ and the precondition of Step 2 (Line 74) is not satisfied. Consequently, $\texttt{open\_incomplete}$ is false (= the $\texttt{OPEN}$ list is complete) when Step 1 is executed, because $\texttt{open\_incomplete}$ was set to false (Line 65) after ComputePath() was executed last and the only place where $\texttt{open\_incomplete}$ is set to true is Line 76, which cannot be executed since its precondition $s_{\text{start}} \neq s_{\text{oldstart}}$ (Line 74) does not hold in this case. There are the following two cases in Step 1:

- Case 1 (Lines 66 - 73): If the goal state of the current search is in the initial $\texttt{CLOSED}$ list (Line 66), then the previous search already determined a cost-minimal path from the start state of the current search to the goal state of the current search, which can be identified in reverse by repeatedly following the parent pointers from the goal state of the current search to the start state of the current search (see A* Property 5 in Section 2.3.4). Then G-FRA* skips Steps 2 to 6.

- Case 2 (80 - 81): If the goal state of the current search is not in the initial $\texttt{CLOSED}$ list (Line 66), then the previous search can be continued to determine a cost-minimal path from the start state of the current search to the goal state of the current search. G-FRA* then skips Steps 2 to 4 and executes Step 5 to calculate the $h$-values of all states in the $\texttt{OPEN}$ list with respect to the goal state of the current search and update their $f$-values before the current search is run since the goal state has changed after the previous search (Line 81).
G-FRA* then starts the current search with the initial OPEN and CLOSED lists (Line 63) to determine a new cost-minimal path from the start state of the current search to the goal state of the current search.

- **Step 2 (Deleting States):** All states in the initial CLOSED list need to satisfy A* Property 4 with respect to the start state of the current search. After the previous search, the initial CLOSED list satisfies A* Property 4 with respect to the start state of the previous search since G-FRA* performs A* for the previous search. Before the current search, if the initial CLOSED list contains exactly all states of the subtree rooted in the start state of the current search, then A* Property 4 holds with respect to the start state of the current search. A* Property 4(a) holds trivially, and A* Property 4(b) holds for the following reason: Consider the cost-minimal path from the start state of the previous search \( s'_{\text{start}} \) to any state \( s \) in the initial CLOSED list that results from following the parent pointers from \( s \) to \( s'_{\text{start}} \) in reverse. Since the start state of the current search \( s_{\text{start}} \) is on this cost-minimal path, it holds that \( \text{dist}^*(s'_{\text{start}}, s) = \text{dist}^*(s'_{\text{start}}, s_{\text{start}}) + \text{dist}^*(s_{\text{start}}, s) \). Since \( s \) satisfies A* Property 4(b) with respect to \( s'_{\text{start}} \), it holds that \( g(s) = g(s'_{\text{start}}) + \text{dist}^*(s'_{\text{start}}, s) \). Since \( s_{\text{start}} \) satisfies A* Property 4(b) with respect to \( s'_{\text{start}} \), it holds that \( g(s_{\text{start}}) = g(s'_{\text{start}}) + \text{dist}^*(s'_{\text{start}}, s_{\text{start}}) \). Thus, \( s \) also satisfies A* Property 4(b) with respect to \( s_{\text{start}} \) since \( g(s) = g(s'_{\text{start}}) + \text{dist}^*(s'_{\text{start}}, s) = g(s_{\text{start}}) - \text{dist}^*(s'_{\text{start}}, s_{\text{start}}) + \text{dist}^*(s'_{\text{start}}, s) = g(s_{\text{start}}) + \text{dist}^*(s_{\text{start}}, s) \).

However, A* Property 4 might not hold for the states that are not in the subtree rooted in the start state of the current search. Thus, G-FRA* performs the following
procedure to delete those states from the search tree of the previous search: G-FRA* first sets the parent pointer of the start state of the current search to NULL (Line 28), then a path from the start state of the previous search to any state in the subtree rooted in the start state of the current search cannot be identified in reverse by repeatedly following the parent pointers from the state to the start state of the previous search. Thus, all states in the subtree rooted in the start state of the current search are not in the subtree rooted in the start state of the previous search. G-FRA* then iterates over all states in the subtree rooted in the start state of the previous search (Line 29):\footnote{One possible implementation of Line 29 is using Breadth-First Search (BFS), which maintains a first-in-first-out (FIFO) queue. Initially, the FIFO queue only contains the start state of the previous search. BFS repeats the following procedure until the FIFO queue is empty: It deletes a state \( s \) from the FIFO queue and expands \( s \) by performing the following operations for each \( s' \in \text{Succ}(s) \). If \( \text{parent}(s') = s \) then BFS generates \( s' \) by inserting \( s' \) into the FIFO queue.} G-FRA* sets the parent pointers of the states to NULL (Line 30), deletes the states from the initial OPEN and CLOSED lists if they were in the lists (Lines 31 - 32), and inserts them into the DELETED list (Line 33).

Figure 4.3(b) visualizes the initial OPEN and CLOSED lists after Step 2. The dotted line represents the states deleted from the initial OPEN list (called “deleted OPEN list”), the yellow area represents the states deleted from the initial CLOSED list (called “deleted CLOSED list”), the solid line represents the states remaining in the initial OPEN list (called “incomplete initial OPEN list”), and the blue area represents the states remaining in the initial CLOSED list (called “initial CLOSED list”).
• **Step 3 (Terminating Early):** If the goal state of the current search is in the initial \textit{CLOSED} list (Line 66), then G-FRA* executes in the same way as Case 1 of Step 1 (= determines a cost-minimal path from the start state of the current search to the goal state of the current search and skips the rest of the steps), except that the \textit{OPEN} list is complete in Case 1 of Step 1 and the \textit{OPEN} list might be incomplete in Step 3, since it might not contain exactly all states that are not in the initial \textit{CLOSED} list but have at least one predecessor state in the initial \textit{CLOSED} list after Step 2 has been executed (shown in Figure 4.3(b)). Thus, G-FRA* has to complete the \textit{OPEN} list in the following steps before performing the current search.

• **Step 4 (Inserting States):** The initial \textit{OPEN} list can be incomplete at the start of Step 4. Thus, Step 4 identifies the states that have to be inserted into the initial \textit{OPEN} list to make it complete. Step 4 iterates over all states in the \textit{DELETED} list and inserts those states that have a predecessor state in the initial \textit{CLOSED} list into the initial \textit{OPEN} list. The \textit{DELETED} list contains all states deleted from the initial \textit{OPEN} and \textit{CLOSED} lists in all executions of Step 2 since G-FRA* ran the last search (Line 33). G-FRA* completes the initial \textit{OPEN} list by performing the following check for all states in the \textit{DELETED} list before it sets the \textit{DELETED} list to empty again (Line 47): If the state has a predecessor state in the initial \textit{CLOSED} list (Line 40), then G-FRA* sets its parent pointer and \textit{g}-value according to A* Property 6 (see Section 2.3.4) and inserts it into the initial \textit{OPEN} list (Lines 41 - 46).
According to A* Property 6, the initial OPEN list has to contain all states that are not in the initial CLOSED list but have at least one predecessor state in the initial CLOSED list. The correctness of G-FRA* follows from the fact that every state that has a predecessor state in the initial CLOSED list when G-FRA* executes Step 4 was in the initial OPEN or CLOSED lists after the last search and thus is in the initial CLOSED list, the initial OPEN list or the DELETED list at the start of Step 4 (Line 34). I distinguish three cases:

- If a state is in the initial CLOSED list at the start of Step 4 (Line 34), then it was in the initial CLOSED list and thus not in the initial OPEN list after the last search and still is not in the initial OPEN list during Step 4. G-FRA* does nothing since the state should indeed not be in the initial OPEN list during Step 4.

- If a state is in the initial OPEN list at the start of Step 4 (Line 34), then it was in the initial OPEN list and thus was not in the initial CLOSED list but had at least one predecessor state in the initial CLOSED list after the last search and this still holds during Step 4 (otherwise it would have been deleted from the initial OPEN list in Step 2). G-FRA* does nothing since the state should indeed be in the initial OPEN list during Step 4 and its parent pointer and g-value still satisfy A* Property 6.

- If a state is in the DELETED list at the start of Step 4 (Line 34), then it was deleted from the initial OPEN or CLOSED lists in Step 2. Thus, if it has at least one predecessor state in the initial CLOSED list, G-FRA* inserts it
into the initial OPEN list in Step 4 and sets its parent pointer and $g$-value according to A* Property 6.

Figure 4.3(c) visualizes the initial OPEN and CLOSED lists after Step 4. The dotted line represents the inserted states that complete the initial OPEN list (called “inserted OPEN list”). The solid and dotted lines together represent the initial OPEN list.

After Step 4, the $h$-values of all states in the initial OPEN list have been updated with respect to the goal state of the current search, since (1) the $h$-values of all states that remained in the initial OPEN list before any state from the DELETED list was inserted into the initial OPEN list have been updated with respect to the goal state of the current search (Line 36), and (2) the $h$-values of all states that were inserted into the initial OPEN list in this step have been updated with respect to the goal state of the current search (Line 41) before the states were inserted into the initial OPEN list. G-FRA* then skips Step 5 and executes Step 6.

Note that the OPEN list, which, if it is implemented as a binary heap, requires the execution of heap operations to keep it sorted only once, namely at the end of Step 4, since it can remain unsorted until then.

• **Step 5 (Updating $h$-values):** G-FRA* executes this step only when the start state remains unchanged since ComputePath() was executed last and the goal state of the current search is not in the initial CLOSED list (see Step 1): Since the goal state of the current search has changed after the previous search, the $h$-values of all states in the initial OPEN list need to be updated with respect to the goal state
of the current search. G-FRA* also updates the $f$-values of all states in the initial OPEN list since they depend on the $h$-values (Lines 49 - 51).

- **Step 6 (Starting A*)**: G-FRA* then starts the current search with the initial OPEN and CLOSED lists (Line 63) to determine a new cost-minimal path from the start state of the current search to the goal state of the current search.

### 4.1.1.4 Example

![Figure 4.4: Legend of Figure 4.5](image)

I now use an example problem of moving target search that is extended from the problem in Figure 2.20 to demonstrate the behavior of G-FRA*. The real terrain and the current states of the agent and target are exactly the same as those in Figure 2.20: Initially, the current states of the agent and target are B1 (marked S) and D5 (marked G), respectively. The only difference is that the terrain now is a known static terrain, where the action costs of the agent do not change between searches. All states have their search-value in the lower right corner. States that have been initialized in InitializeState() also have their $g$-values in the upper left corner, $h$-values in the lower left corner, and $f$-values in the upper right corner (shown in Figure 4.4). G-FRA* breaks ties between states with the same $f$-values in favor of states with larger $g$-values. The outgoing arrows from states are the parent pointers of the states and point to their parents.
Figure 4.5: Example Trace of Generalized Fringe-Retrieving $A^*$

G-FRA* performs its first search from B1 to D5 (shown in Figure 4.5(a)), which starts from scratch. The expanded states (= states in the initial CLOSED list) are shaded grey. After the first search, the agent moves along the path from B1 to B2, and the target moves off the path from D5 to E5. G-FRA* then executes the following steps before the second search:

G-FRA* skips Step 1 and executes Step 2 since the agent moved along the path from B1 to B2, and thus the start state changed between searches. Thus, the conditions of Step 1 are not satisfied.

In Step 2 (shown in Figure 4.5(b)), G-FRA* deletes all states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search by deleting them from the initial OPEN and CLOSED lists. The state that is deleted from the initial OPEN list is A1, and the states that are deleted from the initial CLOSED list are B1, C1, C2, C3, D1, D2, D3, E1, E2 and E3. All states that are deleted have their parent pointer set to NULL and are inserted into the DELETED list. Thus, the DELETED list contains A1, B1, C1, C2, C3, D1, D2, D3, E1, E2, and E3.
G-FRA* skips Step 3 and executes Step 4, since the goal state of the current search changed after the previous search and is not in the initial CLOSED list. Thus, the conditions of Step 3 are not satisfied.

In Step 4 (shown in Figure 4.5(c)), G-FRA* completes the initial OPEN list by checking all states $s$ in the DELETED list: If $s$ has a predecessor state in the initial CLOSED list, then G-FRA* sets its parent pointer and $g$-value according to A* Property 6 (see Section 2.3.4) and inserts it into the initial OPEN list. After Step 4, B1, C2, and C3 have been inserted into the initial OPEN list with $g(B1) = 2$, $g(C2) = 2$, and $g(C3) = 3$, and $parent(B1) = B2$, $parent(C2) = B2$, and $parent(C3) = B3$.

G-FRA* skips Step 5 and executes Step 6, since the start state of the current search has changed after the previous search. Thus, the conditions of Step 5 are not satisfied.

In Step 6 (shown in Figure 4.5(d)), G-FRA* starts the second search, which performs 7 state expansions, namely C3, D3, E3, C2, D2, E2 and E5. On the other hand, A* would perform 17 state expansions if one performed A* from scratch for the second search.

### 4.1.2 Fringe-Retrieving A*

Fringe-Retrieving A* (FRA*) (Sun et al., 2009) is a new search tree transforming incremental search algorithm that optimizes G-FRA* for moving target search in known static grids (see Section 1.1.2).

#### 4.1.2.1 Applicability

Like G-FRA*, FRA* requires the start state of the current search to remain in the search tree of the previous search. Thus, FRA* can only perform forward (but not backward)
searches. FRA* applies only to moving target search in known static grids, since it uses geometric properties that are specific to grids and thus does not apply to other state space representations, such as state lattices (see Section 2.2.2).

4.1.2.2 Principle

FRA* and G-FRA* share the same principle: Whenever the target moves off the path, FRA* performs a new search (= current search) by assigning the current states of the agent and target to the start and goal states of the current search, respectively. Instead of performing the current search from scratch, FRA* transforms the search tree of the previous search, which is given by the initial OPEN and CLOSED lists as well as the parent pointers and g-values of the states in them, to the search tree of the current search. FRA* changes the initial OPEN and CLOSED lists as well as the parent pointers of the states in them to guarantee that A* Properties 4, 5 and 6 hold again and then starts the current search with these initial OPEN and CLOSED lists to find a cost-minimal path from the start state of the current search to the goal state of the current search.

The difference between FRA* and G-FRA* is that the initial CLOSED list of G-FRA* contains only the expanded states of the previous search that are in the subtree rooted in the start state of the current search, while FRA* uses geometric properties that are specific to grids to insert additional states into the initial CLOSED list and guarantee that A* Properties 4, 5 and 6 hold again after the insertions. This optimization enables FRA* to reuse more states from the previous search and thus perform fewer state expansions than G-FRA* in the current search, which runs faster than G-FRA* (Sun et al., 2009).
I now define some terminologies for the \textit{CLOSED} list of A* that are specific to grids. These terminologies will be used to describe the operations of FRA*:

- The inner perimeter of the \textit{CLOSED} list contains all states that are in the \textit{CLOSED} list and have at least one neighboring state that is not in the \textit{CLOSED} list.
- The outer perimeter of the \textit{CLOSED} list contains all states that are not in the \textit{CLOSED} list and share at least one corner with at least one state that is in the \textit{CLOSED} list.

A* Property 6 implies that the states in the \textit{OPEN} list belong to the outer perimeter of the \textit{CLOSED} list. Figure 4.6 illustrates what we call the inner and outer perimeters of the \textit{CLOSED} list (The \textit{CLOSED} list is shaded grey). The solid black line is the outer perimeter of the \textit{CLOSED} list, and the dashed black line is the inner perimeter of the \textit{CLOSED} list. The outer and inner perimeters consist of three parts each.

![Figure 4.6: Definition of Perimeter](image)

4.1.2.3 Operations

Figures 4.7 and 4.8 give the pseudo code of FRA*, that extends the pseudo code of G-FRA* in Figures 4.1 and 4.2. Similar to G-FRA*, FRA* initializes the \(g\)- and \(h\)-values of a state \(s\) whenever they are needed during a search by executing \texttt{InitializeState}(s) (Line
function CalculateKey(s)
return g(s) + h(s);

procedure InitializeState(s)
if search(s) ≠ counter
  g(s) := ∞;
  h(s) := H(s, s_goal);
  search(s) := counter;

procedure UpdateState(s)
if s ∈ OPEN
  OPEN.Update(s, CalculateKey(s));
else
  OPEN.Insert(s, CalculateKey(s));

function ComputePath()
while OPEN ≠ ∅
  s := OPEN.Pop();
  CLOSED.Insert(s);
  for all s' ∈ Succ(s)
    if s' ∉ CLOSED
      InitializeState(s');
    if g(s') > g(s) + c(s, s')
      g(s') := g(s) + c(s, s');
      parent(s') := s;
      UpdateState(s');
  if s = s_goal
    return true;
  return false;

function UpdateParent(direction)
for all s ∈ Succ(cell) in direction order, starting with parent(cell)
  if g(s) = g(cell) + c(cell, s) AND s ∈ CLOSED
    parent(s) := cell;
    cell := s;
    return true;
  return false;

procedure AdditionalStep()
cell := s_start;
while UpdateParent(counter-clockwise) /* body of while-loop is empty */;
cell := s_start;
while UpdateParent(clockwise) /* body of while-loop is empty */;
procedure Step2()
parent(s_start) := NULL;
for all s ∈ S in the subtree rooted in s_oldstart /* After parent(s_start) := NULL, all states in the subtree rooted in s_start are not in the subtree rooted in s_oldstart anymore. */
  parent(s) := NULL;
  if s ∈ OPEN then OPEN.Delete(s);
  if s ∈ CLOSED then CLOSED.Delete(s);

procedure Step4()
for all s ∈ S on the outer perimeter of CLOSED, starting with anchor
  if s is unblocked AND s ∉ OPEN AND ∃s' ∈ Pred(s) : s' ∈ CLOSED
    OPEN := OPEN ∪ {s};
  for all s ∈ OPEN
    InitializeState(s);
    for all s' ∈ Pred(s)
      if s' ∈ CLOSED AND g(s) > g(s') + c(s', s)
        g(s) := g(s') + c(s', s);
        parent(s) := s';
        UpdateState(s);

Figure 4.7: Pseudo Code of Fringe-Retrieving A* (Part 1)
procedure Step5()
for all $s \in OPEN$
\[ h(s) := H(s, s_{goal}) \]
UpdateState($s$);

function Main()
counter := 1;
$s_{start}$ := the current state of the agent;
$s_{goal}$ := the current state of the target;
for all $s \in S$
search($s$) := 0;
InitializeState($s_{start}$);
$g(s_{start}) := 0$;
$OPEN := CLOSED := \emptyset$;
$OPEN$.Insert($s_{start}$, CalculateKey($s_{start}$));
while $s_{start} \neq s_{goal}$
if ComputePath() = false /* Step 6 */
\[ \text{return false; /* Target cannot be caught */} \]
open_incomplete := false;
while $s_{goal} \in CLOSED$
if agent not caught AND target on path from $s_{start}$ to $s_{goal}$
agent follows the path from $s_{start}$ to $s_{goal}$;
if agent caught target
\[ \text{return true;} \]
$s_{oldstart} := s_{start}$;
$s_{start}$ := the current state of the agent;
$s_{goal}$ := the current state of the target;
if $s_{start} \neq s_{oldstart}$
AdditionalStep();
anchor := parent($s_{start}$);
Step2();
open_incomplete := true;
if open_incomplete
counter := counter + 1;
Step4();
else
Step5();
return true;

Figure 4.8: Pseudo Code of Fringe-Retrieving A* (Part 2)

19), and the main search routine is given in ComputePath() (Lines 13 - 26), which remains unchanged from G-FRA*. Different from G-FRA*, FRA* uses geometric properties that are specific to grids to insert more states into the initial CLOSED list and guarantee that A* Properties 4, 5 and 6 hold again after the insertions. There are two major changes from the pseudo code of G-FRA*: (1) FRA* performs an additional step between Step 1 and Step 2, named “Additional Step” to insert more states into the initial CLOSED list, and (2) FRA* performs a version of Step 4 that is different from Step 4 of G-FRA* to complete the initial OPEN list. Both Additional Step and Step 4 of FRA* use geometric properties that are specific to grids, and thus do not apply to other state space
representation, such as state lattices. I now show how FRA* obtains the initial OPEN and CLOSED lists with a focus on the difference between the operations of G-FRA* and FRA*:

- **Step 1 (Starting A* Immediately):** Step 1 of FRA* is the same as Step 1 of G-FRA*: FRA* executes this step if the agent has not moved (and hence $s_{\text{start}} = s_{\text{oldstart}}$) since ComputePath() was executed last (Line 71). There are the following two cases in Step 1:
  
  - Case 1 (Lines 74 - 81): If the goal state of the current search is in the initial CLOSED list, then FRA* skips Additional Step to Step 6.
  
  - Case 2 (Lines 90 - 91): If the goal state of the current search is not in the initial CLOSED list, then the previous search can be continued to determine a cost-minimal path from the start state of the current search to the goal state of the current search. FRA* then skips Additional Step to Step 4 and executes Step 5 to calculate the $h$-values of all states in the initial OPEN list with respect to the goal state of the current search and update their $f$-values before the current search is run since the goal state has changed after the previous search (Line 81). FRA* then starts the current search with the initial OPEN and CLOSED lists (Line 71) to determine a new cost-minimal path from the start state of the current search to the goal state of the current search.

- **Additional Step (Changing Parent Pointers):** The initial CLOSED list at the start of the current search should be as large a subset of the initial CLOSED list after the previous search as possible, so that the current search performs fewer
state expansions and runs faster. FRA* executes this step (which does not exist for G-FRA*) to insert additional states into the subtree rooted in the start state of the current search by using geometric properties that are specific to grids, so that there are more states in the initial \textit{CLOSED} list at the start of the current search: States from the initial \textit{CLOSED} list after the previous search often satisfy A* Property 4(b) with respect to the start state of the current search but not A* Property 4(a) since there are typically many alternative cost-minimal paths from the start state of the previous search to a state \(s\) in the initial \textit{CLOSED} list after the previous search. If the start state of the current search is on the cost-minimal path from the start state of the previous search to \(s\) that results from following the parent pointers from \(s\) to the start state of the previous search, then \(s\) is in the subtree rooted in the start state of the current search and thus in the initial \textit{CLOSED} list at the start of the current search. If the start state of the current search is not on the cost-minimal path, then \(s\) is not in the subtree rooted in the start state of the current search. FRA* finds such states and changes their parent pointers to make them part of the subtree rooted in the start state of the current search if possible so that they can be part of the initial \textit{CLOSED} list at the start of the current search.

First, FRA* makes the start state of the current search its current state \(s\) (Line 35), faces its parent and performs checks in the counter-clockwise direction. It turns counter-clockwise to face the next neighboring state \(s'\) of its current state \(s\) that is in the initial \textit{CLOSED} list and checks whether it holds that \(g(s') = g(s) + c(s, s')\) (Lines 28 - 29).
If the check is successful, then FRA* sets the parent of $s'$ to $s$ (Line 30), which is possible because this change does not affect its g-value. Due to this change, all states in the subtree rooted in $s'$ now belong to the subtree rooted in the start state of the current search and their g-values remain unaffected. FRA* then makes $s'$ its current state (Line 31), faces its new parent $s$ and repeats the process of turning and checking the neighboring state that it faces.

If the check is unsuccessful, then FRA* repeats the process of turning and checking the neighboring state that it faces.

If, during the process of turning and checking the neighboring state that it faces, FRA* faces the parent of its current state again, then it makes the start state of the current search its current state again (Line 37), faces its parent and now performs similar checks in the clockwise direction (Line 38). The resulting search tree is one that a search of A* could have generated if it had broken ties among states with the same $f$-value appropriately.

It is guaranteed that the process of checking the neighboring state terminates. This is because during the checks in the counter-clockwise direction, whenever making a state $s'$ the current state (except when $s'$ is set to the start state of the current search at the start of the process), its $g$-value must be larger than the $g$-value of the previous current state $s$ (Lines 28 - 29 and $c(s, s') > 0$). Thus, the $g$-value of the current state must increase monotonically during the process. Since the $g$-values of the states in the search tree must be finite, the process must terminate. Similar situation holds for the checks in the clockwise direction.
• **Step 2 (Deleting States):** Before Step 2, FRA* remembers the parent of the start state of the current search, called the anchor state, that will be used in Step 4 to complete the initial OPEN list (Line 84).

Step 2 of FRA* is similar to Step 2 of G-FRA*: FRA* first sets the parent pointer of the start state of the current search to NULL (Line 40), then all states in the subtree rooted in the start state of the current search are not in the subtree rooted in the start state of the previous search anymore. FRA* then iterates over all states in the subtree rooted in the start state of the previous search: FRA* sets the parent pointers of the states to NULL (Line 42) and deletes the states from the initial OPEN and CLOSED lists if they were in the lists (Lines 43 - 44). The only difference to G-FRA* is that FRA* does not maintain a DELETED list, and thus does not insert all deleted states into the DELETED list.

• **Step 3 (Terminating Early):** Step 3 of FRA* is as same as Step 3 of G-FRA*:

If the goal state of the current search is in the initial CLOSED list (Line 74), then the previous search already determined a cost-minimal path from the start state of the current search to the goal state of the current search (see A* Property 5 in Section 2.3.4). Thus, FRA* skips Steps 4 to 6.

• **Step 4 (Inserting States):** Step 4 of FRA* is different from Step 4 of G-FRA*:

This step uses geometric properties that are specific to grids: The initial OPEN list needs to contain exactly all states that are on the outer perimeter of the initial CLOSED list that have at least one predecessor state in the initial CLOSED list according to A* Property 6. The initial CLOSED list forms a contiguous area in
grids according to A* Property 5. FRA* thus completes the initial \textit{OPEN} list by circumnavigating the outer perimeter of the initial \textit{CLOSED} list that contains the anchor state, starting with the anchor state, and inserting every visited unblocked state that has at least one predecessor state (= neighboring state on the grid) in the initial \textit{CLOSED} list into the initial \textit{OPEN} list if the unblocked state is not yet in the initial \textit{OPEN} list (Lines 46 - 48). Afterwards, the initial \textit{OPEN} list is complete.\footnote{FRA* does not need to circumnavigate the outer perimeters of the initial \textit{CLOSED} list that do not contain the anchor state (if any) to complete the initial \textit{OPEN} list, because the states belong to those outer perimeters have not been deleted from the search tree of the previous search in Step 2, and remain in the initial \textit{OPEN} list after Steps 2 and 3.}

However, the parent pointers and g-values of some states in the initial \textit{OPEN} list might not satisfy A* Properties 6(a) and (b). FRA* therefore sets the parent of every state \(s\) in the initial \textit{OPEN} list to the state \(s'\) in the initial \textit{CLOSED} list that minimizes \(g(s') + c(s', s)\) and then the g-value of \(s\) to \(g(parent(s)) + c(parent(s), s)\) (Lines 49 - 54). The parent pointers and g-values of all states in the initial \textit{OPEN} list now satisfy A* Properties 6(a) and (b). FRA* also updates the \(f\)-values of all states in the initial \textit{OPEN} list using their current \(g\)- and \(h\)-values (Line 55).

The initial \textit{OPEN} list, which, if it is implemented as a binary heap, requires the execution of heap operations to keep it sorted only once, namely at the end of Step 4, since it can remain unsorted until then.

- **Step 5 (Updating \(h\)-values):** Step 5 of FRA* is as same as Step 5 of G-FRA*: If the start state remains unchanged since ComputePath() was executed last, and the goal state of the current search is not in the initial \textit{CLOSED} list, FRA* updates the \(h\)-values of all states in the initial \textit{OPEN} list with respect to the goal state of
the current search. G-FRA* also updates the $f$-values of all states in the initial
\textit{OPEN} list since their $f$-values depend on their current $h$-values (Lines 57 - 59).

- **Step 6 (Starting A\*)**: Step 6 of FRA* is as same as Step 6 of G-FRA*: It then
  starts the current search with the initial \textit{OPEN} and \textit{CLOSED} lists (Line 71) to
determine a new cost-minimal path from the start state of the current search to the
goal state of the current search.

4.1.2.4 Example

![Figure 4.9: Legend of Figure 4.10](image)

I now use the same example problem of moving target search as for G-FRA* (shown
in Figure 4.5) to demonstrate the behavior of FRA*. Initially, the current states of the
agent and target are B1 (marked S) and D5 (marked G), respectively. All states have their
\textit{search}-value in the lower right corner. States that have been initialized in InitializeState()
also have their $g$-values in the upper left corner, $h$-values in the lower left corner, and
$f$-values in the upper right corner (shown in Figure 4.9). FRA* breaks ties among states
with the same $f$-values in favor of states with larger $g$-values. The outgoing arrows from
states are the parent pointers of the states and point to their parents.

Like G-FRA*, FRA* performs its first search from B1 to D5 from scratch (shown in
Figure 4.10(a)). The expanded states (= states in the initial CLOSED list) are shaded
Figure 4.10: Example Trace of Fringe-Retrieving A*

This figure illustrates the process of fringe retrieving in the A* algorithm. The trace consists of several steps, each labeled with a counter indicating the order of execution.

(a) First Search
*counter = 1*

(b) Additional Step
*counter = 1*

(c) Step 2
*counter = 1*

(d) Step 4
*counter = 2*

(e) Step 6
(Second Search)
*counter = 2*

grey. After the first search, the agent moves along the path from B1 to B2, and the target moves off the path from D5 to E5. Then, FRA* executes the following steps before the second search:

FRA* skips Step 1 and executes Additional Step since the agent moved along the path from B1 to B2, and thus the start state changed between searches. Thus, the conditions of Step 1 are not satisfied.

In Additional Step (shown in Figure 4.10(b)), FRA* inserts states that were not in the subtree rooted in the start state of the current search into the subtree, so that there are more states in the initial CLOSED list at the start of the second search: Before this
step, the expanded states in the subtree rooted in the start state B2 of the current search and in the initial CLOSED list are A3, A4, A5, B2, B3, B5, C5 and D5 (shown in Figure 4.10(a)). However, C2 could have B2 as parent rather than C1 since this change does not affect its g-value. Then, all states in the subtree rooted in C2 belong to this subtree, and their g-values remain unaffected. FRA* starts at the start state of the current search B2 and performs checks in the counter-clockwise direction. First, FRA* is at B2 facing its parent B1 and checks C2 successfully. It then sets the parent of C2 to B2. Second, FRA* is at C2 facing its parent B2, checks C1 unsuccessfully and then checks D2 successfully. It then sets the parent of D2 to C2. Third, FRA* is at D2 facing its parent C2, checks D1 unsuccessfully and then checks E2 successfully. It then sets the parent of E2 to D2. Fourth, FRA* is at E2 facing its parent D2, checks E1 unsuccessfully and then checks E3 successfully. It then sets the parent of E3 to E2 (which does not change its parent). Fifth, FRA* is at E3 facing its parent E2 and checks D3 unsuccessfully. FRA* then starts again at the start state of the current search B2 and performs similar checks in the clockwise direction until it reaches D5. The states in the subtree rooted in the start state B2 of the current search that are in the initial CLOSED list after Additional Step are A3, A4, A5, B2, B3, B5, C2, C3, C5, D2, D3, D5, E2 and E3.

In Step 2 (shown in Figure 4.10(c)), FRA* deletes all states in the subtree rooted in the start state of the previous search but not in the subtree rooted in the start state of the current search by deleting them from the initial OPEN and CLOSED lists. The state that is deleted from the initial OPEN list is A1, and the states that are deleted from the initial CLOSED list are B1, C1, D1, and E1. All states that are deleted have their parent pointer set to NULL.
FRA* skips Step 3 and executes Step 4, since the goal state of the current search changed after the first search and is not in the initial CLOSED list. Thus, the conditions of Step 3 are not satisfied.

In Step 4 (shown in Figure 4.10(d)), FRA* completes the initial OPEN list by circumnavigating the outer perimeter of the initial CLOSED list that contains the anchor cell B1, starting with B1. The grey arrows point towards the states that FRA* visits during the circumnavigation. During the circumnavigation, FRA* inserts B1, C1, D1 and E1 into the initial OPEN list, since they have at least one predecessor state in the initial CLOSED list, and then corrects all of their parent pointers and g-values. For example, before this step, parent(C1) = NULL and g(C1) = 1. After this step, parent(C1) = C2 and g(C1) = 3.

FRA* skips Step 5 and executes Step 6, since the start state of the current search changed after the first search. Thus, the conditions of Step 5 are not satisfied.

In Step 6 (shown in Figure 4.10(d)), FRA* starts the second search, in which FRA* performs only one state expansion, namely E5. On the other hand, G-FRA* performs 7 state expansions, namely C3, D3, E3, C2, D2, E2 and E5 (shown in Figure 4.5(d)).

### 4.2 Moving Target D* Lite

In this section, I introduce Moving Target D* Lite for moving target search in terrains where the action costs of the agent can change between searches. I first introduce a simple version of Moving Target D* Lite, referred to as Basic Moving Target D* Lite (Basic MT-D* Lite), for moving target search in Section 4.2.1. I then introduce an optimized
version of Moving Target D* Lite, referred to as Moving Target D* Lite (MT-D* Lite), in Section 4.2.2, that runs faster than Basic MT-D* Lite.

### 4.2.1 Basic Moving Target D* Lite

Basic MT-D* Lite is a new search tree transforming incremental search algorithm for moving target search.

#### 4.2.1.1 Applicability

Basic MT-D* Lite requires the start state of the current search to remain in the search tree of the previous search. Thus, Basic MT-D* Lite can only perform forward (but not backward) searches by assigning the current states of the agent and target to the start and goal states of each search, respectively, since the current state of the agent can only move along the path found by the previous search and hence cannot move outside of the search tree of the previous search, while the current state of the target can move to a state that is outside of the search tree of the previous search. Basic MT-D* Lite applies to moving target search in all three classes of terrains (see Section 1.1.2).

#### 4.2.1.2 Principle

\[
\text{rhs}(s) = \begin{cases} 
    c & \text{if } s = s_{\text{start}} \quad (\text{Eq. 1'}) \\
    \min_{s' \in \text{Pred}(s)} (g(s') + c(s', s)) & \text{otherwise} \quad (\text{Eq. 2'})
\end{cases}
\]
\[ parent(s) = \begin{cases} 
  \text{NULL} & \text{if } s = s_{\text{start}} \text{ OR } \text{rhs}(s) = \infty \\
  \underset{s' \in \text{Pred}(s)}{\arg\min} \left( g(s') + c(s', s) \right) & \text{otherwise} 
\end{cases} \quad \text{(Eq. 3')} \]

Basic MT-D* Lite shares its principle with D* Lite, and it extends D* Lite to the situation where the start state can change between searches: D* Lite requires that all states satisfy Eqs. 1-4 described in Section 2.4.2.2. The correctness proof of D* Lite continues to hold if \( c \) is an arbitrary finite constant in Eq. 1. instead of zero. In Basic MT-D* Lite, if the start state changed between searches, the rhs-value of the start state of the current search can be an arbitrary finite value, including its current rhs-value, since its current rhs-value is finite.\(^4\) Since Basic MT-D* Lite performs forward searches while D* Lite performs backward searches, Basic MT-D* Lite simply converts Eqs. 1-4 described in Section 2.4.2.2 to Eqs. 1'-4', respectively.\(^5\) Basic MT-D* Lite therefore calculates the rhs-value of the previous start state (according to Eq. 2' since it is no longer the start state of the current search), its parent pointer (according to Eqs. 3'-4') and its membership in the OPEN list. Then, Basic MT-D* Lite performs a new search using similar operations to D* Lite.

\(^4\)The rhs-value of the start state was initialized to 0 (Line 11), and the rhs-value of the start state of the current search is guaranteed to be finite because (1) if there exists a path after the previous search, then the start state of the current search is on the cost-minimal path from the start state to the goal state of the previous search. The rhs-values of all states on this path are no larger than the rhs-value of the goal state of the previous search, which is finite due to Lines 45 - 46 of the Basic MT-D* Lite pseudo code, and (2) if there does not exist a path after the previous search, then the start state remains unchanged because the start state does not move in this case. Therefore, the rhs-value of the start state is still finite.

\(^5\)This conversion is straightforward, Basic MT-D* Lite checks the predecessor states rather than the successor states for setting the rhs-values and parent pointers of all states in Eqs. 2 and 4 when performing forward searches, resulting in Eqs. 2' and 4', respectively. Eqs. 1 and 3 are identical to Eqs. 1' and 3', respectively.
4.2.1.3 Operations

Figures 4.11 and 4.12 give the pseudo code of Basic MT-D* Lite that extends the pseudo code of D* Lite in Figures 2.17 and 2.18. Like D* Lite, Basic MT-D* Lite maintains an $h$-value ($= H(s, s_{goal})$), $g$-value, $rhs$-value, and parent pointer for every state $s$. The $rhs$-value is defined in a similar way to D* Lite (see Eqs. 1’ and 2’). The main difference is that $c$ can be an arbitrary finite constant during each search for Basic MT-D* Lite, while $c$ must be 0 during each search for D* Lite (see Section 2.4.2.2). The parent pointer of state $s$ is also defined in a similar way to D* Lite (see Eqs. 3’ and 4’). The main search routine is ComputePath() (Lines 20 - 55), which is similar to ComputePath() in D* Lite. The main differences of the operations between D* Lite and Basic MT-D* Lite are caused by the change in search direction:

- When expanding a state $s$, Basic MT-D* Lite needs to check the successor states (rather than the predecessor states) of $s$ (Lines 30 and 37).

- When calculating the user-provided $h$-value of a state $s$, Basic MT-D* Lite uses $H(s, s_{goal})$ (rather than $H(s_{goal}, s)$) (Line 02).

- When calculating $k_m$, Basic MT-D* Lite uses $k_m := k_m + H(s_{goal}, s_{oldgoal})$ (rather than $k_m := k_m + H(s_{oldgoal}, s_{goal})$) (Line 70).

ComputePath() determines a cost-minimal path from the start state to the goal state of each search. Before it is executed, states can have arbitrary $g$-values but their $rhs$-values and parent pointers have to satisfy Eqs. 1’-4’ and the OPEN list has to contain all locally inconsistent states. The main necessary modifications to the pseudo code of D* Lite in Figures 2.17 and 2.18 are as follows:
01 function CalculateKey(s)
02 return \(\min(g(s), \text{rhs}(s)) + H(s, s_{goal}) + k_m; \min(g(s), \text{rhs}(s))\);

03 procedure Initialize()
04 OPEN := \emptyset;
05 \(k_m := 0\);
06 for all \(s \in S\)
07 \(\text{rhs}(s) := g(s) := \infty\);
08 parent(s) := NULL;
09 \(s_{start} := \) the current state of the agent;
10 \(s_{goal} := \) the current state of the target;
11 \(\text{rhs}(s_{start}) := 0\);
12 OPEN.Insert(\(s_{start}, \text{CalculateKey}(s_{start})\));

13 procedure UpdateState(u)
14 if \(g(u) \neq \text{rhs}(u)\) AND \(u \in \text{OPEN}\)
15 OPEN.Update(u, \text{CalculateKey}(u));
16 else if \(g(u) \neq \text{rhs}(u)\) AND \(u \notin \text{OPEN}\)
17 OPEN.Insert(u, \text{CalculateKey}(u));
18 else if \(g(u) = \text{rhs}(u)\) AND \(u \in \text{OPEN}\)
19 OPEN.Delete(u);

20 function ComputePath()
21 while \(\text{OPEN}.\text{TopKey()} < \text{CalculateKey}(s_{goal})\) OR \(\text{rhs}(s_{goal}) > g(s_{goal})\)
22 \(u := \text{OPEN}.\text{Top();}\)
23 \(k_{old} := \text{OPEN}.\text{TopKey();}\)
24 \(k_{new} := \text{CalculateKey}(u);\)
25 if \(k_{old} < k_{new}\)
26 OPEN.Update(u, \(k_{new}\));
27 else if \(g(u) > \text{rhs}(u)\)
28 \(g(u) := \text{rhs}(u);\)
29 OPEN.Delete(u);
30 for all \(s \in \text{Succ}(u)\)
31 if \(s \neq s_{start}\) AND \(\text{rhs}(s) > g(u) + c(u, s)\)
32 \(\text{parent}(s) := u;\)
33 \(\text{rhs}(s) := g(u) + c(u, s);\)
34 UpdateState(s);
35 else
36 \(g(u) := \infty;\)
37 for all \(s \in \text{Succ}(u) \cup \{u\}\)
38 if \(s \neq s_{start}\) AND \(\text{parent}(s) = u\)
39 \(\text{rhs}(s) := \min_{s' \in \text{Pred}(s)}(g(s') + c(s', s));\)
40 if \(\text{rhs}(s) = \infty\)
41 \(\text{parent}(s) := \text{NULL;}\)
42 else
43 \(\text{parent}(s) := \arg \min_{s' \in \text{Pred}(s)}(g(s') + c(s', s));\)
44 UpdateState(s);
45 if \(\text{rhs}(s_{goal}) = \infty\)
46 return false;
47 return true;

48 procedure BasicDeletion()
49 parent(s_{start}) := \text{NULL;}
50 \(\text{rhs}(s_{\text{oldstart}}) := \min_{s' \in \text{Pred}(s_{\text{oldstart}})}(g(s') + c(s', s_{\text{oldstart}}));\)
51 if \(\text{rhs}(s_{\text{oldstart}}) = \infty\)
52 parent(s_{\text{oldstart}}) := \text{NULL;}
53 else
54 \(\text{parent}(s_{\text{oldstart}}) := \arg \min_{s' \in \text{Pred}(s_{\text{oldstart}})}(g(s') + c(s', s_{\text{oldstart}}));\)
55 UpdateState(s_{\text{oldstart}});

Figure 4.11: Pseudo Code of Basic Moving Target D* Lite (Part 1)
function Main()
Initilize();
while s\_start \neq s\_goal
    s\_oldstart := s\_start;
    s\_oldgoal := s\_goal;
    if ComputePath() = true
        while target not caught AND target on path from s\_start to s\_goal AND action costs do not change
            agent follows path from s\_start to s\_goal;
            if agent caught target
                return true;
        s\_start := the current state of the agent;
    else
        wait until some action costs decrease;
        s\_goal := the current state of the target;
        km := km + H(s\_goal * s\_oldgoal);
        if (s\_oldstart \neq s\_start)
            BasicDeletion();
        for all actions whose costs changed from c'(u,v) to c(u,v)
            if c'(u,v) > c(u,v)
                if v \neq s\_start AND rhs(v) > g(u) + c(u,v)
                    parent(v) := u;
                    rhs(v) := g(u) + c(u,v);
                    UpdateState(v);
                else
                    if v \neq s\_start AND parent(v) = u
                        rhs(v) := min_{s' \in \text{Pred}(v)} (g(s') + c(s',v));
                        if rhs(v) = \infty
                            parent(v) := NULL;
                        else
                            parent(v) := arg \min_{s' \in \text{Pred}(v)} (g(s') + c(s',v));
                            UpdateState(v);
            return true;

Figure 4.12: Pseudo Code of Basic Moving Target D* Lite (Part 2)

• First, D* Lite performs backward searches. On the other hand, Basic MT-D* Lite performs forward searches by assigning the current states of the agent and target to the start and goal states of each search, respectively (Lines 09, 10, 66 and 69). This is so because, like G-FRA*, Basic MT-D* Lite requires the start state of the current search to remain in the search tree of the previous search to reuse the search tree. If Basic MT-D* Lite performed backward searches, the start state (= the state of the target) might move outside of the search tree of the previous search. Basic MT-D* Lite might not reuse any part of the search tree from the previous search when the start state moves outside of the search tree of the previous search.
• Second, D* Lite requires the start state to remain unchanged between searches. On the other hand, the start state of Basic MT-D* Lite can change between searches. If the agent moved after the previous search, and thus the start state of the current search is different from the start state of the previous search (Line 71), Basic MT-D* Lite executes BasicDeletion(), which is a new procedure that does not exist for D* Lite: Before ComputePath() is executed, the rhs-values and parent pointers of all states have to satisfy Eqs. 1’-4’ and the OPEN list has to contain all locally inconsistent states. Fortunately, the rhs-values and parent pointers of all states already satisfy these invariants, with the possible exception of the start states of the previous and current searches. Basic MT-D* Lite therefore calculates the rhs-value of the start state of the previous search (according to Eq. 2’ since it is no longer the start state of the current search), its parent pointer (according to Eqs. 3’-4’) and its membership in the OPEN list (Lines 50 - 55). The correctness proofs of D* Lite continue to hold if c is an arbitrary finite value in Eq. 1’. instead of zero (see Section 4.2.1.2). Thus, the rhs-value of the start state of the current search can be an arbitrary finite value, including its current rhs-value, since its current rhs-value is finite. Basic MT-D* Lite therefore does not change the rhs-value of the start state of the current search nor its membership in the OPEN list and only sets its parent pointer to NULL (according to Eq. 3’) (Line 49).
4.2.1.4 Example

I now use an example problem of moving target search that extends the problem in Figure 4.5 to demonstrate the behavior of Basic MT-D* Lite. The only difference is that the terrain now is a known dynamic terrain. Initially, the current states of the agent and target are B1 (marked S) and D5 (marked G), respectively. All unblocked states have their $g$-values in the upper left corner and their $rhs$-values in the upper right corner (shown in Figure 4.13). For ease of illustration, I use $h$-values that are zero for all states and thus do not show them in each state. The key of each state is thus the minimum of its $g$- and $rhs$-values. The outgoing arrows from states are the parent pointers of the states and point to their parents.

Basic MT-D* Lite performs its first search from B1 to D5 (shown in Figure 4.14(a)), which starts from scratch. The cost-minimal path is B1, B2, B3, A3, A4, A5, B5, C5, and D5. The agent then moves along the path to B2, the target moves off the path to E5, and B4 becomes unblocked, which changes the action costs of the agent $c(B3,B4)$, $c(B4,B3)$, $c(B4,B5)$, $c(B5,B4)$, $c(B4,A4)$ and $c(A4,B4)$ from infinity to one (shown in Figure 4.14(b)). Since the target moved off the path and the action costs of the agent changed, Basic MT-D* Lite finds a cost-minimal path from the current state of the agent B2 to the current state of the target E5 using the following steps. Basic MT-D* Lite

---

6It is possible to execute BasicDeletion() after updating the action costs of the agent (by executing Lines 71 - 72 after Line 86) without affecting the correctness of Basic MT-D* Lite, since Eqs. 1'-4' still hold for all states.
(a) First Search (b) After Target Moved and B4 unblocked (c) After BasicDeletion and action costs updated

(e) After A1 Expanded (f) After C1 Expanded (g) After D1 Expanded (h) After B1 Expanded

(i) After E1 Expanded (j) After C1 Expanded (k) After A1 Expanded (l) After B4 Expanded

(m) After D1 Expanded (n) After B5 Expanded (o) After E1 Expanded (p) After C5 Expanded

(q) After D5 Expanded

Figure 4.14: Example Trace of Basic Moving Target D* Lite
sets the parent pointer of the start state of the current search B2 to NULL, updates the rhs-value and parent pointer of the start state of the previous search B1 to 2 and B2, respectively, and inserts B1 into the OPEN list (shown in Figure 4.14(c)). Basic MT-D* Lite then starts the second search, which performs 14 state expansions, namely B1, A1, C1, D1, B1, E1, C1, A1, B4, D1, B5, E1, C5 and D5 (shown in Figures 4.14(d-q)). The cost-minimal path is B2, B3, B4, B5, C5, D5 and E5.

All states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search, namely A1, B1, C1, D1 and E1, are expanded twice during the second search. Expanding these states twice can be computationally expensive (for example, expanding a state in D* Lite and its variants involves many operations for updating the g- and rhs-values of the state and its successor states, and updating their keys and memberships in the OPEN list), which is the issue addressed by Moving Target D* Lite (MT-D* Lite).

### 4.2.2 Moving Target D* Lite

Moving Target D* Lite (MT-D* Lite) is a new search tree transforming incremental search algorithm for moving target search that optimizes Basic MT-D* Lite.

#### 4.2.2.1 Applicability

Like Basic MT-D* Lite, MT-D* Lite can only perform forward (but not backward) searches by assigning the current states of the agent and target to the start and goal states of each search, respectively. MT-D* Lite applies to moving target search in all three classes of terrains (see Section 1.1.2).
4.2.2.2 Principle

MT-D* Lite shares its principle with Basic MT-D* Lite, except that Basic MT-D* Lite uses BasicDeletion() to lazily expand the states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search to set their $g$-values to infinity one at a time when needed. Whenever it sets the $g$-value of a state to infinity, it needs to update the $rhs$-values, parent pointers and memberships in the OPEN list for all successor states of the state, so that all states satisfy Eqs. 1'-4'. Expanding the states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search can be computationally expensive. On the other hand, MT-D* Lite uses OptimizedDeletion(), a more sophisticated procedure that combines the principles of Basic MT-D* Lite and G-FRA* to delete all states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search (= deleted states) by eagerly setting their $g$- and $rhs$-values to infinity and setting their parent pointers to $NULL$, so that all states satisfy Eqs. 1'-4'. MT-D* Lite avoids expanding the deleted states in the current search twice and thus can be computationally less expensive than Basic MT-D* Lite.

4.2.2.3 Operations

Figures 4.15 and 4.16 give the pseudo code of MT-D* Lite that extends the pseudo code of Basic MT-D* Lite in Figures 4.11 and 4.12. There is only one modification to the pseudo code of Basic MT-D* Lite, namely that MT-D* Lite uses OptimizedDeletion() (Lines 48 - 60) instead of BasicDeletion() to ensure that all states satisfy Eqs. 1'-4' after the start state has changed between searches:
01 function CalculateKey(s)
02 return \( \min(g(s), rhs(s)) + H(s, s_{\text{goal}}) + k_m: \min(g(s), rhs(s)) \);  
03  
04 procedure Initialize()
05 OPEN := \emptyset;  
06 \( k_m := 0; \)  
07 for all \( s \in S \)
08 \( rhs(s) := g(s) := \infty; \)
09 \( \text{parent}(s) := \text{NULL}; \)
10 \( s_{\text{start}} := \) the current state of the agent;
11 \( s_{\text{goal}} := \) the current state of the target;
12 \( rhs(s_{\text{start}}) := 0; \)
13 OPEN.Insert(\( s_{\text{start}}, \text{CalculateKey}(s_{\text{start}}); \))
14  
15 procedure UpdateState(u)
16 if \( g(u) \neq rhs(u) \) AND \( u \in OPEN \)
17 OPEN.Update(u, CalculateKey(u));
18 else if \( g(u) \neq rhs(u) \) AND \( u \notin OPEN \)
19 OPEN.Insert(u, CalculateKey(u));
20 else if \( g(u) = rhs(u) \) AND \( u \in OPEN \)
21 OPEN.Delete(u);
22  
23 function ComputePath()
24 while \( OPEN.\text{TopKey}() \not< \text{CalculateKey}(s_{\text{goal}}) \) OR \( rhs(s_{\text{goal}}) > g(s_{\text{goal}}) \)
25 \( u := OPEN.\text{Top}(); \)
26 \( k_{\text{old}} := OPEN.\text{TopKey}(); \)
27 \( k_{\text{new}} := \text{CalculateKey}(u); \)
28 if \( k_{\text{old}} < k_{\text{new}} \)
29 OPEN.Update(u, k_{\text{new}});
30 else if \( g(u) > rhs(u) \)
31 \( g(u) := rhs(u); \)
32 OPEN.Delete(u);
33  
34 for all \( s \in \text{Succ}(u) \)
35 if \( s \neq s_{\text{start}} \) AND \( rhs(s) > g(u) + c(u, s) \)
36 \( \text{parent}(s) := u; \)
37 \( rhs(s) := g(u) + c(u, s); \)
38 UpdateState(s);
39 else
40 \( g(u) := \infty; \)
41 for all \( s \in \text{Succ}(u) \cup \{u\} \)
42 if \( s \neq s_{\text{start}} \) AND \( \text{parent}(s) = u \)
43 \( rhs(s) := \min_{s' \in \text{Pred}(s)}(g(s') + c(s', s)); \)
44 if \( rhs(s) = \infty \)
45 \( \text{parent}(s) := \text{NULL}; \)
46 else
47 \( \text{parent}(s) := \arg \min_{s' \in \text{Pred}(s)}(g(s') + c(s', s)); \)
48 UpdateState(s);
49 if \( rhs(s_{\text{goal}}) = \infty \)
50 return false;
51 return true;
52  
53 procedure OptimizedDeletion()
54 DELETED := \emptyset;
55 parent(s_{\text{start}}) := \text{NULL};
56 for all \( s \in S \) that belong to the search tree rooted in \( s_{\text{oldstart}} \) but not the subtree rooted in \( s_{\text{start}} \)
57 \( \text{parent}(s) := \text{NULL}; \)
58 \( rhs(s) := g(s) := \infty; \)
59 insert \( s \) into \( DELETED; \)
60 for all \( s \in DELETED \)
61 for all \( s' \in \text{Pred}(s) \)
62 if \( rhs(s) > g(s') + c(s', s) \)
63 \( rhs(s) := g(s') + c(s', s); \)
64 \( \text{parent}(s) := s'; \)
65 UpdateState(s);
66  
Figure 4.15: Pseudo Code of Moving Target D* Lite (Part 1)
function Main()
Initialize();
while $s_{\text{start}} \neq s_{\text{goal}}$
    $s_{\text{oldstart}} := s_{\text{start}}$;
    $s_{\text{oldgoal}} := s_{\text{goal}}$;
    if ComputePath() = true
        while target not caught AND target on path from $s_{\text{start}}$ to $s_{\text{goal}}$ AND action costs do not change
            agent follows path from $s_{\text{start}}$ to $s_{\text{goal}}$;
            if agent caught target
                return true;
            $s_{\text{start}} :=$ the current state of the agent;
        else
            wait until some action costs decrease;
            $s_{\text{goal}} :=$ the current state of the target;
    $k_m := k_m + H(s_{\text{goal}}^*, s_{\text{oldgoal}})$;
    if ($s_{\text{oldstart}} \neq s_{\text{start}}$) OptimizedDeletion();
    for all actions whose costs changed from $c'(u, v)$ to $c(u, v)$
        if $c'(u, v) > c(u, v)$
            $\text{parent}(v) := u$;
            $\text{rhs}(v) := g(u) + c(u, v)$;
            UpdateState(v);
        else
            if $v \neq s_{\text{start}}$ AND $\text{parent}(v) = u$
                $\text{rhs}(v) := \min_{s' \in \text{Pred}(v)} (g(s') + c(s', v))$;
            if $\text{rhs}(v) = \infty$
                $\text{parent}(v) := \text{NULL}$;
            else
                $\text{parent}(v) := \arg \min_{s' \in \text{Pred}(v)} (g(s') + c(s', v))$;
            UpdateState(v);
    return true;

Figure 4.16: Pseudo Code of Moving Target D* Lite (Part 2)

MT-D* Lite maintains a \textit{DELETED} list in OptimizedDeletion(). The \textit{DELETED} list contains all states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search, which is exactly the definition that G-FRA* uses. First, MT-D* Lite sets the parent pointer of the start state of the current search to \textit{NULL} (Line 50). Then, MT-D* Lite executes two phases to ensure that all states satisfy Eqs. 1'-4':

- Phase 1 (Lines 51 - 54): MT-D* Lite eagerly sets the parent pointers of all states in the search tree of the previous search that are not in the subtree rooted in the
start state of the current search to NULL, sets their $g$- and $rhs$-values to infinity and inserts them into the \textit{DELETED} list in one pass (Lines 51 - 54).

- Phase 2 (Lines 55 - 60): MT-D* Lite then updates the $rhs$-values according to Eqs. 1'-2', parent pointers according to Eqs. 3'-4', and memberships in the \textit{OPEN} list according to the definition of local consistency for all states in the \textit{DELETED} list in one pass (Lines 55 - 60). Their $rhs$-values, parent pointers and memberships in the \textit{OPEN} list can be updated in one pass since they depend only on the $g$-values of their predecessor states, which do not change in Phase 2.

Overall, Phase 1 is similar to Step 2 (Deleting States) of G-FRA*, and Phase 2 is similar to Step 4 (Inserting States) of G-FRA*. Phases 1 and 2 iterate over all states in the \textit{DELETED} list. Phase 2 also iterates over all predecessor states of the states in the \textit{DELETED} list. Thus, each state in the \textit{DELETED} list can require $n$ operations on $n$-neighbor gridworlds. Both phases manipulate the \textit{OPEN} list, which, if it is implemented as a binary heap, requires the execution of heap operations to keep it sorted only once, namely at the end of Phase 2, since it can remain unsorted until then.

### 4.2.2.4 Example

![Legend of Figure 4.18](image)

Figure 4.17: Legend of Figure 4.18

I now use the same example problem of moving target search as used for Basic MT-D* Lite (shown in Figure 4.14) to demonstrate the behavior of MT-D* Lite. Initially, the
current states of the agent and target are B1 (marked S) and D5 (marked G), respectively. All unblocked states have their $g$-values in the upper left corner and their $rhs$-values in the upper right corner (shown in Figure 4.17). For ease of illustration, I use $h$-values that are zero for all states and thus do not show them in each state. The key of each state is thus the minimum of the $g$- and $rhs$-values of the state. The outgoing arrows from states are the parent pointers of the states and point to their parents.
MT-D* Lite runs its first search from B1 to D5 (shown in Figure 4.18(a)), which is exactly the same as for Basic MT-D* Lite (shown in Figure 4.14(a)). The cost-minimal path is B1, B2, B3, A3, A4, A5, B5, C5, and D5. The agent then moves along the path to B2, and the target moves off the path to E5 and B4 becomes unblocked, which changes the action costs $c(B3,B4)$, $c(B4,B3)$, $c(B4,B5)$, $c(B5,B4)$, $c(B4,A4)$ and $c(A4,B4)$ from infinity to one (shown in Figure 4.18(b)). Since the target moved off the path and the action costs of the agent changed, MT-D* Lite finds a cost-minimal path from the current state of the agent B2 to the current state of the target E5 using the following steps.

Different from Basic MT-D* Lite that lazily expands the states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search to set their $g$-values, $rhs$-values and parent pointers and updates their memberships in the OPEN list one at a time, so that all states satisfy Eqs. 1'-4' (see Section 2.4.2.2), MT-D* Lite uses OptimizedDeletion() to eagerly set the $g$-values, $rhs$-values and parent pointers and update the memberships in the OPEN list of these states, so that they satisfy Eqs. 1'-4', which can be computationally less expensive. (For example, in order to expand a state, Basic MT-D* Lite needs to not only set the $g$-value of the state itself, but also update the $rhs$-values of all successor states of that state.) First, MT-D* Lite sets the parent pointer of the start state of the current search B2 to NULL. Second, in Phase 1, MT-D* Lite deletes all states in the search tree of the previous search that are not in the subtree rooted in the start state of the current search, namely A1, B1, C1, D1, and E1, by setting their $g$- and $rhs$-values to infinity, setting their parent pointers to NULL and inserting them into the DELETED list. Third, in Phase 2, MT-D* Lite updates the $rhs$-values according to Eqs. 1'-2' and parent pointers according to Eqs. 3'-4'
for all states in the \textit{DELETED} list. For example, $rhs(B1) = 2$ and $parent(B1) = B2$.

Then, it inserts A1, B1, C1, D1, and E1 into the \textit{OPEN} list, since they have become locally inconsistent.

MT-D* Lite then starts the second search, which performs 9 state expansions (compared to 14 state expansions performed by Basic MT-D* Lite in Figures 4.14(d-q)), namely B1, A1, C1, B4, D1, B5, E1, C5 and D5 (shown in Figures 4.18(d-l)). The cost-minimal path is B2, B3, B4, B5, C5, D5 and E5.

\subsection*{4.3 Summary}

In this chapter, I introduced new search tree transforming incremental search algorithms for moving target search. In Section 4.1.1, I introduced G-FRA*, a new search tree transforming incremental search algorithm for moving target search in known static terrains. In Section 4.1.2, I introduced FRA*, that uses geometric properties specific to grids to optimize G-FRA* for moving target search in known static grids. Both G-FRA* and FRA* transform the search tree of the previous search to the search tree of the current search by reusing the subtree rooted in the start state of the current search. Thus, they can be computationally more efficient than Repeated A* if a large number of states belong to the subtree rooted in the start state of the current search. In Section 4.2.1, I introduced Basic MT-D* Lite, a new search tree transforming incremental search algorithm that generalizes D* Lite to moving target search. In Section 4.2.2, I introduced MT-D* Lite that optimizes Basic MT-D* Lite by combining the principles of G-FRA* and Basic MT-D* Lite. Similar to G-FRA* and FRA*, both Basic MT-D* Lite and MT-D* Lite
transform the search tree of the previous search to the search tree of the current search by reusing the subtree rooted in the start state of the current search. Thus, Basic MT-D* Lite and MT-D* Lite can be computationally more efficient than Repeated A* if a large number of states are in the subtree rooted in the start state of the current search, since they can reuse a large number of states from the previous search.
Chapter 5: Experimental Evaluation

In this chapter, I measure the runtimes of the developed incremental search algorithms for moving target search. The structure of this chapter is as follows: In Section 5.1, I introduce the objective of the experimental evaluation. In Section 5.2, I introduce the experimental settings. In Section 5.3, I describe the experimental results and analyze them. In Section 5.4, I discuss the strengths and weaknesses of the developed incremental search algorithms and provide guidelines for potential users when to choose one algorithm over another.

5.1 Objective

The objective of the comparison of the developed incremental search algorithms is to find out their strengths and weaknesses in terms of their runtime per search in different scenarios of moving target search and to demonstrate that the developed incremental search algorithms are able to satisfy the runtime requirements imposed by computer game company Bioware (= 1-3 ms per search). I do not measure their memory requirements since they are in linear in the number of states in the state space for all of them as for $A^*$,
which can often be satisfied in different applications, such as computer games (Rabin, 2002). Specifically, I compare their runtime per search with respect to:

- different search directions of the incremental search algorithms, namely forward and backward searches (if applicable);
- different state spaces, namely randomly generated grids, randomly generated mazes and game maps;
- different classes of terrains, namely the ones discussed in Section 1.1;
- different numbers of actions whose costs change between searches in known dynamic terrains; and
- different movement strategies of the target, namely moving to randomly selected states and using TrailMax, a state-of-the-art evasion algorithm for known static terrains, that generates movement strategies for the target that result in long trajectories of the agent necessary to catch the target (Moldenhauer & Sturtevant, 2009a), as discussed in Section 2.5.

Then, I discuss their strengths and weaknesses and provide guidelines for when to choose a particular algorithm over another. In order to ensure that the style of the pseudo codes of all compared search algorithms are consistent throughout the dissertation, I use the style of the pseudo code of D* Lite (Koenig & Likhachev, 2002) for all compared search algorithms. The implementation of a search algorithm can be slightly different from the pseudo code of the algorithm. For example, rather than using UpdateState(s), I have implemented Forward and Backward Repeated A*, GAA*, G-FRA* and FRA* with
inline code to update the membership of a state in the OPEN list. The source code of all compared search algorithms is available at (Sun, 2012). The experimental results showed that the runtimes per search of these algorithms with slightly different implementation are similar. Some developed incremental search algorithms, such as GAA* and MT-D* Lite, have also been implemented by other researchers, who have gained similar experimental results to the ones demonstrated in this chapter (Belghith et al., 2010; Anand, 2011). I will provide more details about their works in Chapter 6.

5.2 Experimental Setups

In this section, I introduce my experimental setups for evaluating the runtime per search of the developed incremental search algorithms.

5.2.1 Algorithms

The search algorithms that I compare in this chapter for moving target search are:

- Forward Repeated A* and Backward Repeated A*;
- Forward Differential A* and Backward Differential A*;
- Forward Generalized Adaptive A* (Forward GAA*) and Backward Generalized Adaptive A* (Backward GAA*);
- Generalized Fringe-RetrievinA* (G-FRA*) and Fringe-Retrieving A* (FRA*); and
5.2.2 State Spaces

I primarily use four-neighbor grids to evaluate the runtime of the developed search algorithms, since they are standard test beds for evaluating the runtime of search algorithms and have been widely used to represent state spaces of search-based path planning problems (Korf, 1990; Ishida & Korf, 1991; Koenig & Likhachev, 2002; Thrun & Buecken, 1996; Likhachev, 2005; Ferguson, 2006; Rabin, 2002; Sun & Koenig, 2007; Moldenhauer & Sturtevant, 2009b). I use the Manhattan distances from each state to the goal state as the user-provided $h$-values in four-neighbor grids.

I perform experiments to evaluate the runtime of the developed incremental search algorithms in three different kinds of grids that have been used to evaluate the runtime of search-based path planning algorithms (Koenig et al., 2004a, 2007; Hernández et al., 2009; Moldenhauer & Sturtevant, 2009b; Sun & Koenig, 2007):
• Random grids of size 500 \times 500 in which 25\% of randomly chosen states were blocked (shown in Figure 5.1 left).

• Random mazes of size 500 \times 500 whose corridors are ten states wide (shown in Figure 5.1 middle). The corridors are generated with depth-first search.

• Two game maps adapted from World of Warcraft of size 512 \times 512 and 676 \times 676, respectively (shown in Figure 5.1 right).

5.2.3 Terrains

I systematically evaluate the runtime of the search algorithms described in Section 5.2.1 in all three classes of terrains discussed in Section 1.1.2.

• Known Static Terrains: In Section 5.3.1, I compare the developed search algorithms described in Section 5.2.1 that apply to known static terrains in known static grids.

• Known Dynamic Terrains: In Section 5.3.2, I compare the developed search algorithms described in Section 5.2.1 that apply to known dynamic terrains in known dynamic grids. I unblock \( k \) blocked states and block \( k \) unblocked states every tenth move of the agent in a way so that there always remains a path from the current state of the agent to the current state of the target, where \( k \) is a parameter whose value I vary from ten to five thousand in order to get a comprehensive comparison of the search algorithms. For each test case in the experiments, if the first search of the test case does not return a path from the current state of the agent to the current state of the target, then I terminate the test case. If the first
search returns a path, then I ensure that a path from the current state of the agent to the current state of the target exists by keeping all states on the first path and the trajectory of the target unblocked. The reason why I terminate a test case if there does not exist a path from the current state of the agent to the current state of the target is that it can take a long time before the state space changes in a way such that a new path exists from the current state of the agent to the current state of the target, which makes it unacceptably time-consuming to perform the experiments.

- **Unknown Static Terrains:** In Section 5.3.3, I compare the developed search algorithms described in Section 5.2.1 that apply to unknown static terrains in unknown static grids. I set the sensor range of the agent to one state, that is, the agent can always observe the blocked state of the four neighboring states of its current state.

### 5.2.4 Target Movement Strategies

I evaluate the runtime of all search algorithms described in Section 5.2.1 with respect to two different movement strategies of the target:

- **Random Waypoint:** The target always follows a cost-minimal path from its current state to a randomly selected unblocked state. I ensure that there always exists a path from the current state of the target to the unblocked state when selected, and this path is kept unblocked during the test case. The target repeats the process once it reaches that unblocked state in all three kind of terrains discussed in Section 1.1.2. This strategy can be seen in computer games. For example, a game
character can repeatedly move to random states selected by the player via mouse clicks.

- **TrailMax**: TrailMax moves the target to a state that is as far away from the current state of the agent as possible in known static terrains (see Section 2.5).

Following the previous literature on moving target search (Ishida & Korf, 1991; Ishida, 1992; Chimura & Tokoro, 1994; Moldenhauer & Sturtevant, 2009a), I choose the initial states of the agent and target randomly for each test case. The target skips every tenth move to enable the agent to catch it.

### 5.2.5 Measures

I run all experiments on an Intel Core 2 Duo 2.53 Ghz Linux PC with 2 GBytes of RAM. I report two measures for the difficulty of the search problems, namely (a) the number of searches per test case and (b) the number of moves of the agent per test case until it catches the target. All search algorithms determine paths of the same cost for each search if the start and goal states of that search are the same, and their numbers of moves and searches per test case are thus approximately the same. They differ slightly since the agent can follow different trajectories due to tie breaking. I report two measures for the efficiency of the search algorithms, namely (c) the number of expanded states (= state expansions) per search and (f) the runtime per search. I calculate the runtime per search by dividing the total runtime until the target is caught by the number of searches.\(^1\)

Unfortunately, the runtime per search depends on low-level machine and implementation details, such as the instruction set of the processor, the optimizations performed by the

\(^1\)The time for calculating the movement strategy of the target is not included in the total runtime.
compiler and coding decisions. This point is especially important since the grids fit into memory and the resulting state spaces are thus small. I do not know of any better method for evaluating search algorithms than to implement them as well as possible, publish their runtimes, and let other researchers validate them with their own and thus potentially slightly different implementations. I also report the standard deviation of the mean for the number of state expansions per search (in parentheses) for all search algorithms described in Section 5.2.1 to demonstrate the statistical significance of my results. I do not report the standard deviation of the mean for the runtime per search because the smallest clock granularity available in the Linux system is one microsecond, which is larger than some runtimes per search when the agent is near the target. In this situation, the runtimes per search measured are all zeros, resulting in an inaccurate measurement of the standard deviation of the mean for the runtime per search. For GAA*, I report (d) the number of $h$-value updates (= state propagations) per search by the consistency procedure as third measure since the runtime per search depends on both the number of state expansions and propagations per search. I calculate the number of state propagations per search by dividing the total number of $h$-value updates until the target is caught by the number of searches. For FRA*, G-FRA* and MT-D* Lite, I report (e) the number of deleted states (= state deletions) per search as third measure since the runtime per search depends on both the number of state expansions and deletions per search. I calculate the number of state deletions per search by dividing the total number of deleted states by the number of searches.
Table 5.1: Experimental Results in Known Static Grids (Target Movement Strategy: Random Waypoint)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Random Grid</th>
<th>Random Maze</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>Forward Repeated A*</td>
<td>165 296</td>
<td>1,726 (31.6)</td>
</tr>
<tr>
<td>Forward GAA*</td>
<td>165 296</td>
<td>1,321 (25.7)</td>
</tr>
<tr>
<td>Forward Differential A*</td>
<td>165 296</td>
<td>1,726 (31.6)</td>
</tr>
<tr>
<td>Backward Repeated A*</td>
<td>169 301</td>
<td>2,310 (45.9)</td>
</tr>
<tr>
<td>Backward GAA*</td>
<td>161 301</td>
<td>802 (17.8)</td>
</tr>
<tr>
<td>Backward Differential A*</td>
<td>169 301</td>
<td>2,310 (45.9)</td>
</tr>
<tr>
<td>FRA*</td>
<td>179 290</td>
<td>199 (4.7)</td>
</tr>
<tr>
<td>G-FRA*</td>
<td>185 288</td>
<td>211 (7.3)</td>
</tr>
<tr>
<td>Basic MT-D* Lite</td>
<td>198 316</td>
<td>431 (4.3)</td>
</tr>
<tr>
<td>MT-D* Lite</td>
<td>203 320</td>
<td>229 (7.6)</td>
</tr>
</tbody>
</table>

(a) = searches until the target is caught; (b) = moves until the target is caught; (c) = state expansions per search (in parenthesis: standard deviation of the mean); (d) = state propagations per search; (e) = state deletions per search; and (f) = runtime per search (in microseconds)

5.3 Experimental Results

In this section, I discuss the experimental results for all search algorithms described in Section 5.2.1.

5.3.1 Experiments in Known Static Terrains

In this section, I compare all search algorithms described in Section 5.2.1 for moving target search in known static terrains.

5.3.1.1 Target Movement Strategy: Random Waypoint

In this section, I compare all search algorithms described in Section 5.2.1 for moving target search when the target uses the Random Waypoint movement strategy.

First, I discuss the experimental results for Forward and Backward Repeated A* and the heuristic learning incremental search algorithms, namely, Forward and Backward
GAA*, which are identical to Forward and Backward MT-Adaptive A*, respectively, in known static grids. Table 5.1 shows the following relationships:

- For all three kinds of grids (see Section 5.2.2), GAA* has a smaller runtime per search than Repeated A* with the same search direction because GAA* updates the h-values to make them more informed over time and hence expands fewer states per search. The smallest runtime per search in each kind of grid is shown in bold. For example, Backward GAA* runs faster than Backward Repeated A* by a factor of 1.75 in random grids, 2.18 in random mazes, 2.45 in the game map of size 512 × 512 and 2.21 in the game map of size 676 × 676.

- For all three kinds of grids, Backward GAA* has a smaller runtime per search than Forward GAA*. The reason is as follows: Assume that after the previous search the current state of the target changed from t to t'. Let d be equal to the minimum cost of moving from the state of the agent in the previous search to t, and d' be equal to the minimum cost of moving from the state of the agent in the previous search to t'. For both Forward and Backward GAA*, there are two possible cases:

  - Case 1: d ≤ d'. For example, the target moves away from the agent after the previous search.
  - Case 2: d > d'. For example, the target moves towards the agent after the previous search.

For Forward GAA*, the minimum costs of moving from most of the expanded states in the previous search to the current state of the target t' (= goal state of the current search) do not decrease in Case 1 after the target moved. However, Forward GAA*
corrects the $h$-value of the state $s$ expanded in the previous searches by assigning
\[ h(s) := \max(H(s, t'), h(s) - h(t')) \] (see Assignment (3.1) on page 68), which likely
decreases the $h$-value and thus make it less informed although the minimum cost of
moving from $s$ to the goal state likely does not decrease.

For Backward GAA* and for both Cases 1 and 2, the agent moves from $s_{goal}$ to $s'_{goal}$
towards the target after the previous search. Thus, the minimum costs of moving
from most of the expanded states in the previous search to the current state of
the agent (= goal state of the current search) decrease after the agent moved.
Backward GAA* corrects the $h$-value of each state $s$ expanded in the previous
search by assigning $h(s) := \max(H(s, s'_{goal}), h(s) - h(s'_{goal}))$, which likely reflects
the change of the $h$-value of $s$ accurately.

I also perform additional experiments with the game map of size $676 \times 676$ to
measure the ratios between the updated $h$-values and the user-provided $h$-values of
Forward GAA* and Backward GAA*. The results show that the ratios of Forward
GAA* and Backward GAA* are 2.13 and 2.39, respectively. Therefore, Backward
GAA* has more informed $h$-values and expands fewer states per search than For-
ward GAA*, which results in a smaller runtime per search than Forward GAA*.
For example, Backward GAA* runs faster than Forward GAA* by a factor of 2.04
in random grids, 1.45 in random mazes, 1.54 in the game map of size $512 \times 512$
and 2.29 in the game map of size $676 \times 676$.

Second, I discuss the experimental results for Forward and Backward Repeated A*
and the search tree transforming incremental search algorithms, that is, Forward and
Backward Differential A*, FRA*, G-FRA*, Basic MT-D* Lite and MT-D* Lite. Table 5.1 shows the following relationships:

- For all three kinds of grids, Repeated A* has a smaller runtime per search than Differential A* with the same search direction. This is because the start state changes between searches for moving target search. After the start state changes, Differential A* deletes the start state of the previous search and creates the start state of the current search. Deleting the start state of the previous search results in the deletion of all states in the search tree of the previous search. Thus, Differential A* cannot reuse any part of the search tree from the previous search but spends effort on deleting the search tree of the previous search and hence can be even slower than Repeated A* with the same search direction (see Section 2.4.2.3). For example, Forward Repeated A* runs faster than Forward Differential A* by a factor of 1.29 in random grids, 1.38 in random mazes, 1.20 in the game map of size $512 \times 512$ and 1.31 in the game map of size $676 \times 676$.

- For all three kinds of grids, both FRA* and G-FRA* have a smaller runtime per search than Forward and Backward Repeated A* because FRA* and G-FRA* do not expand the states in the subtree of the previous search rooted in the start state of the current search. Forward and Backward Repeated A*, on the other hand, expand some of these states. For example, in random grids, FRA* expands only about 8.52% of the states that Forward Repeated A* expands. It thus runs by a factor of 5.71 faster per search than Forward Repeated A*. Similarly, FRA* runs
faster than Forward Repeated A* by a factor of 5.06 in random mazes, 4.96 in the
game map of size 512 × 512 and 8.85 in the game map of size 676 × 676.

- For all three kinds of grids, MT-D* Lite has a smaller runtime per search than
  Basic MT-D* Lite because Basic MT-D* Lite expands the states in the search
tree of the previous search that are not in the subtree rooted in the start state
of the current search to set their $g$-values to infinity. MT-D* Lite, on the other
hand, uses OptimizedDeletion() instead, which is demonstrated to be faster with
the experimental results. For example, MT-D* Lite runs faster than Basic MT-D*
Lite by a factor of 1.64 in random grids, 1.93 in random mazes, 1.78 in the game
map of size 512 × 512 and 1.69 in the game map of size 676 × 676.

- For all three kinds of grids, FRA* and G-FRA* have smaller runtimes per search
  than Basic MT-D* Lite and MT-D* Lite because of two reasons: (1) FRA* and
  G-FRA* have a smaller runtime per state expansion than Basic MT-D* Lite and
  MT-D* Lite. For example, in random grids, the approximate runtime per state
  expansion (calculated by dividing the runtime per search by the number of expanded
  states per search) is 0.32 and 0.34 microseconds for FRA* and G-FRA*, respectively,
  while the approximate runtime per state expansion is 0.50 and 0.51 microseconds for
  Basic MT-D* Lite and MT-D* Lite, respectively. (2) FRA* and G-FRA* expand
  fewer states than Basic MT-D* Lite and MT-D* Lite in the following case: If the
target moves to a state in the subtree of the previous search that is rooted in the
start state of the current search, FRA* and G-FRA* terminate without expanding
states due to Step 3 (Terminating Early). I also perform additional experiments for
FRA* with the game map of size 676 × 676 to measure the ratios of the number of early terminations and the total number of searches per test case. The results show that 64 out of 351 searches of FRA* (= 18.23%) terminate early. Basic MT-D* Lite and MT-D* Lite, on the other hand, do not terminate early but have to expand all locally inconsistent states whose keys are smaller than the key of the goal state. For example, FRA* runs faster than MT-D* Lite by a factor of 1.45 in random grids, 2.44 in random mazes, 1.73 in the game map of size 512 × 512 and 1.78 in the game map of size 676 × 676.

- For all three kinds of grids, FRA* has a smaller runtime per search than G-FRA* because G-FRA* reuses only the subtree of the previous search that is rooted in the start state of the current search. FRA*, on the other hand, uses the Additional Step for grids (see Section 4.1.2.3), that allows it to reuse more of the search tree of the previous search. Thus, FRA* deletes and expands fewer states per search than G-FRA*. For example, in random grids, FRA* expands only about 92.38% of the states per search that G-FRA* expands. It thus runs by a factor of 1.17 faster per search than G-FRA*. Similarly, FRA* runs faster than G-FRA* by a factor of 1.05 in random mazes, 1.04 in the game map of size 512 × 512 and 1.04 in the game map of size 676 × 676.

- For all three kinds of grids, FRA* has the smallest runtime per search among Forward and Backward Repeated A*, FRA*, G-FRA*, Basic MT-D* Lite and MT-D* Lite.
Third, I compare the search tree transforming incremental search algorithms with the heuristic learning incremental search algorithms. Table 5.1 shows the following relationships:

- For all three kinds of grids, G-FRA* and FRA* have smaller runtimes per search than Forward and Backward GAA* because G-FRA* and FRA* expand fewer states per search than Forward and Backward GAA*. I focus only on the comparison of FRA* against Backward GAA* since FRA* runs faster than G-FRA* and Backward GAA* runs faster than Forward GAA*. In random grids, FRA* expands about 19.92% of the states per search that Backward GAA* expands. It thus runs by a factor of 2.74 faster per search than Backward GAA*. Similarly, FRA* runs faster than Backward GAA* by a factor of 2.40 in random mazes, 2.81 in the game map of size 512 × 512 and 3.02 in the game map of size 676 × 676.

- For all three kinds of grids except the random mazes, MT-D* Lite has a smaller runtime per search than Forward and Backward GAA*. I focus only on the comparison of MT-D* Lite against Backward GAA* since MT-D* Lite runs faster than Basic MT-D* Lite and Backward GAA* runs faster than Forward GAA*. MT-D* Lite runs faster than Backward GAA* by a factor of 1.89 in random grids, 1.63 in the game map of size 512 × 512 and 1.70 in the game map of size 676 × 676. However, Backward GAA* runs faster than MT-D* Lite by a factor of 1.02 in random mazes.

I perform additional experiments with MT-D* Lite and Backward GAA* to gain more insight into the operations of both search algorithms. I first measure the ratio
$r$ of the number of states deleted from the search tree and the total number of states in the search tree after each search of MT-D* Lite. I calculate $r$ by dividing the total number of states deleted from the search tree by the total number of states in the search tree before the deletion is performed. $r$ can significantly affect the runtime of MT-D* Lite because, the smaller $r$ is, the smaller a portion of the search tree of the previous search is deleted and hence the larger a portion of the search tree of the previous search can be reused for the current search. Thus, a smaller $r$ tends to result in a smaller runtime per search for MT-D* Lite. The results show that $r$ is equal to 4.63% and 21.12% in random grids and random mazes, respectively, which explains why MT-D* Lite runs faster than Backward GAA* by a factor of 1.89 in random grids, but runs more slowly than Backward GAA* by a factor of 1.02 in random mazes.

To explain why $r$ is much larger in random mazes than random grids, I measure the informedness of the user-provided $h$-values in each kind of grid as follows: I randomly select 100 pairs of start state $s_{start}$ and goal state $s_{goal}$ in each kind of grid. Then, I calculate the informedness of the user-provided $h$-values in each kind of grid by using the ratio $r_{in}$ between the average ratio of the user-provided $h$-values $H(s_{start},s_{goal})$ and $dist^*(s_{start},s_{goal})$ between each pair of $s_{start}$ and $s_{goal}$. Since the user-provided $h$-values (= Manhattan Distances) are admissible, $0 \leq r_{in} \leq 1$ and, the larger $r_{in}$ is, the more informed the $h$-values are. Experimental results show that $r_{in}$ is equal to 0.61 in random mazes, 0.95 in random grids, 0.85 in the game map of size 512 $\times$ 512 and 0.90 in the game map of size 676 $\times$ 676. Thus, the order of informedness of the user-provided $h$-values in the different kinds of grids, from
Figure 5.2: Reusable and Deleted States of Moving Target D* Lite

the most to the least well-informed, is in random grids, the game map of size 676 × 676, the game map of size 512 × 512, and random mazes. In this dissertation, the user-provided $h$-values in random mazes are regarded as “ill-informed”, since they are much smaller than the ones in random grids, and the two game maps, which are regarded as “well-informed”. The informedness of the user-provided $h$-values explains why $r$ is much larger in random mazes than random grids: Assume that MT-D* Lite performs a search from the current state of the agent $S$ to the current state of the target $G$ and, after the search, the current states of the agent and target change to $S'$ and $G'$, respectively. Figure 5.2(a) visualizes the expanded states of a search with well-informed $h$-values. In this case, the search is more focused on the area that contains the cost-minimal path from $S$ to $G$. When the agent moves from $S$ to $S'$, a large portion of expanded states belongs to the subtree rooted in $S'$ (= reusable states) since the search was more focused. MT-D* Lite only needs to delete a small portion of expanded states from the search tree of the previous search, resulting in a smaller runtime per search than Backward GAA*. Figure 5.2(b) visualizes the expanded states of a search with ill-informed $h$-values.
In this case, the search is less focused on the area that contains the cost-minimal path from \( S \) to \( G \). When the agent moves from \( S \) to \( S' \), only a relatively small portion of expanded states belongs to the subtree rooted in \( S' \) (= reusable states) since the search was less focused. MT-D* Lite needs to delete a large portion of expanded states from the search tree of the previous search, resulting in a larger \( r \) and hence a larger runtime per search than Backward GAA*.

5.3.1.2 Target Movement Strategy: TrailMax

I now evaluate the runtime of all search algorithms described in Section 5.2.1 when the target uses TrailMax as its movement strategy to find out how different movement strategies of the target affect the runtimes of the compared search algorithms. I do not report the results of Differential A*, since its runtime per search is larger than the one of Repeated A* (with the same search direction) and hence it is not competitive.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Random Grid</th>
<th>Random Maze</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>Forward Repeated A*</td>
<td>806</td>
<td>1,001</td>
</tr>
<tr>
<td>Forward GAA*</td>
<td>813</td>
<td>1,008</td>
</tr>
<tr>
<td>Backward Repeated A*</td>
<td>820</td>
<td>1,015</td>
</tr>
<tr>
<td>Backward GAA*</td>
<td>812</td>
<td>1,017</td>
</tr>
<tr>
<td>Forward GAA*</td>
<td>810</td>
<td>1,005</td>
</tr>
<tr>
<td>Backward GAA*</td>
<td>812</td>
<td>1,017</td>
</tr>
<tr>
<td>FRA*</td>
<td>810</td>
<td>1,005</td>
</tr>
<tr>
<td>G-FRA*</td>
<td>785</td>
<td>9,85</td>
</tr>
<tr>
<td>Basic MT-D* Lite</td>
<td>841</td>
<td>1,030</td>
</tr>
<tr>
<td>MT-D* Lite</td>
<td>844</td>
<td>1,032</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
</tr>
</tbody>
</table>

(a) = searches until the target is caught; (b) = moves until the target is caught; (c) = state expansions per search (in parenthesis: standard deviation of the mean); (d) = state propagations per search; (e) = state deletions per search; and (f) = runtime per search (in microseconds)

Table 5.2: Experimental Results in Known Static Grids (Target Movement Strategy: TrailMax)
I first describe the similarity between the results in Table 5.1, where the target uses the Random Waypoint movement strategy, and the results in Table 5.2, where the target uses the TrailMax movement strategy. The explanations given earlier in Section 5.3.1.1 still apply to the following relationships:

- First, for heuristic learning incremental search algorithms, both tables show the following similar relationships:
  
  – For all three kinds of grids, GAA* has a smaller runtime per search than Repeated A* with the same search direction. For example, Backward GAA* runs faster than Backward Repeated A* by a factor of 1.65 in random grids, 1.72 in random mazes, 2.14 in the game map of size $512 \times 512$ and 2.27 in the game map of size $676 \times 676$.
  
  – For all three kinds of grids, Backward GAA* has a smaller runtime per search than Forward GAA*. For example, Backward GAA* runs faster than Forward GAA* by a factor of 2.43 in random grids, 1.65 in random mazes, 2.79 in the game map of size $512 \times 512$ and 2.21 in the game map of size $676 \times 676$.

- Second, for search tree transforming incremental search algorithms, both tables show the following similar relationships:
  
  – For all three kinds of grids, both FRA* and G-FRA* have a smaller runtime per search than Forward and Backward Repeated A*. For example, FRA* runs faster than Forward Repeated A* by a factor of 6.43 in random grids,
3.11 in random mazes, 5.04 in the game map of size $512 \times 512$ and 10.98 in the game map of size $676 \times 676$.

- For all three kinds of grids, MT-D* Lite has a smaller runtime per search than Basic MT-D* Lite. For example, MT-D* Lite runs faster than Basic MT-D* Lite by a factor of 1.86 in random grids, 1.77 in random mazes, 1.87 in the game map of size $512 \times 512$ and 1.74 in the game map of size $676 \times 676$.

- For all three kinds of grids, FRA* and G-FRA* have smaller runtimes per search than Basic MT-D* Lite and MT-D* Lite. For example, FRA* runs faster than MT-D* Lite by a factor of 1.31 in random grids, 1.87 in random mazes, 2.48 in the game map of size $512 \times 512$ and 1.52 in the game map of size $676 \times 676$.

- For all three kinds of grids, FRA* has a smaller runtime per search than G-FRA*. For example, FRA* runs faster than G-FRA* by a factor of 1.24 in random grids, 1.08 in random mazes, 1.44 in the game map of size $512 \times 512$ and 1.03 in the game map of size $676 \times 676$.

- For all three kinds of grids, FRA* has the smallest runtime per search among Forward and Backward Repeated A*, FRA*, G-FRA*, Basic MT-D* Lite and MT-D* Lite.

- Third, when comparing the heuristic learning incremental search algorithms to the search tree transforming incremental search algorithms, both tables show the following similar relationships:
– For all three kinds of grids, FRA* and G-FRA* have smaller runtimes per search than both Forward and Backward GAA*. I focus only on the comparison of FRA* against Backward GAA* since FRA* runs faster than G-FRA* and Backward GAA* runs faster than Forward GAA*. FRA* runs faster than Backward GAA* by a factor of 2.39 in random grids, 1.33 in random mazes, 1.52 in the game map of size $512 \times 512$ and 4.59 in the game map of size $676 \times 676$.

– For random grids and the game map of size $676 \times 676$ (where the user-provided $h$-values are well-informed, see Section 5.3.1.1), MT-D* Lite has a smaller runtime per search than Backward GAA*. For example, MT-D* Lite runs faster than Backward GAA* by a factor of 1.83 in random grids and 3.02 in the game map of size $676 \times 676$.

– For random mazes (where the user-provided $h$-values are ill-informed, see Section 5.3.1.1) and the game map of size $512 \times 512$ (where the user-provided $h$-values are less well-informed than random grids and the game map of size $676 \times 676$), Backward GAA* has a smaller runtime per search than MT-D* Lite. For example, Backward GAA* runs faster than MT-D* Lite by a factor of 1.40 in random mazes and 1.63 in the game map of size $512 \times 512$.

I now describe the differences between the results in Table 5.1, where the target uses the Random Waypoint movement strategy, and the results in Table 5.2, where the target uses the TrailMax movement strategy:
• For all three kinds of grids, the number of searches of all search algorithms and the number of moves to catch the target are larger when the target uses the TrailMax rather than the Random Waypoint movement strategy, which demonstrates the power of TrailMax to avoid the agent. For example, in random grids, the agent performs 252 Forward Repeated A* searches and 401 moves to catch the target when the target uses the Random Waypoint movement strategy. On the other hand, the agent performs 806 Forward Repeated A* searches (= an increase of a factor of 3.20) and 1,001 moves (= an increase of a factor of 2.50) to catch the target when the target uses the Random Waypoint movement strategy.

• For all three kinds of grids, the runtime per search of all search algorithms can be smaller or larger when the target uses different movement strategies, since different movement strategies of the target can result in different trajectories of both the agent and the target. In this case, a search algorithm can expand different states when searching for a cost-minimal path between the current states of the agent and the target, resulting in a different runtime per search. For example, the runtime per search of MT-D* Lite is 94 microseconds in random grids when the target uses the Random Waypoint movement strategy, and it decreases to 64 microseconds when the target uses the TrailMax movement strategy. Different from the situation in random grids, the runtime per search of MT-D* Lite is 83 microseconds in the game map of size 512 × 512, when the target uses the Random Waypoint movement strategy, and it increases to 176 microseconds when the target uses the TrailMax movement strategy.
5.3.1.3 Summary

<table>
<thead>
<tr>
<th>Terrain</th>
<th>Fastest Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Grid</td>
<td>FRA* → G-FRA* → MT-D* Lite → Backward GAA*</td>
</tr>
<tr>
<td>Random Maze</td>
<td>FRA* → G-FRA* → Backward GAA* → MT-D* Lite → Forward GAA*</td>
</tr>
<tr>
<td>Game Map 512x512</td>
<td>FRA* → G-FRA* → MT-D* Lite → Backward GAA* → Basic MT-D* Lite</td>
</tr>
<tr>
<td>Game Map 676x676</td>
<td>FRA* → G-FRA* → MT-D* Lite → Basic MT-D* Lite → Backward GAA*</td>
</tr>
</tbody>
</table>

Figure 5.3: Runtime Relationships in Known Static Terrains (Target Movement Strategy: Random Waypoint)

<table>
<thead>
<tr>
<th>Terrain</th>
<th>Fastest Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Grid</td>
<td>FRA* → G-FRA* → MT-D* Lite → Backward GAA* → Basic MT-D* Lite</td>
</tr>
<tr>
<td>Random Maze</td>
<td>FRA* → G-FRA* → Backward GAA* → MT-D* Lite → Forward GAA*</td>
</tr>
<tr>
<td>Game Map 512x512</td>
<td>FRA* → G-FRA* → Backward GAA* → MT-D* Lite → Backward Repeated A*</td>
</tr>
<tr>
<td>Game Map 676x676</td>
<td>FRA* → G-FRA* → MT-D* Lite → Basic MT-D* Lite → Backward GAA*</td>
</tr>
</tbody>
</table>

Figure 5.4: Runtime Relationships in Known Static Terrains (Target Movement Strategy: TrailMax)

Finally, I summarize the results by combing the experimental results from Tables 5.1 and 5.2. The runtime relationships of the fastest five search algorithms when the target uses the Random Waypoint and TrailMax movement strategies are shown in Figures 5.3 and 5.4, respectively. An arrow points algorithm A to algorithm B iff the runtime per search of algorithm A is smaller than the one of algorithm B. Figure 5.3 shows the following relationships:
- For all three kinds of grids except for random mazes (where the user-provided $h$-values are well-informed, see Section 5.3.1.1), FRA* and G-FRA* have the smallest and second smallest runtime per search, respectively. MT-D* Lite has the third smallest runtime per search.

- For random mazes (where the user-provided $h$-values are ill-informed, see Section 5.3.1.1), FRA* and G-FRA* have the smallest and second smallest runtime per search, respectively. Backward GAA* has the third smallest runtime per search.

Figure 5.4 shows the following relationships:

- For all three kinds of grids except for random mazes (where the user-provided $h$-values are well-informed, see Section 5.3.1.1), FRA* and G-FRA* have the smallest and second smallest runtime per search, respectively. For random grids and the game map of size 676 $\times$ 676, MT-D* Lite has the third smallest runtime per search.

- For the game map of size 512 $\times$ 512, Backward GAA* has the third smallest runtime per search.

- For random mazes (where the user-provided $h$-values are ill-informed, see Section 5.3.1.1), FRA* and G-FRA* have the smallest and second smallest runtime per search, respectively. Backward GAA* has the third smallest runtime per search.

To summarize, FRA* and G-FRA* have the smallest runtime per search among all compared search algorithms for all three kinds of grids and for both movement strategies of the target. Thus, they are the fastest incremental search algorithms for moving target search in known static terrains.
5.3.2 Experiments in Known Dynamic Terrains

In this section, I compare all search algorithms described in Section 5.2.1 for moving target search in known dynamic terrains. Since TrailMax was designed only for known static terrains, I compare all search algorithms in case the target uses the Random Waypoint movement strategy.

First, I discuss the experimental results for Forward and Backward Repeated A* and the heuristic learning incremental search algorithms, that is, Forward and Backward GAA*. Table 5.3 shows the following relationships:

- For all three kinds of grids and for all \( k \), the number of state expansions per search of GAA* is smaller than the one of Repeated A* with the same search direction because GAA* updates the \( h \)-values to make them more informed over time and hence expands fewer states per search. For example, in random grids, when \( k = 10 \), Backward GAA* expands only about 51.38% of the states per search that Backward Repeated A* expands.

- For all three kinds of grids, the number of state propagations per search of GAA* increases as \( k \) increases since a larger number of action costs then decrease. Thus, the \( h \)-values of more states in the state space are affected by the action cost decreases. The consistency procedure needs to perform more state propagations to maintain the consistency of the \( h \)-values. For example, in random grids, the number of state propagations per search of Forward GAA* is 7 when \( k = 10 \), and it increases to 916 when \( k = 5,000 \).
<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Grid</td>
<td>Random Maze</td>
<td>Random Grid</td>
<td>Random Maze</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td>Game Map 512 × 512</td>
<td>Game Map 676 × 676</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) = searches until the target is caught; (b) = moves until the target is caught; (c) = state expansions per search (in parenthesis: standard deviation of the mean); (d) = state propagations per search; (e) = state deletions per search; and (f) = runtime per search (in microseconds)

**Table 5.3: Experimental Results in Known Dynamic Grids**

- For all three kinds of grids, when the number of action cost changes is small (that is, when \( k \leq 100 \)), GAA* has a smaller runtime per search than Repeated A* with
the same search direction because GAA* updates the $h$-values to make them more informed over time and hence expands fewer states per search. The computational effort of the consistency procedure (i.e., the overhead of the consistency procedure), that is, the number of state propagations to maintain the consistency of the $h$-values, is small when $k$ is small. Thus, the savings of GAA* over Repeated A* in state expansions dominates the overhead of the consistency procedure, resulting in a smaller runtime per search than Repeated A*. For example, when $k = 10$ in random grids, Backward GAA* expands only about 51.38% of the states per search that Backward Repeated A* expands. It thus runs by a factor of 1.73 faster per search than Backward Repeated A*. Similarly, Backward GAA* runs faster per search than Backward Repeated A* by a factor of 2.44 in random mazes, 2.57 in the game map of size $512 \times 512$ and 2.51 in the game map of size $676 \times 676$.

- For all three kinds of grids, when the number of action cost changes is large (that is, when $k = 5,000$), GAA* has a larger runtime per search than Repeated A* with the same search direction because the overhead of the consistency procedure increases as $k$ increases. When $k$ is large, the overhead of the consistency procedure dominates the savings of GAA* over Repeated A* in state expansions, resulting in a larger runtime per search than Repeated A*. For example, when $k = 5,000$ in the game map of size $676 \times 676$ (and thus 2.19% of states change their blockage status), Backward GAA* expands only about 53.39% of the states per search that Backward Repeated A* expands. However, the overhead of the consistency procedure dominates the savings of Backward GAA* over Repeated A* in state
expansions. Thus, Backward GAA* runs by a factor of 1.11 more slowly per search than Backward Repeated A*.

- For all three kinds of grids and for all $k$, Backward GAA* has a smaller runtime per search than Forward GAA* for the same reason as discussed in Section 5.3.1. For example, when $k = 10$, Backward GAA* runs faster than Forward GAA* by a factor of 1.80 in random grids, 1.45 in random mazes, 1.18 in the game map of size $512 \times 512$ and 2.16 in the game map of size $676 \times 676$.

Second, I discuss the experimental results for Forward and Backward Repeated A* and the search tree transforming incremental search algorithms, that is, Basic MT-D* Lite and MT-D* Lite. Table 5.3 shows the following relationships:

- For all three kinds of grids and for all $k$, MT-D* Lite has a smaller runtime per search than Basic MT-D* Lite for the same reason as discussed in Section 5.3.1. For example, when $k = 10$, MT-D* Lite runs faster than Basic MT-D* Lite by a factor of 1.67 in random grids, 2.05 in random mazes, 1.82 in the game map of size $512 \times 512$ and 1.73 in the game map of size $676 \times 676$.

- For all three kinds of grids, Basic MT-D* Lite and MT-D* Lite have runtimes per search that generally increase as $k$ increases because they generally update more $g$-values and hence expand more states per search. Thus, the savings in the runtime per search of Basic MT-D* Lite and MT-D* Lite over Forward and Backward Repeated A* generally decrease as $k$ increases. For example, when $k = 10$, in random grids, Basic MT-D* Lite and MT-D* Lite run faster per search than

\footnote{FRA* and G-FRA* do not apply to known dynamic terrains.}
Forward Repeated A* by a factor of 2.73 and 4.56, respectively. These factors decrease to 2.62 and 4.31 when $k = 100$, 2.06 and 2.64 when $k = 1,000$, and 1.39 and 1.65 when $k = 5,000$.

- For all three kinds of grids except the random mazes (where the user-provided $h$-values are well-informed, see Section 5.3.1.1) and for all $k$, MT-D* Lite has the smallest runtime per search among all compared search algorithms. However, in random mazes (where the user-provided $h$-values are ill-informed, see Section 5.3.1.1), the savings in runtime per search of MT-D* Lite over Forward Repeated A* are not as large as the ones in random grids and game maps due to the large ratio $r$ of the states deleted from the search tree and the total number of states in the search tree after each search. The reason for this is similar to that discussed for known static terrains in Section 5.3.1: A smaller $r$ tends to result in a smaller runtime per search for MT-D* Lite. I perform additional experiments to compare the ratio $r$ of MT-D* Lite in random grids and in random mazes: For example, when $k = 10$, $r$ is equal to 4.05% in random grids (where the user-provided $h$-values are well-informed, see Section 5.3.1.1), and 23.91% in random mazes (where the user-provided $h$-values are ill-informed, see Section 5.3.1.1). MT-D* Lite thus runs faster than Forward Repeated A* by a factor of 4.56 in random grids (with a small $r$), but a factor of only 2.18 in random mazes (with a large $r$). Similar results hold for all $k$, which explains why MT-D* Lite is slower in random mazes (where the user-provided $h$-values are ill-informed, see Section 5.3.1.1) compared to random grids (where they are well-informed, see Section 5.3.1.1).
Third, I compare the search tree transforming incremental search algorithms with the heuristic learning incremental search algorithms. Table 5.3 shows the following relationships:

- For all three kinds of grids except the random mazes (where the user-provided $h$-values are well-informed, see Section 5.3.1.1) and for all $k$, MT-D* Lite has a smaller runtime per search than both Forward and Backward GAA*. The reason is as follows:

  - For Forward and Backward GAA*, their runtimes per search generally increase as $k$ increases because the number of state propagations per search increases as $k$ increases since a larger number of action costs then decrease, which affects the $h$-values of more states. The consistency procedure needs to perform more state propagations to maintain the consistency of the $h$-values, resulting in a larger runtime per search. Moreover, Forward and Backward GAA* need to construct the search tree from scratch for each search.

  - For MT-D* Lite, its runtime per search generally increases with $k$ (which is similar to both Forward and Backward GAA*), because it updates more $g$-values as $k$ increases and hence expands more states per search. However, since the user-provided $h$-values are well-informed, the ratio $r$ between the number of states deleted from the search tree and the total number of states in the search tree after each search is very small. For example, when $k = 10$, $r$ is equal to 4.05% in random grids, which means that 95.95% of states belong to the reusable search tree. Thus, although the runtime per search of MT-D*
Lite generally increases with \( k \), MT-D* Lite still runs faster than Forward and Backward GAA*, since a large portion of the search tree can be reused. For example, in random grids, MT-D* Lite runs faster per search than Forward and Backward GAA* by a factor of 3.75 and 2.09, respectively, when \( k = 10 \), 3.91 and 2.08 when \( k = 100 \), 2.85 and 1.58 when \( k = 1,000 \), and 2.94 and 2.28 when \( k = 5,000 \).

- For random mazes (where the \( h \)-values are ill-informed, see Section 5.3.1.1), there are three different cases as \( k \) increases:

  - First, when the number of action cost changes is small (that is, when \( k \leq 100 \)), Backward GAA* has a smaller runtime per search than MT-D* Lite and both Forward and Backward Repeated A*, because Backward GAA* updates the \( h \)-values to make them more informed over time and hence focuses the search better than both Forward and Backward Repeated A*. The number of state propagations of the consistency procedure is small when \( k \) is small. Thus, the savings of Backward GAA* over both Forward and Backward Repeated A* in state expansions dominates the overhead of the consistency procedure, resulting in a smaller runtime per search than both Forward and Backward Repeated A*. For example, when \( k = 10 \), Backward GAA* runs faster than Forward and Backward Repeated A* by factors of 2.32 and 2.44, respectively. On the other hand, since MT-D* Lite has a large ratio \( r \) of states deleted from the search tree and the total number of states in the search tree after
each search in random mazes, Backward GAA* runs faster than MT-D* Lite in random mazes when \( k \) is small.

- Second, when the number of action cost changes is medium (that is, when \( k = 1,000 \)), Backward GAA* has a larger runtime per search than MT-D* Lite because the number of state propagations of the consistency procedure increases as \( k \) increases. Although the runtime per search of MT-D* Lite also increases as \( k \) increases, the increase in the runtime per search of MT-D* Lite is smaller than the one of Backward GAA*. When \( k = 1,000 \), MT-D* Lite has a smaller runtime per search than both Forward and Backward Repeated A* and Forward and Backward GAA*.

- Third, when the number of action cost changes is large (that is, when \( k = 5,000 \)), Forward Repeated A* has a smaller runtime per search than both Forward and Backward GAA* and MT-D* Lite: Forward Repeated A* has a smaller runtime per search than Forward and Backward GAA* because the number of state propagations per search of the consistency procedure dominates the savings of GAA* over Forward Repeated A* in state expansions per search. Forward Repeated A* has a smaller runtime per search than MT-D* Lite because the runtime per search of MT-D* Lite increases as \( k \) increases. When \( k = 5,000 \), the computational effort of MT-D* Lite needed to reuse the information from previous searches for the current search is larger than the computational savings gained from reusing the information. Thus, neither Forward or Backward GAA* nor MT-D* Lite should be used in this case.
5.3.2.1 Summary

Finally, I summarize the results in Table 5.3. The runtime relationships of the fastest five search algorithms are shown in Figure 5.5. An arrow points algorithm A to algorithm B iff the runtime per search of algorithm A is smaller than the one of algorithm B. Figure 5.3 shows the following relationships:

- For all three kinds of grids except the random mazes (where the user-provided $h$-values are well-informed, see Section 5.3.1.1), MT-D* Lite has the smallest runtime per search among all compared search algorithms.

- For random mazes (where the user-provided $h$-values are ill-informed, see Section 5.3.1.1), there are three different cases as the number of action cost changes increases:
  
  - When the number of action cost changes is small (that is, when $k \leq 100$), Backward GAA* has the smallest runtime per search among all compared search algorithms.
  
  - When the number of action cost changes is medium (that is, when $k = 1,000$), MT-D* Lite has the smallest runtime per search among all compared search algorithms.
  
  - When the number of action cost changes is large (that is, when $k = 5,000$), Forward Repeated A* has the smallest runtime per search among all compared search algorithms.
Figure 5.5: Runtime Relationships in Known Dynamic Terrains (Target Movement Strategy: Random Waypoint)
### Table 5.4: Experimental Results in Unknown Static Grids

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Random Grid</th>
<th>Random Maze</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>Forward Repeated A*</td>
<td>544</td>
<td>704</td>
</tr>
<tr>
<td>Forward GAA*</td>
<td>535</td>
<td>694</td>
</tr>
<tr>
<td>Backward Repeated A*</td>
<td>531</td>
<td>690</td>
</tr>
<tr>
<td>Backward GAA*</td>
<td>529</td>
<td>690</td>
</tr>
<tr>
<td>Basic MT-D* Lite</td>
<td>536</td>
<td>663</td>
</tr>
<tr>
<td>MT-D* Lite</td>
<td>537</td>
<td>686</td>
</tr>
</tbody>
</table>

(a) = searches until the target is caught; (b) = moves until the target is caught; (c) = state expansions per search (in parenthesis: standard deviation of the mean); (d) = state propagations per search; (e) = state deletions per search; and (f) = runtime per search (in microseconds)

### 5.3.3 Experiments in Unknown Static Terrains

In this section, I compare all search algorithms described in Section 5.2.1 for moving target search in unknown static terrains. Since the TrailMax algorithm has been designed only for known static terrains, I compare all search algorithms in case the target uses the Random Waypoint movement strategy. All search algorithms are optimized for unknown static grids as discussed in Section 2.3.6.2: They replan only when either the target moves off the path or the action costs of the agent increase between the agent and the target along the path.

First, I discuss the experimental results for Forward and Backward Repeated A* and heuristic learning incremental search algorithms, that is, Forward and Backward GAA*.

The consistency procedure is not performed in unknown static terrains, since action costs are non-decreasing between searches in unknown static grids. Table 5.4 shows the following relationships:

- For all three kinds of grids, GAA* has a smaller runtime per search than Repeated A* with the same search direction because GAA* updates the $h$-values to make
them more informed over time and hence expands fewer states per search, resulting in a smaller runtime per search than A*. For example, Backward GAA* runs faster than Backward Repeated A* by a factor of 1.16 in random grids, 1.22 in random mazes, 1.12 in the game map of size $512 \times 512$ and 1.39 in the game map of size $676 \times 676$.

- For all three kinds of grids, Forward Repeated A* has a smaller runtime per search than Backward Repeated A* because Forward Repeated A* has a smaller number of state expansions per search than Backward Repeated A*. Similarly, Forward GAA* has a smaller runtime per search than Backward GAA*. For example, Forward Repeated A* has a smaller runtime per search than Backward Repeated A* by a factor 6.90 in random grids, and Forward GAA* has a smaller runtime per search than Backward GAA* by a factor of 6.61 in random grids. The reason why the search algorithms that search forward have smaller runtimes per search than the ones that search backward was discussed in previous literature: “We typically assign optimistic costs to edges whose costs we do not know. As a result, areas of the graph that have been observed have more expensive edge costs than the unexplored areas. This means that, when searching forwards, as soon as the search exits the observed area it can rapidly progress through the unexplored area directly to the goal. However, when searching backwards, the search initially rapidly progresses to the observed area, then once it encounters the more costly edges in the observed area, it begins expanding large portions of the unexplored area trying to find a cheaper path. As a result, it can be significantly more efficient to use forward A*
rather than backward A* when replanning from scratch.” (Ferguson, Likhachev, & Stentz, 2005).

Second, I discuss the experimental results for Forward and Backward Repeated A* and the search tree transforming incremental search algorithms, that is, Basic MT-D* Lite and MT-D* Lite. Table 5.4 shows the following relationships:

- For all three kinds of grids, MT-D* Lite has a smaller runtime per search than Basic MT-D* Lite for the same reason as discussed in Section 5.3.1. For example, MT-D* Lite runs faster than Basic MT-D* Lite by a factor of 1.06 in random grids, 1.19 in random mazes, 1.23 in the game map of size $512 \times 512$ and 1.06 in the game map of size $676 \times 676$.

- For all three kinds of grids, MT-D* Lite has a larger runtime per search than Forward Repeated A*. The reason is as follows: MT-D* Lite, as a variant of D* Lite, has a smaller runtime per search than Repeated A* when the action cost changes are close to the goal state (see Section 2.4.2.2). However, for moving target search, MT-D* Lite has to perform forward (but not backward) searches in order to reuse the search tree rooted in the current state of the agent. When the agent observes new blocked states near it, the resulting action cost increases are close to the current state of the agent and thus the start state of MT-D* Lite. MT-D* Lite then has to update a large number of $g$-values and hence expands a large number of states per search, resulting in a large runtime. Therefore, MT-D* Lite runs more

---

3FRA* and G-FRA* are designed for known static terrains and hence do not apply to unknown static terrains.

slowly than Forward Repeated A*. For example, Forward Repeated A* runs faster than MT-D* Lite by a factor of 2.23 in random grids, 1.58 in random mazes, 2.29 in the game map of size 512 \times 512 and 2.33 in the game map of size 676 \times 676. Thus, MT-D* Lite and Basic MT-D* Lite should not be used in unknown static terrains.

Third, I compare the search tree transforming incremental search algorithms with the heuristic learning incremental search algorithms. Table 5.4 shows the following relationships:

For all three kinds of grids, MT-D* Lite has a larger runtime per search than Forward GAA*. This is because Forward GAA* updates the $h$-values to make them more informed over time and hence expands fewer states per search than Forward Repeated A*, resulting in a smaller runtime per search than Forward Repeated A*. Moreover, as discussed earlier, MT-D* Lite has a larger runtime per search than Forward Repeated A*. Thus, Forward GAA* runs faster than MT-D* Lite. For example, Forward GAA* runs faster than MT-D* Lite by a factor of 2.46 in random grids, 5.44 in random mazes, 2.62 in the game map of size 512 \times 512 and 2.44 in the game map of size 676 \times 676.

5.3.3.1 Summary

Finally, I summarize the results in Table 5.4. The runtime relationships of the compared search algorithms are shown in Figure 5.6. An arrow points algorithm A to algorithm B iff the runtime per search of algorithm A is smaller than the one of algorithm B. Figure 5.6 shows the following relationships:

---

5I compare MT-D* Lite against Forward GAA* only, because MT-D* Lite runs faster than Basic MT-D* Lite and Forward GAA* runs faster than Backward GAA* in unknown static terrains.
For all three kinds of grids, Forward GAA* has the smallest runtime per search among all compared search algorithms.

For all three kinds of grids, search algorithms that search forward have smaller runtimes per search than search algorithms that search backward.

For all three kinds of grids, MT-D* Lite and Basic MT-D* Lite have larger runtimes per search than Forward Repeated A*. Thus, MT-D* Lite and Basic MT-D* Lite should not be used for moving target search in unknown static terrains.

Overall, Forward GAA* is the fastest incremental search algorithm for moving target search in unknown static terrains.

5.4 Conclusions

In this chapter, I have systematically evaluated the runtime of the developed incremental search algorithms for moving target search with respect to (1) different terrains introduced in Section 1.1; (2) different target movement strategies; (3) different numbers of action
cost changes in known dynamic terrains and (4) different kinds of grids introduced in Section 5.2.2. Nevertheless, there are limitations on the experimental setups:

• First, the only state space representation used for the experiments is four-neighbor grids. I have not performed more experiments with different state space representations, such as eight-neighbor grids and polygonal maps (Rabin, 2002) in order to get more comprehensive comparison of the developed incremental search algorithms. As demonstrated in this chapter, the developed incremental search algorithms run even more slowly than both Forward and Backward Repeated A* in terrains where the number of action cost changes is large. Thus, it is expected that the developed incremental search algorithms do not work well in high dimensional state spaces with large branching factors (= the number of successor states of each state) in which the action costs can change between searches, because even a small change in the state space (for example, one observed obstacle) can typically affect a large number of action costs in this state space.

• Second, the sizes of the grids used for all experiments are about $500 \times 500$, which are about the same size as the game maps used by the game company Bioware.

Table 5.5: Best Incremental Search Algorithms for Different Terrains

<table>
<thead>
<tr>
<th>Known Terrains</th>
<th>User-provided h-value</th>
<th>Small Number of Cost Changes</th>
<th>Medium Number of Cost Changes</th>
<th>Large Number of Cost Changes</th>
<th>Static Terrains</th>
</tr>
</thead>
</table>

Unknown Terrains

| Well-informed | MT-D* Lite [1.05] |
| Ill-informed  | MT-D* Lite [3.44] |

183
have not varied the size of the grids in order to get more comprehensive comparison of the developed incremental search algorithms.

- Third, the action costs can only change from one to infinity or from infinity to one in the experiments. I have not varied the values that the action costs can change to (For example, the action cost can change from one to a finite value.) in order to get more comprehensive comparison of the developed incremental search algorithms.

- Fourth, the speed of the target is smaller than the agent. I have not varied the ratio of the speed of the target and the agent in order to get more comprehensive comparison of the developed incremental search algorithms.

- Fifth, the sensor range of the agent is set to one cell in unknown static terrains. I have not varied the sensor range of the agent in order to get more comprehensive comparison of the developed incremental search algorithms in unknown static terrains.

- Sixth, the priority queues of the developed incremental search algorithms are implemented by using binary heaps only. I have not varied the data structure of the priority queues by using other containers, such as buckets, in order to get more comprehensive comparison of the developed incremental search algorithms.

- Seventh, the memory usage of the developed incremental search algorithms are not evaluated, since they have memory requirements that are only in linear in the number of states in the state space, which often can be satisfied by many applications, such as computer games. This measurement can become important and should not
be skipped when applying the developed incremental search algorithms to applications with high dimensional state spaces with large branching factors because those applications requires a large amount of memory, which might not be easily satisfied.

- Eighth, the experiments focus mainly on the test cases where there exists a path from the agent to the target. I have not focused on the test cases where there does not exist a path from the agent to the target, for which the developed incremental search algorithms do not run faster (or may even run more slowly) than Forward and Backward Repeated A*.

Table 5.5 lists the search algorithms with the smallest runtime per search in different scenarios. The numbers in square brackets are the smallest ratio of the runtime per search of Repeated A* and the runtime per search of the fastest search algorithm in different kind of grids. The runtime per search of Repeated A* is the smaller runtime per search between Forward Repeated A* and Backward Repeated A*.

- In known static terrains, FRA* and G-FRA* have the smallest runtime per search of all search algorithms described in Section 5.2.1. FRA* is optimized for grids and does not apply to other state space representations, such as state lattices, while G-FRA* applies to arbitrary state space representations.

---

For example, in known static terrains, when the user-provided $h$-values are well-informed and the movement strategy of the target is either Random Waypoint (see Table 5.1) or TrailMax (see Table 5.2), FRA* has the smallest runtime per search among all compared search algorithms. When the movement strategy of the target is Random Waypoint, the smallest ratio of the runtime per search of Repeated A* and the runtime per search of FRA* ($= 4.96$) is calculated by dividing the runtime per search of Forward Repeated A* ($= 238$ microseconds) by the one of FRA* ($= 48$ microseconds) in the game map $512 \times 512$. When the movement strategy of the target is TrailMax, the smallest ratio of the runtime per search of Repeated A* and the runtime per search of FRA* ($= 3.25$) is calculated by dividing the runtime per search of Backward Repeated A* ($= 231$ microseconds) by the one of FRA* ($= 71$ microseconds) in the game map $512 \times 512$. Thus, the smallest ratio reported is $\min(4.96, 3.25) = 3.25$. 

185
In known dynamic terrains, if the user-provided \( h \)-values are well-informed, MT-D* Lite has the smallest runtime per search, because the ratio of the number of states deleted from the search tree and the total number of states in the search tree after each search is very small. Therefore, MT-D* Lite can reuse a large portion of the search tree from the previous search for the current search. If the user-provided \( h \)-values are ill-informed, Backward GAA* has the smallest runtime per search when the number of action cost changes is small. When the number of action cost changes is medium, MT-D* Lite has a smaller runtime per search than Backward GAA* because the overhead of the consistency procedure of Backward GAA* becomes large. Finally, when the number of action cost changes is large, both Backward GAA* and MT-D* Lite have larger runtimes per search than Forward Repeated A*. In such a situation, incremental search algorithms should not be used, since their computational effort to reuse information from previous searches to speed up the current search is larger than the computational savings gained from reusing the information.

In unknown static terrains, Forward GAA* has the smallest runtime per search because it updates the \( h \)-values to make them more informed over time. Basic MT-D* Lite and MT-D* run more slowly than Forward Repeated A* and Forward GAA* since the action cost changes are close to the current state of the agent (= start state) between searches. Basic MT-D* Lite and MT-D* Lite then have to update a large number of \( g \)-values and hence expand more states per search than Forward and Backward Repeated A*. 

186
In this chapter, it has been demonstrated that the developed incremental search algorithms have smaller runtimes per search than both Forward and Backward Repeated A* for moving target search in different classes of terrains. However, there are scenarios in which their runtimes per search can be larger than both Forward and Backward Repeated A*. We need to be aware of these worst case scenarios and avoid using the developed incremental search algorithms in such scenarios:

- Heuristic learning incremental search algorithms, such as GAA*, have a smaller runtime per search than Repeated A* with the same search direction in known static and unknown static terrains, because GAA* updates its $h$-values to make them more informed and hence focus its search better. However, when a larger number of action costs decrease in known dynamic terrains, the consistency procedure needs to perform more state propagations to maintain the consistency of the $h$-values, which can be computationally expensive. Thus, GAA* has a larger runtime per search than Repeated A* with the same search direction because of the overhead of the consistency procedure.

- Search tree transforming incremental search algorithms, such as MT-D* Lite, have a smaller runtime per search than both Forward and Backward Repeated A* in known static terrains, because MT-D* Lite reuses the part of search tree of the previous search rooted in the start state of the current search. However, MT-D* Lite has a larger runtime per search than both Forward and Backward Repeated A* in the following scenarios:
– In unknown static terrains, when the agent observes new blocked states near it, the resulting action cost increases are close to the current state of the agent and thus the start state of MT-D* Lite. MT-D* Lite then has to update a large number of $g$-values and hence expands a large number of states per search, resulting in a large runtime per search.

– In known dynamic terrains, if the user-provided $h$-values in the state space are ill-informed, then the ratio $r$ of the number of states deleted from the search tree and the total number of states in the search tree after each search of MT-D* Lite is large. Thus, MT-D* Lite can only reuse a small portion of the search tree of the previous search. Moreover, when a larger number of action costs change, MT-D* Lite also has to update a large number of $g$-values and hence expands a large number of states per search. Thus, when both the user-provided $h$-values in the state space are ill-informed and a large number of action costs change, MT-D* Lite has a larger runtime per search than both Forward and Backward Repeated A*.

Overall, the fastest incremental search algorithms that I have developed achieve runtimes per search below 1 ms in known static terrains, unknown static terrains and known dynamic terrains when the number of action cost changes is small (that is, when $k \leq 100$). For example, FRA* achieves a runtime of 0.127 ms per search in the game map $676 \times 676$, which is one order of magnitude faster than Repeated A* in known static terrains. This runtime speedup is important given that all experiments have been performed on a relatively fast computer available in 2011, a Linux PC with an Intel Core 2 Duo 2.53 Ghz.
CPU. On the other hand, many current mobile devices, such as cell phones or touchpads, have slower CPUs than the one I used for the experiments. However, game developers are designing more and more computer games based on mobile devices nowadays, which means that the search algorithms have to satisfy the runtime requirements with relatively slow CPUs. Thus, the developed incremental search algorithms have a good potential to be applied to computer games on mobile devices given their large speedups over Repeated A*.
Chapter 6: Applications

In this chapter, I give an overview of applications of the developed incremental search algorithms. First, I describe an application of GAA*, namely, the Flexible Adaptive Probabilistic RoadMap planner (FAPRM planner) (Belghith et al., 2010) in Section 6.1. Second, I describe an application of G-FRA*, namely, applying G-FRA* to unmanned ground vehicle (UGV) navigation on state lattices (Sun et al., 2010a) in Section 6.2. Third, I describe an application of MT-D* Lite, namely, the JPathPlan path planner (Anand, 2011) in Section 6.3.

6.1 Generalized Adaptive A* Application

Figure 6.1: Interface of the Roman Tutoring System (Belghith et al., 2010)
In this section, I describe an application of GAA*, namely, the Flexible Adaptive Probabilistic RoadMap planner (FAPRM planner) (Belghith et al., 2010), which has been developed in the context of the ROMAN Tutoring System (shown in Figure 6.1) by Khaled Belghith and Froduald Kabanza from the University of Sherbrooke (Canada) and Leo Hartman from the Canadian Space Agency for training astronauts to operate the Space Station Remote Manipulator System (SSRMS). I describe the operations of the FAPRM planner in a way that is sufficient for understanding its principles. A detailed description of the application can be found in (Belghith et al., 2010) and Khaled Belghith’s dissertation (Belghith, 2010).

6.1.1 Background

The Space Station Remote Manipulator System (SSRMS) (shown in Figure 6.2) onboard the International Space Station (ISS) is a 17 meter long articulated robot arm (Currie & Peacock, 2002). It has a complex geometry, with seven rotational joints, each with a range of 270 degrees. There are fourteen cameras covering different parts of the ISS,
which are able to observe the operations of the SSRMS. The SSRMS is a key component of the ISS and has been used in the assembly, maintenance and repair of the station, and also for moving payloads from visiting shuttles. Astronauts operate the SSRMS through a workstation located inside one of the ISS compartments.

The ROMAN Tutoring System is used to train the operators of the SSRMS before they are sent to the ISS to accomplish their missions. The tutoring system is important for providing an unlimited supply of training examples for astronauts to gain and maintain their skills for long duration missions. In essence, the ROMAN Tutoring System observes an astronaut operating a simulation of the SSRMS to accomplish a task in the simulation environment. Every few seconds the system has to determine whether the astronaut is keeping the simulated SSRMS close to a path that achieves the task. If the simulated SSRMS is too far away from such a path, the tutoring system intervenes and shows the astronaut a more appropriate path generated by the FAPRM planner embedded in the tutoring system. The action costs of the SSRMS can change over time depending on the orbit of the ISS and the visibility of the corresponding area via the camera on-board the ISS since the SSRMS prefers a path to avoid regions with limited visibility in order to see the robot motion well. The visibility of an area can change over time. For example, a camera may point towards the sun as the SSRMS moves, resulting in visibility changes of the corresponding area that the camera views (Belghith et al., 2010). I do not provide a detailed description of how the action costs of the SSRMS are modeled since the pseudo code of GAA* is independent of how the action costs are modeled.

Since the FAPRM planner uses a version of Backward GAA*, the goal state (= the current state of the agent) can change over time as the agent moves towards the
target (Belghith et al., 2010). The start state (= the current state of the target) does not change in the experiments performed in (Belghith et al., 2010), although the FAPRM planner can solve problems with moving targets.\(^1\) The FAPRM planner has to be able to generate new paths for the SSRMS as fast as possible as the action costs change to guarantee that the tutoring system runs smoothly and without delay, while A* is generally not fast enough to satisfy the runtime requirements of the tutoring system.\(^1\) In this section, I describe the FAPRM planner with an emphasis on the version of GAA* implemented in the planner.

6.1.2 State Space Representation

The FAPRM planner uses Probabilistic RoadMaps (PRMs) (Lavalle, 2006), a sampling-based representation of the terrain that is different from the discretization-based representation described in Section 2.2. PRMs are most commonly used to efficiently generate paths between any two configurations (Lavalle, 2006). A configuration of an articulated robot with \(n\) degrees of freedom is an \(n\)-element vector of the robot joint positions. The basic idea behind the sampling-based representation is to generate a set of sample configurations and try to connect these configurations to each other through sequences of actions. Each configuration corresponds to a state in the state space representation.

6.1.3 Overview of the Experimental Results

Experiments were performed in the context of training an astronaut to manipulate the SSRMS from a given start configuration to a given goal configuration, which remains

\(^1\) I thank Professor Froduald Kabanza (University of Sherbrooke), one of the designers of the ROMAN Tutoring Systems, to confirm this.
unchanged over time (Belghith, 2010). Since the FAPRM planner uses a version of Backward GAA*, searches are performed from the goal configuration (= start state) to the current configuration of the SSRMS (= goal state). Thus, the goal state can change over time as the SSRMS moves towards the goal configuration. The simulated SSRMS has 7 degrees of freedom and the terrain consists of 75 obstacles modeled with 85,000 triangles. The experiments were run on a 2.86 GHZ Core 2 Processor with 2GB of RAM. The experiments were run three times with the operator doing exactly the same manipulations to reach the goal configuration. The average runtime per search of the FAPRM planner and the SBL planner (Sanchez & Latombe, 2001) are reported. Except for the first few searches, the FAPRM planner runs faster than the SBL planner, because the FAPRM planner uses GAA* to update the $h$-values after each search to make them more informed and thus future searches more focused. The FAPRM planner runs faster than the SBL planner by up to a factor of four (shown in Figure 1.3 of (Belghith, 2010)).

6.1.4 Discussion

This application demonstrated that the FAPRM planner that uses Backward GAA* achieves a smaller runtime per search than the SBL planner in a state space representation that is different from grids. The FAPRM planner combines the incremental search algorithm (Backward GAA*) with the sampling-based representation of the state space so that the planner can be applied to robotic applications with high dimensional state spaces. The sampling-based representation of the state space can significantly reduce the number of states in the state space compared to the discretized representation of the state space (see Chapter 2) at the expense of potentially increase the cost of the
path of each search. I believe that combing incremental search with the sampling-based representation of the state space is a promising direction to apply incremental search algorithm to applications with high dimensional state spaces in the future. Nevertheless, there are limitations on the experimental setups to demonstrate the runtime per search of the FAPRM planner:

- First, all experiments are performed in situations where the target does not move between searches, although the FAPRM planner can solve problems with moving targets. Thus, the experimental results do not directly demonstrate the performance of the FAPRM planner for moving target search.

- Second, the FAPRM planner uses a sampling-based representation of the state space, resulting in an increased cost of the path found by each search. The experimental results have not shown the ratio of the cost of the path found and the minimum cost.

- Third, the runtime per search of Backward GAA* is affected by the informedness of the $h$-values used during each search. It is unclear how informed the user-provided $h$-values are in this application, and by how much the updated $h$-values are more informed than the user-provided $h$-values.

### 6.2 Generalized Fringe-Retrieving A* Application

In this section, I describe an application of G-FRA*, namely, applying G-FRA* to moving target search on state lattices, which are popular kinds of state spaces for unmanned ground vehicle (UGV) navigation (Pivtoraiko & Kelly, 2005a, 2005b; Kelly, Howard, &
Green, 2007; Pivtoraiko, Knepper, & Kelly, 2009; Likhachev & Ferguson, 2009; Kushleyev & Likhachev, 2009). A detailed description of the application can be found in (Sun et al., 2010a).

6.2.1 Background

Moving target search is important for robotics applications. For example, UGVs often have to follow other friendly or hostile UGVs or move from one surveillance position to the next one as their surveillance target moves. FRA* and G-FRA* are experimentally the fastest and the second fastest incremental search algorithms for moving target search in known static grids, respectively (see Chapter 5). Although grids are popular test domains for agent navigation in artificial intelligence (Ishida & Korf, 1991; Moldenhauer & Sturtevant, 2009b; Hernández & Meseguer, 2005; Chimura & Tokoro, 1994; Undeger & Polat, 2007), they are rather unrealistic models for UGV navigation since they are unable to model motion constraints. Unfortunately, FRA* uses geometric properties that are specific to grids and thus does not apply to arbitrary graphs. State lattices (described in Section 2.2.2) are popular state spaces for UGV navigation that are able to model motion constraints of UGVs. Fortunately, G-FRA* does not suffer from being specific to grids and still runs fast on state lattices. I therefore apply G-FRA* to moving target search in known static state lattices.

6.2.2 State Space Representation

State lattices can accurately model the motion constraints of UGVs and guarantee that the paths found are feasible for them to follow. This makes state lattices well-suited
for path planning for UGVs, and thus they have been used widely for UGV navigation (Likhachev & Ferguson, 2009; Kushleyev & Likhachev, 2009; Pivtoraiko et al., 2009; Rufli & Siegwart, 2010).

I implemented G-FRA* in the Search-Based Planning Library (SBPL) (Likhachev, 2009), a publicly available library for UGV navigation with state lattices, which is written in C++. SBPL discretizes the orientation of the UGV into multiples of 22.5° and constructs the state lattices from three inputs, namely the size of the UGV, a terrain
definition file and a motion primitive definition file. The terrain definition file specifies
the size of the grid, the size of each cell, the cells of the obstacles, the translational and
rotational velocities of the UGV and the start and goal states of the UGV. The motion
primitive definition file defines the motion primitives for each state. The cost of a motion
primitive is its execution time. Figure 6.3 shows an example UGV path for a state lattice
constructed by SBPL. I use the default size of the UGV, namely 20 centimeters \times 4
centimeters. I use one of the example terrain definition files (\texttt{env2.cfg}), which defines
the size of the grid to be 1200 cells \times 100 cells, the size of each cell to be 2.5 centimeters
\times 2.5 centimeters, the translational velocity of the UGV to be 1 meter per second and
the rotational velocity of the UGV to be 22.5° per second.

### 6.2.3 Overview of the Experimental Results

I use four motion primitive definition files together with the default $h$-values, namely the
amount of time needed for the UGV to move to the target at maximal speed in a straight
line in the absence of obstacles and motion constraints. Figure 6.4 shows the motion
primitives defined in the four files for state $(8, 2, 90°)$. The target moves randomly but
skips every tenth move to enable the UGV to catch it. The problem is solved when the
UGV and the target are in the same state. Experimental results show that G-FRA* has
a smaller runtime per search than Forward and Backward Repeated A* and Forward and
Backward GAA* for all motion primitive definition files, because it reuses parts of the
search tree of the previous search and hence expands fewer states per search than Forward
and Backward Repeated A* and Forward and Backward GAA*. The runtime per search
of G-FRA* is smaller than the one of Forward Repeated A* by factors of 12.54 to 21.28
and smaller than the one of Forward GAA* by factors of 6.52 to 11.45. The runtime per search of G-FRA* is smaller than the one of Backward Repeated A* by factors of 10.58 to 20.32 and smaller than the one of Backward GAA* by factors of 3.28 to 4.58. Overall, the experimental results of G-FRA* on state lattices are similar to the experimental results on grids that were shown in Section 5.3.1. G-FRA* runs up to one order of magnitude faster than both Forward and Backward Repeated A* (Sun et al., 2010a). The source code of SBPL that I extended for this application is available at (Sun, 2012).

### 6.2.4 Discussion

The implementation of G-FRA* in SBPL demonstrated that G-FRA* is up to one order of magnitude faster than both Forward and Backward Repeated A* when applied to UGV navigation on state lattices, which extends the applicability of the newly developed incremental heuristic search algorithms for moving target search in a different state space from grids. Moreover, the priority queues of the compared search algorithms in SBPL are implemented by using the hashed buckets, which are different from the binary heaps used in Chapter 5. Thus, the experimental results demonstrate that G-FRA* runs faster than both Forward and Backward Repeated A* when the priority queues are implemented with different data structures. Nevertheless, since I wanted to use similar experimental settings as earlier experiments in Chapter 5, I did not attempt to build on a realistic simulation of moving target search with UGVs, and there are limitations on the experimental setups:

- First, I used a coarse-grained discrete simulation on the motion-primitive level.
- Second, the UGV and target took turns executing uninterruptible actions with potentially different execution times.

- Third, the UGV and target had symmetrical motion capabilities.

- Fourth, the motion strategy of the target was simplistic.

- Fifth, I assumed the exact position for the target, and did not try to run into the target.

It is therefore my future work to evaluate G-FRA* as part of a realistic simulation of moving target search with UGVs, where (a) a motion simulator provides a continuous simulation, (b) the UGV and target are able to move in parallel, (c) the UGV and target have different motion capabilities, and (d) the target uses a more sophisticated motion strategy.

6.3 Moving Target D* Lite Application

In this section, I describe an application of MT-D* Lite, the JPathPlan path planner, which has been implemented by Abhijeet Anand from the Royal Melbourne Institute of Technology (RMIT) in the RACT-GOAL agent system for the International Multi-Agent Contest (Dastani, Dix, & Novak, 2005). A detailed description of the application can be found in Abhijeet Anand’s Master’s thesis (Anand, 2011).
6.3.1 Background

The International Multi-Agent Contest started in 2005 and provides a platform for researchers from all over the world to demonstrate their research in multi-agent systems design, development and programming (Anand, 2011).

The gold-mining game in the International Multi-Agent Contest involves two teams of six agents each, interacting with a remote simulation server. The game runs on a grid. The agents need to collect gold and deposit it into a depot. They need to deposit gold when they reach their maximum carrying capacity. They are not allowed to enter the depot if they are not carrying any gold. However, if they are carrying gold and enter the depot but do not deposit it, they are punished by being teleported to a random location on the grid. The agents initially have no information about certain areas of the grid. Moreover, moving target search is necessary since the gold piece (= target) can be moved from its current cell to one of the neighboring cells before the agent arrives. Finally, path planning must be fast to allow the agent to collect the gold quickly (Anand, 2011).

The RACT (= RMIT Agent Contest Team) group of the Royal Melbourne Institute of Technology has designed the RACT-GOLD system for participating in the gold-mining game (Yadav, Zhou, Sardina, & Rönnquist, 2010). However, the path planner in the RACT-GOLD system was too slow. Thus, they developed JPathPlan, a new path planner for the RACT-GOLD system based on MT-D* Lite.

6.3.2 State Space Representation

The gold-mining game maps are represented as grids. The agents, gold pieces and the depot are placed in unblocked cells. The content of a cell is initially unknown to an
agent. In each time step, the game server sends each agent a request-action message. A request-action message contains information about the map, the agent’s current $x$- and $y$- coordinate on the grid, the time step number and the number of gold pieces that the agent carries. The agent can respond to a request-action message by sending an action message. This action message can either encode a movement action (up, left, down or right), a pick up or drop gold action, or a skip action. The game server imposes a timeout period for every request-action message that is sends to the agents. If an agent fails to reply within the timeout period, the game server considers this as a skip action. If an agent sends an action message with a skip action while it is in the depot, it is teleported to a random unblocked cell on the grid (Anand, 2011).

6.3.3 Overview of the Experimental Results

Both Forward Repeated A* and MT-D* Lite have been implemented in the JPathPlan planner (in JAVA) and compared in the context of the gold-mining game. The following performance measures have been reported in (Anand, 2011):

- total time spent on search;
- time per search, which measures how quickly an agent responds to the server after it receives a request-action message from the server; and
- the number of time steps skipped due to path planning taking longer than the timeout period for a time step.

Simulation results show that MT-D* Lite outperforms Forward Repeated A* with respect to all performance measures (Anand, 2011). For example, in four-neighbor grids
of size $300 \times 300$ (shown in Table 4.2 in (Anand, 2011)), MT-D* Lite runs faster than Forward Repeated A* by a factor of 3.47 when 8\% of randomly chosen cells are blocked, 5.86 when 9\% of randomly chosen cells are blocked, and 7.96 when 10\% of randomly chosen cells are blocked. The developer of the JPathPlan planner concluded that MT-D* Lite provides a substantial improvement over Forward Repeated A* in the context of the gold-mining game of the International Multi-Agent Contest (Anand, 2011).

### 6.3.4 Discussion

The JPathPlan planner demonstrated that MT-D* Lite can achieve a smaller runtime per search than Forward Repeated A* in the context of the gold-mining game, which is similar to the results reported in Chapter 5. Nevertheless, there are limitations on the experimental setups to demonstrate the feasibility of MT-D* Lite when applied to the moving target search applications in multi-agent systems:

- First, although the RACT-GOLD system consists of six agents, the developer only assigned one of the agents the task of path planning. The other agents in the system move randomly and provide location updates to the single agent that is assigned the task of path planning.

- Second, only one incremental heuristic search algorithm is implemented in the RACT-GOLD system. Since the runtime per search of MT-D* Lite can be affected by the informedness of the user-provided $h$-values used during each search, it is unclear how informed the user-provided $h$-values are in the terrain used, and whether
Backward GAA* (which has not been implemented in the JPathPlan planner) can have a smaller runtime per search than MT-D* Lite or not.

6.4 Conclusions

In this chapter, I described applications of the developed incremental search algorithms. In Section 6.1, I described the Flexible Adaptive Probabilistic RoadMap planner (FAPRM planner), that uses Backward GAA* for moving target search. The FAPRM planner has been implemented in the ROMAN Tutoring System for training astronauts to operate the Space Station Remote Manipulator System. Results showed that the FAPRM planner runs faster than the SBL planner by up to a factor of four. In Section 6.2, I described how I applied G-FRA* to moving target search for unmanned ground vehicle navigation on state lattices. Results showed that G-FRA* runs faster than both Forward and Backward Repeated A* by up to one order of magnitude. In Section 6.3, I described the JPathPlan path planner that uses MT-D* Lite for moving target search. The JPathPlan path planner has been implemented by researchers from the Royal Melbourne Institute of Technology in their RACT-GOLD system, a multi-agent system designed for the International Multi-Agent Contest. Results showed that JPathPlan with MT-D* Lite runs faster than JPathPlan with repeated A* by up to a factor of 7.96. Thus, the developed incremental search algorithms provide a substantial improvement over repeated A* in applications of moving target search.
Chapter 7: Conclusions

In moving target search, an agent’s information about the terrain can change and the target can move over time. Thus, the agent often needs to repeatedly perform searches to find new cost-minimal paths to the target. Incremental search algorithms can reuse information from previous searches to speed up the current search and hence find cost-minimal paths for series of similar path planning problems faster than is possible by solving each path planning problem from scratch. Incremental search algorithms generally fall into two classes:

- Heuristic learning incremental search algorithms reuse information from the previous searches to update the h-values of the current search so that they become more informed and focus the current search better.

- Search tree transforming incremental search algorithms reuse the search tree of the previous search for the current search so that the current search does not need to start from scratch.

However, there did not exist incremental search algorithms that apply to moving target search, where both the start and goal states can change between searches, and run faster than A*. Thus, I have developed new incremental search algorithms that
significantly speed up search-based path planning for moving target search. The key contributions of this dissertation are as follows:

- In Chapter 3, I developed new heuristic learning incremental search algorithms for moving target search. In particular, I developed Generalized Adaptive A* (GAA*) (Sun et al., 2008), an incremental A* variant that extends MT-Adaptive A* (Koenig et al., 2007) to moving target search in terrains where the action costs of the agent can both increase and decrease between searches. Thus, GAA* applies to known static, unknown static and known dynamic terrains. I demonstrated experimentally that Forward GAA* runs faster than Backward GAA*, Forward and Backward Repeated A*, Basic MT-D* Lite and MT-D* Lite for moving target search in unknown static terrains, and Backward GAA* runs faster than Forward GAA*, Forward and Backward Repeated A*, Basic MT-D* Lite and MT-D* Lite for moving target search in known dynamic terrains when the user-provided $h$-values are ill-informed and the number of action cost changes is small.

- In Chapter 4, I developed new search tree transforming incremental search algorithms for moving target search:

  - First, I developed Generalized Fringe-Retrieving A* (G-FRA*), an incremental A* variant for moving target search in known static terrains. G-FRA* is currently the fastest incremental search algorithm for moving target search in known static terrains, and runs up to one order of magnitude faster than Forward and Backward Repeated A*.
Second, I developed Fringe-Retrieving A* (FRA*) (Sun et al., 2009), an incremental A* variant that optimizes G-FRA* to apply to moving target search in known static grids only. FRA* is currently the fastest incremental search algorithm for moving target search in known static grids, and runs up to one order of magnitude faster than Forward and Backward Repeated A*.

Finally, I developed Moving Target D* Lite (MT-D* Lite) by combining the principles of G-FRA* and D* Lite. MT-D* Lite applies to moving target search in terrains where the action costs of the agent can increase and decrease between searches. Thus, MT-D* Lite applies to known static, unknown static and known dynamic terrains. MT-D* Lite is currently the fastest incremental search algorithm for moving target search in known dynamic terrains when the user-provided $h$-values are well-informed, and runs by up to a factor of eight faster than Forward and Backward Repeated A*.

In Chapter 5, I systematically evaluated the runtime of the developed incremental search algorithms for moving target search with respect to (1) different terrains, including the known static, unknown static and known dynamic terrains; (2) different target movement strategies, including the random waypoints and TrailMax strategies; (3) different numbers of action cost changes in known dynamic terrains and (4) different kinds of grids, including random grids, random mazes and two game maps. I discussed the strengths and weaknesses of them and provided guidelines for when to choose a particular algorithm over others with Table 5.5.
In Chapter 6, I gave an overview of the following applications of the developed incremental search algorithms.

- For GAA*, I described the Flexible Adaptive Probabilistic RoadMap (FAPRM) planner by researchers from the University of Sherbrooke and the Canadian Space Agency, that uses Backward GAA*. The FAPRM planner has been used in the ROMAN Tutoring System for training astronauts to operate the Space Station Remote Manipulator System.

- For G-FRA*, I applied G-FRA* to unmanned ground vehicle (UGV) navigation in state lattices.

- For MT-D* Lite, I described the JPathPlan path planner by researchers from the Royal Melbourne Institute of Technology (RMIT). The JPathPlan path planner has been used in their RACT-GOLD system, a multi-agent system designed for the International Multi-Agent Contest.

For future work, I now discuss possible approaches to speed up path planning for moving target search even further:

- Currently, there does not exist any incremental search algorithm that both learns $h$-values and reuses search trees from the previous searches for moving target search. Thus, one possible approach to speed up path planning for moving target search is to combine heuristic learning incremental search algorithms with search tree transforming incremental search algorithms.
• Anytime search algorithms (Likhachev et al., 2007), such as Anytime Repairing A* (ARA*) (Likhachev, Gordon, & Thrun, 2003), solve a single search problem by quickly finding an initial, error-bounded suboptimal path and then improving this path until the available time runs out (Likhachev et al., 2007). Thus, one possible approach to speed up path planning for moving target search is to combine anytime search algorithms with incremental search algorithms, resulting in anytime incremental search algorithms.

• State abstraction techniques (Holte et al., 1996; Bulitko, Sturtevant, Lu, & Yau, 2007) map several states in the state space to a single abstract state. The abstract state space consists of all abstract states. Thus, the number of abstract states in the abstract state space can be much smaller than the number of states in the state space. The abstract states that the start and goal states are mapped to become the abstract start and goal states. One can first find a path between the abstract start and goal states in the abstract state space, which can be faster than finding a path between the start and goal states in the state space since the number of abstract states is smaller than the number of states, and then refine this path to a path in the state space, which can be used by the agent for navigation.

• Currently, the developed search tree transforming incremental search algorithms, such as FRA* and MT-D* Lite, require the agent to move along the path found by the previous search so that the agent (= start state of each search) cannot move outside the search tree of the previous search. However, in realistic applications, such as UGV navigation, there are situations where the agent can accidentally move
off the path found by the previous search and move outside the search tree of the
previous search. In these situations, the developed search tree transforming incre-
mental search algorithms cannot reuse any part of the search tree of the previous
search. Thus, one potential research direction is to investigate situations where the
agent can move outside the search tree of the previous search, and develop new
search tree transforming incremental search algorithms that can reuse the search
trees in such situations.
Bibliography


