REAL-TIME PLANNING FOR ROBOTS

BY

Bence Cserna

MS of Computer Engineering, Budapest University of Technology and Economics,
Budapest, Hungary 2013
BS of Computer Engineering, Budapest University of Technology and Economics,
Budapest, Hungary 2011

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

May, 2019

This dissertation has been examined and approved in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science by:

On April 12th, 2019

Approval signatures are on file with the University of New Hampshire Graduate School.

To my wife and family.

ACKNOWLEDGMENTS

This long journey far from home would have not been possible without the endless and unconditional support of my loving wife Dóri. The courage she gave me guided me through the most challenging times of my PhD.

I'm forever grateful for the mentorship of Wheeler Ruml. I have learned more from Wheeler than I ever anticipated, far beyond the limits of heuristic search and research. His vast knowledge is truly humbling. He was always available for me when I needed guidance and could point me in the right direction even in the most confusing moments. Wheeler taught me that there is no substitute for hard work and always encouraged me to achieve the research goals I was most passionate about.

This work would have not been possible without the long discussions and help of Will, Sammie, Jordan, Kevin, Tianyi, Marek, Reazul, and many others. Their unique perspective shaped my research.

I could not have made it this far without the support and care of all people who became my friends during my stay at UNH and those who didn't let the distance fade our friendship. They kept me sane, helped me find joy even after long hours of research.

Lastly, I gratefully acknowledge support from the NSF (grant 1150068), the UNH Department of Computer Science, and the UNH Graduate School.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

REAL-TIME PLANNING FOR ROBOTS

by

Bence Cserna

University of New Hampshire, May, 2019

Autonomous robotic systems are becoming widespread in the form of self-driving cars, drones, and even as consumer appliances. These systems all have a planner that makes them autonomous. The planner defines the way these robots evaluate and select among the possible actions that are available to them. This dissertations is about a specific type of planning called on-line real-time planning that is especially applicable to autonomous robots.

The central thesis of this work is that real-time heuristic search can be a viable planning method for complex state spaces. Planning for autonomous agents requires novel methods that are not direct adaptations of off-line planning methods but designed specifically for the task to provide fast and reliable execution while keeping the agent safe to guarantee that the goal will be reached. While there are many approaches to planning for embodied agents, my work pursues a class of on-line planning methods called real-time heuristic search that provide real-time bounds on action selection time.

This dissertation makes three major contributions. Traditional real-time search algorithms are not guaranteed to recognize a subgraph from which the goal is not reachable before entering it, thus they are inherently incomplete in domains with dead-ends when a time bound is imposed on the planner. The first contribution of my dissertation addresses this issue by introducing a real-time search method with completeness guarantees in domains with dead-ends. Real-time planning in the presence of local minima is particularly challenging due to the bounded rationality of real-time decision making. The second contribution of my dissertations is to introduce novel methods to mitigate this deficiency of real-time search.

Real-time search methods should be designed specifically to optimize the metric of interest: goal achievement time. Having more time to think leads to more informed and better quality decisions that can improve the overall goal achievement time. The third contribution of my dissertation is the introduction of a real-time metareasoning technique that considers actions that do not lead to the best discovered node, according to the commonly used f metric, in order to provide more time to the agent to plan the upcoming iteration.

Together these contributions support that: real-time heuristic search is a viable planning method for complex state spaces.

CHAPTER 0

Introduction

Autonomous robotic systems are becoming widespread in the form of self-driving cars, drones, and even as consumer appliances. These systems all have a planner that makes them autonomous (Siegwart, Nourbakhsh, & Scaramuzza, 2011). The planner defines the way these robots evaluate and select among the possible actions that are available to them. Planning for robotics is a challenging subject and an active source of open questions in the research community (Siegwart et al., 2011). Planning finds a sequence of actions that enables the autonomous system to reach its goal. This is a particularly hard problem due to the fact that in most cases the number of possible action sequences is more than the number of atoms in the universe, thus it is not possible to consider all these sequences and sophisticated techniques are required if we wish to find the best or even a very good plan that will lead to the goal. My work is about a specific type of planning called on-line real-time planning that is especially applicable to autonomous robots.

Traditionally, planning has been considered from an off-line perspective, in which a complete plan is created before its execution begins. However, time is often a very limited resource for humans. For us, the fine details of the plan the autonomous system carries out are often irrelevant, but the time it takes the robot to complete the task matters. Consider an autonomous vacuum cleaner that has to clean up a spot in a room: it is preferred to plan a 10s trajectory in 2s than a 5s trajectory in 10s, and it would be even better to begin the execution before finding a complete plan. Therefore, many AI systems, such as autonomous robots, demand an on-line planning approach, in which the objective is to achieve the goal as quickly as possible and planning takes place concurrently with execution. The robot may begin the execution of a partial plan before a complete plan is formulated. In these scenarios, our central performance metric is goal achievement time (GAT), which measures the time starting from when the agent receives a goal specification and begins planning to achieve it and ending when the agent reaches a goal state and is done with the task (Kiesel, Burns, & Ruml, 2015). This metric provides a natural way to compare different types of planning algorithms.

While there are many approaches to planning for embodied agents, my work pursues a class of on-line planning methods called real-time heuristic search that provide real-time bounds on action selection time.

The central thesis of my dissertation is that real-time heuristic search can be a viable planning method for complex state spaces. Current real-time search methods suffer from significant drawbacks that limit their viability and potential applications. First, they are incomplete in domains with dead end states. Existing real-time search methods derived from off-line search algorithms tend to fall into inescapable spaces due to the limited planning time. As a direct consequence, current real-time search algorithms cannot be deployed in safety critical systems that exhibit potential dead ends. Second, popular real-time search methods are prone to getting lost in large local minima, which increases the time to satisfy the goal condition. While there are existing methods aimed at avoiding getting lost in local minima, they introduce a new set of issues specific to them. Finally, on-line planning has a characteristic that is truly unique to the setting – that time passes during planning. In off-line planning, planning time and execution time are independent measures, while in on-line planning this is no longer true. Planning time and execution time have the same dimension and effect the time-to-goal directly. Existing methods ignore this fundamental aspect of on-line planning.

Planning for autonomous agents can be enhanced by designing novel methods that are not direct adaptations of off-line planning methods but designed specifically for the task to provide fast and reliable execution while keeping the agent safe to guarantee that the goal will be reached. The chapters of my dissertation inspect the three central limitations and deficiencies of real-time heuristic search and offer novel solutions to address and mitigate those, and to enable new applications for these planning methods.

0.1 Potential of Real-time Heuristic Search

The first chapter analyzes the performance of real-time planning under the GAT metric compared to the leading current approach in robotics, known as anytime planning (Likhachev & Ferguson, 2009). In anytime planning, the planner quickly finds a suboptimal plan, and then continues to find better plans as time passes, until eventually converging on an optimal plan. However, anytime methods have a significant disadvantage: the time until the first plan is returned is not bounded, so such methods inherently imply that the robot cannot start moving until the first complete plan is formulated. In some systems, a delay in starting execution is merely undesirable, but in others such as fixed-wing aircraft, it is inherently infeasible, as the system cannot pause indefinitely without moving. In the first contribution of my dissertation, I compare anytime and real-time heuristic search methods in their ability to allow agents to achieve goals quickly. The results suggest that real-time search is more broadly applicable and often achieves goals faster than anytime search, thus the remainder of my dissertation focuses on real-time heuristic search techniques.

This work appeared in the Proceedings of the Symposium on Combinatorial Search (SoCS-16) (Cserna, Bogochow, Chambers, Tremblay, Katt, & Ruml, 2016).

0.2 Real-time Heuristic Search in the Presence of Dead-ends

The central limitation of current real-time heuristic search algorithms it that they are inherently incomplete in directed domains with dead-end states due to the time constraints. In each planning step, the agent has a limited amount of time to decide which action to take next, thus the number of states explored per planning step is bounded. A traditional real-time search algorithm is not guaranteed to recognize a subgraph from which the goal is not reachable before entering it. Chapter 2 addresses this by introducing a novel and general method that guarantees the safety of the agent, and thus completeness, under certain assumptions. This work appeared in the Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18) (Cserna, Doyle, Ramsdell, & Ruml, 2018).

0.3 Safe Temporal Planning for Urban Driving

While the methods of Chapter 2 are general, it is interesting to examine important special cases. For example, a self-driving car must always have a plan for safely coming to a halt. Often, finding these safe plans is treated as an after-thought. In this chapter, we demonstrate that techniques explicitly designed for safety can yield higher quality plans and lower latency than conventional planners in an urban driving setting. We adopt ideas from Chapter 2 on safe online real-time heuristic search to the spatiotemporal state lattices used when planning for autonomous driving. We experimentally compare our proof-of-concept implementation to conventional methods and find significantly improved performance while still maintaining passenger comfort and safety. This work appeared in the Proceedings of the AAAI-19 Workshop on Artificial Intelligence Safety (SafeAI-19) (Cserna, Doyle, Gu, & Ruml, 2019).

0.4 Framework for Safe Real-time Heuristic Search

The main focus of Chapter 2 is to ensure that the agent is able to safely reach a goal in domains with dead-ends. In this chapter, we introduce a more general real-time heuristic search framework for safety that relies on the foundation outlined in Chapter 2 and incorporates two novel methods to minimize the overhead of dealing with dead-ends. We prove the efficiency of this framework in theory and show that its promise is fulfilled in practice. In addition, this chapter offers a new domain designed to evaluate safety oriented real-time search methods.

0.5 Real-time Planning in the Presence of Local Minima

The second fundamental limitation of real-time heuristic search methods is that they are prone to getting lost in local-minima. Real-time planning in the presence of local minima is particularly challenging due to the bounded rationality of real-time decision making. The local minima have to be filled by the agent before it can avoided or escaped. This behavior (also known as scrubbing) negatively impacts the goal achievement time. Chapter 5 makes contributions to mitigate this deficiency of real-time search.

0.6 Optimization for Goal Achievement Time

Finally, Chapter 6 addresses the third central characteristic of online planning – that time passes during planning. Real-time search methods should be designed specifically to optimize the metric of interest: goal achievement time. Having more time to think leads to more informed and better quality decisions that can improve the overall goal achievement time. Chapter 6 of my dissertation introduces a real-time metareasoning technique that considers actions that do not lead to the best discovered node, according to the commonly used f metric, in order to provide more time to the agent to plan the upcoming iteration. This work appeared in the Proceedings of the Twenty-eighth International Conference on Automated Planning and Scheduling (ICAPS-17) (Cserna, Ruml, & Frank, 2017).

0.7 Conclusion

Together, these contributions address the central limitations of real-time heuristic search and support the main thesis of my dissertation: Real-time heuristic search can be a viable method for complex states spaces. This dissertation demonstrates that real-time heuristic search is not defined by its current limitations, but it can guarantee the safety of a controlled agent in domains with dead-ends states, it can operate in large domains with heuristic depressions, and that it can exploit meta-information specific to the on-line setting in partial search trees to achieve the goal faster.

CHAPTER 1

Potential of Real-time Heuristic Search

In this chapter, I analyze the performance of real-time planning under the GAT metric compared to the leading current approach in robotics, known as anytime planning (Likhachev & Ferguson, 2009). In anytime planning, the planner quickly finds a suboptimal plan, and then continues to find better plans as time passes, until eventually converging on an optimal plan. The framework of anytime planning was used by the winning entry of the DARPA Urban Challenge (Ferguson, Howard, & Likhachev, 2008; Ferguson et al., 2008).

However, anytime methods have a significant disadvantage: the time until the first plan is returned is not bounded, so such methods inherently imply that the robot cannot start moving until the first complete plan is formulated. In some systems, a delay in starting execution is merely undesirable, but in others such as fixed-wing aircraft, it is inherently infeasible, as the system cannot pause indefinitely without moving. In the first contribution of my dissertation, I compare the currently popular anytime and real-time heuristic search methods in their ability to allow agents to achieve goals quickly. This work appeared in the Proceedings of the Symposium on Combinatorial Search (SoCS-16) (Cserna et al., 2016).

The results of this chapter suggest that real-time search is more broadly applicable and often achieves goals faster than anytime search, thus the remainder of my dissertation focuses on real-time heuristic search techniques.

1.1 Introduction

Many AI systems, such as robots, must plan under time constraints. Anytime heuristic search methods have seen wide use in robotics applications. With these methods, the agent can begin execution as soon as the first plan is found and then switch to better plans as they are discovered. An alternative approach is real-time heuristic search. These methods offer guaranteed low latency and allow the agent to begin acting even before a complete plan is found. In this chapter, we compare real-time and anytime search in their ability to allow agents to achieve goals quickly. We discuss how to adapt published algorithms to a real-time environment and present results in simulated domains ranging from discrete grid pathfinding to a continuous double integrator point robot and the non-linear Acrobot. Our results suggest that real-time search is more broadly applicable and often achieves goals faster than anytime search, while anytime search finds smaller length plans and does not suffer from dead-ends. This work provides guidance for those interested in using heuristic search techniques in time-sensitive applications, such as robotics.

Heuristic search is a fundamental planning method for intelligent agents. Not only does it underlie many of the state-of-the-art domain-independent AI task planners, but it has also been used to directly select motion actions for robots (Ruml, Do, Zhou, & Fromherz, 2011). In this chapter, we investigate the application of heuristic search methods to control a range of simulated robot systems of increasing complexity.

Because many robots operate under time pressure, to optimize productivity or because a human is waiting, for example, there is much interest in search methods that allow the system to solve problems quickly. The most popular search approach applied in robotics so far is anytime search, in which the algorithm quickly finds a suboptimal plan, and then continues to find better and better plans as time passes, until eventually converging on an optimal plan (Dean & Boddy, 1988). However, the time until the first plan is returned is not controllable, so such methods inherently involve idling the system's operation before 'real' execution can begin. Real-time search methods provide hard real-time bounds on action selection time, yet to our knowledge, they have not yet been demonstrated for robotic systems. In this chapter, we provide a comparison of the performance of anytime and real-time heuristic search methods in a variety of simulated domains, illustrating the benefits of using real-time search over anytime search.

The performance of offline suboptimal heuristic search algorithms are often measured in planning time (wall-clock time or the number of expanded nodes) and execution time (Thayer & Ruml, 2011; Gilon, Felner, & Stern, 2016). Figure 1-1 shows that for off-line planning algorithms these metrics

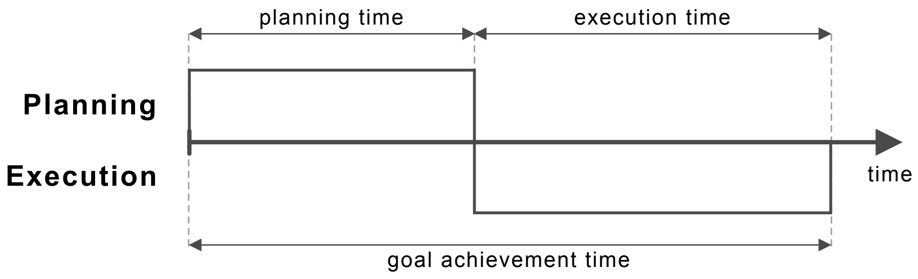


Figure 1-1: Goal achievement time of off-line planning algorithms.

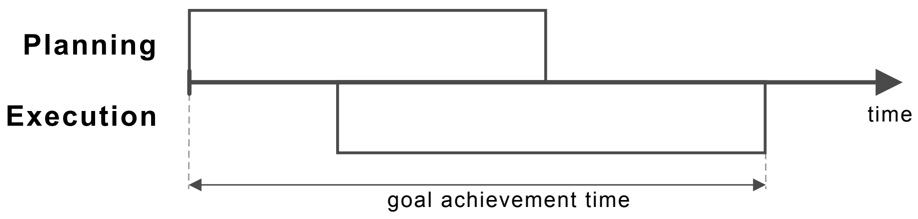


Figure 1-2: Goal achievement time of on-line planning algorithms.

clearly represent an algorithm's performance as the planning time and the execution time are not overlapping. However, for on-line environments in which the planning and execution happen in parallel these metrics become impractical as the execution begins before the termination of the planner (Figure 1-2).

Our central performance metric is goal achievement time (GAT), which measures the time starting from when the agent receives a goal specification and begins planning to achieve it and ending when the agent reaches a goal state and is done with the task (Kiesel et al., 2015). This metric provides a natural way to compare off-line algorithms like A*, which forms a complete plan before beginning execution, with on-line algorithms like real-time search.

We compare real-time and anytime search algorithms in a variety of domains, including those involving the control of dynamical systems. Our results suggest that real-time search can be a better alternative to anytime search for minimizing GAT. It is also applicable in a broader variety of domains. On the other hand, anytime search executes shorter trajectories and does not suffer from dead-ends in the state space. More broadly, this work encourages a closer look at on-line heuristic search and real-time methods in particular.

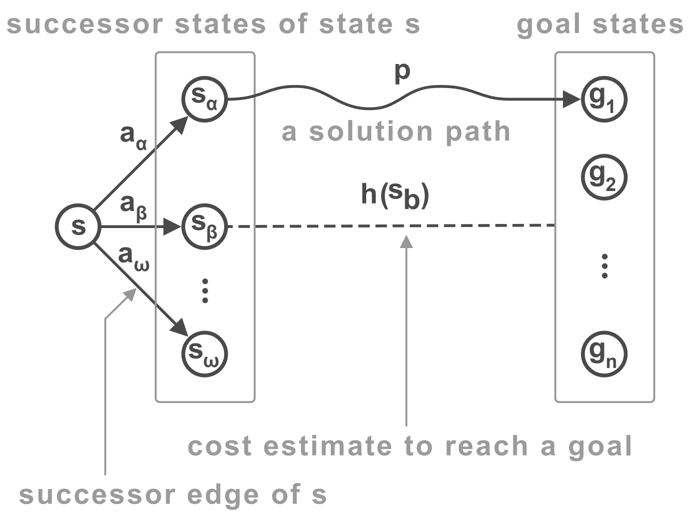


Figure 1-3: Demonstration of state space search.

1.2 State Space Search

This section is a brief introduction to state space search. In this dissertation we rely on the following formulation of state space search to discuss heuristic search methods: Given a set of states S, a set of goal states G, a set of actions A, a successor function (Definition 1), a cost function (Definition 2), and a heuristic function h (Definition 3) and the objective is to find a sequence of actions such that applying those actions starting from the initial state lead to one of the goal states. These concepts are shown on Figure 1-3.

Definition 1 *The* ***successor function*** *returns a set of all possible direct successor states* s' *from a given source state* s *along with the action* a *that leads to the states* s' *from* s*:*

$$successor : s \in S \to X \subseteq S$$

Definition 2 *The* ***cost function*** c *defines the cost of taking an action* a *from a given state* s *to a state* s'*. Due to the deterministic nature of the domains discussed in this dissertation this formulation is redundant and the following invocations of this function refer to the same value:*

$$c(s, a, s') = c(s, a) = c(s, s')$$

Definition 3 *The **heuristic function*** h *estimates the cost of the lowest cost path between a state* s *to a goal state. This is also known as* s*'s heuristic value or simply the heuristic of* s. *The heuristic value of state* s *is denoted as* h(s).

Definition 4 *A heuristic function* h *is **admissible** iff it never overestimates the true cost of reaching a goal state from any given node or more formally:*

$$admissible(h) \Leftrightarrow h(s) \leq h^{*}(s) : \forall s \in S$$

Where $h^{*}(s)$ *is the true cost of reaching the closest goal from state* s.

Definition 5 *A heuristic function* h *is **consistent** iff the heuristic value* h(s) *of any state* s *is less than or equal to the heuristic value of any successor* s' *of* s *plus the cost of getting to* s' *from* s*:*

$$consistent(h) \Leftrightarrow h(s) \leq h(s') + c(s, s') : \forall s' \in successor(s)$$

This formulation implicitly defines a graph where the states are the vertices and the edges are given by the successor function. The size of this implicit graph is super-astronomical for many problems. Search algorithms discussed in the following chapters rely on an explicit tree (or graph) referred to as the search tree (or graph). The search tree is incrementally induced by the search algorithm. The search tree wraps the states (or set of states) from the state space with search nodes and connects these nodes with edges that may or may not exist in the original search space. The structure of the search tree is algorithm specific. The node is a simple data structure containing meta information recorded by the search method (e.g. updated heuristic value or parent node). While there is a clear distinction between states and nodes we will use them interchangeably if there exists a one-to-one mapping between them.

In the context of the search trees, we will refer to the invocation of the successor function on a state as expansion or node expansion. Expanding a node means that the successors of the state represented by the node are added to the search tree as well as to the open list (see below) of the search algorithm, unless otherwise noted.

Throughout this work the algorithms used for state space exploration are based on best-first search. Best-first search algorithms expand nodes until a goal state is reached (expanded) starting from the initial state. The node to be expanded is selected from the successors of the previously expanded nodes based on a priority function that is defined by the algorithm. We refer to this set as open nodes and the data structure that stores it as open list.

Definition 6 *Given a search tree, the* g *value defines the cost of the path leading from the initial state to a node defined by the tree structure.*

Definition 7 *The* f *value of a node* n *is the sum of the* g *value of the node and the* h *value of the state* s *represented by* n.

$$f(n) = g(n) + h(s)$$

f is a commonly used function for best-first search algorithms (Hart, Nilsson, & Raphael, 1968).

The notation above of state space search is used throughout this dissertation. Additional functions (such as distance or d, safety distance d_{safe}, $safety$, $\hat{h}$, and $\hat{f}$) will be defined in the following chapters as needed.

1.3 Anytime Search

Anytime search represents one approach to planning under time pressure. It is perhaps best thought of as planning under an unknown deadline. Anytime search quickly finds a suboptimal plan, and continually finds improved plans as time allows until either the algorithm is terminated or an optimal plan is found.

One of the most well-known anytime search algorithms is Anytime Repairing A* (Likhachev, Gordon, & Thrun, 2004). ARA* searches using weighted A*, which is equivalent to A* with the heuristic inflated by a factor ϵ. The solution achieved by weighted A* is provably within a factor of ϵ of the optimal solution cost. Therefore, the intuition is that when ϵ decreases, the time taken to find the plan increases and the solution converges towards the optimal plan (with Provable Bounds on Sub-Optimality,). ARA* starts by finding a suboptimal plan using Weighted A* with a high ϵ. Then ϵ is lowered, the open list is resorted, and the search continues until an ϵ-admissible solution is found. In this way, ARA* reuses the results of each weighted A* iteration.

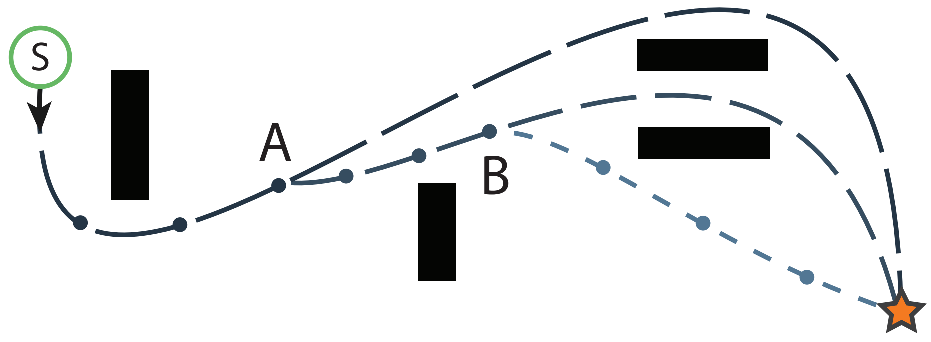


Figure 1-4: ARA* regression planning with $n = 3$

1.3.1 Concurrent Planning and Execution

Although ARA* is presented as a generic anytime algorithm, how it can be used for concurrent planning and execution is also described in Likhachev et al. (2004). As soon as the first plan is found, the agent can begin moving. The advantage of this approach is that the agent can begin moving sooner than if it had to wait for an optimal search to complete. In order for the previous search effort to remain applicable, however, the search is actually performed starting at the goal state and searching towards the agent. This means that, even though the agent moves, the g-values (cost to a node from the original goal state) are still useful in the subsequent searches.

It is unclear how the goal state is set for each planning iteration in Likhachev et al. (2004). Recall that the agent is moving during planning and the planner is searching backward from the goal to the agent. However, the planner cannot reliably predict how long planning will take, and hence it is not clear where the agent will be at the time planning has completed. To handle this, Likhachev (2016) recommends that the state that is planned to is a constant time (or, equivalently in this case, a constant number of actions) forward from the start state of the previous plan. As the agent moves along its current path, the planner concurrently begins planning backwards to this point from the goal with a decreased ϵ, updates the current plan of the agent, and starts planning again. This process is repeated until the goal is reached. Concurrent planning and execution in ARA* is described visually in Figure 1-4. Assuming that the planning time is predicted to be three actions worth of time, after the first planning iteration is complete, a regression plan begins from the goal to state A in the figure. Once the second iteration is complete, a final regression plan begins from the goal to state B. If a planning iteration takes longer than expected, planning restarted with the newly updated agent location. In this sense, each ARA* iteration is now subject to a real-time bound on its completion. Because the agent will approach the goal, planning iterations are likely to get faster and eventually complete on time. In the worst case, the agent will just follow the first suboptimal plan to the goal.

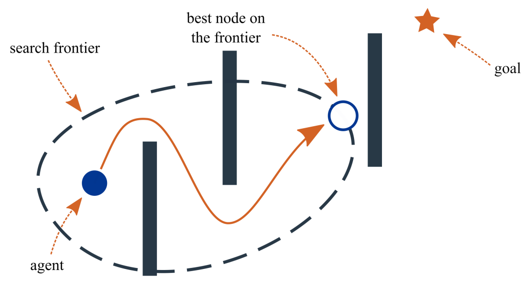
1.4 Real-time Search

Systems that interact with the external physical world often must be controlled in real time. Examples include systems that interact with humans and robotic systems, such as autonomous vehicles. In this dissertation, we address real-time planning, where the planner must return the next action for the system to take within a specified wall-clock time bound. We assume the system has fully observable state, discrete time, and discrete deterministic actions. We adopt a heuristic state-space search approach: while the system transitions from state s_1 to s_2, the planner exploits a heuristic cost-to-go function to explore promising parts of the state-space graph starting from s_2 to find an appropriate action to execute once the transition completes. The duration of the transition gives rise to the real-time bound for the planner. The problem domain is represented by an initial starting state, a successor state generator, and a goal predicate on states.

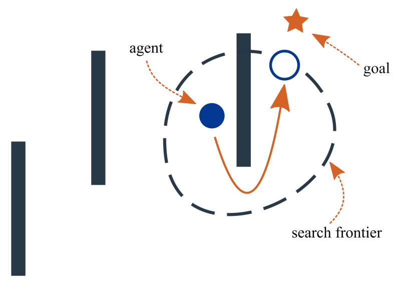
Real-time search adheres to a given hard real-time constraint by performing a bounded lookahead. The agent expands the search space until a time bound is reached, at which point the agent will then take either a single step or a sequence of steps towards the edge of the local search space that was expanded. While this partial plan is being executed, the search computes the next actions to take. In this way, the agent incrementally constructs a solution path.

RTA*

Real-Time A* (RTA*) selects a single action in each iteration (Korf, 1990). After a lookahead is performed, the next stage of RTA* is action selection. The f-value of all nodes on the edge of the frontier are backed up the tree by each node in the LSS taking the minimum f-value of each of its children. The agent then picks the smallest f-value from the direct children of its current state and



(a)



(b)

Figure 1-5: Iterations of LSS-LRTA*.

executes that action.

In order to prevent loops while still allowing for the agent to return to a previous state when it is beneficial to do so, a hash table of the heuristic values of the nodes actually visited by the agent is maintained. The heuristic function is applied to each node not in the hash table, and the agent chooses its child with the minimum f-value, as discussed previously. Before starting this process again, the previous state of the agent is stored in the hash table with an updated f-value equal to the second smallest f-value from its direct children. The intuition behind choosing the second best f-value is that this value represents the estimated cost to go if the agent returned to this state.

LSS-LRTA*

LSS-LRTA* performs an A* lookahead, limited by a fixed number of nodes to expand (Koenig & Sun, 2008). The progress of LSS-LRTA* shown on Figure 1-5. Intuitively, this is superior to a simple breadth first search towards a fixed depth because it takes the heuristic into account. LSS-LRTA* also performs Dijkstra's algorithm to update the h-values of all nodes in the local search space. Dijkstra's algorithm is started from the open list of the previous lookahead search. The algorithm terminates when the goal is expanded during the A* expansion stage.

Single Step LSS-LRTA* (LSS-LRTA* SS) is a variation of LSS-LRTA* in which the agent takes only a single step on the path that A* generated, rather than committing to an entire path (Burns, Kiesel, & Ruml, 2013). Only taking a single step allows LSS-LRTA* to update the heuristic values of nearby nodes more often. This creates a tradeoff between execution time and action confidence.

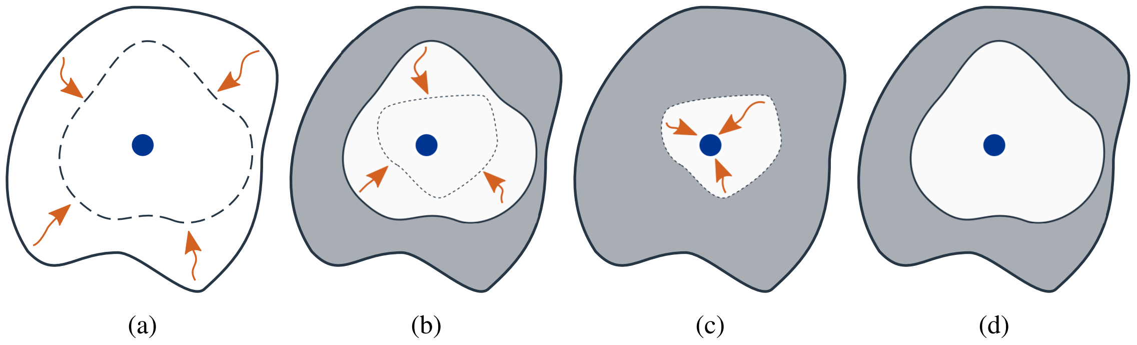


Figure 1-6: Learning step of LSS-LRTA*. a-c: Progress of learning within one learning phase. d: Partially updated state space as a result of interrupted learning.

For more details on LSS-LRTA* please refer to Chapter 2.

Dynamic $\hat{f}$

A variation of LSS-LRTA*, Dynamic $\hat{f}$orders the open list during lookahead search on $\hat{f}$, which is an inadmissible estimate of the cost of the best plan passing through a node n (Kiesel et al., 2015). Multiplying the distance from a node to the goal by the average single step error in the heuristic intuitively accounts for the inherent heuristic error at every node.

1.4.1 Real Time Bounds

In most previous real-time search research, time bounds are given in node expansions. In a real application, time bounds need to be in wall clock time. This is non-trivial in algorithms like LSS-LRTA*, in which there is significant work — the heuristic update backups — to be performed after the lookahead search phase. To handle this, our implementation defers the learning phase until the beginning of the next planning episode. If the time bound becomes imminent while the algorithm is still performing backups, the next action is selected via one-step lookahead. (This never actually happened during our experiments.)

An interrupted learning phase could create artificial local minima as a side effect in which only the heuristic values close to the search frontier would be raised and the rest of the space would be intact. Figure 1-6d demonstrates this phenomena. On Figure 1-6 the blue circle marks the agent, the outer region the local search space, the gray area denotes the space where the heuristic values were updated in the current learning iteration, and the light area surrounding the agent is the intact space.

1.5 Theoretical Comparison

This paper aims to determine what problem characteristics might cause one family of algorithms to prevail over the other. We do this both theoretically and empirically.

1.5.1 Anytime Search

There are three important limitations of anytime search for concurrent planning and acting. First, recall that, because the agent is moving, in ARA* a regression plan is formed from the goal state to a start state in the previous plan. This requires the domain to have a predecessor function andd a concrete or a small set of goal states, which may be non-trivial. For example, we did not use the acrobot domain with anytime search because it was not clear how to create a predecessor function. In contrast, real-time search can be used with any domain in which a forward simulator is available.

Second, the agent cannot start moving until the entire first planning iteration is complete. This implies an unavoidable delay at the start of execution while the first complete plan is formulated. In some systems, a delay in starting execution is merely undesirable, but in others it is inherently infeasible. This unavoidable delay in anytime search makes its application on certain domains impossible, whereas real-time search has no such delay. Third, because the time it will take for the search to find a new plan is not known in advance, it is not obvious what the goal state for the backwards search should be. In practice, a state along the current plan that is some predefined distance ahead of the agent can be selected, but this is ad hoc.

1.5.2 Real-time Search

Real-time search also suffers from inherent limitations. First, real-time search is incomplete in any domain in which there are states from where the goal is unreachable. These states are known as dead ends. For example, in the double integrator domain, it is entirely possible that an agent cannot leave a particular state without colliding with an obstacle based on its current speed and acceleration. In contrast, this would not happen in anytime search because the plan is always complete and guaranteed to reach a goal. Second, because real-time search algorithms select actions based on the f values of the nodes at the lookahead frontier, they are prone to falling into heuristic local minima, regions where states' h values are much lower than they should be. The only way the search can escape such minima is by performing enough lookahead to update the heuristic values inside the minimum. if the lookahead is smaller than the minimum size, many iterations of learning can be required, causing the algorithm to repeatedly traverse the same states (Sturtevant & Bulitko, 2014). This behavior appears irrational to an outside observer and is undesirable in many applications, including robotics (Likhachev, 2016).

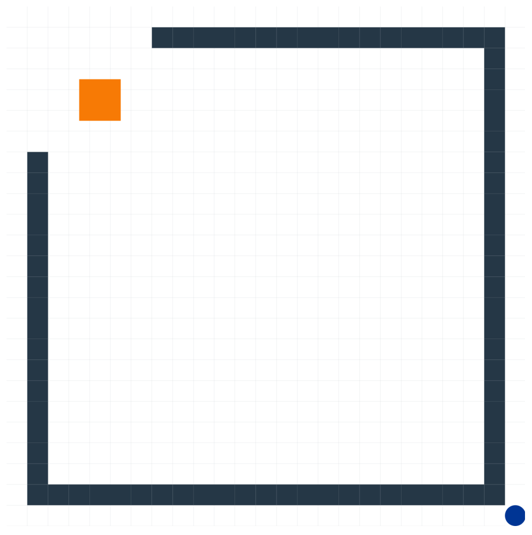
1.6 Experimental Comparison

We implemented A*, RTA*, LSS-LRTA*, Dynamic $\hat{f}$, and ARA* in Kotlin using the IBM J9 Virtual Machine using the Metronome real-time garbage collector. All experiments were performed on machines equipped with Intel Core 2 duo E8500 3.16GHz processors and 8GB of RAM. Data structures were statically sized to prevent long copying pauses. Adherence to time bounds was strictly checked.

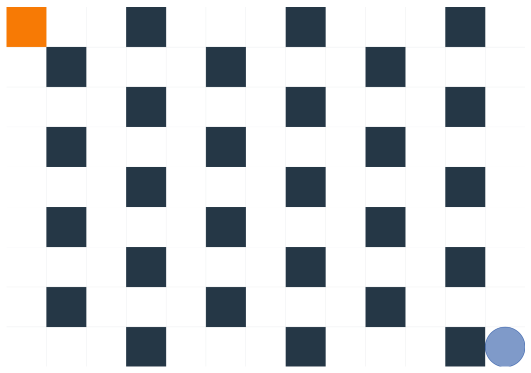
1.6.1 Grid Pathfinding

The grid pathfinding domain consists of a grid of cells in which each cell can be free or blocked. Each action moves the agent one cell in a cardinal direction, for a total of 4 distinct actions. The goal is to reach a specific cell. The size of the state space is $w \cdot h - b$, where h is the height of the domain, w is the width of the domain, and b is the number of blocked cells. Each action takes one unit of time and the heuristic used is the Manhattan distance from the agent's current location to the goal. We ran our experiments on the wall, slalom, and uniform instances of O'Ceallaigh and Ruml (2015), small versions of some of which are shown in Figure 6-3. The open box domain is 400x400, the hbox domain is 400x400, the squiggle domain is 800x800, the uniform domain is 1200x1200, the slalom domain is 37x124, and the wall domain is 41x21.

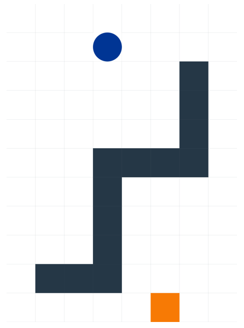
While the algorithms showed similar results in most grid pathfinding instances, certain maps cause one family of algorithms to dominate the other. The open box map shown in Figure 6-3 is



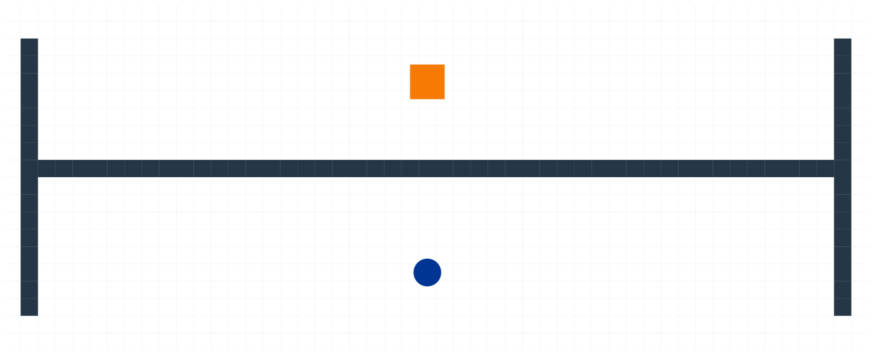
(a)



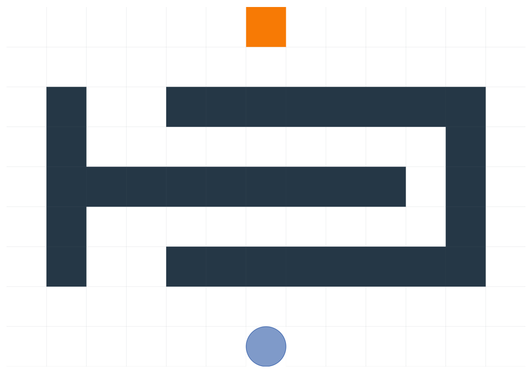
(b)



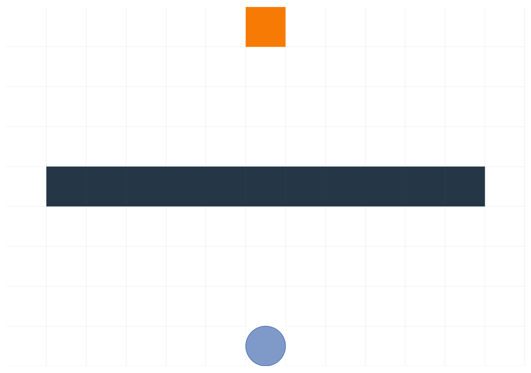
(c)



(d)



(e)



(f)

Figure 1-7: Grid instances: open box (a), uniform (b), squiggle (c), h-box (d), slalom (e), wall (f). The orange box marks the start and the blue circle the goal state.

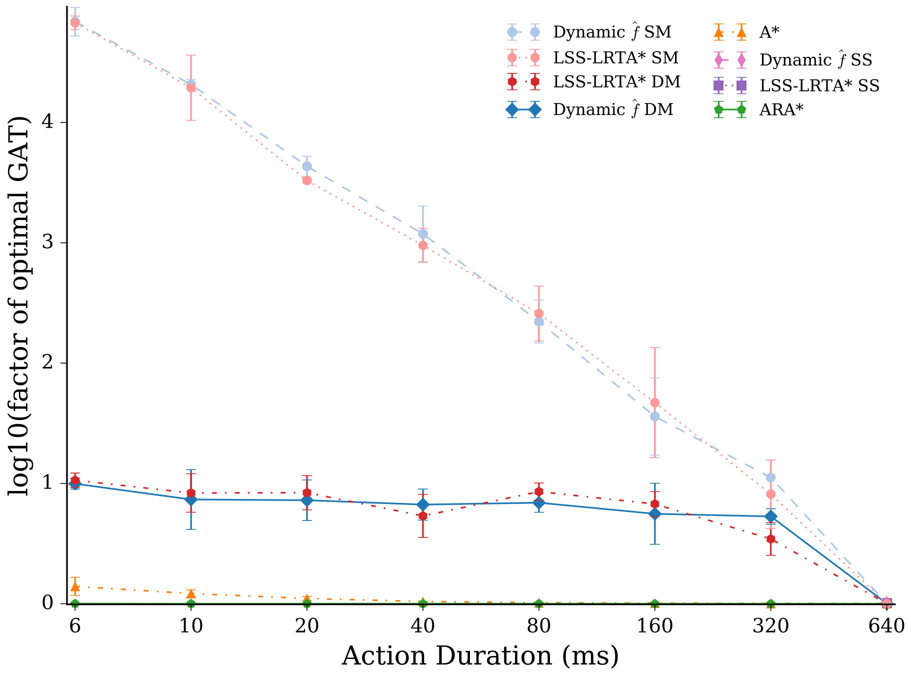


Figure 1-8: GAT on the open box instance of grid pathfinding.

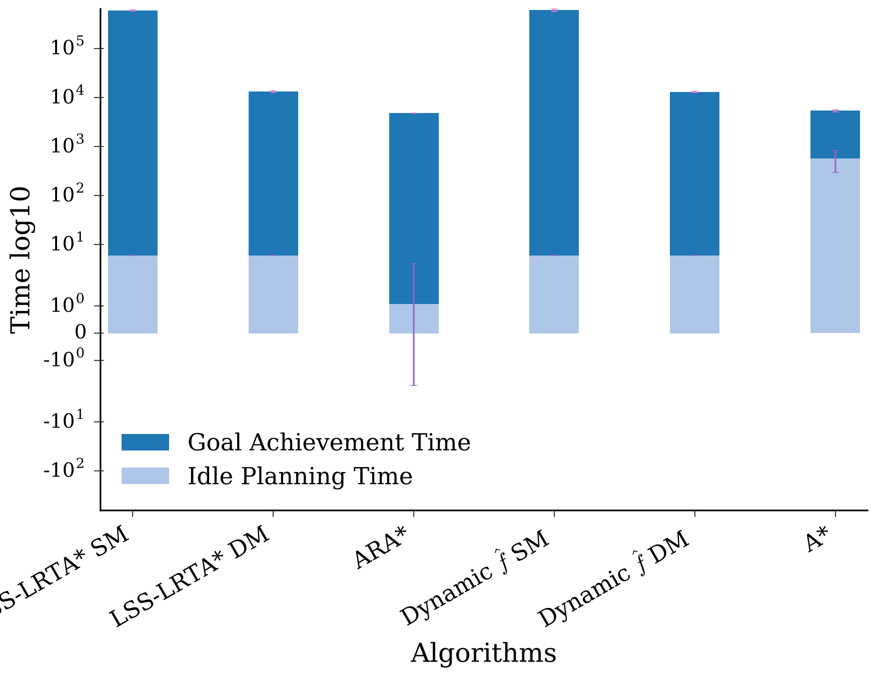


Figure 1-9: GAT breakdown for the open box instance.

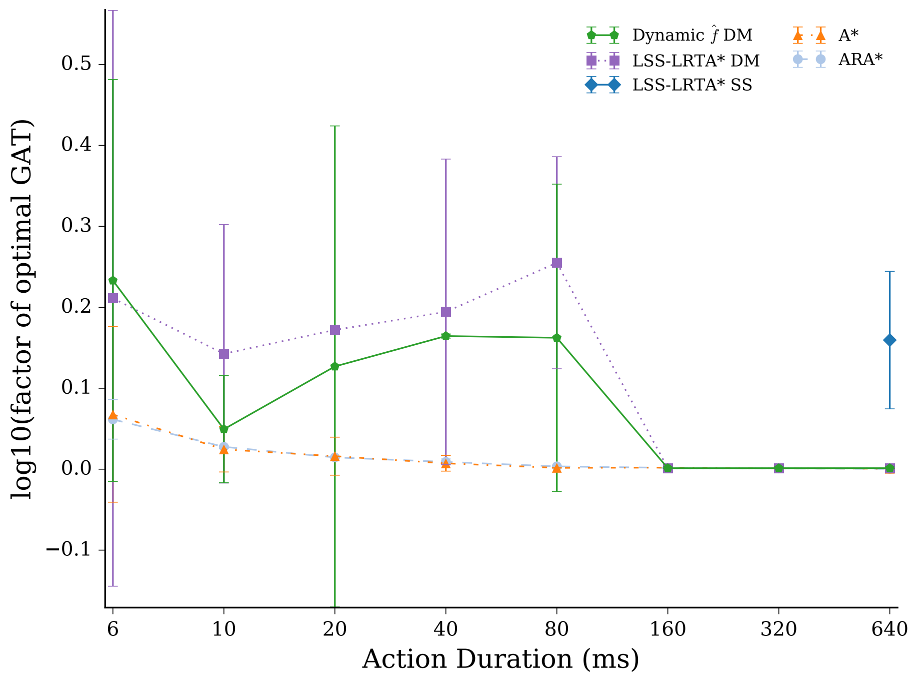


Figure 1-10: GAT on the hbox instance of grid pathfinding.

an instance with poor heuristic accuracy. Figure 1-8 shows the results of the algorithms run on this domain. The goal achievement times are normalized by the optimal execution time for the particular action duration in the particular instance and then plotted along the y axis. The x axis represents how long each action takes. ARA* significantly outperforms the real-time search algorithms. Real-time search follows the deceptive greedy path towards the goal and therefore struggles with leaving the middle of the box. However, ARA* benefits from reverse planning from the goal to the start state because there are very few states to explore on the outside of the box. Therefore, ARA* finds an efficient path to the goal almost immediately as shown in Figure 1-9 by its low idle planning time (the time it takes to find its initial plan). A similar, but less extreme map is the symmetric hbox domain, shown in Figure 6-3. ARA* still performed slightly better than real-time search, as shown in Figure 1-10 but the results were much more even.

The map shown in Figure 6-3 is an instance in which anytime search struggles. ARA* initially finds a plan along the left side of the squiggle and towards the start state at the bottom of the figure. ARA* then begins executing this plan and plans to a new point on the left of the squiggle. By the time ϵ is lowered to find a more optimal plan, it has already committed too far into a bad path.

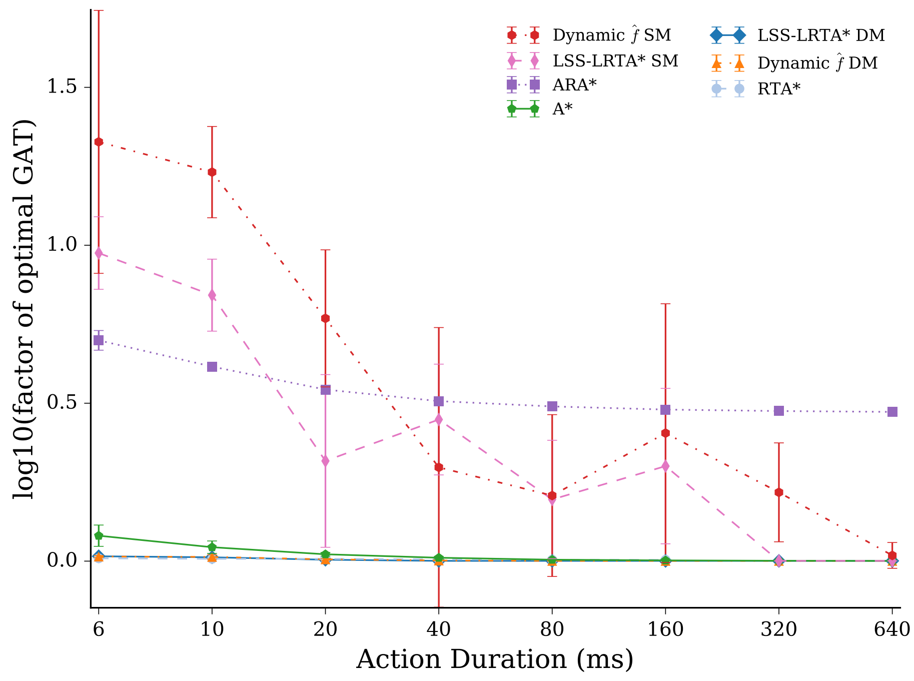


Figure 1-11: GAT on the squiggle instance.

However, in real-time search the agent greedily moves down the center of the grid towards the goal, thereby committing to the right side of the grid along a suboptimal path. Figure 1-11 shows real time search outperforming ARA* on this domain.

All algorithms were run on the uniform, slalom, and wall instances. All of the algorithms performed similarly, which we suppose is due to the very small branching factor of the grid pathfinding domain.

1.6.2 Racetrack

The racetrack domain is very similar to the grid pathfinding domain, but features additional actions and inertia (Barto, Bradtke, & Singh, 1995). Each action modifies the acceleration of the agent by -1, 0, or 1 in both the horizontal and vertical directions, making for a total of 9 distinct actions. There is no limit on the agent's speed. The goal is a contiguous set of cells. The size of the state space is $(h \cdot w - b) \cdot mxs \cdot mys$, where h is the height of the domain, w is the width of the domain, b is the number of blocked cells, and mxs and mys are the maximum speeds in the x and y directions, respectively, that could possibly be attained given the domain size. The mxs value is the largest integer value such that $mxs(mxs + 1)/2 \leq w$ is still true. Similarly, mys value is the largest integer value such that $mys(mys + 1)/2 \leq h$ is still true. A stochastic version of racetrack is popular to use for testing offline MDP algorithms. In that approach no heuristic is used. In our case, the heuristic used for the Racetrack domain is

$$h(s, c) = \frac{\max(|s.x - c.x|, |s.y - cy|)}{\max(mxs, mys)}$$

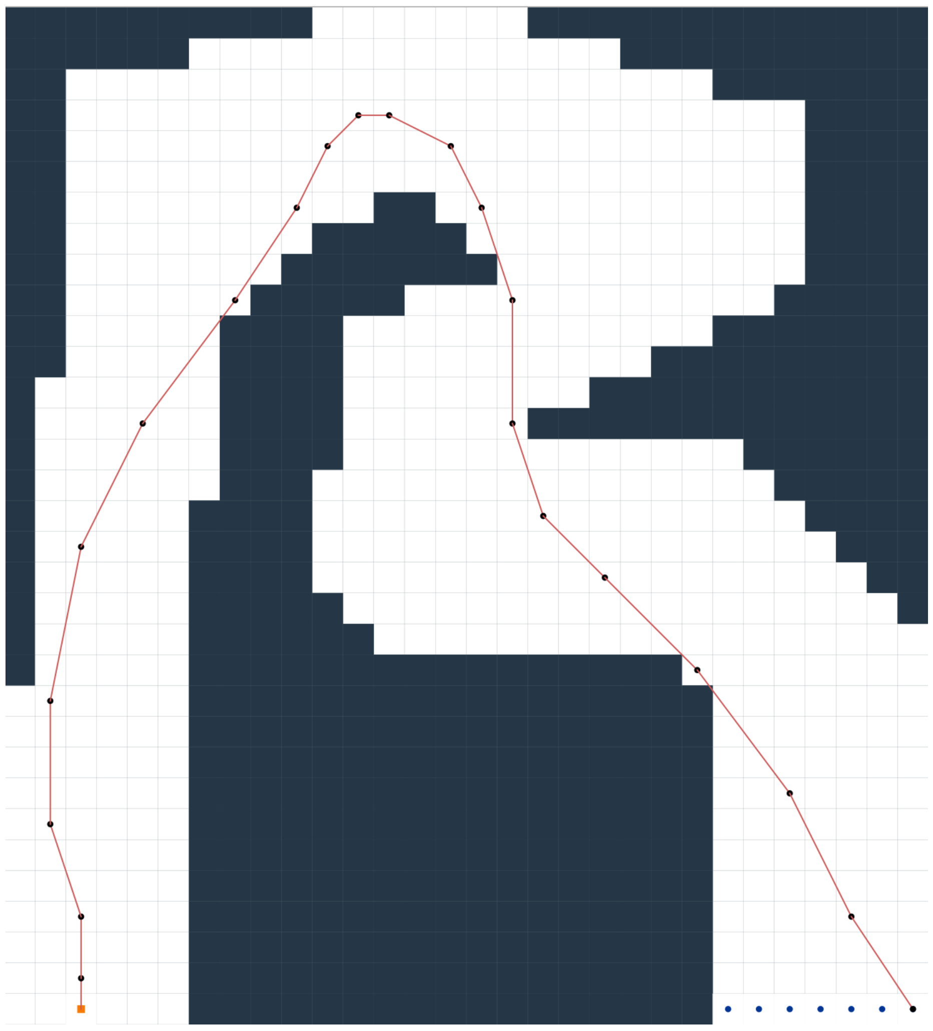
where s is the current state, and c is the closest goal on the finish line to s.

Figure 1-13 shows the results of all algorithms run on the large racetrack instance shown in Figure 1-12. Multiple step LSS-LRTA* with static lookahead (LSS-LRTA* SM) performs poorly, timing out with certain action durations. Multiple step LSS-LRTA* with dynamic lookahead (LSS-LRTA* DM) performs quite well, presumably because the increased lookahead shies the agent away from dead ends.

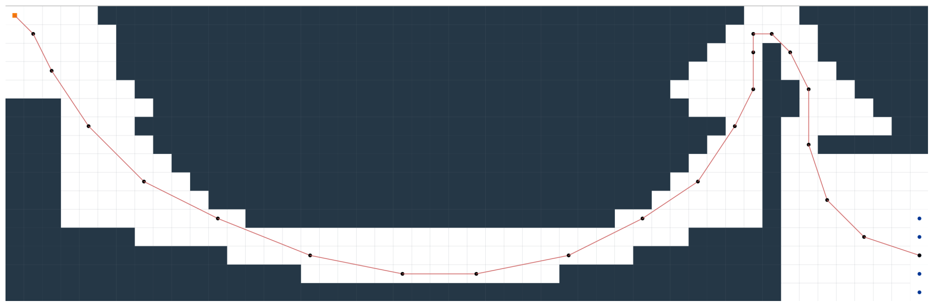
ARA* also performs poorly, which may be due to the heuristic being inaccurate since the agent must travel away from the goal first in order to traverse the track. Figure 1-14 shows that ARA* has a longer idle planning time than A* in the large track. This suggests that the initial plan of ARA* was inflating an inaccurate heuristic. To test this hypothesis, ARA* was run on the long track domain which causes the heuristic to appear more accurate since the agent, overall, travels in the general direction of the goal, as shown in Figure 1-12. The results of the long track, shown in Figure 1-15, show the algorithms performing similarly to each other, which supports the inaccurate heuristic hypothesis.

1.6.3 Point Robot

The point robot domain has the same layout as the grid pathfinding and racetrack domains, but each state is a floating point (x, y) location and thus the state space is continuous. Since the state space is continuous, the agent could visit an infinite number of states. Therefore, grid based discretization is applied to this domain in order the keep the state size finite. Each action is a pair of integer numbers representing an increase in speed in the x and y directions, respectively. Because the agent can instantaneously accelerate in the point robot domain, if there is not a maximum speed bound put on the agent, then the agent would be able to effectively teleport to the best position in its line of sight.



(a)



(b)

Figure 1-12: Large (a) and long (b) racetrack instances. A potential solution path is marked with red.

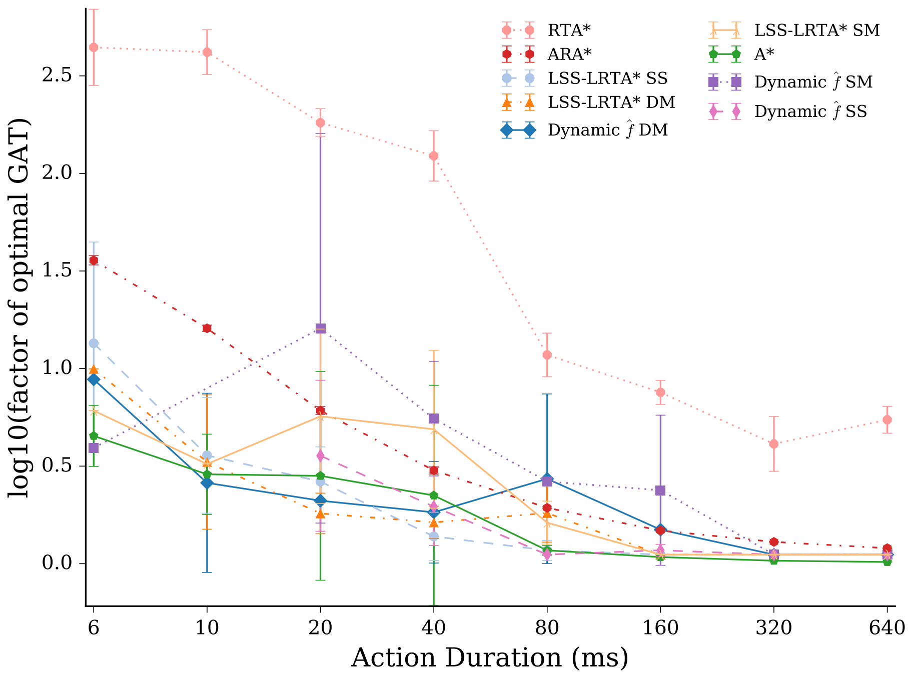


Figure 1-13: GAT on the large racetrack instance.

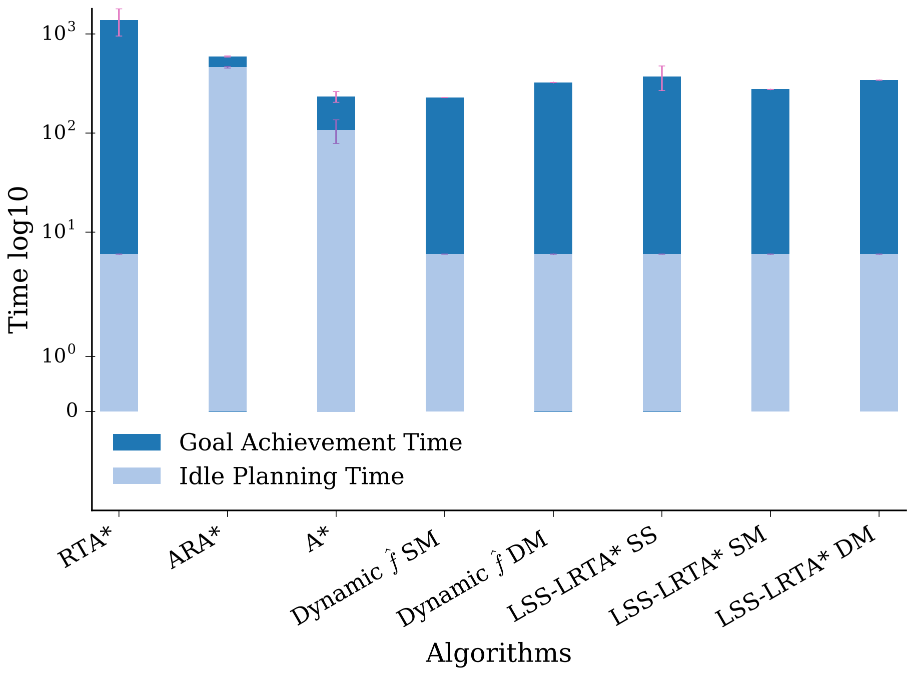


Figure 1-14: GAT breakdown on the large racetrack.

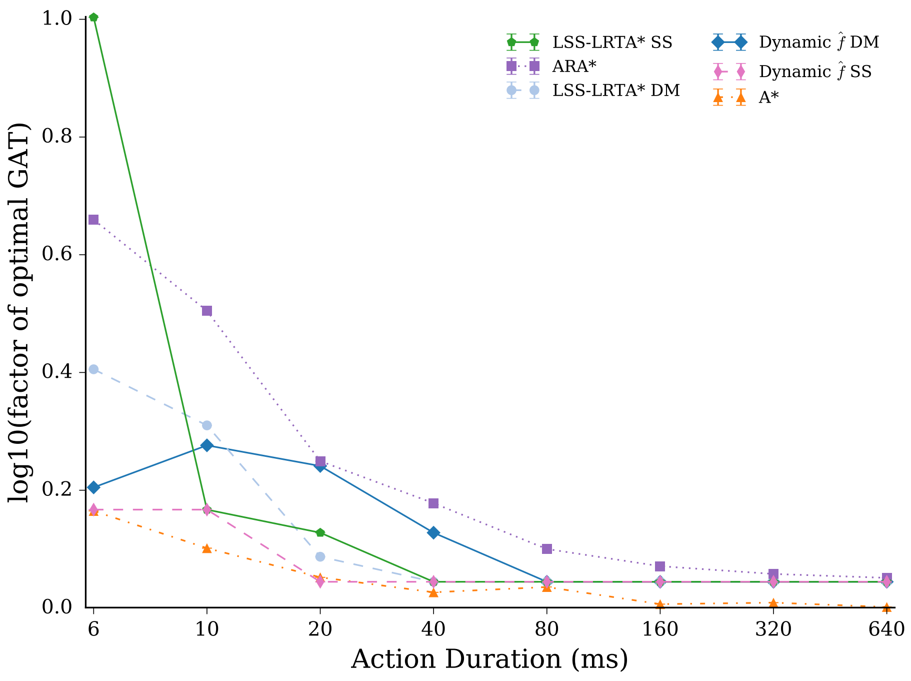


Figure 1-15: GAT on the long racetrack.

Therefore, the agent is given a maximum speed of 3. The state space equation is $h \cdot w - b$, where h is the height of the domain, w is the width of the domain, and b is the number of blocked cells. The goal is a defined radius around a point in the domain. A heuristic of

$$h(s, g) = \frac{\sqrt[2]{(g.x - s.x)^2 + (g.y - s.y)^2} - R}{MaxS}$$

is used, where s is the current state, g is the goal, R is the goal radius, and $MaxS$ is the maximum speed.

Figure 1-16 shows the results of all algorithms run on the uniform instance shown in Figure 6-3. In the point robot domain, A* performs poorly, which matches the results of O'Ceallaigh and Ruml (2015) in that A* performs poorly due to the large branching factor in the point robot.

The algorithms were also performed on point robot with the open box, hbox, and squiggle instance shown in Figure 6-3. In all three instances, the algorithms yielded similar results to those from the grid pathfinding domain.

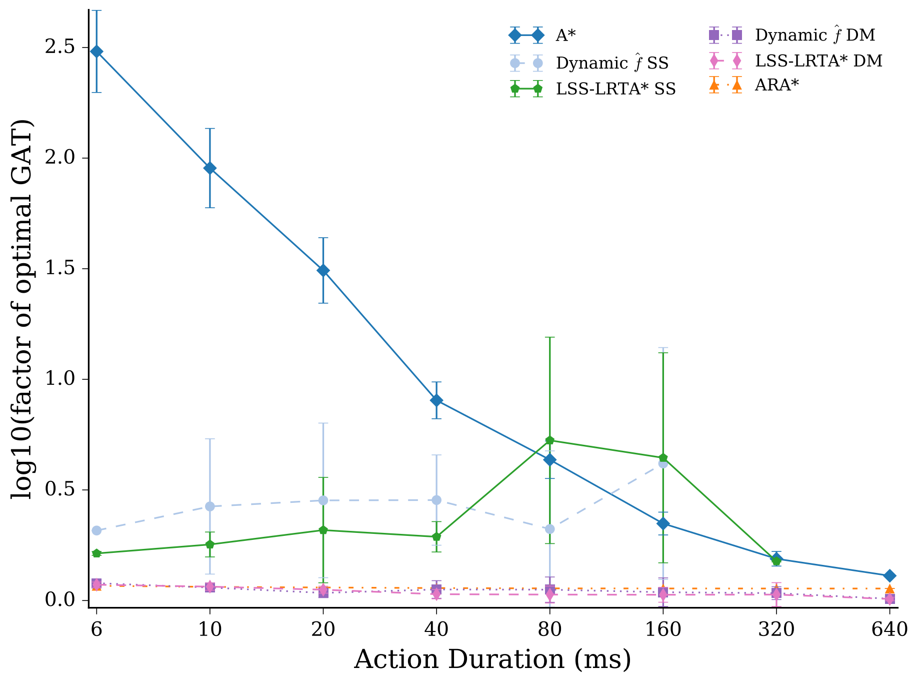


Figure 1-16: GAT for a point robot on the uniform grid.

1.6.4 Double Integrator

The double integrator domain is identical to the point robot domain, except it incorporates inertia which means that the agent cannot instantaneously accelerate. Each action is a pair of floating point numbers representing a change in acceleration in the x and y directions, respectively. For every positive acceleration, there is a corresponding negative acceleration allowing for the agent to be able to decelerate at the same rate it can accelerate. Successor states are computed by adding the acceleration to the current speed in n integration steps. Each time the current speed is updated, the agent moves forward by the current speed times $1/n$. In our implementation we used $n = 100$. The agent's starting speed and goal speed are 0 in both the x and y directions. The size of the state space is $(h \cdot w - b) \cdot d^2 \cdot mxs \cdot mys$ where h is the height of the domain, w is the width of the domain, and b is the number of blocked cells, and mxs and mys are the same as in the point robot without inertia domain, and d is the discretization of states. For example, if the discretization is in integer increments, then $d = 1$. However, if the discretization is in 0.25 increments, then $d = 4$. The heuristic for the double integrator is based off of the dynamics equation $s = v \cdot t + a \cdot t^2$ where s is

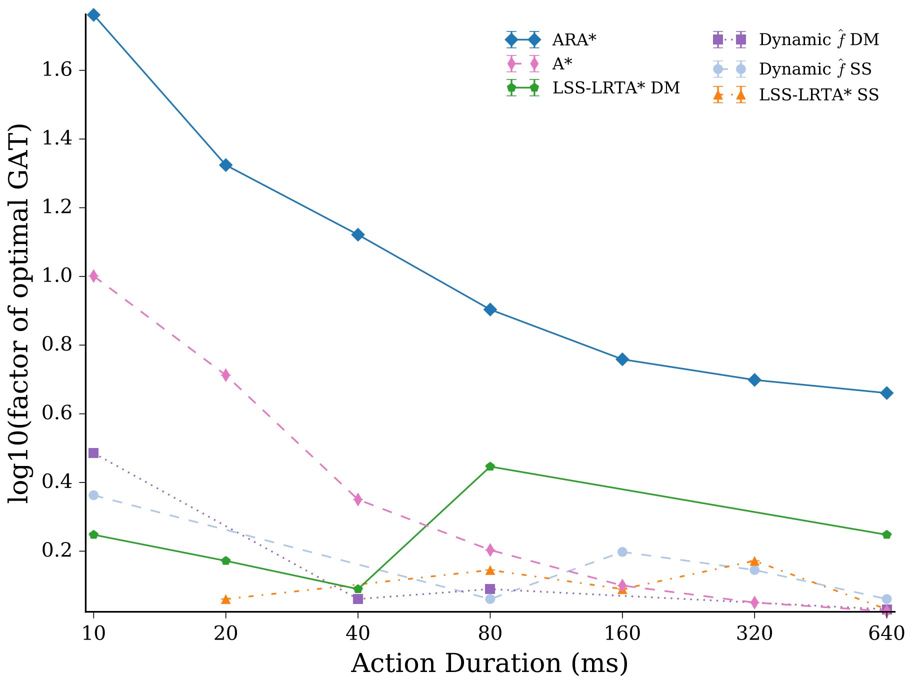


Figure 1-17: GAT for the double integrator on the slalom grid.

the distance between two points, v is the current speed, a is the maximum acceleration, and t, in this case, is the number of actions required to get between the two points. This equation is then solved for t. The heuristic is

$$h = \max(\min(t_{-xa}, t_{xa}), \min(t_{-ya}, t_{ya}))$$

where t_{-ax} and t_{ax} are the t result from the dynamics equation using the maximum negative and positive accelerations, respectively, and the distance in the x direction. Similarly, t_{-ay} and t_{ay} are the maximum negative and positive accelerations, respectively, and the distance in the y direction.

Figure 1-17 shows the results of all algorithms run on the slalom instance shown in Figure 6-3. Real-time search outperformed ARA* since ARA* requires a considerable amount time to finish its first planning iteration as shown in Figure 1-18 and also generates substantially more nodes, whereas the real-time search algorithms immediately begin moving towards the goal. Additionally, all of the algorithms ignored the slalom by moving around the edge of the map.

Figure 1-19 shows the results of all algorithms run on the wall instance shown in Figure 6-3 which are similar to those for the slalom instance.

Figure 1-20 shows the results for RTA* and single step LSS-LRTA* with static lookahead

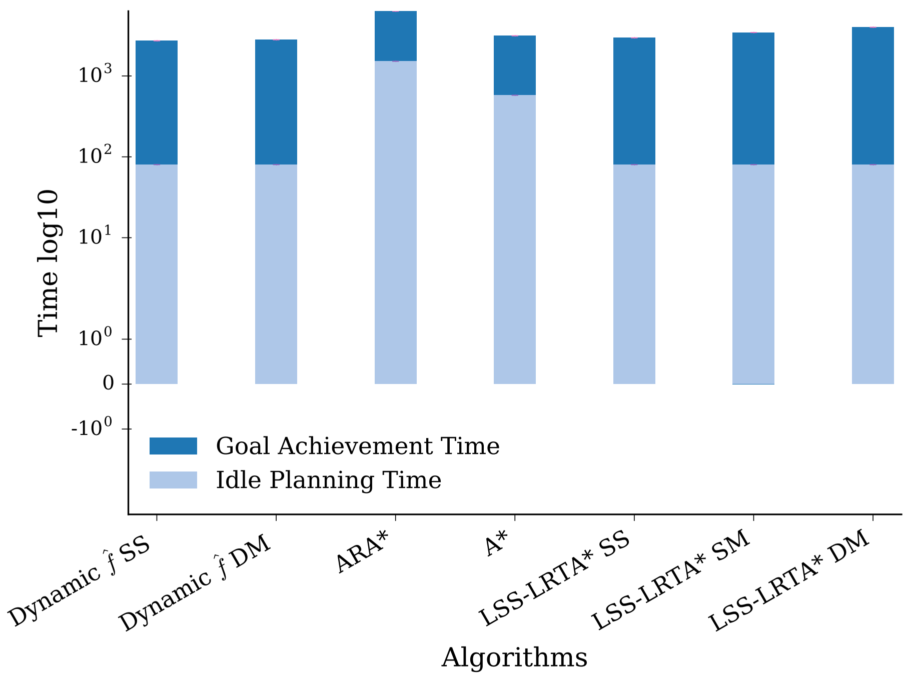


Figure 1-18: GAT breakdown for the double integrator on the slalom grid.

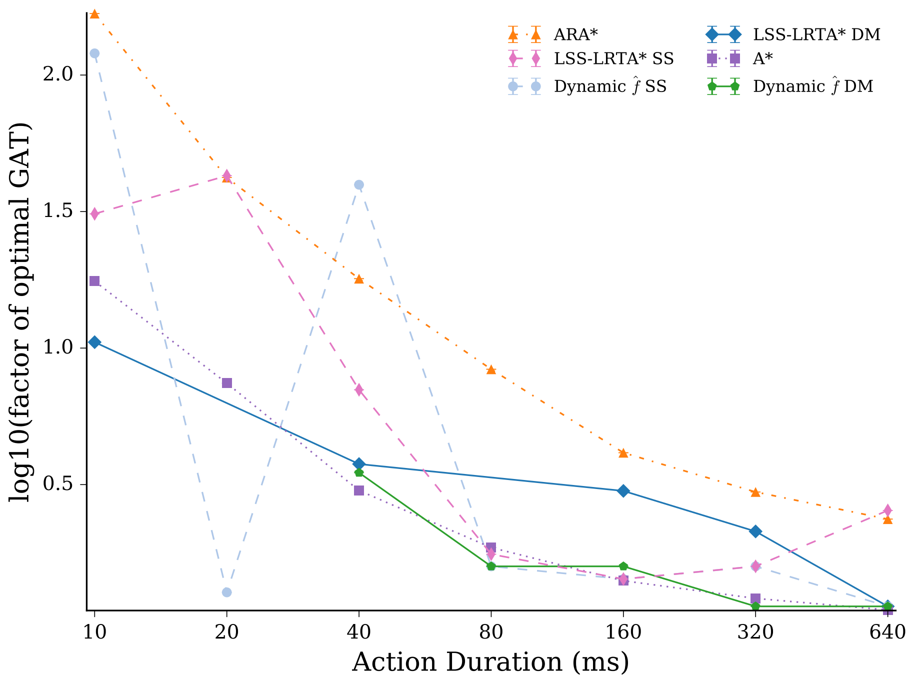


Figure 1-19: GAT for the double integrator on the wall grid.

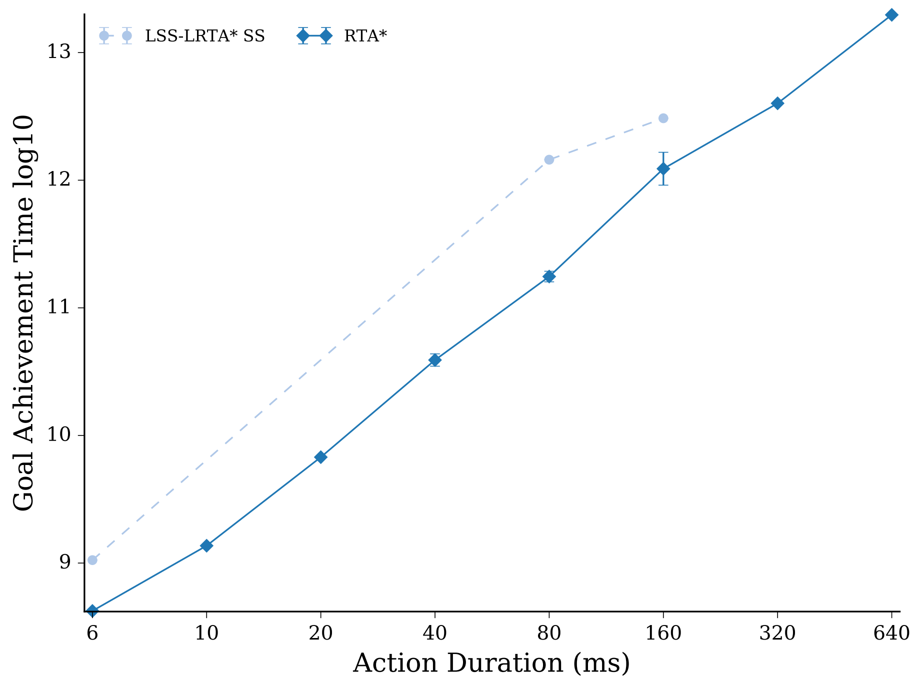
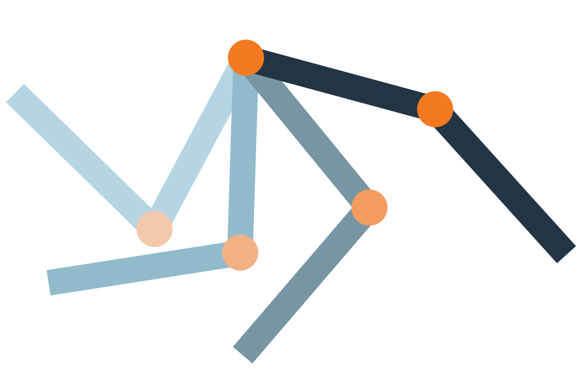


Figure 1-20: GAT for the double integrator on the uniform grid.

performed on the uniform instance shown in Figure 6-3. All other algorithms failed to solve the domain likely due to the state space being too large in the case of A* and ARA*, and in the case of the other real-time search algorithms, the large amount of obstacles causing dead ends as the heuristic encouraged the agent to move quickly toward the goal.

1.6.5 Acrobot

The acrobot (Murray & Hauser, 1991) domain is a two-link, underactuated system, shown in Figure 1-21. Torque can be applied solely to the center joint between the two links. The goal of the domain is to stand both links up vertically within a specified goal radius. A state is defined as the angles (θ_1, θ_2) and angular velocities $(\dot{\theta}_1, \dot{\theta}_2)$ of the two links. The torques which can be applied as actions are limited to positive torque ($\tau = 1.0$), no torque ($\tau = 0.0$), and negative torque ($\tau = -1.0$). The constant values in the equations of motion as well as the velocity bounds used are the same as those used by Boone (1997). The state space is continuous, but grid-based discretization is applied to this domain in order for the algorithms to feasibly find solutions. Our heuristic estimates remaining time by taking the maximum angular distance to the goal state divided by maximum angular velocity:


(a)

(b)

Figure 1-21: a): The acrobot agent. b): The goal state of the acrobot in an upright position in which both links are vertical above the center joint.

$$\max\left\{\frac{\theta_{g1}-\theta_1}{\dot{\theta}_{1max}}, \frac{\theta_{g2}-\theta_2}{\dot{\theta}_{2max}}\right\}$$

where θ_{gn} is the closest goal position for link n and $\dot{\theta}_{nmax}$ is the maximum velocity of link n and the angle differences are naturally wrapped around 0 and 2π.

To our knowledge, real-time heuristic search has never been tried on the acrobot domain. Results were gathered for both A* and real-time search algorithms as shown in Figure 1-22.

Addition video demonstration of the racetrack, acrobot, and point robot can be found at `https://youtu.be/oPTQuvDFVjU`.

1.7 Discussion

Our study has found that using real-time heuristic search planning algorithms can have significant benefits, even in complicated domains. In almost all instances in each of the domains used, real-time search outperforms anytime search in terms of goal achievement time. Real-time search begins executing a path immediately, however, the path is incomplete and could be wrong or inefficient. ARA* always has a complete path to the goal but this usually incurs a longer idle planning time to find an initial plan to the goal. The initial delay incurred by anytime search outweighs the typically shorter path to the goal in terms of the overall goal achievement time.

The one major exception to this was the open box instance. In this instance, ARA* greatly

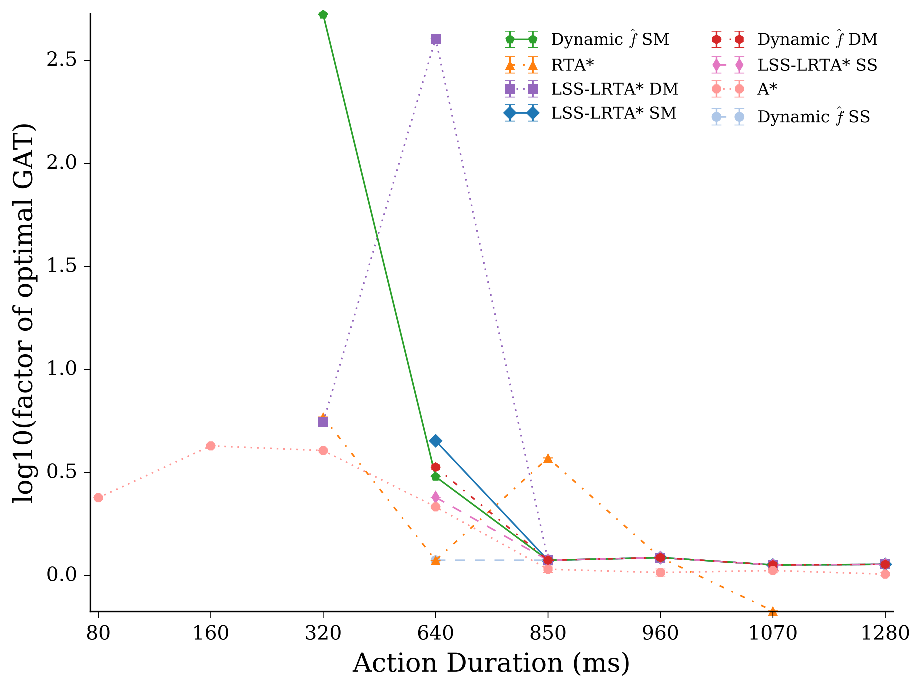


Figure 1-22: GAT for the acrobot problem.

benefited from planning backwards from the goal. ARA* should perform well on domains in which heuristic values near the goal are more accurate than near the start and planning backwards is advantageous. However, when the backwards planning advantage was removed by creating the hbox instance, ARA* performed similarly to the real-time algorithms.

One downside of real-time search is that it is susceptible to dead ends. If the lookahead is too small in real-time search, then the planner cannot detect obstacles in enough time to correct its path. This is especially evident in the double integrator domain with instances that have long open spaces since real-time search will attempt to increase its velocity as much as possible towards the goal but if there are obstacles between the agent and the goal, the agent may not be able to slow down in time which would cause the agent to collide with the obstacles.

One additional benefit of real-time search is that it is able to plan in domains without obvious predecessor states. The acrobot is one such domain. ARA* could not be run on the acrobot because planning backwards from the goal requires calculating predecessor states, which is non trivial.

Finally, our results showed that, as the action durations are increased, the goal achievement times of the algorithms converge. All of the algorithms have a sufficient time to plan and find good solutions the higher the action duration thus increasing the quality of the solution as well as the likelihood that a solution is found.

1.8 Discussion

To our knowledge, this is the first time heuristic search has been used for the robotics-oriented double integrator and Acrobot domains, and the first time that real-time heuristic search has been implemented to follow a strict hard real-time bound. The results show that real-time search generally outperforms anytime search in terms of goal achievement time in the domains we tested. Anytime search is currently the leading heuristic search approach taken in advanced robotics work where time is important, so these results suggest that further investigation of the real-time approach is warranted.

The main drawback of real-time heuristic search illustrated in our results is their incompleteness in the presence of dead ends.

This deficiency motivated the following three chapters that focus on the completeness of real-time heuristic search methods. Chapter 2 lays down the groundwork for safe real-time search and introduces a practical method that was specifically designed for safety. Then, Chapter 3 utilizes the underlying principles from Chapter 2 to create a proof-of-concept spatiotemporal planner for urban driving that is not only capable to ensure completeness but has higher performance than competing methods. Lastly, the final chapter addressing safety, Chapter 4, introduces several techniques that are exploiting the properties of the real-time setting to make more informed decisions in the context of safety to improve the goal achievement time, and to reduce the overhead introduced by dealing with dead ends.

CHAPTER 2

Real-time Planning in the Presence of Dead-ends

In many applications of planning, the objective is not only to achieve the goal as quickly as possible but also to ensure safe operation of the system. In practice, safety and speed are often contradicting objectives. For example, a very fast self-driving car is inherently unsafe. Current real-time heuristic search algorithms are incomplete in domains with dead-ends, which severely limits their applicability. This chapter of my dissertation presents a novel planner called SafeRTS that guarantees the safety of the agent while is able to optimize the time to achieve the goal. SafeRTS lays down the foundation for real-time search methods that can operate in domains with dead ends and expands the potential application of real-time heuristic search to safety-critical environments. This work appeared in the Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18) (Cserna et al., 2018).

2.1 Introduction

Because the planner cannot necessarily find a complete path to a goal state within the time bound, there is no guarantee that the selected action leads to a goal. In fact, it may be necessary to return to a previously visited state in order to reach a goal, so real-time heuristic search methods update their heuristic function to avoid becoming trapped in cycles (Korf, 1990). If the state space is finite and a goal is reachable from every state, most real-time search methods are complete and will eventually reach a goal. In practice, the state spaces of many real-world applications can be considered finite, but unfortunately they often contain dead-end states from which a goal is not reachable. For example, if a robot is traveling too fast towards an obstacle, it may be impossible to avoid a crash. As we will see below, conventional real-time search methods quickly succumb to dead ends. This prevents existing search methods from being applied to many on-line planning problems.

In this chapter, we study general methods for avoiding dead ends in real-time search. In contrast to traditional off-line safety verification, we aim to avoid a pre-specified controller and lengthy preprocessing and rather allow an agent to dynamically plan for its current goals and prove on-line that its next action will be safe. We define a general problem setting for safe on-line planning and several new real-time search algorithms and prove conditions under which they can keep an agent safe. We define a new safety-oriented node evaluation function and develop practical methods that exploit its information to efficiently preserve safety while selecting appropriate actions. We empirically test these methods on two simulated domains, traffic crossing and controlling a vehicle with inertia, and find that the new methods dramatically outperform the conventional ones.

2.2 Preliminaries

The algorithms we discuss are based on a modern general-purpose real-time search algorithm, Local Search Space Learning Real-Time A* (LSS-LRTA*) (Koenig & Sun, 2008). (Note that incremental approaches, such as D* (Koenig & Likhachev, 2002), or anytime approaches, such as ARA* (Likhachev et al., 2004), have planning times that are not tightly bounded, rendering them inapplicable.) We begin by introducing LSS-LRTA* and studying its behavior in domains with dead ends.

2.2.1 Real-time Heuristic Search

Psuedocode for LSS-LRTA* is sketched in Figure 1. The algorithm proceeds in two phases: planning and learning. The planning phase is similar to A*: expanding nodes in best-first order, preferring low f values, where $f(n) = g(n) + h(n)$, the cost-so-far plus an estimate of the cost-to-go. To obey the real-time bound, only a pre-specified number of nodes are expanded, forming a local search space. In the learning phase, a Dijkstra-like propagation updates the heuristic values of all expanded nodes backwards from the search frontier. Nodes with no successors are given an h value of ∞. In the original version of LSS-LRTA*, the agent then commits to all the actions leading to the node on the frontier with the lowest f value. In the experiments below, we also evaluate a more conservative version that commits only to the first action along the most promising path. If one or more goal

Algorithm 1: LSS-LRTA*.

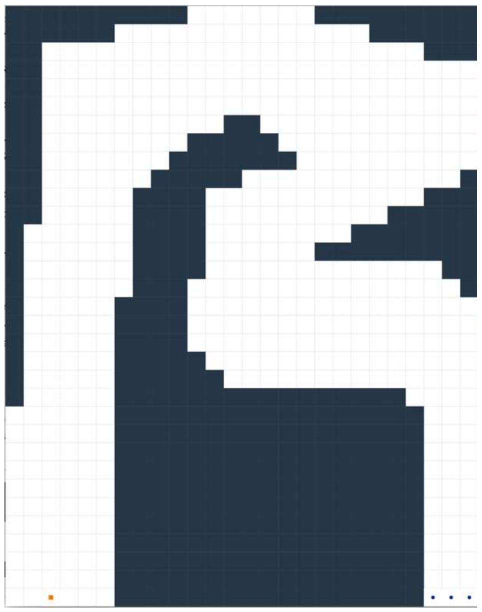
```
Input: $s_{root}$, bound
1 while the agent is not at a goal do
2     perform bound expansions of A* search from $s_{root}$
3     if open becomes empty, terminate with failure
4     $s \leftarrow$ node on open with lowest $f$ value
5     update $h$ values of nodes in closed
6     commit to actions along path from $s_{root}$ to $s$
7     $s_{root} \leftarrow s$
```

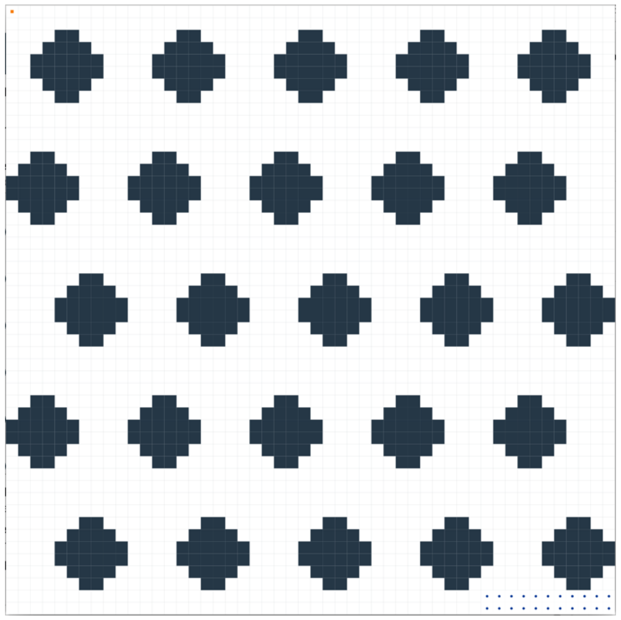
states are discovered during planning, the agent commits to the path to the goal that is closes. This path is optimal if the heuristic is admissible and consistent as the lookahead search of LSS-LRTA* relies on A*. If the open list becomes empty, a goal state is not reachable and we say the agent has failed.

LSS-LRTA* is complete if *h* is consistent and the cost of every action is bounded from below by a constant (Koenig & Sun, 2008). It is quite general: it does not assume that the state space is undirected (every predecessor of a state is also a successor), that action costs are uniform, that there is a single goal state or a small number of them, or even that goal states are given explicitly. Many other algorithms build on LSS-LRTA*, including RTAA* (Koenig & Likhachev, 2006) and daLSS-LRTA* (Hernández & Baier, 2012).

2.2.2 Empirical Performance

While LSS-LRTA* is complete in finite state-spaces that lack dead ends, we have found that it can perform poorly if they are present. We empirically examined its behavior in two domains. Both feature directed state spaces, in the sense that it is not always possible to return to the previous state, and dead end states, which have no successors. The first is a dynamic world that presents exogenous dangers to the agent. It is an extension of the traffic domain used by Kiesel et al. (2015): reminiscent of the video game Frogger, the agent must navigate a grid from the upper-left to the lower-right using


(a)


(b)

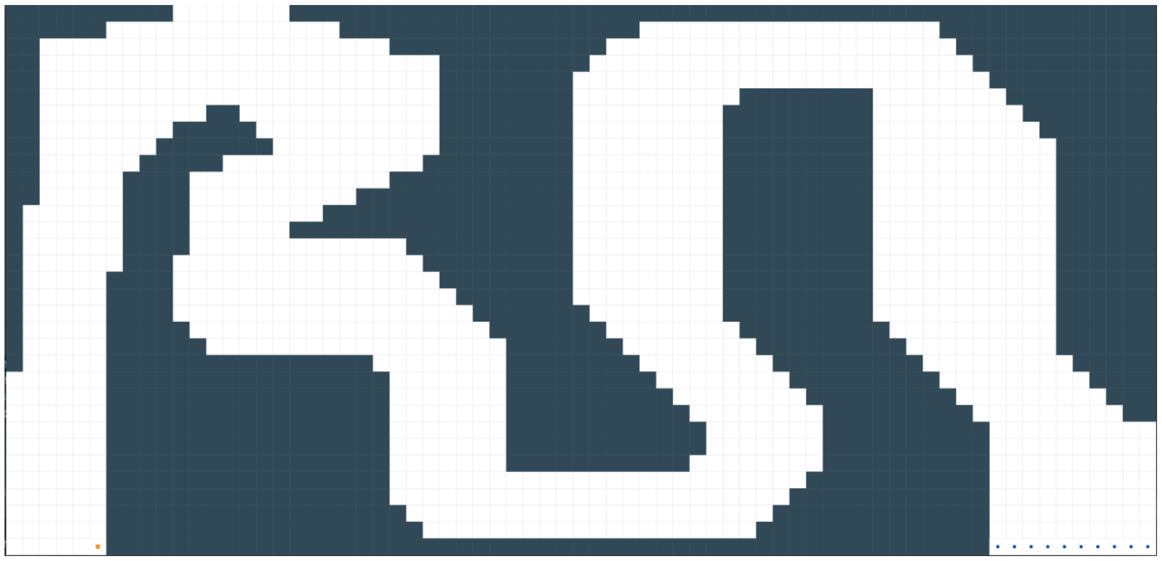
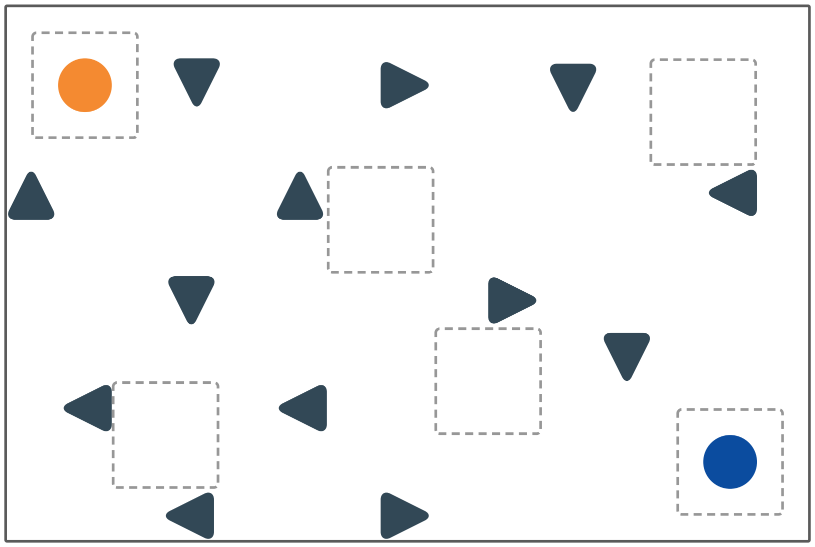

(c)

Figure 2-1: Racetrack domain instances: Barto racetrack (a), Uniformly cluttered track(b), and Hansen-Barto combined track(c).



(a)

Figure 2-2: Traffic domain with bunker cells.

four-way movement while avoiding moving obstacles. A cartoon sketch is shown on Figure 2-2a (the white cells are bunkers, described below). Each obstacle moves either vertically or horizontally, one cell per timestep. Obstacles bounce off the edges of the grid but pass through each other. While these velocities are known to the agent, and hence the domain is deterministic, the system state space is large, involving both the location of the agent and the locations of the obstacles (or equivalently, the timestep). In addition to moving in the four directions, the agent can also execute a no-op action and remain stationary. We extend the domain to include special bunker cells, off of which dynamic obstacles bounce, protecting the agent. The objective is to minimize the number of moves to reach the goal and the cost-to-go heuristic h is the Manhattan distance, which is perfect in the absence of obstacles. We created 100 random instances of 50 by 50 cells in which each cell has a 50% chance to be the starting position of an obstacle or a 10% chance of being a bunker.

The second domain is reminiscent of autonomous driving and is a variant of the popular racetrack problem (Barto et al., 1995) which was first introduced in Chapter 1. The agent moves in a grid attempting to reach one of a set of goal locations while avoiding static obstacles. Figure 2-4 shows the three racetrack maps used in the experiments. The track on Figure 2-1a was created by Hansen and Zilberstein (2001a). The combination of the first with a track by Barto et al. (1995) on Figure 2-

1c. Lastly, a new uniformly cluttered track to provide variety on Figure 2-1b. The agent's velocity in the *x* and *y* directions can be adjusted by at most one at each timestep. While acceleration is tightly bounded, velocity is limited only by the size of the grid. The system state includes the agent's location and velocity. The objective is to minimize the number of time steps until a goal cell is reached. The heuristic function is the maximum, either horizontally or vertically, of the distance to the goal divided by an estimate of the maximum achievable velocity in that dimension. This is admissible. This domain differs from traffic in that the heuristic is inherently dangerous: it prefers states in which the agent is closer to a goal, which are likely to be those in which it is moving faster and hence more likely to crash. For each of the three maps, we created 25 instances with starting positions chosen randomly among those cells that were at least 90% of the maximum distance from a goal.

We compared the strategy of committing to all the actions to the lookahead frontier to merely committing to one action at a time. In both domains, multiple-action commitment was consistently superior, so this is what we show below. When committing to multiple actions, the search algorithm devotes all the time until the last committed action ends to its planning phase (the 'dynamic' lookahead strategy of Kiesel et al. (2015)).

We implemented all domains and algorithms in Kotlin (JetBrains, 2017) and ran them on a modern Xeon server. Each algorithm was given a maximum of 10 CPU minutes and 20 GB of RAM. Runs exceeding those resources were marked as failed. Figure 2-3 and Figure 2-4 show the fraction of instances in which LSS-LRTA* (blue line) successfully guided the agent to a goal. The x-axis represents different durations for actions in the world: longer durations allow more node expansions during lookahead before the next action must be returned. (As in many studies of real-time search, for simplicity and reproducibility, we measure time using node expansions.) Error bars represent 95% confidence intervals around the mean. As one might expect, in both domains, for smaller action durations when there is less time to plan, LSS-LRTA* collides with obstacles more often. But even with 1,000 expansions, it cannot keep the agent safe. Clearly, safety is currently an issue for real-time search algorithms.

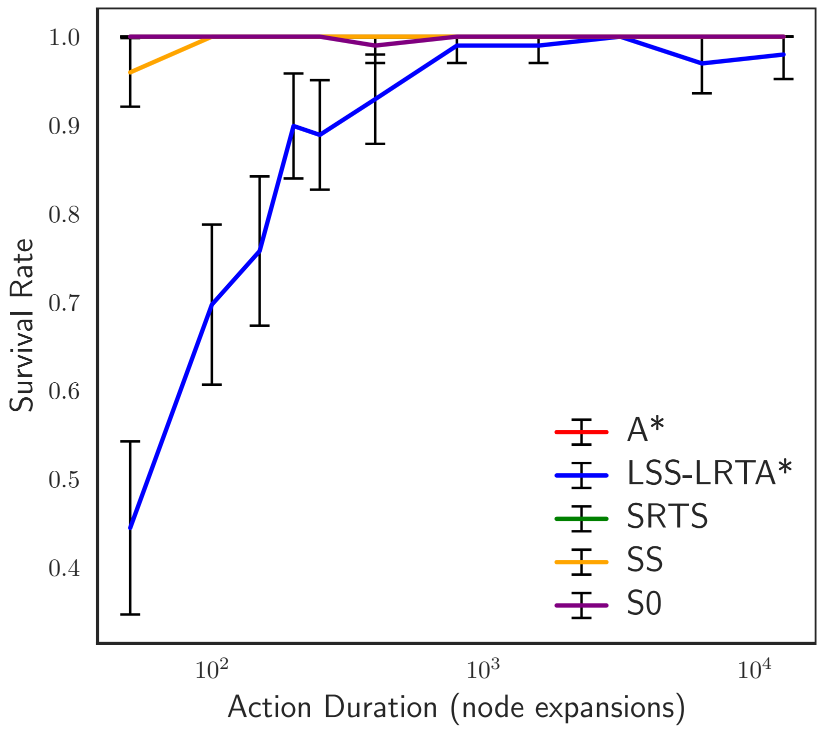


Figure 2-3: Success rates in the traffic domain.

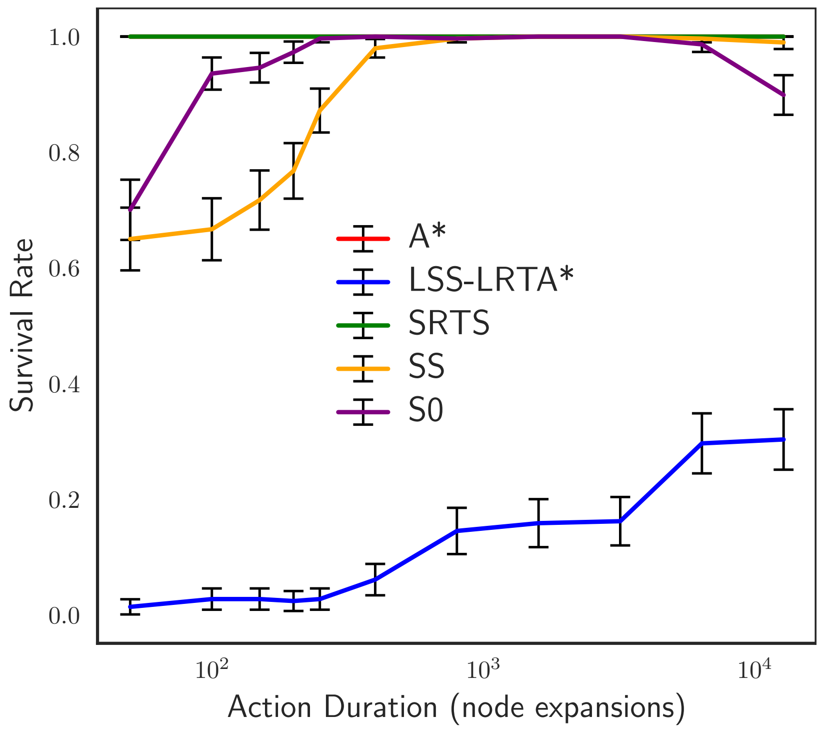


Figure 2-4: Success rates in the racetrack domain.

2.2.3 Problem Setting

As the foundation for our approach to safety, we assume that the user supplies a predicate that identifies certain states as *safe*. We take this to indicate that a goal is likely to be reachable from such states, and thus that it can benefit the agent to maintain a feasible plan to reach a safe state in case the other states it is considering turn out to be dead ends. We will say that a state that is safe or that is known by the agent to have a safe descendant is a *comfortable* state and that an action leading to a comfortable state is a *safe action*. If an agent never goes to an uncomfortable state then we say it *stays safe*. We call a node with no safe descendants *unsafe*; note that determining unsafety may be impractical. We emphasize that in some domains the safety predicate might be merely heuristic. Although (as we will see below) a guarantee that a goal is reachable from every safe state can endow certain algorithms with desirable properties, safety can in some cases merely indicate that such states might be useful in avoiding failure. In the traffic domain, for example, the safety predicate marks states in which the agent is located in a bunker as safe. However, it could be that a multitude of dynamic obstacles will prevent the agent from ever actually reaching a goal. In racetrack, states in which the agent has velocity zero in both dimensions are marked as safe. Assuming the goal states are not completely blocked by obstacles, this safety predicate actually does guarantee goal reachability.

If the user has a predicate that can detect some of the dead-end states, we will not consider it explicitly in this chapter. We assume that, if available, the detector is applied to every state at generation time and that those for which it returns true are pruned. Unless the detector is always perfect, the agent may still encounter dead-end states.

2.3 Basic Safe Real-Time Search

We now turn to safe search algorithms. We begin by defining a conservative search algorithm to be one that prefers safe actions. More precisely, it adheres to this general schema: 1) expand nodes, starting from the agent's current state, to generate a local reachable portion of the state space, and call the safety predicate on each generated state, 2) back up comfort by marking all predecessors of any comfortable state as comfortable, and 3) commit to a safe action, if one exists. All conservative search algorithms have the appealing property that they keep the agent comfortable under certain conditions.

Theorem 1 *For a given conservative search algorithm a, if the agent starts in a comfortable state and every comfortable state s has a comfortable descendant that is reachable within the local search space generated by a from s, then a keeps the agent comfortable.*

Proof: Assume the agent becomes uncomfortable. This means a non-safe action was selected. Any safe node that is generated in the local search space was reached via a feasible sequence of actions from the current state, so the absence of a safe action implies that no safe nodes were generated. This is a contradiction, as the local search space was guaranteed to contain one. □

The applicability of Theorem 1 hinges on ensuring that a comfortable descendant will be in the local search space. We now introduce a conservative search that we will call simple safe search (SS). It is similar to LSS-LRTA*, except that in its planning phase, it first performs a breadth-first search until either a safe state is generated or a prespecified depth bound k is reached. It then uses any remaining expansion budget to perform best-first search on f, starting from the already-developed breath-first search frontier. After the learning phase, simple safe search marks the ancestors (via all predecessors) of all generated safe states (from both the breadth-first and best-first searches) as comfortable and all top-level actions leading from the initial state to a comfortable state as safe. To select actions, it chooses the best f node on the frontier, checks along its path for a comfortable node, and commits to the actions leading to the deepest such node. If no comfortable node is found, the path to the next-best frontier node is checked, and so on. If no safe states are found, simple safe search behaves as LSS-LRTA* would, committing to the best frontier node (or the first action toward it in the case of single commitment). We will refer to this action selection strategy as safe-toward-best.

More sophisticated variations on this simple algorithm are possible, but we study this one for its simplicity. For example, Theorem 1 is easy to apply — we need only ensure there is at least one safe node within distance k:

Theorem 2 *If the agent starts in a comfortable state, every comfortable state has a comfortable* *descendant no more than k steps away, and the expansion bound is large enough to allow a complete depth-k breadth-first search, then simple safe search keeps the agent comfortable.*

Proof: An immediate corollary of Theorem 1. □

Remarkably, despite its strong assumptions, Figure 2-3 and Figure 2-4 show that simple safe search (yellow line denoted SS) is able to keep the agent much safer than LSS-LRTA* in both of the benchmark domains. The parameter k was arbitrarily set to 10 for these experiments.

Theorem 2 might seem trivial, but it can be applied in both of the benchmark domains. In traffic, note that the no-op action allows an agent to remain in a bunker location, meaning that safe states always have an immediate safe successor. So if the agent starts in a safe state, the theorem applies and the agent will remain comfortable. Of course, if the lookahead is too small to allow the agent to find a comfortable state that is closer to a goal, the agent will remain comfortable at the expense of reaching a goal (we note that it fails a few trials at low action durations), but this is a fundamental and unavoidable trade-off. Similarly, in racetrack, applying zero acceleration allows the agent to stay in a safe state, so the agent will only leave a safe state when it has a feasible plan to reach another. Successor states with velocity k will never have a safe descendant within $k - 1$ steps so the agent will never accelerate beyond $k - 1$. Intuitively, because the agent only chooses actions that lead to states from which it has looked far enough ahead to know that it can come to a stop if necessary, then it will remain safe. However, it will take a long time to reach the goal (and again we note some failures at low durations).

Although it requires severe assumptions, one can exploit the similarity to LSS-LRTA* for a completeness result:

Theorem 3 *If h is consistent, all action costs are bounded from below, the goal is reachable from every state, and the most attractive top-level action can always be proven safe by k-step lookahead, then simple safe search is complete.*

Proof: Because the learning phase in simple safe search behaves like LSS-LRTA*'s, then if the most attractive top-level action (the one LSS-LRTA* would take) can be proven safe by k-step lookahead, the algorithm reduces to LSS-LRTA* (albeit with a smaller lookahead). So if the domain adheres to the assumptions of LSS-LRTA*'s completeness proof, simple safe search inherits the property. □

2.3.1 A Simpler Variant of LSS-LRTA*

Not all domains are guaranteed to have safe states within a constant distance of any comfortable state, and as a practical matter, it seems profligate to expand nodes breadth-first even if they have poor f values. So we next examine an even simpler strategy we call S0. It behaves like LSS-LRTA*, but checks for safe states during node generation in the planning phase. In the learning phase, S0 propagates back the comfort of each node to its predecessors that have been discovered in the previous exploration phase. Like simple safe, S0 uses safe-toward-best action selection. When a state is determined to be comfortable, this information is cached (along with its h value as usual for real-time search) so that we do not need to rederive this if we encounter the state again, in the next search iteration for example.

As shown in Figure 2-4, S0 dominates LSS-LRTA* and surpasses simple safe search in racetrack. In the traffic domain on Figure 2-3, S0 occasionally fails to reach a goal within the time limit. Although S0 performs better than LSS-LRTA*, it is still not able to reliably keep the agent safe in practice. In short, merely watching over LSS-LRTA*'s shoulder is not sufficient for an effective safe real-time search.

2.4 Heuristic Search for Safety

Having seen the ineffectiveness of simple methods, we now investigate a more sophisticated approach that we call SafeRTS (sketched in Algorithm 2). In contrast to S0, SafeRTS explicitly tries to prove nodes safe, but in contrast to simple safe search, SafeRTS only does this for states that appear to be promising. To enable explicitly finding a plan to a safe state, we introduce a safety heuristic, $d_{safe}(n)$, that estimates the distance, in state transitions, from a given node n forward through the state space to the nearest safe state. For example, in traffic, d_{safe} is the Manhattan distance to the nearest bunker. Because of both static and dynamic obstacles, this is an optimistic estimate. In racetrack, it is the maximum of the absolute values of the x and y velocities of the agent, representing the number of decelerations required to bring the agent to a stop. In the absence of obstacles, this estimate is perfect, although with obstacles the true value may be ∞.

Algorithm 2: SafeRTS

```
Input: $s_{root}$, $bound$
while $s_{root} \neq s_{goal}$ do
    $C \leftarrow \emptyset$
    $b \leftarrow 10$ ◃ initialize expansion budget
    while expansion limit bound is not reached do
        perform Astar for $b$ expansions
        perform Best-First Search on $d_{safe}$
            from the top node $t$ of open
            until comfortable node $c$ is found
            or until $b$ expansions
        if such c found then
            cache comfort of nodes in path from $t$ to $c$
            $b \leftarrow 10$ ◃ reset budget
            $C \leftarrow C \cup \{t\}$
        else
            $b \leftarrow 2 * b$ ◃ increase budget
    for $c \in C$ do
        propagate comfort to ancestors of $c$
    choose node $s$ in open with lowest $f$ value that has $s_{safe}$ safe predecessor
    if such $s$ and $s_{safe}$ exists then
        $s_{target} \leftarrow s_{safe}$
    else if identity action is available at $s_{root}$ then
        $s_{target} \leftarrow s_{root}$ ◃ apply identity action
    else
        Terminate ◃ no safe path is available
    use Dijsktra to update $h$ values of the nodes
    move the agent along the path from $s_{root}$ to $s_{target}$
    $s_{root} \leftarrow s_{target}$
```

SafeRTS distributes the total expansion allowance available for the planning iteration between two alternating stages: exploring the state space via a best-first search on f (Figure 2-5) and attempting to prove that the currently most promising frontier node is safe via a best-first search on d_{safe} (Figure 2-6).

Figure 2-5 shows the state of the search at the end of an exploration stage. The agent is on the left side and a goal is on the right. The black area marks the state space covered by the current and the prior exploration stages within the current search iteration and the gray area shows the area that the algorithm will explore by the end of the search iteration. To put this in perspective there are multiple alternating stages (exploration and proof) within a search iteration and usually it takes multiple search iteration for the agent to reach the goal.

At the end of the exploration stage the node with the lowest f value (breaking ties toward low h) is chosen as the promising node for the subsequent proving stage. The following proving stage will attempt to prove the safety property of this node. The selected node is marked by the *safety search* label and already proven or naturally safe node with green on Figure 2-5.

The proving stage succeeds when the chosen node is determined to be a predecessor of a comfortable or safe node. Figure 2-6 demonstrates that the search tree and the search frontier (denoted with orange color) of the safety search is independent from the search tree used for exploration. The safety oriented search terminates if it expands a safe or a comfortable node (green node on Figure 2-6)[line 8] or the budget allocated for the search runs out [line 9].

We do not know how much effort will be required to prove the chosen node safe, or even if this is possible, so both the exploration and proving stages are subject to a stage expansion budget, initially set arbitrarily to ten nodes. Nodes generated during the proving stage are not added to the search tree, as they may have suboptimal g values that could mislead the learning phase. The proving stage ends when it succeeds or its budget is exhausted. If the proof is successful, the stage budget is reset to its original value [line 12] and the nodes used for the proof are marked as comfortable and this information is cached for future use [line 11]. If no comfortable descendant is found, the stage budget is doubled, effective with the next exploration stage [line 15]. In this way, SafeRTS limits its effort in case it tries to prove the safety of an unsafe node, while also focusing its proving expansions

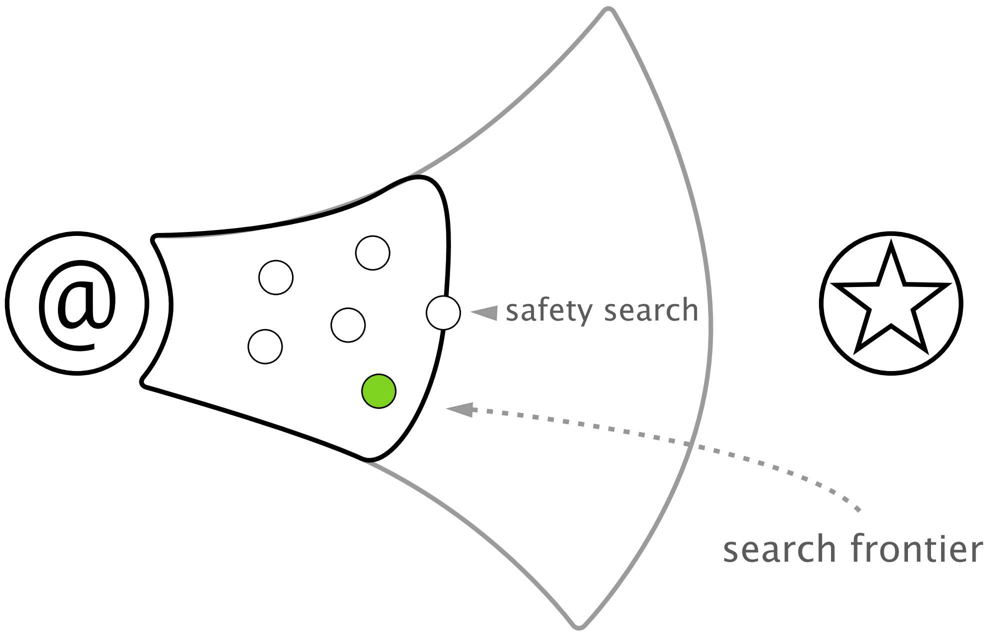


Figure 2-5: Mid iteration best-first search on f is "interrupted" to attempt a safety proof on the current best f node.

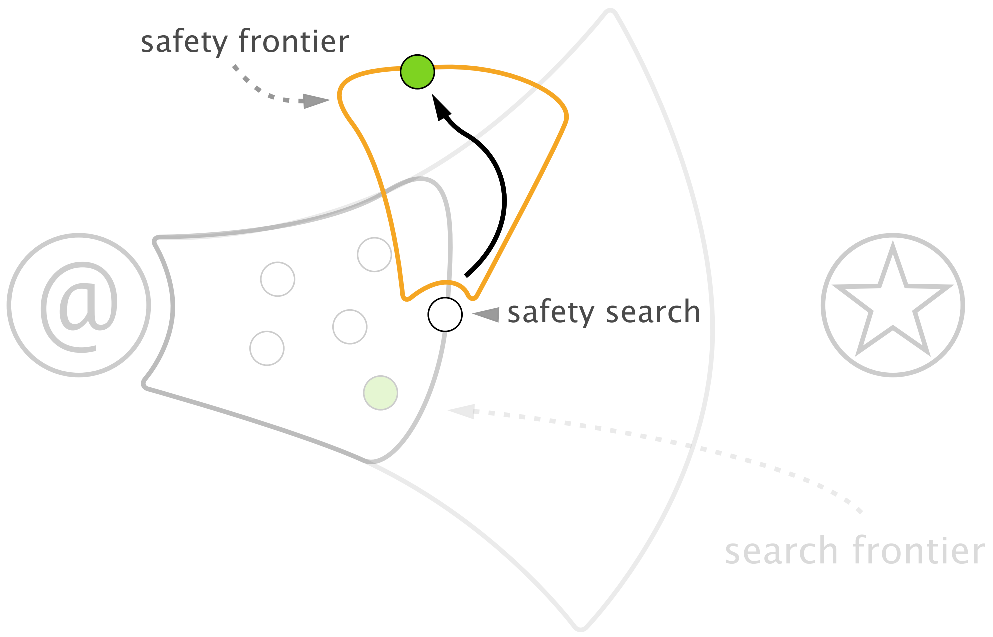


Figure 2-6: Safety proof: Best-first search on d_{safe} started from the current best f node.

on promising states. When the overall time bound is exhausted, SafeRTS performs learning as in LSS-LRTA* [line 25] and backs up comfort through predecessors as in S0 [line 17].

We experimented with two action selection strategies: safe-toward-best, as used in simple safe and S0 and shown in Algorithm 2 [line 18], and best-safe, in which we commit to the actions leading to the safe node that was most recently expanded in the A* search (and was therefore most recently judged promising). Figure 2-7 illustrates their differences. The best-safe strategy would select the best safe node B from all safe nodes and safe-toward-best would commit towards the lowest f node A on the search frontier with a safe ancestor node C.

With either strategy, if no comfortable (or safe in the case of best-safe) nodes are encountered, SafeRTS will take an identity action that takes the agent directly back to the current state if one exists [line 22]. In this case, SafeRTS preserves its current search tree and augments it during the next planning phase.

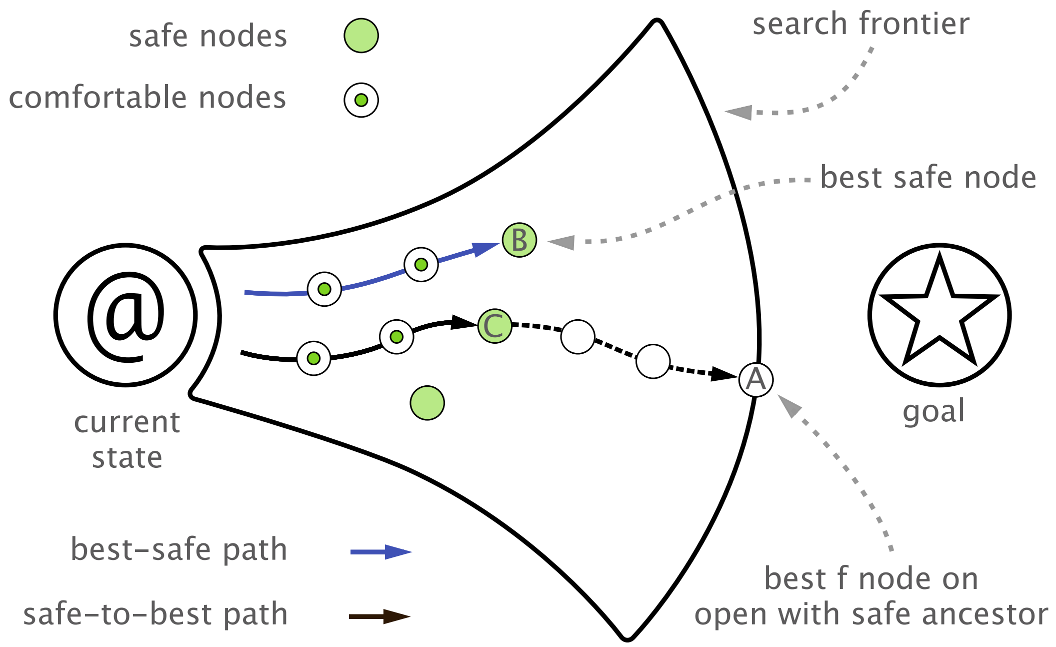


Figure 2-7: Safe action selection strategies.

2.4.1 Theoretical Properties

The properties of SafeRTS depend on the state space and on the action commitment strategy. For the following results, we assume SafeRTS with best-safe action commitment, and we make the following assumptions about the state space:

A0 the state space is finite,

A1 the heuristic is admissible and consistent,

A2 the cost of transitioning between two states is bounded from below by a positive constant,

A3 the start and goal states are safe,

A4 a goal is reachable from every safe state, and

A5 an identity action is available at each safe state.

First, we show that SafeRTS keeps the agent safe.

Theorem 4 *Until it reaches a goal, the agent will always move to a safe state different from its current state.*

Proof: The agent starts in a safe state (A3). If the search expands one or more safe states, the agent will move to one of them. Otherwise, identity actions are taken until a safe node is selected for expansion. Since the space is finite (A0) and the goal is safe (A3) and reachable (A4), this will eventually happen. □

For a proof of completeness, we need a few preliminary results, which are similar to those for previous algorithms.

Theorem 5 *The h values will stay admissible and consistent.*

Proof: The heuristic values are updated using the same learning phase as used by LSS-LRTA*. As the heuristic values are initially admissible and consistent (A1) and action costs are positive (A2), the same proof as used for LSS-LRTA* (Koenig & Sun, 2008, Theorem 2) applies. □

Theorem 6 *Increases in h values are bounded from below.*

Proof: During learning, every h value is set to the sum of one of the original h values, of which there are a finite number, plus a subset of the action costs (no action will be included more than once in a shortest path due to A2). Because only a finite number of such sums are possible, only a finite number of differences between such sums are possible, and thus the increases in h values are bounded from below by a problem-dependent constant. □

Unlike similar results for previous real-time search algorithms, we need to handle the fact that the agent might not move to the best successor. And unlike in Korf (1990)'s proof for RTA*, it is not the case that the h value of the agent's previous state is always updated. Our approach relies on committing only to states that were expanded during lookahead, allowing us to say something about their value.

Theorem 7 *If the agent moves from current state a to any expanded state b inside the LSS, incurring cost $c(a, b)$, and if $h(a) < c(a, b) + h(b)$, then $h(a)$ will be increased during the learning phase.*

Proof: Let $h'(a)$ denote $h(a)$ after learning and let *best* be the node on the LSS frontier with the lowest f value. By the back-propagation of h values from the LSS frontier during learning, $h'(a) = g(best) + h(best) = f(best)$. Due to the consistent h (theorem 5) and A*'s expansion order when forming the LSS, $f(b) \leq f(best)$. We assumed $h(a) < c(a, b) + h(b)$, so we have $h(a) < f(b) \leq f(best) = h'(a)$. □

Theorem 8 *For any looping sequence of states L visited by the agent in which the first state s_1 and the last state $s_{|L|}$ are the same, the h value of at least one state in L will be increased by the learning phase.*

Proof: Assume, for sake of contradiction, that the agent travels in a loop L and no h values are increased. By the consistency of h (Theorem 5), $h(s_1) \leq c(s_1, s_2) + h(s_2)$. But since there is no h increase, Theorem 7 implies $h(s_1) \geq c(s_1, s_2) + h(s_2)$. So we have

$$\begin{aligned} h(s_1) &= c(s_1, s_2) + h(s_2) \\ &= c(s_1, s_2) + c(s_2, s_3) + h(s_3) \\ &= \cdots \\ &= \left(\sum_{i=1}^{|L|-1} c(s_i, s_{i+1}) \right) + h(s_{|L|}) \\ &= \left(\sum_{i=1}^{|L|-1} c(s_i, s_{i+1}) \right) + h(s_1) \end{aligned}$$

Because costs are positive (A2), the sum is positive and this is a contradiction. □

Following the spirit of Korf's proof, the key is to show that infinite traps cannot exist, although we use the properties of h to derive the contradiction:

Theorem 9 *There does not exist a finite set S of non-goal nodes such that, after a finite time, the agent will move within S forever after.*

Proof: Assume, for sake of contradiction, that S exists. Without loss of generality, restrict S to be as small as possible. If there are states in S that are visited only a finite number of times, we consider only times after the last such visit and shrink S, removing such states. Hence, each state is visited an infinite number of times. This implies that, for any state, we will visit it again. By Theorems 8 and 6, this implies that an h value will be increased by an amount bounded from below. There will be an infinite number of such increases, contradicting the admissibility of h (Theorem 5). □

We are now ready for completeness itself:

Theorem 10 *The agent will eventually reach a goal.*

Proof: The agent stays safe (Theorem 4), so a goal is always reachable (A4). For the agent to not reach a goal, there would have to exist a set of non-goal nodes within which the agent traveled forever. Theorem 9 disallows this. □

While the completeness proof made use of several assumptions, it is straightforward to see that some assumptions are required:

Theorem 11 *There does not exist an algorithm that can keep the agent safe in all domains or can be complete in all domains.*

Proof: Assume, for sake of contradiction, that such an algorithm a exists. Construct a simple domain whose state space is a single directed path forking at node n with one successor leading to a dead end and the other to the only goal. To be complete and keep the agent safe, a must make the correct decision at n. If a is able to expand k nodes within the time bound, add $k + 1$ nodes with identical h and d_{safe} values after n along each path. Whichever arbitrary way a breaks the tie, place the goal on the other path. □

2.4.2 Empirical Performance

In the experiments, SafeRTS seemed to perform slightly better with the safe-toward-best action selection strategy, so that is what we present here. Figure 2-3 and Figure 2-4 show that, as expected from Theorem 10, SafeRTS keeps the agent safe while achieving a goal in racetrack. However, SafeRTS can not always be guaranteed keep the agent safe in traffic, because the start state is not always safe in the benchmark set. In the experiments, using a multiple-commitment action selection strategy, SafeRTS never failed. Empirically, it seems that SafeRTS can reliably keep the agent safe, unlike every other real-time method tested.

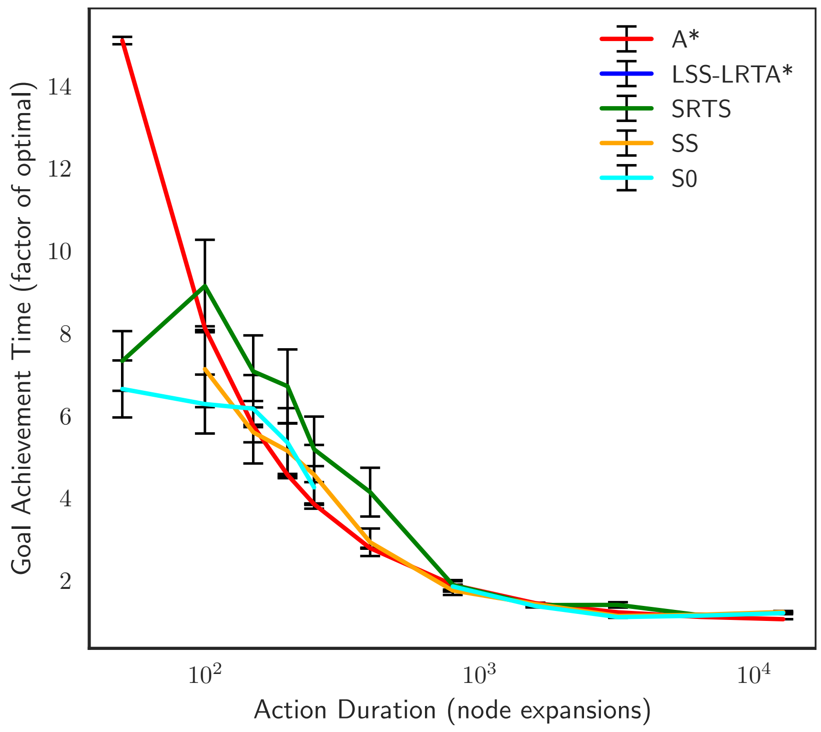


Figure 2-8: Goal achievement times in the traffic domain.

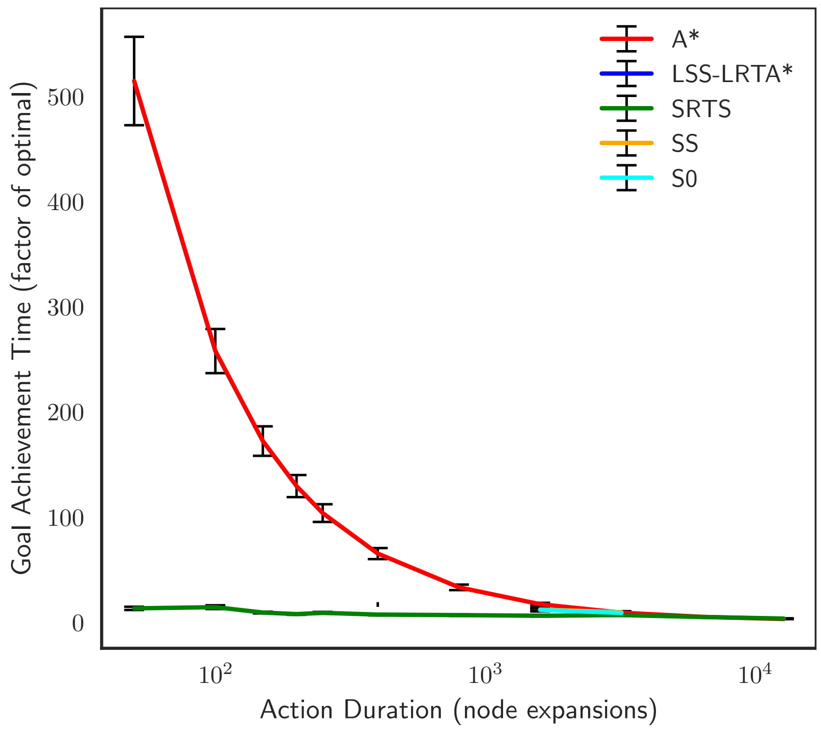


Figure 2-9: Goal achievement times in the racetrack domain.

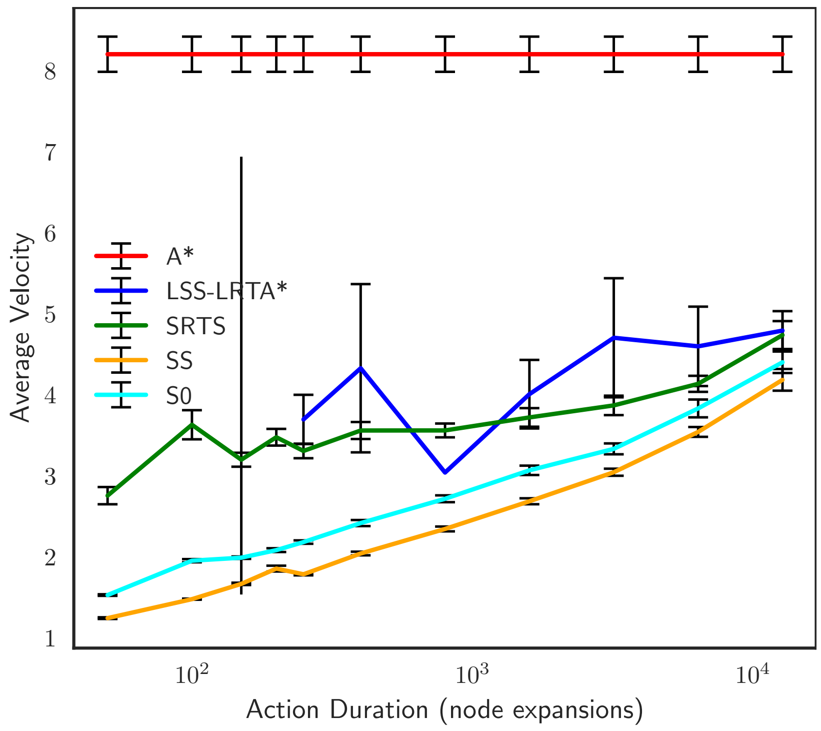


Figure 2-10: Average velocity in the uniform racetrack instance.

As in Chapter 1, to give a sense of the quality of the behavior generated by different planners, we also measured the time (in number of expansions) from the beginning of planning until a goal is reached, known as the goal achievement time (Kiesel et al., 2015). This measure is also well-defined for off-line methods like A*, allowing us to compare them to real-time search. The top and center panels of Figure 2-9 presents goal achievement time for the racetrack and Figure 2-8 for the traffic domains, expressed as a factor of the time taken to execute an optimal plan (A* without planning time). Data points are omitted for action durations for which an algorithm did not solve all the benchmark instances in a domain, thus S0 and SS are rarely seen in racetrack and LSS-LRTA* is rare in both. The results show that SafeRTS surpasses the other real-time methods tested. The plots also include the success rate and goal achievement time of A* for reference, but we note that A* cannot actually be used without non-trivial modification in the traffic domain, as its plan for the initial state will be out of date by the time it is returned because time has passed and the obstacles will have moved. Nevertheless, SafeRTS performs almost as well as A* in traffic while still adhering to the real-time bound and allowing responsive behavior.

To confirm our understanding of these algorithms' behavior, we also measured the average velocity achieved by the agent in the racetrack domain. The bottom panel of Figure 2-10 shows that, as we would expect, the more sophisticated SafeRTS method is able to maintain a higher velocity than the basic safe real-time methods SS and S0, while being more robust than LSS-LRTA*. All methods achieve higher speeds when they are allowed greater lookahead. Again, A* represents perfect off-line performance.

2.5 Discussion

The specific methods we have investigated each have their limitations. Simple safe search can keep an agent safe, but this can come at the expense of completeness. SafeRTS can provide provable completeness, but only under best-safe action commitment and certain domain assumptions. More generally, if the safety predicate can identify very few states as safe, then staying safe and making progress toward a goal will be difficult for any algorithm. And of course, real-time search algorithms can suffer from poor behavior if h is highly misleading—this results in the agent revisiting states, or scrubbing (Sturtevant & Bulitko, 2016), as it updates its h estimates. (Existing methods that avoid this behavior, such as EDA* (Sharon, Felner, & Sturtevant, 2014), assume undirected state spaces.)

Zilberstein (2015) notes the importance of avoiding dead ends when building reliable AI systems, and indeed it has been studied in many contexts. For example, dead end detectors have been proposed in domain-independent deterministic planning (Lipovetzky, Muise, & Geffner, 2016). There has been work in real-time search on identifying so-called 'dead' states and 'swamps' that can be pruned without removing an optimal path (Sturtevant & Bulitko, 2011; Sharon, Sturtevant, & Felner, 2013)—this differs from our focus on dead ends in directed state spaces. Musliner, Durfee, and Shin (1995) use non-real-time planning to generate safe controllers. In planning under uncertainty, work has addressed detecting dead ends (Kolobov, Mausam, & Weld, 2010) and avoiding them in off-line planning (Camacho, Muise, & McIlraith, 2016). Safety has also been addressed in reinforcement learning, where it is important not to take an unsafe action even if it yields important information about the underlying model being learned. Moldovan and Abeel (2012), for example, address safe learning but assume the existence of a policy that can always bring the system back to the starting state. Work on safety in robotics has focused mainly on collision avoidance. The work of Bareiss and van den Berg (2015), for example, proposes a centralized algorithm to control multiple robots to avoid collisions, and is not real-time in the hard bounded sense we use here. Bekris and Kavraki (2007) address safety for a single agent among dynamic obstacles, although their approach is limited to motion planning. Schmerling and Pavone (2013) consider uncertain dynamics. Dey, Sadigh, and Kapoor (2016) combine motion planning with higher-level temporal logic mission plans.

Traditionally, safety in real-time systems has been concerned with formally verifying off-line the behavior of relatively simple pre-specified finite-state controllers against a known system model. In contrast, our aim is to enable a retaskable agent to plan its actions on-line in light of its current goals and knowledge, while still avoiding dead ends. The agent and the world may have a combined state space that is too large to completely verify off-line. The approach we are taking should be easy to adapt to other kinds of reachability maintenance goals, such as 'emcee an exciting party, but ensure that it is always possible to return the house to its pre-party condition by dawn.'

We address safety in on-line planning, as this allows the agent's goals, the system model, or the agent's information about the world to change without necessitating a lengthy pause for replanning. By taking a heuristic search approach, our methods are quite general and not limited to particular state representations or planning formalisms. LSS-LRTA* has been used in partially known environments (Koenig & Likhachev, 2006) and adapted for dynamic environments (Bond, Widger, Ruml, & Sun, 2010), and we expect that our methods should be adaptable to these settings. Furthermore, on-line planning often scales to larger and more complex problems better than off-line planning, because an entire plan or policy does not need to be constructed (Korf, 1990).

2.6 Conclusion

As the results show, susceptibility to dead ends is a serious liability of previous real-time heuristic search methods. LSS-LRTA* failed to solve a significant fraction of instances in both the traffic and racetrack domains. This chapter presented a setting for investigating safety in the context of on-line planning. A user-provided safety predicate is used to encourage the agent to avoid dead ends and a user-provided safety heuristic d_{safe} can be used to speed the search for safe states. This chapter showed that naively adding safety to LSS-LRTA* was not effective. We introduced simple safe search and proved conditions under which it can keep an agent safe, and presented the more sophisticated SafeRTS algorithm that provided high performance on both of our very different benchmark domains while guaranteeing completeness in certain domains. While no real-time method can guarantee safety in every domain, these methods apply in a wide variety of situations.

CHAPTER 3

Safe Temporal Planning for Urban Driving

A self-driving car must always have a plan for safely coming to a halt. Often, finding these safe plans is treated as an after-thought. In this chapter, we demonstrate that techniques explicitly designed for safety can yield higher quality plans and lower latency than conventional planners in an urban driving setting. We adopt ideas from Chapter 2 on safe online real-time heuristic search methods to the spatiotemporal state lattices used when planning for autonomous driving. We experimentally compare our proof-of-concept implementation to conventional methods and find significantly improved performance while still maintaining passenger comfort and safety. This work appeared in the Proceedings of the AAAI-19 Workshop on Artificial Intelligence Safety (SafeAI-19) (Cserna et al., 2019).

3.1 Introduction

A central goal of artificial intelligence is the construction of autonomous systems. It is becoming more common that these systems interact closely with humans. Perhaps the most intimate way that humans can interact with an autonomous system is to climb inside it and put their lives in the hands of its control system. Interestingly, it is exactly such systems, in the form of self-driving automotive mobility systems, that are predicted to become widespread in the coming decades. It is crucial that the AI systems that decide on the actions of autonomous vehicles be designed with safety as a fundamental aspect.

Self-driving vehicle technology has many potential benefits for individuals, such as reduced collisions, improved mobility, and reclaiming time spent driving. They also have the potential for broad benefits to society and the environment, such as reduced car ownership, space devoted to parking, and the number of vehicles required. However, there are still significant problems to be surmounted to achieve this vision. For example, in addition to avoiding actual collisions, for this technology to be successfully adopted, individuals need to feel subjectively safe and comfortable while using an autonomous vehicle. In this chapter, we address the problem of planning trajectories for autonomous vehicles, taking into account both objective safety, given the predicted trajectories of nearby vehicles and pedestrians, and subjective passenger comfort.

3.1.1 Background

Trajectory planning for urban driving has proven to be a complex problem because of the inherently dynamic environment. A typical driver can encounter hundreds of other vehicles and many more pedestrians. Not only does the planner need to consider the spatial location of each part of this environment but their temporal evolution needs to be built into the planning techniques that are employed. These two aspects need to be intelligently combined to produce high quality and computationally feasible algorithms to address the urban driving problem.

Hierarchical methods are state-of-the-art for addressing the difficulty of autonomous urban driving (Paden, Čáp, Yong, Yershov, & Frazzoli, 2016). At the highest level, a vehicle must be able to select a route from a road network based upon its current position in that network and the desired destination of the passenger. This network can be represented as a graph with millions of weighted edges. Traditional offline algorithms such as A* are computationally infeasible, as planners are expected to run at a rate of ten times per second. Once a route has been identified, the vehicle must select a sequence of behaviors to use along that route. These behaviors correspond to situations such as ‘highway cruising’ or ‘stopping at a stop sign’ and identify the pertinent rules of the road with respect to other aspects of the environment, such as pedestrians and other drivers. Next, a motion planning algorithm will decide the vehicle trajectory used to achieve the next behavior. This motion plan then becomes a reference trajectory for a low-level controller to achieve using throttle and steering inputs while correcting for errors and inaccuracies from the vehicle model. These four components of route planning, behavior selection, motion planning, and control comprise the hierarchical approach to autonomous driving.

Our work addresses urban driving at the level of motion planning: we want to send trajectories to the vehicle controller such that the motions are comfortable for passengers while simultaneously ensuring that they are safe. We define a safe trajectory as one that reaches a safe state. In urban driving, a safe state is one in which the vehicle is stopped (Shalev-Shwartz, Shammah, & Shashua, 2017).[1] We assume that the vehicle replans frequently, thus only the first part of a safe trajectory will be executed in most cases before a new trajectory is computed to replace the current plan.

The planner also needs to take into account the comfort of its passengers. Specific ranges of acceleration, when applied to the autonomous vehicle, create an uncomfortable riding experience. Allowing a high amount of acceleration to be chosen by the planner and executed on the vehicle leads to passenger discomfort due to the physical stress the human body undergoes when the vehicle quickly accelerates.

McNaughton, Urmson, Dolan, and Lee (2011) devise acceleration profiles for a spatiotemporal lattice. These profiles can be used to construct a constraint on the planner to maintain the acceleration of the autonomous vehicle within comfortable ranges. Our work addresses these two crucial aspects of the urban driving problem: We construct a planner that can remain safe while limiting the control of the vehicle for the passengers to be comfortable.

In this chapter, we study a general online method for guaranteeing that the planner will find a series of safe actions for the agent to execute. We dynamically allow the agent to plan for its current goals and devise a way to balance the safety of the trajectories the agent is executing while being as close to the edge of non-safe action execution as possible. We define a general problem setting for the use of this technique and study the use of it in the domain of simple urban driving. We empirically test this method on a simulated vehicle with inertia and find that the new techniques dramatically outperform the conventional ones. Unlike, in Chapter 2, we assume that the environment might change between planning iteration due other moving agents and unforeseen static obstacles. Thus, the planning graph is discarded after every planning iteration as the previously computed node costs are potentially invalid.

[1]For highway driving, a safe state might be one where the car is pulled over at the side of the road. The identification and analysis of traffic dependent safe states are beyond the scope of this chapter.

3.2 Previous Work

A state lattice (Pivtoraiko, Knepper, & Kelly, 2009) is a discretization of a continuous state space. A Spatiotemporal state lattice (Ziegler & Stiller, 2009) is the result of combining a traditional state lattice with time and velocity dimensions. The urban driving domain requires the planner to be able to consider both time and space while planning. The state lattice gives us a method for searching through a static environment; however, adding in time and velocity to the state space lattice can lead to an exponential blowup of the size of the search space. Even assuming a modest number of possible accelerations applied to a vehicle, the state space can contain nearly 12 million trajectory edges that would need to be evaluated during each planning iteration (Pivtoraiko et al., 2009). On the other hand, it is a difficult task to compose search heuristics for the complex cost function of urban driving. For example, McNaughton et al. (2011) applied exhaustive search on a spatiotemporal state lattice.

CL-RRT (Kuwata, Teo, Fiore, Karaman, Frazzoli, & How, 2009) is a real-time motion planning algorithm that can guarantee safety. To deal with dynamic obstacles, in each planning iteration, the algorithm cleans the motion tree by removing the invalid tree nodes and saving unconnected subtrees into a stand-by forest. A random state is sampled by biasing the nearest tree in the forest or the goal. If the new sampled state is in one of the subtrees, then it will try to connect the nearest state on the current motion tree to the sampled state by solving a boundary value problem. Otherwise, it will perform the conventional RRT extend routine. They guarantee safety by ensuring that the vehicle is stopped and safe at the end of the trajectory. However, the random feasible solution that is constructed by the RRT could be a highly sub-optimal plan (Karaman & Frazzoli, 2011). We will refer to approaches that expands the search space until a stopped goal state is found as plan-to-stop.

Chapter 2 introduced SafeRTS (Cserna et al., 2018) a real-time search algorithm that can guarantee safety. It explicitly tries to prove nodes are safe and finds a plan to a safe node. SafeRTS searches for a partial real-time plan that has a frontier node with the most promising f value and guarantees that plan could be lead to a safe node in the lattice. The safety heuristic $d_{safe}(n)$, estimates the distance through the state space from a given node n forward to the nearest safe state. As an online search algorithm, SafeRTS distributes the expansion allowance between exploring the best f

Algorithm 3: SafeTLP

```
    Input: $s_{root}$
 1  perform Best-First Search on lattice to find a potentially unsafe trajectory $T$ from $s_{root}$ to a
      partial goal
 2  $s_{current} \leftarrow T.last$
 3  while $s_{current}$ exists do
 4      perform Best-First Search on $d_{safe}$
            from the node $s_{current}$ to $s_{goal}$
 5      if $s_{goal}$ is found then
 6          cache the safe partial path from $s_{current}$ to $s_{goal}$
 7          return $\langle s_{root} \ldots s_{current} \rangle \langle s_{current} \ldots s_{goal} \rangle$
 8      else
 9          $s_{current} \leftarrow s_{current}.predecessor$ in $P$
10  Best-First Search to find a safe trajectory $T$ from $s_{root}$ to $s_{goal}$
11  return $T$
```

state and attempting to prove its safety. The safety proof performs a best-first search on d_{safe}.

3.3 Safe Temporal Lattice Planning

We now turn to applying Chapter 2's notion of safe planning to spatiotemporal state lattice planning. The planner's goal state is a location a predefined distance along the current route with a zero velocity. Our technique, Safe Temporal Lattice Planning (SafeTLP), first performs a best-first-search from the root state s_{root} until a partial goal state is expanded (line 1 of Alg. 3). A partial goal state is a state that matches the location of the goal state but may not have the correct speed. The priority function of the best-first search prioritizes states with lower distance-to-goal, earlier goal achievement time, and higher speed.

This naïve solution is inherently unsafe as it does not consider upcoming obstacles and dead-end states beyond the explored spaces or partial goal. To ensure safety SafeTLP proves that a prefix of the naïve trajectory is safe by constructing a safe trajectory starting with this prefix. The algorithm first attempts to prove that the last state of the trajectory is safe (line 4). If the safety proof is not successful, SafeTLP falls back to the preceding state on the trajectory(line 9). The last few states in the naïve trajectory may be skipped if a maximum allowed deceleration would not allow the agent to come to a full stop from these states.

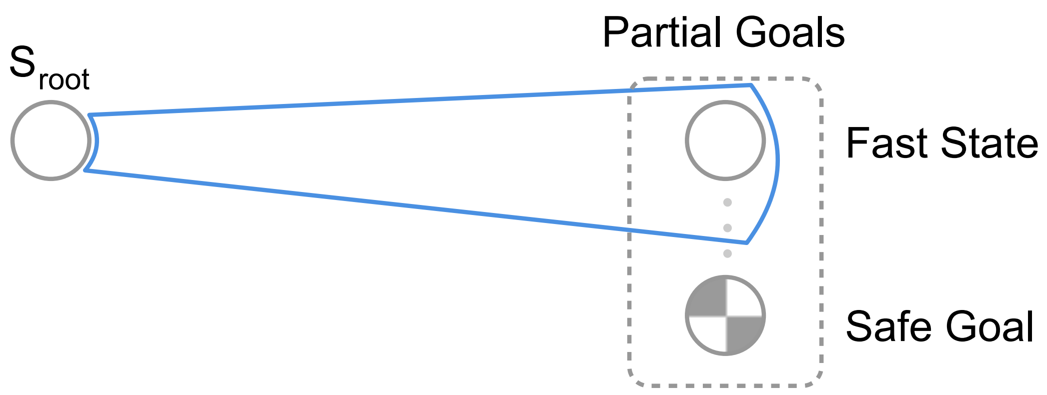
As we attempt to prove safety, we store every node encountered in a special safety closed list. If reencountered during a subsequent attempt, such nodes are not expanded as we have already attempted to prove safety from them.

Similar to SafeRTS's d_{safe} heuristic, SafeTLP prioritizes states that are closer to the goal state. The safety proof expands the search tree under the state $s_{current}$ (Cserna et al., 2018) with expansion ordered based on the distance to goal and speed (the lower, the better). If the safety open list becomes empty, the proof is unsuccessful and $s_{current}$ is labeled as unsafe. Upon the expansion of the goal node, the safety search terminates and returns the trajectory leading to the goal from $s_{current}$. $s_{current}$ is now proven safe and its naïve partial plan is augmented by the discovered safe trajectory to the goal to form a complete safe plan from the agent's current state to a goal state (line 7).

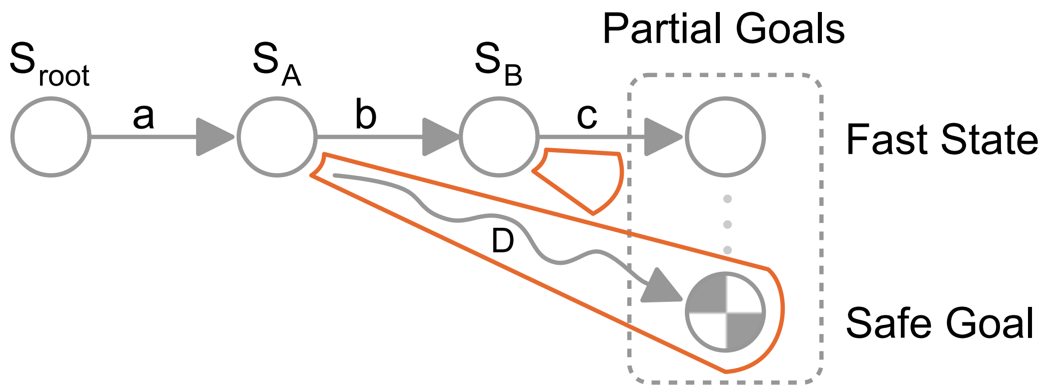
We use Fig. 3-1 to demonstrate the behavior of SafeTLP. SafeTLP first expands a search tree (blue on Fig. 3-1) from s_{root} that optimizes for velocity until it reaches a state in the partial goal set. In our example the sequence of a, b, and c trajectory segments represents a trajectory that leads to a partial goal state $s_{partial\_goal}$. Then SafeTLP attempts to find a safe trajectory to s_{goal} from each intermediate state identified by the $\langle a, b, c\rangle$ trajectory starting from the state closest to the partial goal. First, it attempts to prove that $s_{partial\_goal}$ is safe, then it falls back to S_b, S_a, and lastly to s_{root} if necessary. These proofs are independent search trees marked with orange on Fig. 3-1. The proof searches are optimized on d_{safe}. In our example the safety proof from $s_{partial\_goal}$ and S_b fails and only succeeds from S_a. SafeTLP constructs a safe trajectory to s_{goal} by appending the prefix $\langle a\rangle$ trajectory that leads to S_a with the trajectory segment sequence D.

Theorem 12 *SafeTLP is guaranteed to find a safe plan if one exists in the state space.*

Proof: If all safety proofs initiated from the states on the naïve path (line 5 of Alg. 3) fail, SafeTLP



(a) SafeTLP phase 1: search optimized for performance (velocity, comfort, etc.).



(b) SafeTLP phase 2: search optimized for safety.

Figure 3-1: SafeTLP search phases.

will perform a best-first search from the agent's current state (line 10). Because best-first search does not prune states from the search tree and the state space is finite, this search will eventually find paths from s_{root} to every other reachable state. Best-first search is exhaustive and complete in finite state spaces, thus eventually it will identify a path to a safe state if one exists. □

Theorem 13 *In the worst case, SafeTLP expands no more than twice as many states as the naïve plan-to-stop method expands in the worst case.*

Proof: Recall that, due to the safety closed list, any node will be expanded at most once by a safety proof attempt. If all the safety proofs fail, then SafeTLP will perform best-first search initiated from the agent's current state. This exhaustive search will expand every state at most once, as the standard

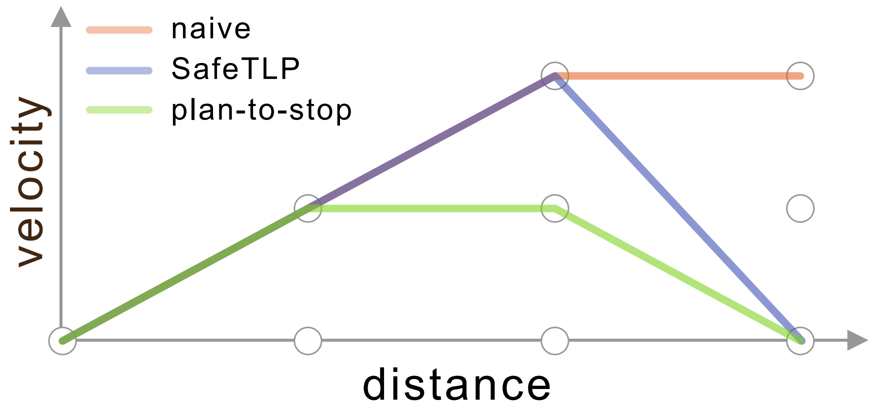


Figure 3-2: An cartoon sketch of algorithm behavior for spatiotemporal planning.

closed list of best-first search will prevent duplicate expansions. Thus any state will be expanded at most twice (once by a safety proof attempt and once by the exhaustive search). The naïve method expands each node at most once. □

In motivating our approach, we present three different approaches in Figure 3-2. The abstract diagram shows the distance towards the goal along the horizontal axis and the velocity of the vehicle along the vertical axis. Ideally, we want an algorithm which allows the vehicle to travel as fast as possible towards the goal while guaranteeing that there is an alternate course of actions to travel to a safe state in the case of an emergency or unexpected situation.

The basic naïve approach, shown in orange, performs a best-first search towards the goal. By optimizing distance-to-goal, this search tends to accelerate the vehicle until it reaches the maximum velocity and then maintain that speed. This technique is clearly not safe as it does not guarantee a way for the vehicle to reach a safe state in case of an emergency. However, it does fulfill the task of accelerating the vehicle to a very high speed. On the other hand, the current state-of-the-art is a more exhaustive approach, shown in green and labeled plan-to-stop. This performs an exhaustive breadth-first search expanding all the possible trajectories for the vehicle to undertake with the goal of completely stopping the vehicle at the end as shown in Fig. 3-3. This approach is able to find a safe way of reaching the goal at the price of many node expansions and a slow velocity for the vehicle. To be able to plan to slow down at the current planning horizon, it needs to plan to have a

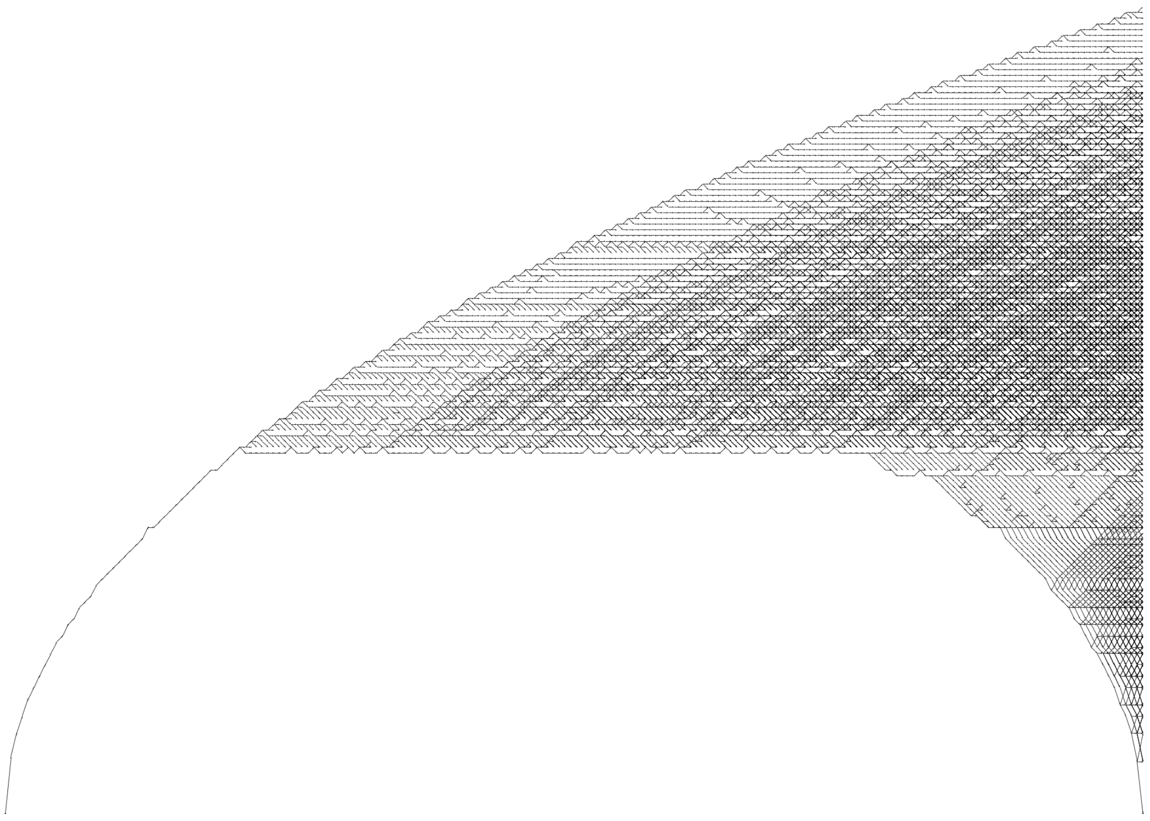


Figure 3-3: Distance-velocity projection of the plan-to-stop search tree. The horizontal axis represents space and the vertical axis velocity. Note that the temporal aspect of the states is not captured by the projection.

	1	2	3	4
comfortable acceleration	0.8	1.0	1.2	1.5
comfortable deceleration	0.8	1.0	1.2	1.5
aggressive deceleration	1.6	1.8	2.0	2.2

Table 3-1: Acceleration limit sets (m/s^2). Inferred from the work of Powell and Palacín (2015), Martin and Litwhiler (2008), and Hoberock (1977)

lower average velocity than the naïve approach would find.

In essence, we have devised a hybrid approach that performs a best-first search to find a trajectory leading to a very high speed, and then performs a search on safety to find a series of safe actions leading us towards the goal safely represented by the blue line. In this fashion, we guarantee that even though we are accelerating the vehicle very quickly towards the goal, there will be a series of actions the vehicle can take to reach a safe state in the event of an emergency or unexpected action. SafeTLP can utilize more aggressive deceleration actions as it is aware that the those are only going to be executed in unlikely case of emergency. This allows the agent to have a higher average travel speed compared to the plan-to-stop methods. This purpose-designed hybrid algorithm combines the low number of expansions of the best-first search's high-velocity actions with the guarantee of safety of the exhaustive search.

3.4 Empirical Evaluation

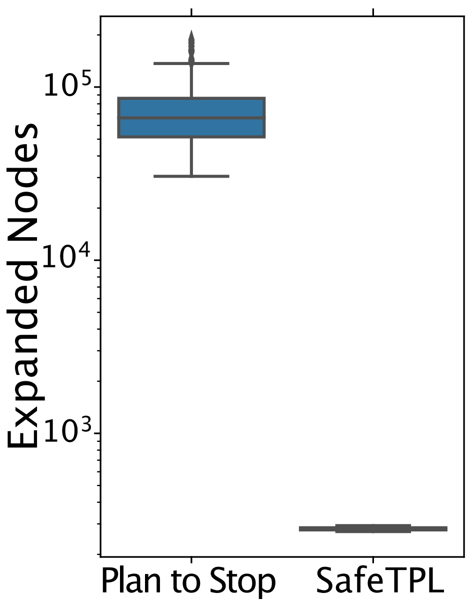
We demonstrate planning in simplified spatiotemporal lattice. The goal is 100 meters away from the start position of the vehicle. The road is discretized by half meters, so there are 200 states in total on the distance dimension of the lattice. From each state, the vehicle can apply three actions: accelerate, maintain the velocity, and decelerate. The studies of Powell and Palacín (2015), Martin and Litwhiler (2008), and Hoberock (1977) show that 0.8 to 1.5 m/s^2 are comfortable acceleration values for urban driving, and people can tolerate up to 2.2 m/s^2 without injury. Following this, we design four sets of comfortable acceleration and deceleration pairs, along with the aggressive deceleration that would be allowed to apply in our approach. Table 3-1 shows the acceleration and deceleration sets we used in our experiment. We fixed the maximum velocity limit at 15 m/s. We vary the starting velocity from 0 to 5 m/s with 0.2 increment.

Figure 3-4a shows the number of expanded node for the algorithms to find a safe trajectory. As we can see, SafeTLP expands about three magnitudes fewer nodes than plan-to-stop. This is because plan-to-stop performs an exhaustive search of the state space while SafeTLP explicitly reasons about the optimal safe solution. Figure 3-4b shows the planning time. As expected, SafeTLP consumes much less time than plan-to-stop to find a safe plan. In a real application setting, fast planning enables the planner to run in a higher frequency, thereby enabling the vehicle to react to the environment more quickly and thus be safer.

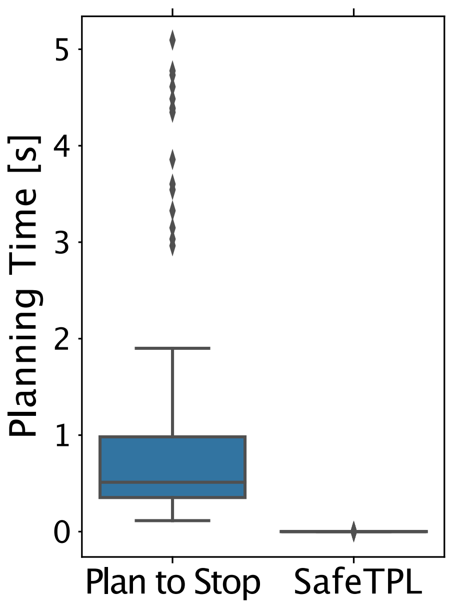
Figure 3-4c present violin plots showing the distribution of the average velocity over the test instances. As we can see, SafeTLP produces higher performance trajectories than plan-to-stop.

3.5 Conclusion

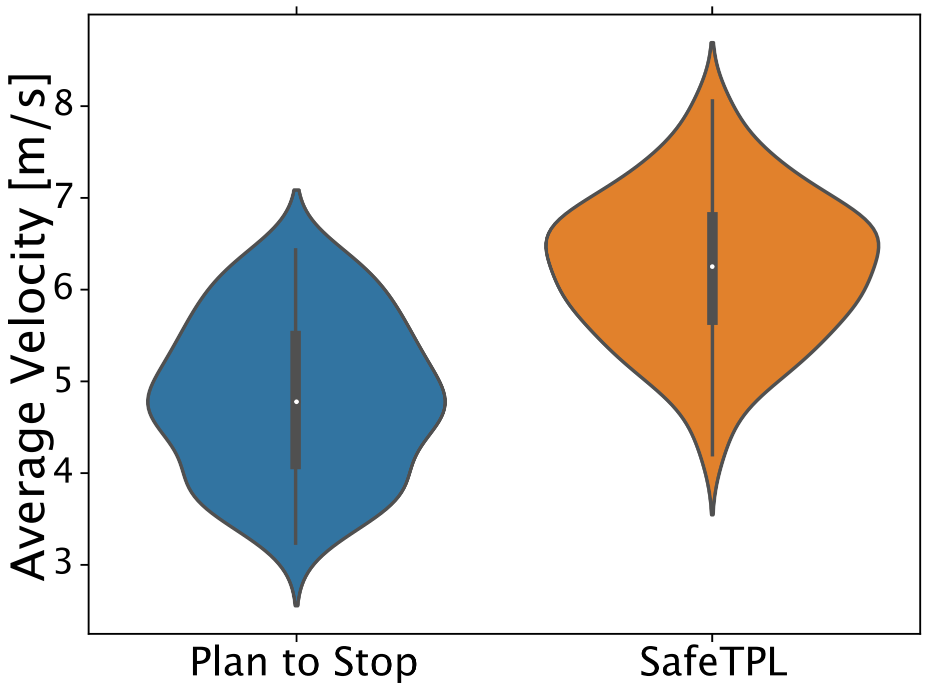
This chapter introduced a new and more effective method for safe action selection in spatiotemporal planning as a practical manifestation of the ideas from Chapter 2. Previous methods are extremely conservative in how they guarantee safety by expanding drastically more nodes than is required. Our planner can quickly generate comfortable and safe plans online. We demonstrate that planning time is drastically reduced while simultaneously being able to achieve a higher average velocity for the vehicle. In combining techniques from real-time heuristic search and spatiotemporal planning, we introduced a method for selecting safe and fast plans quickly.



(a)



(b)



(c)

Figure 3-4: (a): Distribution of the number of expanded nodes during search. (b):Planning time distribution. (c): Average vehicle velocity distribution.

CHAPTER 4

Framework for Safe Real-time Heuristic Search

SafeRTS exploits a user-provided predicate to identify safe states, from which a goal is likely reachable, and attempts to maintain a backup plan for reaching a safe state at all times. In this chapter, we introduce a more general real-time heuristic search framework for safety that relies on the foundation outlined in Chapter 2 and incorporates two novel methods to minimize the overhead of dealing with dead-ends. We prove that the new approach performs at least as well as SafeRTS and present experimental results showing that its promise is fulfilled in practice. In addition, this chapter offers a new domain designed to evaluate safety oriented real-time search methods.

4.1 Introduction

In Chapter 2, we proposed the first real-time heuristic search method, SafeRTS, that is able to reliably reach a goal in domains with dead-ends. Prior real-time methods focus their search effort on a single objective that minimizes the cost to reach the goal. A single objective is insufficient to provide completeness and minimize the time to reach the goal as these often contradict each other. Thus, SafeRTS distributes the available time between searches optimizing the independent objectives of safety and finding the goal.

The contribution of this chapter is four-fold. First, we argue that benchmark domains used for real-time planning may not be good indicators of performance in the context of safe real-time planning. We present a new set of benchmarks that overcome the deficiencies of previous benchmark domains. Second, we show how to utilize meta information presented by safety oriented real-time search methods to reduce redundant expansions during both the safety and goal-oriented searches. This improvement marginally reduces the goal achievement time (GAT) while it does not increase the space and time complexity of the safe real-time search method. Third, we prove inefficiencies in the approach taken by SafeRTS by examining properties of local search spaces and the changing priority of which nodes to prove safe as an LSS grows. Lastly, introduce a new framework for safe real-time heuristic search that utilizes the time bound unique to real-time planning. This framework follows the same basic principle of search effort distribution as SafeRTS but does so more efficiently. We empirically demonstrate the potential of the new framework.

4.2 Safety in Heuristic Search

A dangerous limitation of most real-time search methods is that in directed domains, no resources are spent on ensuring that the path being committed does not lead to a dead-end. If a terminal state s_t (i.e. one with no successors) is just beyond the expanded search frontier, the agent may still commit actions toward an immediate predecessor of s_t.

Chapter 2 introduced the notion of safety as a way of evaluating which states are less likely to lead to dead-ends. Here we expand on this notion and provide formalized definitions for safety concepts.

Definition 8 *Any path* p *in the set of all possible paths through the state space* P *is a sequence of nodes such that node* $\mathrm{n}_{\mathrm{i+1}}$ *is a successor of* n_i *terminating in some arbitrary node* n_{k}.

$$\begin{aligned} p = &\langle n_1, n_2, \ldots n_k\rangle \in P \Leftrightarrow \\ &\quad \forall_{i\in[1\ldots k]} n_i \in N \wedge \forall_{i\in[1\ldots k-1]} n_{i+1} \in successor(n_i) \end{aligned}$$

Definition 9 *A node n is* safe *iff there exists a path* p *that begins at n and ends with a goal.*

$$n \textit{ is safe iff } \exists\, p \in P :\; p_1 = n \wedge isGoal(p_{|p|})$$

Definition 10 *A* dead-end *node is a node* n *from which there is no path* p *to a goal.*

$$n \textit{ is dead-end iff } \neg\, \exists\, p \in P : p_1 = n \wedge isGoal(p_{|p|})$$

A node that is likely to be safe according to some criterion we will denote $safe_L$.

Definition 11 *A safety predicate* $\mathrm{f_{safe}}$ *determines whether a given node is likely to be safe or its safety property is unknown.*

$$f_{safe} : N \rightarrow \{safe_L, unknown\}$$

f_{safe} is a user provided function without any particular guarantees, merely to guide the search algorithm towards states that are *likely* to not lead to dead-end states.

Definition 12 *A safety predicate* $\mathrm{f_{safe}}$ *is strong iff there exist a path* p *to the goal from every node* n *that is marked* $\mathrm{safe_L}$ *by the function.*

$$\forall n \in N : f_{safe}(n) \rightarrow safe_L$$
$$\exists p \in P : p_1 = n \wedge isGoal(p_{|p|})$$

Definition 13 *A node* n *is* explicitly safe *if* $f_{safe}(n) = safe_L$*. A node* n' *is implicitly safe if it is a predecessor of a safe node n.*

$$(f_{safe}(n) = safe_L) \wedge \exists p \in P : p_1 = n' \wedge n \in p$$

We refer to both *explicitly safe* and *implicitly safe* nodes simply as *safe* when the distinction is unimportant.

Definition 14 *A safety proof of a node* n *is a path* p *that begins at* n *and ends at a safe node.*

$$proof(n) = p \in P \ : p_1 = n \wedge f_{safe}(p_{|p|}) = safe_L$$

Definition 15 $\mathrm{proof}^*(\mathrm{n})$ *is an optimal safety proof of* n *if it has minimum number of states.*

Definition 16 *A safety heuristic function* $\mathrm{d_{safe}(n)}$ *is a function that estimates the minimum distance in state transitions between* n *and any* safe *node* $\mathrm{n'}$.

$$d_{safe} : N \rightarrow \mathbb{N}^0$$

d_{safe} is a user-defined heuristic function that does not require any suboptimality bound on its estimate of the distance to a safe state.

4.2.1 SafeRTS

As a reminder, SafeRTS splits planning resources between exploration for a goal via best-first search on f and attempting to prove promising nodes as safe using d_{safe}. The planning phase alternates between these two tasks: first, nodes are expanded as in A* to some expansion limit b. Then, the node on top of the open list is selected as the target of the safety proof. Nodes are expanded using a best-first search on d_{safe}, starting at the target node until the same node expansion limit b is reached or until a *safe* node is discovered. Nodes expanded by the proving stage are not added to the search tree because their g values are not based on the agent's location. If no *safe* node was discovered in the proving stage, the resource limit b is doubled and the algorithm switches back to goal-finding with expansions on f. Once *bound* total expansions occur between both exploration and proving stages, the algorithm shifts to the learning phase.

In the learning phase, the safety of all discovered safe nodes is propagated back to their ancestors. Then h values are propagated back through the local search space in exactly the same manner as in LSS-LRTA*. Actions are selected based on a strategy referred to as "safe-toward-best" whereby the agent would select as its target the deepest *safe* node along the path to the node on the open list that a) has a *safe* predecessor, and b) has the best f value of all other nodes on the frontier with a *safe* predecessor. If no nodes on the open list have a *safe* predecessor, the agent takes an "identity action" defined as an action which transitions to the exact same state, if such an action exists.

The approach of using half the expansions on proving safety instead of exploration means that the search tree of SafeRTS does not go as deep as that of LSS-LRTA*. Empirical results showed that the tradeoff was well worth the effort as SafeRTS was able to avoid dead-ends at far greater rates than the baseline LSS-LRTA*. However, the technique of switching between exploration and proving stages within the planning phase is inefficient in that nodes may be expanded in the proving stage that will be expanded in a subsequent exploration stage. Below, we will present theorems supporting this assertion and we explore alternatives. But before we discuss and evaluate new algorithms in detail, we introduce a new and more efficient benchmark domain.

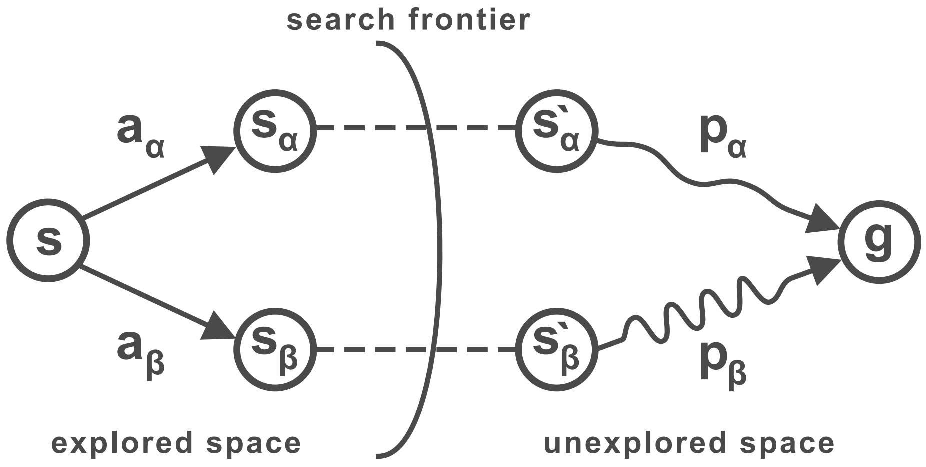


Figure 4-1: Demonstration of actions with long term effects.

4.3 A Benchmark Domain for Evaluating Safety

Real-time planning algorithms construct a solution iteratively and start to move the agent immediately after the first completed iteration. In the chain of decisions, the starting point of each decision is the result of all prior decisions. We argue that it is important to balance the impact of each decision on the overall GAT.

Consider the example domain in Figure 4-1. The circles mark the states, the straight arrows the actions, the dotted lines two identical segments, and the squiggle arrows long segments. The segments consist of a sequence of states connected by actions. The agent is currently at state s and the planner has to decide between selecting action a_α and b_β which lead to states s_α and s_β respectively. Given the real-time setting, the planner has limited time to inspect the possibilities beyond these states and would not be able to switch between the α and β path after the first step. Assuming that the dotted segments and the leading actions are identical and that the exploration does not reach $s^{'}_\alpha$ and $s^{'}_\beta$ states, deciding between the two alternative actions is not informed because p_β is much cheaper than p_α. This arbitrary decision would penalize or reward the planner despite the fact that it does not reflect intelligent decision-making. In domains where similar settings exist single actions have a disproportional impact on the GAT.

As an illustration of how to reduce the long term impact of actions in benchmark domains, we introduce a new benchmark domain called Airspace. In Airspace the state is two dimensional

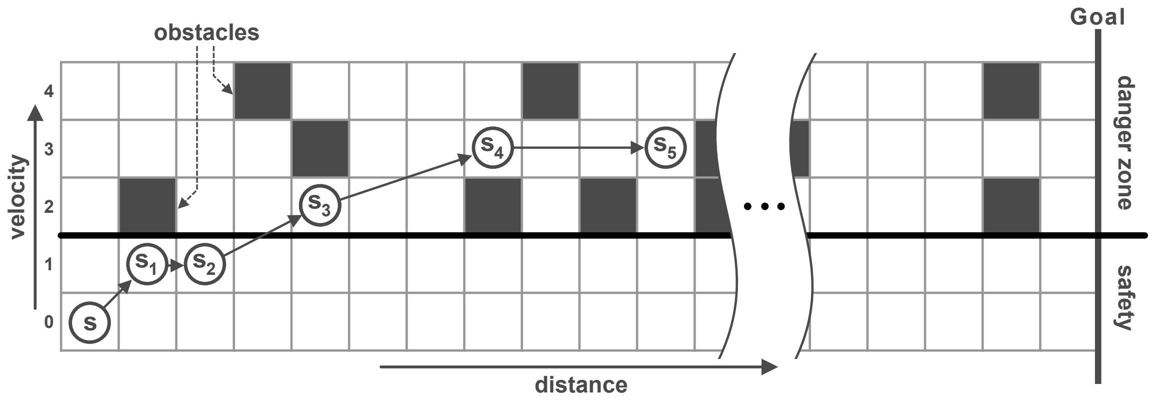


Figure 4-2: Airspace domain.

consisting of distance and altitude. The agent starts with zero speed and altitude. The goal is to traverse a predefined distance. In each step, the agent moves a distance matching its altitude. In other words, the higher the altitude the larger the agent's velocity towards the goal. Above altitude 1, obstacles are blocking the way of the agent with uniform probability p_{obs}, thus making it more difficult for the agent to traverse the space with higher speeds. The agent has three possible actions: increase, keep or decrease the altitude. The agent can only take an action if it does not lead out of the bounds of the map and if the linearly interpolated state between the source and target state is not in collision. Figure 4-2 shows a sketch of the Airspace domain with an example path that demonstrates the dynamics of the agent. The agent starts from state s in the bottom left corner and the goal is to cross the goal line on the right. The *safety predicate* of the domain marks all states at altitudes 0 and 1 as safe. The *safety heuristic* of a state is its altitude -1. The heuristic function returns the remaining distance from a state to the finish line divided by the maximum speed.

Properties of the Airspace Domain

One of the key principles behind Airspace is to avoid allowing any single decision to have an outsized effect on a planner's overall performance. Airspace exhibits this property because each altitude layer is connected with reasonable probability to the layers above and below. While, at any one time, some of these connections may be blocked by the obstacles, there will be enough options available that the agent still has the possibility of reaching any layer in the long term. Ideally, the agent is able

altitude	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
safety probability	.95	.94	.89	.88	.86	.80	.74	.70	.64	.58	.51	.43	.35	.27	.19	.12	.06
proof length	4	5	6	8	9	11	12	14	16	18	20	23	25	27	29	31	32
successful proof expansions	4	5	7	8	10	12	14	18	21	26	32	38	45	53	60	65	68
failed proof expansions	1	1	1	1	2	2	2	3	4	5	8	10	13	14	14	11	7

Table 4-1: Difficulty of safety proofs in the Airspace domain ($p_{obs} = 0.05$) averaged over 100,000 states per altitude.

to return to safe low altitudes regardless of its position as there are no long separators present in the state space between high and low altitudes. In fact, using an obstacle probability of $p_{obs}0.05$ allows a perfect agent to achieve a velocity of 13 on average, meaning the agent can vertically traverse the domain using only a small number of actions. We recommend choosing p_{obs}, the horizontal and vertical dimensions of the domain such that the number of steps it takes to vertically traverse the domain is negligible compared to the horizontal distance. The overall effect is that the agent is only forced to fail if it makes a series of multiple poor decisions, rather than a single uninformed blunder.

Altitude layers higher than 1 contain dead-end states due the velocity of the agent and the obstacles in the space. The probability of hitting an obstacle at altitude a taking an altitude keeping action for any such layer is $p^a_{collision} = 1-(1-p_{obs})^a$. Thus, higher altitudes lead to better performance, as the agent travels faster towards the goal, but they are increasingly more difficult to navigate. It is not only more difficult to find a feasible path at high altitudes, but it makes it more difficult to complete a safety proof due both to the distance from the safe states and the probability of hitting an obstacle.

Table 4-1 shows the probability of a state being safe, the average number of steps to reach a safe state, the average number of nodes expanded during both successful and failed proofs for each altitude. These average values were empirically measured from all states of a 100,000 long and 20 high Airspace domain with $p_{obs} = 0.05$.

Additionally, Airspace guarantees that the agent will not visit a state more than once, thus it eliminates scrubbing (Sturtevant & Bulitko, 2016), focusing the benchmark on exploration and safety rather than on learning.

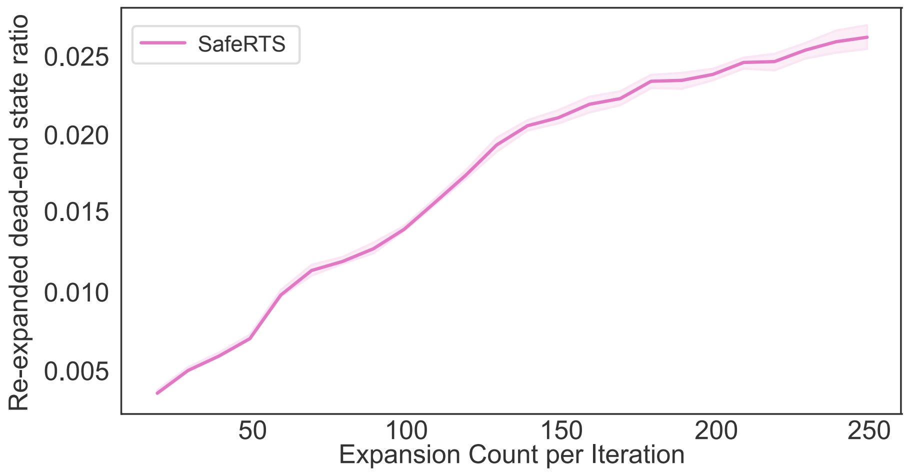


Figure 4-3: Ratio between dead-end re-expansions and total number of node expansions.

4.4 Propagation of Dead-ends

As our second contribution, we will introduce a method of using information gathered about dead ends to reduce future planning effort. Safety in real-time search relies on identifying the safety property of states using the safety predicate and propagating this information to predecessors. The converse of this problem can be formulated, given a predicate for dead-ends, as identifying and propagating dead-ends. Proving whether a state is safe or unsafe is as difficult as the original problem (e.g. consider a domain where the only safe states are the goal states). We argue that focusing on safety is more practical for planning than detecting dead-ends due to their propagation properties. While a state is considered to be implicitly safe if any of its successors are safe, a state can only be considered an implicit dead-end if all successors are proven to be dead-ends. Thus, having a safety predicate that is able to identify a subset of safe states is more practical than a similar dead-end detector.

Attempting to prove the safety property of a state has three potential outcomes. First, if a safe descendant is found then the state is safe. If the allocated budget for the safety proof is exhausted then the proof is non-conclusive: the state could be safe or unsafe. Lastly, when all descendants are expanded without leading to a safe state, then the state and all of its descendants are considered to be unsafe. We propose to cache this information to ensure that these states are not re-expanded in the future by the goal-oriented search effort or by the safety proofs. Marking states unsafe following a safety proof does not increase the space or time complexity of the algorithm, as all of the touched states have already been visited. It requires only an extra flag per state to be stored.

To assess the impact of this simple enhancement, we measure the ratio between dead-end re-expansions and the total number of expansions. In the augmented SafeRTS this quantifies the avoidable additional work. Figure 4-3 shows this ratio on the Airspace domain (band indicates 95% confidence interval around the estimate). Eliminating the re-expansions would yield 0.5 – 2.5% additional expansions. The improvement did not lead to statistically significant results in our experimental setting. However, the yield is highly domain dependent. Our intuition is that the gain could be higher in domains where the agent has to revisit the same sub-spaces frequently and/or which contain large or recurring pockets of dead-end states.

4.5 Proving Safety in Real-time Search

Now we turn to our analysis of SafeRTS. As our third contribution, we prove that it is more efficient to allocate all resources for proving node safety at the end of the planning phase, rather than iteratively within it.

Theorem 14 *Any safety proof that does not pass through the open list requires no expansions.*

Proof: For every node n internal to the Local Search Space, all immediate successors have been discovered. This follows from the fact that an LSS is constructed by expanding nodes on the frontier, at which point those frontier nodes become internal. A safety proof is a sequence of successor nodes, the last of which is a safe node. Any proof that does not pass through the open list must terminate in some successor that has already been expanded. Since the safety of all discovered safe nodes is propagated via a Dijkstra-like backup to all predecessors which are members of the LSS, no expansions are required to discover such a proof. □

Theorem 15 *Any safety proof that passes through the open list requires additional expansions.*

Proof: If a safety proof passes through the open list, that means it is required to examine unexplored nodes which necessitates additional expansions. □

Theorem 16 *For each node* x *internal to the LSS (i.e. already closed) whose safety status cannot be proven without passing through the open list, there exists a node* y *on the open list such that proving the safety of* y *is strictly less difficult than proving the safety of* x.

Proof: If x is not a dead-end *w.l.o.g.*, let $\mathit{proof}^*(x) = \langle x, x_1, \ldots, x_{k-1}, y, \ldots, x_{(k+l-1)}, z \rangle$ where x_i is the ith successor of x, y is a node on the open list, and z is a *safe* node. Note that y is k state transitions away from x, and z is l transitions from y. It is trivial to see that $|\mathit{proof}^*(x)| = k + l + 1$ and $|\mathit{proof}^*(y)| = l$. k and l represent a number of state transitions, which by thair nature must be ≥ 0. Since x is not on the open list, $x \neq y \therefore k > 0$, meaning $|\mathit{proof}^*(x)| > |\mathit{proof}^*(y)|$. Note that in the edge case where $y = z$, meaning $l = 0$, the above statement still holds. □

Theorem 17 *Given a graph* G *with internal node* x *and frontier node* y*, and given* proof(y) ⊂ proof(x)*,* proof(y) *is equivalently impactful as* proof(x).

Proof: Let us say that $\mathit{proof}(x)$ and $\mathit{proof}(y)$ terminate in safe node z. Any proof p that terminates in z will result in the propagation of safety from z extending back to all its predecessors including but not limited to both y and its predecessor x, regardless of the start node in proof p. Therefore, the impact of $\mathit{proof}(y)$ and $\mathit{proof}(x)$ will be identical. □

Theorem 18 *Let* $\mathrm{G_i}$ *be a search graph expanded to* i *nodes, and let* $\mathrm{G_j}$ *be a subsequent search graph expanded from* $\mathrm{G_i}$ *to* j *nodes such that* j > i*. Let* $\mathrm{Pr_i} \subset \mathrm{P}$ *be any set of optimal proofs for all nodes in* $\mathrm{G_i}$ *that can be proven safe. Finding* $\mathrm{Pr_i}|\mathrm{G_j}$ *requires equal or fewer expansions than finding* $\mathrm{Pr_i}|\mathrm{G_i}$.

Proof: We will first address the case where $j = i + 1$, then show that this proof extends to all cases. First, the set of nodes N that are internal nodes of G_i (and hence also G_j) and whose proofs do not pass through G_i's open list require no additional effort by Theorem 14.

Now let us examine the extra node n_j expanded in G_j. Note that since n_j was just added to the search graph, it is on the open list of G_j. There are 3 possibilities:

1. n_j is part of a proof $\mathit{proof}^*(n_k) \in Pr_i$, but n_j is not *explicitly safe*. Theorem 16 proves $|\mathit{proof}^*(n_j)| < |\mathit{proof}^*(n_k)|\ \forall\ k$ where n_k is not on the open list. For any node n_k on the open list whose optimal safety proof passes through n_j, the same principle applies in that $|\mathit{proof}^*(n_k)| > |\mathit{proof}^*(n_j)|$ by at least 1.

2. n_j is part of a proof $proof^*(n_k) \in Pr_i$, and n_j is *safe*. This is a special case of the above where $|proof^*(n_j)| = 0$

3. n_j is not part of any $proof^*(n_k) \in Pr_i$. In this case the expanded G_j has no effect on the effort of computing Pr_i. Note that $n_j \notin G_i$, and therefore computing Pr_i does not require us to find $proof^*(n_j)$.

Clearly, the theorem holds for $i = 1$. Now by induction, any two graphs $G_i, G_j : j > i, i \geq 1$ will display these same characteristics. □

SafeRTS interleaves exploration and safety proofs during its planning phase. As a direct consequence, it attempts safety proofs on nodes that become internal to the LSS by the end of the search iteration. As shown in Theorem 18, it would be equally or less difficult to achieve the same or better safety coverage by doing safety proofs after the LSS expansions. SafeRTS has an anytime behavior but does not effectively utilize the real-time bound given.

4.6 A Real-time Framework for Safety

Given this analysis of SafeRTS, we now introduce as our fourth contribution a general scheme called Real-time Framework for Safety (RTFS). This framework, shown in Algorithm 4, composes an algorithm from four elements: a parameter that determines the ratio between goal and safety oriented search, and three main functions: EXPLORE, ALLOCATEPROOF, and SELECTTARGET. First, we formalize our notion of these functions.

Definition 17 *An exploration strategy is a function* EXPLORE $: s_{root}, budget \rightarrow G$ *that constructs a local search space given a root state and an exploration budget, returning a graph G representing the Local Search Space.*

Definition 18 *A safe target selection strategy is a function* SELECTTARGET $: G \rightarrow \langle n_1, n_2, \ldots, n_k \rangle$ *that defines a natural ordering for a set of nodes N of size k structured as a Local Search Space graph G.*

Note that when used to retrieve a single target n_{target}, the first node of the ordering is returned.

Algorithm 4: Real-time Framework for Safety

```
Input: $s_{root}$, $iterationBound$, $explorationRatio < 1$
1  $bound \leftarrow iterationBound$
2  while $s_{root} \neq s_{goal}$ do
3      $explorationBudget \leftarrow bound * explorationRatio$
4      $safetyBudget \leftarrow bound - explorationBudget$
5      $lss \leftarrow$ EXPLORE($s_{root}$, $explorationBudget$)
6      ALLOCATEPROOF(SELECTTARGET, $lss$, $safetyBudget$)
7      PROPAGATEH($lss$)
8      PROPAGATEDEADENDS($lss$)
9      PROPAGATESAFETY($lss$)
10     $s_{target} \leftarrow$ SELECTTARGET($lss$)
11     if $s_{target}$ is null then
12         TERMINATE ◁ no safe path is available
13     move the agent along the path from $s_{root}$ to $s_{target}$
14     $s_{root} \leftarrow s_{target}$
15     $bound \leftarrow iterationBound + unusedBudget$
```

Definition 19 *A safety proof allocation strategy is a function* $\textsc{allocateProof} : \textsc{selectTarget}, G, budget \mapsto \perp$ *that allocates resources to proving the safety of nodes. It takes a safety target selection strategy, a search graph G, and a budget of time to allocate.*

Such a function is free to allocate resources in any way it chooses, but the selectTarget parameter allows it to prioritize proof effort on nodes that will be chosen as the target to which the agent will commit.

RTFS exploits the real-time bound of the problem to pre-allocate the time spent on exploration and safety proofs [line 3, 4]. RTFS takes the *explorationRatio* as an input parameter. A higher value allows for more aggressive exploration, but decreases the likelihood of completing any safety proofs. The appropriate value should reflect the total available time per iteration and the difficulty of proving that a node is safe in a given domain.

The explore function defines the way the algorithm uses the expansion budget to build the local search space [line 5]. A trivial example of such function is A*, but any exploration method that is capable of constructing a search tree could be used, such as wA* (Pohl, 1970; Rivera, Baier, & Hernández, 2015), GBFS (Pearl, 1984), Beam search (Russell & Norvig, 2010), and Speedy (Burns, Ruml, & Do, 2013). Using an exploration method that leads to a narrow and deep tree makes each safety proof more consequential as upon a successful proof every ancestor of the node will become safe.

Given a search tree, a selectTarget function, and an expansion budget, the allocateProof function distributes the given budget among the frontier nodes of the tree to prove their safety [line 6]. Using selectTarget, allocateProof can allocate resources based on the ordering provided. This function is highly non-trivial.

The learning function in line 7 is the same as that of LSS-LRTA*. The dead-end propagation function in line 8 removes all nodes from the local search tree that were found to be unsafe or whose successors are all unsafe. Lastly, similar to SafeRTS the safety propagation function in line 9 marks as safe every node with a safe successor as discovered during exploration or as proven safe by allocateProof.

Given a search tree in which the safe and dead-end nodes are marked, the selectTarget function, in line 10, selects a node which the agent should commit towards. The authors of SafeRTS described multiple examples for such target selection strategies and claimed best results with the previously described safe-towards-best.

The invocation of these functions might require less time than the given bound for the iteration. The unused time is used towards the next planning iteration as shown in line 15. Alternatively, in domains that do not allow such budget transfer, the EXPLORE and ALLOCATEPROOF functions are called with remaining budget distributed between them using the original ratio.

The structure of RTFS is designed to utilize the property proven in Theorem 18. The full LSS is constructed before any effort is spent on proving safety, which efficiently allocates available time such that it is more likely to prove more promising nodes than SafeRTS, as we will see below.

4.7 Empirical Evaluation

To ascertain the performance gain of RTFS, we create a configuration RTFS-0 with *target selection* and *safety proof allocation* functions that mimic SafeRTS. SafeRTS allocates at least 50% of its expansions towards the construction of the LSS, hence we set the *exploration ratio* of RTFS-0 to 0.5. Though both algorithms select the node on the open list with the lowest f value at the time the proof is attempted, RTFS-0 differs from SafeRTS as it only attempts safety proofs after expanding the LSS. As such, its proofs are only limited by the iteration bound. If a target node is proven to be an implicit dead-end, RTFS-0 removes it and all other discovered dead-end nodes from the graph, then attempts to prove the next best node on the open list.

In our experiments we include two additional oracles, A* and Safe-LSS-LRTA*, to provide reference points. A* is executed offline and its execution time is not included in its GAT. This serves as a lower bound on the GAT and an upper bound on the velocity. Safe-LSS-LRTA* is a version of LSS-LRTA* that has access to an ideal dead-end detector, thus it only considers nodes that are safe. This offers the behavior of an agent-centered real-time search method that only has to focus on reaching the goal.

To evaluate the performance of our methods, we test them on the racetrack and traffic domains used in Chapter 2, along with our new Airspace domain. As a reminder, in racetrack, a derivative of

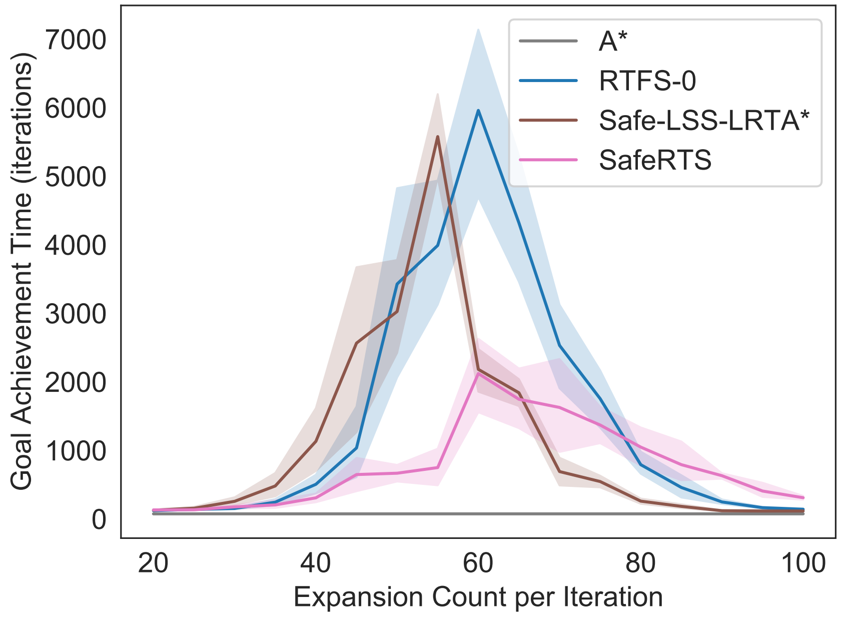


Figure 4-4: GAT on the Hansen–Barto racetrack.

the benchmark of Barto et al. (1995), an agent with inertia and limited acceleration traverses a grid. The agent's state is $\langle x, y, \dot{x}, \dot{y} \rangle$. Dead-ends are reached if the agent collides with any blocked cell. Notably, the heuristic in this domain encourages the agent to achieve high velocity, as that produces the lowest estimate of GAT. 10 randomly sampled start positions were used for both instances we tested on. In the traffic domain, an extension of the domain used by Matters (), a agent moves in a grid, avoiding moving obstacles. A dead-end is reached if an obstacle collides with the agent before it reaches a goal state. In the racetrack and Airspace domains, the planners select one action per planning iteration, while in traffic the planners commit to multiple actions to match the results presented in Chapter 2. All tested planners were able to successfully avoid dead-ends in all benchmark instances, thus our results solely focus on performance indicators such as GAT and velocity.

First, we turn to the comparison of SafeRTS and RTFS-0. While both SafeRTS and RTFS-0 solved all instances of the traffic domain, the results are non-conclusive and stochastic. The GAT of both algorithms is highly fluctuating with no algorithm dominating the other.

Figure 4-4 shows the GAT on the Hansen-Barto racetrack domain (Chapter 2). All algorithms

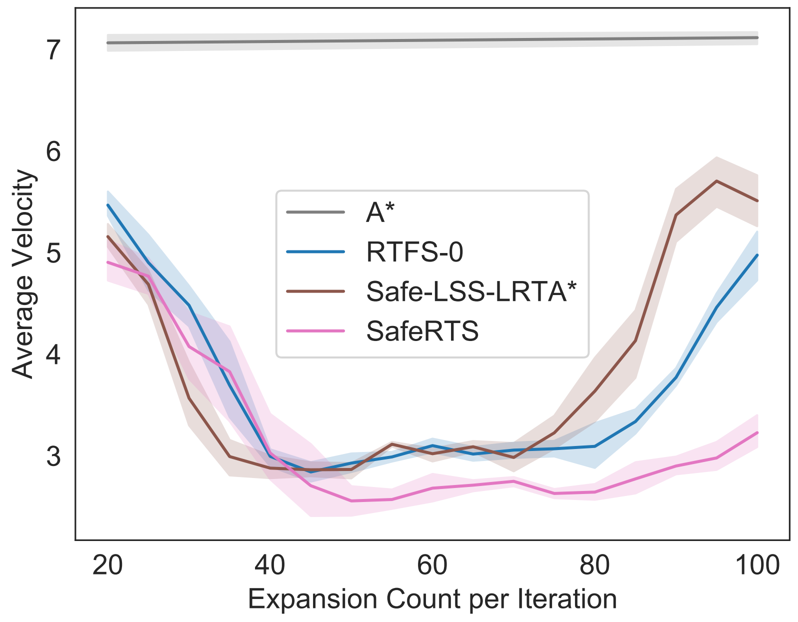
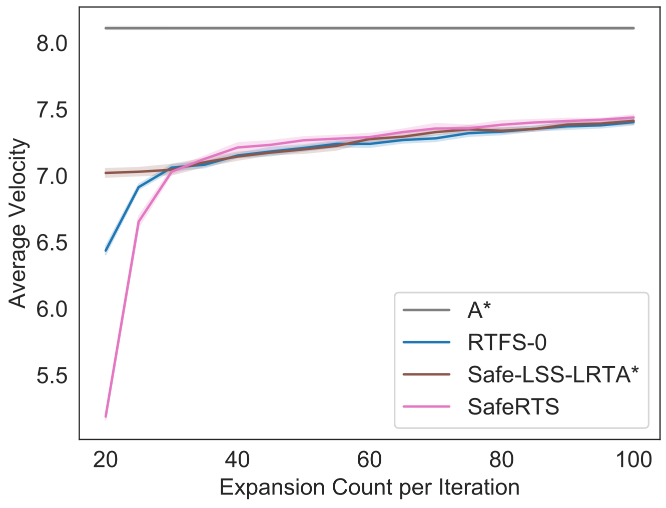
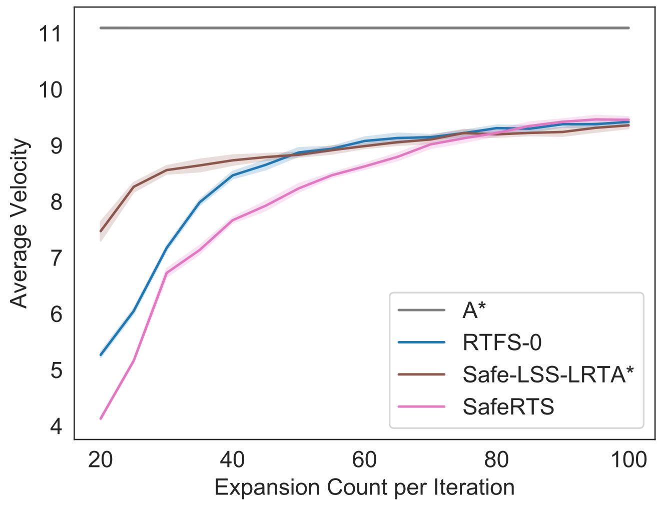


Figure 4-5: Average velocities on the Hansen–Barto racetrack.

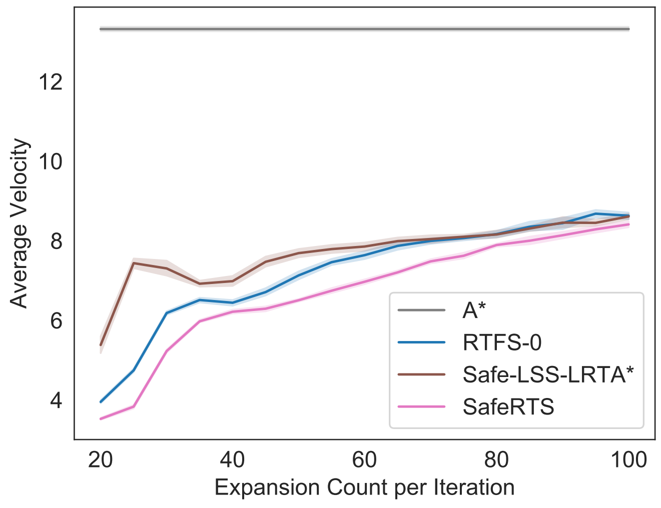
perform close to optimal with small action durations, however increasing the action duration drastically decreases the performance of all algorithms. The oracle-based Safe-LSS-LRTA* becomes 50 times slower when more planning time is provided. We speculate that this is a result of actions with long term effects that lead to significant scrubbing (Sturtevant & Bulitko, 2016). Empirically, the GAT in Figure 4-4 imply that the agent revisited states many times before finding the path to the goal. The GAT of SafeRTS is superior to RTFS-0 and to Safe-LSS-LRTA* for low to medium lookaheads, which suggests that SafeRTS is simply a better heuristic for this particular domain. Figure 4-5 shows the average velocity of the agent, computed as the total distance travelled divided by the travel time. The velocity of RTFS-0 is higher than SafeRTS's and similar to that of the real-time oracle. Results on the uniform racetrack instance were similar thus those results are omitted. Both SafeRTS and RTFS-0 use the safe towards best target selection strategy. Ultimately, this strategy is intended to commit the agent towards the top node on open if safety can be inferred. RTFS-0 managed to pick the top node in every single experiment on the Hansen-Barto racetrack, while SafeRTS selected targets of descendants of nodes with the average position on the open list of 7.2 at action duration 20. Yet even though SafeRTS falls far short of selecting the top node, it performs better in this domain,



(a)



(b)



(c)

Figure 4-6: Average velocity on the Airspace domain of length 100,000 with altitude limits of 10 (a), 14 (b), and 20 (c).

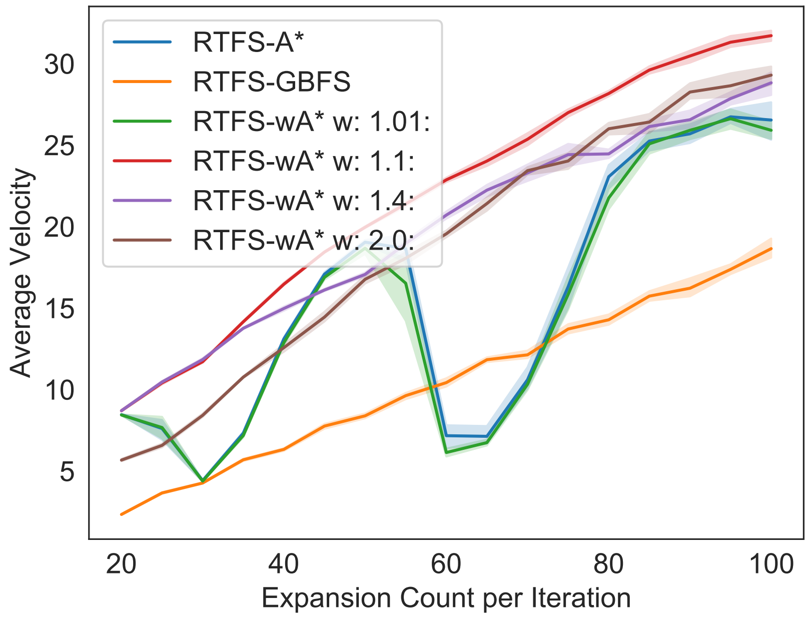


Figure 4-7: Average velocity of different exploration function RTFS variants on the Airspace domain of length 100,000 ($p_{obs} = 0.01$)

which implies to us that this domain is not suitable for evaluateing the performance of safety oriented methods.

To further evaluate the performance of RTFS-0 and SafeRTS, We created 10 random Airspace instances with a horizontal distance of 100,000, maximum altitudes of 10, 14, 20, and $p_{obs} = 0.05$. In this domain we only focus on the average horizontal velocity as it directly corresponds to the GAT.

Finding a solution in Airspace is trivial as it can be completed by navigating the agent at the obstacle free altitude 1. Finding a good solution is increasingly more difficult as the altitude limit increases. Figure 4-6 shows the convergence of methods towards the oracle real-time search, and the average velocity shows a clear increasing trend as the time available per iteration increases. The convergence of SafeRTS and RTFS-0 slows down as the difficulty of the problem increases (Figure 4-6c). RTFS-0 has faster average velocity and it closes the gap faster than SafeRTS.

We demonstrate the flexibility of RTFS by instantiating variants of it using two of the 4 degrees of freedom and evaluate them on 5 Airspace domains instances with a less dense obstacle probability of ($p_{obs} = 0.05$). The upper velocity bound achievable by A* is 70. We investigate the effect of

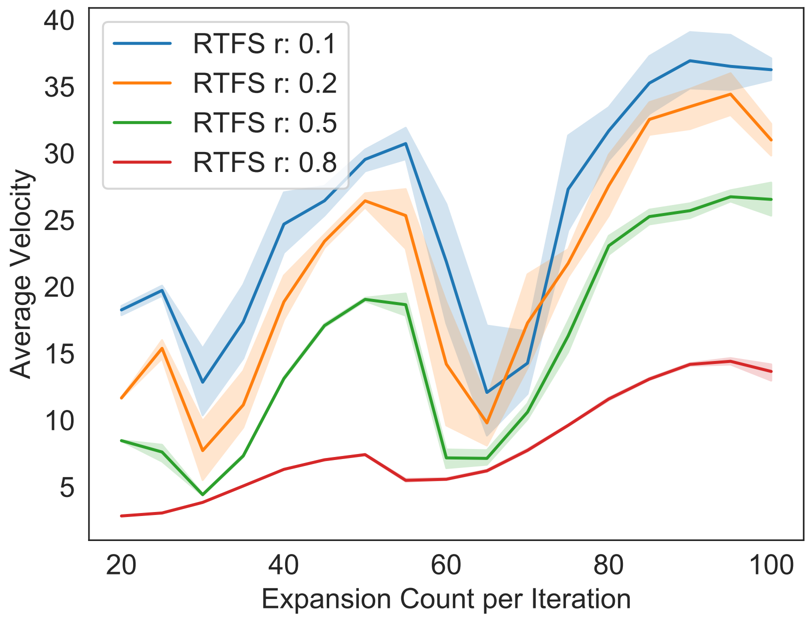


Figure 4-8: Average velocity of different exploration ratio RTFS variants on the Airspace domain of length 100,000 ($p_{obs} = 0.01$)

different exploration functions(A*, Weighted-A*, and Greedy Best First Search (GBFS)) as well as the impact of different *explorationRatio*s. Deviating from a locally optimal A* exploration of RTFS–0 can not only improve the average velocity but can reduce the variance of the outcome as shown on Figure 4-7. In this context, RTFS–0 is denoted as RTFS-A*. The performance of RTFS-A* plummets periodically as the size of the search frontier and the available expansions are aligned in a way that a low performing node is selected. This is likely a consequence of selecting nodes from incomplete f layers (Kiesel et al., 2015). Weighted A* seems to break this alignment by making the search tree deeper. Additionally, the overall performance improved up to 20% when a weight of 1.1 is used. Further increases in the weight reduced the average velocity, converging the performance of GBFS, which achieved the lowest average velocity in our experiments.

In addition to the exploration function, we tested a range of *explorationRatio*s, that determines how much time should be spent on exploration and safety proofs in each iteration. *explorationRatio* = 0.1, or $r = 0.1$ for short, means that 10% of the time is spent on expansions. Figure 4-8 shows that decreasing the exploration time increased the performance as higher altitudes require longer and

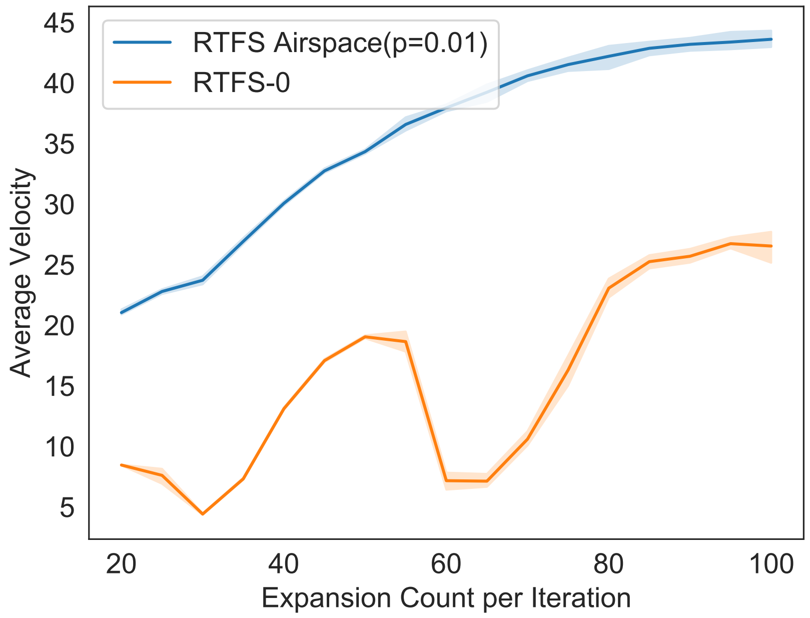


Figure 4-9: Average velocity of RTFS-0 and its domain fitted variant on the Airspace domain of length 100,000 ($p_{obs} = 0.01$)

more difficult safety proofs. However, it also amplified the fluctuations discussed above.

Lastly, the performance of RTFS composed from the wA* ($w = 1.1$) exploration function and an *explorationRatio* of 0.1 is shown on Figure 4-9. The remaining function are the same as in RTFS-0. While the union of these modifications increases the performance by 100%, it only demonstrates the flexibility of RTFS and it is not intended to serve as a general recommendation of this particular configuration.

4.8 Discussion

While the above results are encouraging, it is important to note that simplistic target selection and safety allocation strategies were used in RTFS with the intention of matching SafeRTS for better comparison. The topic of selecting the node to commit to is an issue fundamental to online real-time planning, and a deep investigation is outside the scope of this chapter, beyond ensuring that the node selected is *safe*. Resource allocation for safety is similar to the problem of a parallel portfolio of algorithms: we may choose any of a number of promising frontier nodes on which to attempt a proof. The safety allocation problem may have additional constraints in that we prefer to prove nodes based on the ordering provided by the target selection strategy, but it is not always clear when a proof of a node should be abandoned or not attempted in the first place in favor of some other promising node which may be easier to prove. Learning based methods have been proposed to address these settings (O'Ceallaigh & Ruml, 2015; Cserna et al., 2017; Petrik & Zilberstein, 2006).

4.9 Conclusion

This chapter has four contributions. First, we introduced a new domain with dead-ends called Airspace that minimizes the long term effects of actions and was designed specifically to evaluate safe real-time methods. Second, we showed that the simple method of caching dead-ends provided a mild performance improvement. Third, we proved that proving safety is more effective when it is done after expanding the local state space. Lastly, we combined these finding into a flexible planning framework, RTFS, to address real-time planning in the presence of dead-ends. We demonstrated that, when configured like SafeRTS, RTFS provides improved performance. Unlike SafeRTS, RTFS can be tuned to each domain to achieve higher performance, and thus is a flexible model for safe real-time search.

CHAPTER 5

Real-time Planning in the Presence of Local Minima

This chapter concerns state spaces with large local minima, in which agent centered real-time heuristic search algorithms potentially exhibit scrubbing which negatively impacts their performance (Sturtevant & Bulitko, 2016). In this chapter, we address state spaces that are deterministic, fully-known, and undirected, allowing us to take a non-agent-centered approach in which each planning iteration continues to expand a growing search frontier, avoiding state re-expansions. Our approach differs from the leading algorithm for this setting, Time-bounded A* (TBA*), in that it carefully structures the explored space, building a graph that provides the agent with efficient navigation in large problems. We prove the completeness of our approach and provide experimental results suggesting that it performs comparably to TBA* in easy problems, and better in hard ones.

5.1 Introduction

In real-time heuristic search, the planner must return the next action for the agent to take within a pre-specified time bound. While the agent is executing the action, the planner can consider the agent's next move. In this way, real-time heuristic search models the integration of incremental plan synthesis and plan execution. As in the previous chapters, the objective is to minimize the agent's goal achievement time (GAT): the wall clock time from when a goal is presented until it is achieved. Applications include controlling autonomous vehicles, robots, and characters in video games. Many algorithms have been proposed for this setting, starting with the seminal work of Korf (1990) (such as LSS-LRTA* (Koenig & Sun, 2008), RTAA* (Koenig & Likhachev, 2006), TBA* (Björnsson, Bulitko, & Sturtevant, 2009), daRTAA* (Hernández & Baier, 2012), and SafeRTS (Cserna et al., 2018)).

Many real-time algorithms are agent-centered, meaning that they repeatedly conduct a search of the portion of the state space that is centered immediately around the agent. This is known as the local search space (LSS) (Koenig, 2001). These algorithms have 3 phases: exploration, during which a local search space is constructed by expanding states, learning, where the heuristic values of the nodes on the LSS frontier and the costs to reach them are used to update the h values of the states in the LSS, and commitment, when the agent commits to actions that take it toward the best node on the LSS frontier.

After each planning iteration, the LSS is typically discarded (Korf, 1990; Koenig & Sun, 2008) and a new one is constructed during the next exploration phase, starting at the agent's new state. While this approach is adequate for domains with shallow local minima in the heuristic function such as grid worlds with sparse or randomly distributed obstacles, in domains that have local minima much larger than the LSS, agent-centered algorithms can revisit states many times in order to update their h values sufficiently to escape. This scrubbing behavior occurs when the size of the LSS is smaller than the size of the local minima. In every iteration, the h values of the nodes captured by the LSS is increased to match the h level at the edge of the LSS. This learning process is slow when the LSS does not capture the edges of the minima. The agent can only leave the local minima when the heuristic values inside surpass the values outside. Sturtevant and Bulitko (2016) prove that LSS-style algorithms must, under certain conditions, perform more state visits than there are states in the state space.

Time-Bounded A* (TBA*) (Björnsson et al., 2009) represents an alternative to agent-centered methods. It constructs a single A* search tree, with each planning iteration picking up where the previous left off. To select an action for execution, it traces from the target node on the search frontier back to the root, following the parent pointers of the search tree. If the agent is on this trace, it steps forward toward the frontier. Otherwise, it backtracks toward the root, following the parent pointer of its current state. Eventually, the agent will intersect the trace and move toward the target.

Because each state is expanded at most once, TBA* performs much better than agent-centered algorithms in difficult problems with large heuristic minima. However, its monolithic search tree leads to two significant drawbacks. First, the tree is anchored to the original start state, which becomes increasingly more irrelevant as the agent moves and results in the expansion of states of low interest. Second, the agent can only follow the tree gradient (the direction of the parent pointers or their opposite) which does not allow for lateral navigation in the search tree.

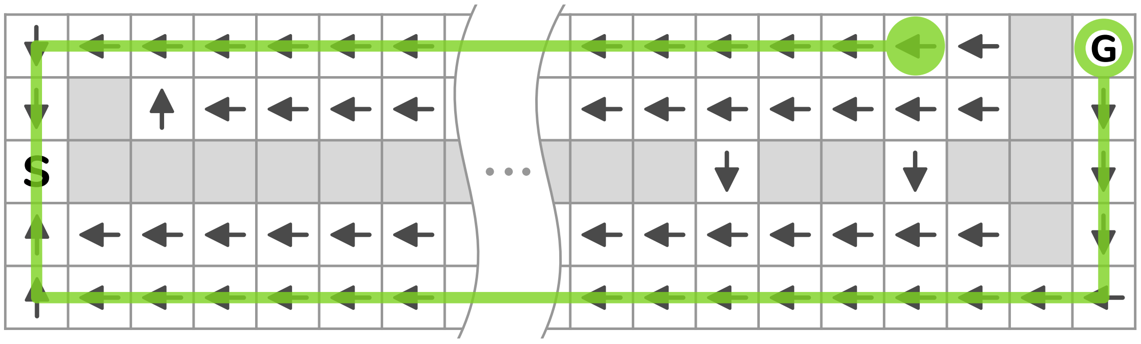
In this chapter, we propose an approach to non-agent-centered real-time heuristic search that addresses both of these concerns by representing more detail in the structure of the explored state space. During the exploration phase, Cluster Real-time Search (Cluster-RTS) incrementally groups the expanded nodes into a graph of small search trees, called *clusters*. The cluster graph can then be used to more efficiently navigate the agent toward promising nodes on the search frontier.

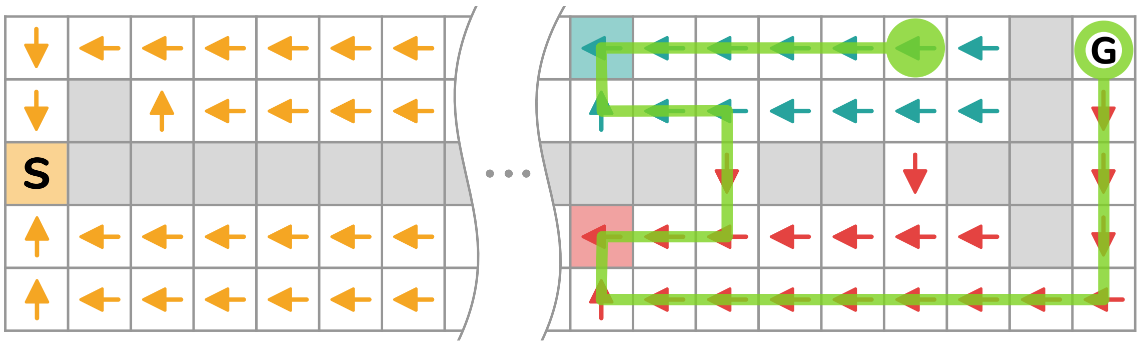
Figure 5-1 illustrates the intuition using two simple scenarios. An S marks the initial state, a G the goal state, and the filled circle is the agent's current location. In TBA*, the agent backtracks all the way (Figure 5-1a) or almost all the way (Figure 5-1c) back to the root of the search tree before intersecting with the path traced back from the goal (arrows indicate parent pointers). In Cluster-RTS (Figure 5-1b and Figure 5-1d), the agent can move between cluster seeds (filled cells, colors represent the associated clusters) to take a more efficient path to the goal.

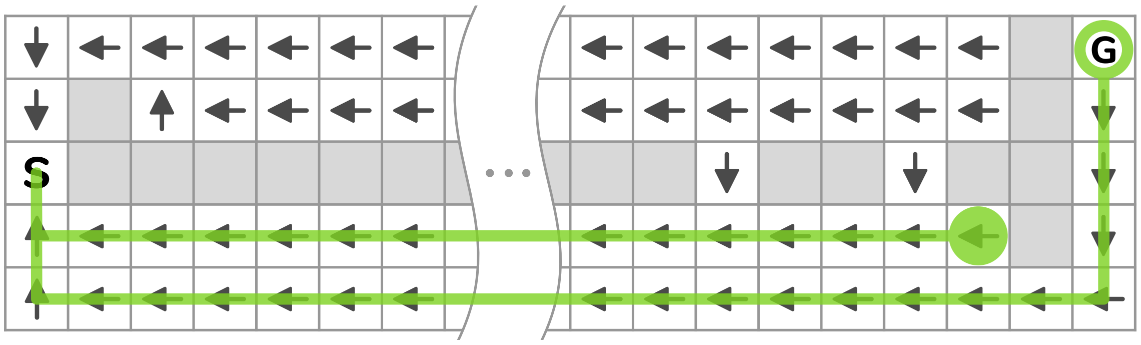
In the remainder of the chapter, we flesh out these ideas in more detail and evaluate their behavior both theoretically and empirically. We find that the running time of TBA*'s backtracing step can be linear in the number of expanded nodes, while that of Cluster-RTS is bounded by the log of the number of expanded nodes. Experimentally, we identify three factors that influence the performance of TBA*. Our results suggest that Cluster-RTS is competitive with TBA* on small problems while surpassing its performance on hard ones.

5.2 Time-Bounded A*

Pseudocode for TBA* appears in Algorithm 5. The algorithm takes as parameters *expansionLimit* and *tracebackLimit*, two fractions between 0 and 1 such that *expansionLimit* + *tracebackLimit* = 1. These fractions define the portion of time within each real-time planning iteration that will be spent on state expansions versus tracebacks (described below) respectively. The original work suggests values of 0.9 for *expansionLimit* and 0.1 for *tracebackLimit*, so that is what is used for our experimental analysis.


(a)


(b)


(c)

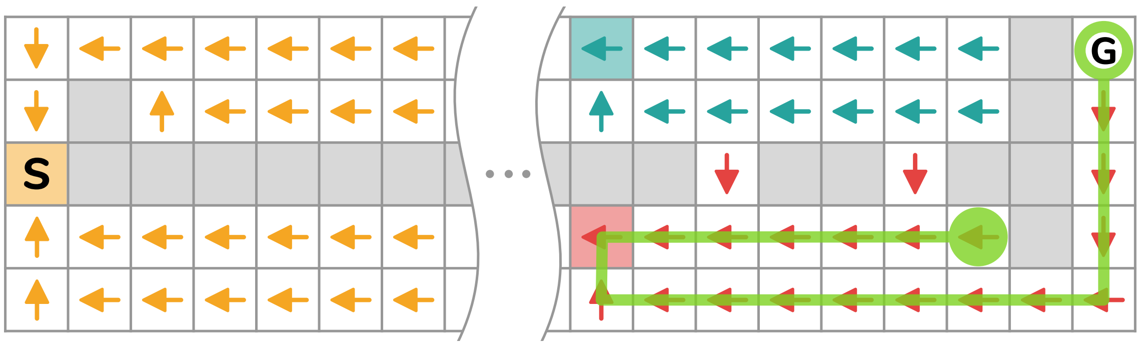

(d)

Figure 5-1: The agent must retreat to the initial state in TBA* (a, c) but can move between clusters in Cluster-RTS (b, d).

Algorithm 5: TBA*

```
input : $expansionLimit, tracebackLimit$
1  $OPEN \leftarrow \{root\}$
2  $loc \leftarrow root$
3  Main while $loc \neq goalState$ do
4      A*$OPEN$ until $expansionLimit$ is exhausted
5      if $goal\ node\ not\ traced$ then
6          if $no\ trace\ in\ progress$ then
7              $target \leftarrow$ best node $\in OPEN$
8              $trace \leftarrow target$
9          while $trace \notin \{loc, root\}$ do
10             $trace \leftarrow trace.parent$
11             if $tracebackLimit\ reached$ then
12                 break
13         if $trace \in \{loc, root\}$ then
14             $agentTarget \leftarrow target$
15     if $loc\ is\ on\ path\ to\ agentTarget$ then
16         $loc \leftarrow loc.next$
17     else
18         $loc \leftarrow loc.parent$
```

TBA* begins precisely as A*: seeding the open list with the root state [line 1] and expanding nodes sorted on f [line 4]. The A* search is paused when *expansionLimit* is reached. Then, the algorithm performs its "Traceback" phase. If no traceback is in progress, TBA* selects the node from the open list with the lowest f value and sets it as the *target* [line 6]. Using parent pointers, it traces back until one of 3 criteria are met: the trace reaches the root state, the trace reaches the agent's location, or the *tracebackLimit* is reached [line 9]. If a traceback has already begun in the previous iteration but not terminated in the root state or the agent's location, TBA* picks up where that trace left off, terminating on the same criteria. Once a traceback terminates at the root or the agent's location, *target* is set as the agent's target [line 14]. The agent moves based on a simple decision: if it is on the path to its current target, continue on that path. If not, move to the previous state on its current path. The movement strategy requires the state space to be undirected as in a directed state space it might not be possible to step back to the parent state if no action is available that would invert the effects of the action that lead to the current state from the parent state.

Björnsson et al. (2009) describe two optimizations to the basic algorithm:

1. After the goal has been found, the shortcutting optimization devotes a limited number of expansions to finding the optimal path from the agent's current state to the traceback path by conducting an A* search to the agent's location. The open list is initialized to all states on the traceback path from the goal to its intersection with the the agent's path from the root. If a shortcut can be found, it precludes the need for the agent to backtrack along its current path. They conclude that this optimization has insignificant effects on empirical performance, as only in about 5% of cases was the agent on an alternate path when the goal was found, and in those cases beneficial shortcuts were rarely discovered. And since the A* search is restarted after every iteration, this approach does not scale to larger domains as finding such shortcut is as difficult as finding a goal from the initial state.

2. The threshold optimization prevents the agent from choosing a new path unless the g value of the path's *target* is at least as high as the g value of the agent's current path. This prevents the agent from switching rapidly between paths that are roughly as good. This optimization was shown to produce 2% better paths in certain circumstances, and so we include it in our

implementation.

5.2.1 Time-Bounded Weighted A*

Hernández, Asín, and Baier (2014) extend TBA* by replacing the A* search with a suboptimal variant, yielding Time-Bounded Weighted A* (TBwA*) and Time-Bounded Greedy Best-First Search (TB-GBFS). They prove an upper bound on the solution cost of TBwA* and show empirically that as its weight increases, total solution cost *decreases* in many domains, especially with smaller time bounds. This is due to Weighted A* finding the goal faster than A*, albeit via a suboptimal path (but see also Wilt and Ruml (2012)). We will confirm below that TBwA* avoids backtracking more effectively TBA*. The agent's target path before a goal is discovered is likely to be extended first as long as the heuristic continues to decrease. These improvements make TBA* the leading non-agent-centered real-time search algorithm for undirected state spaces such as grid pathfinding.

5.2.2 Behavior of TBA*

As a simulation of A*, TBA* is complete. However, TBA* anchors the root of the search tree at the initial state at the start of problem-solving, despite the fact that the agent must commit to actions leading it away from initial state as the search continues. TBA* has no mechanism to guide the search based on the current location of the agent, and therefore the practical effectiveness of the algorithm depends on the assumption that an initially promising path will ultimately be on or near the discovered path to a goal. Since there can be no guarantee that this assumption is correct, it may prove costly if the goal is discovered via an alternate path.

The structure of the TBA* tree compounds this issue by discovering only paths that connect back to the root state with no inter-path connections. In the worst case, TBA* will direct the agent all the way back to the root state in order to switch to a new target path. No strategy has been proven effective for connecting divergent paths. Instead, the improvements presented by TBwA* and TB-GBFS pull the agent forward through the state-space with the aim of discovering the goal more quickly.

Theorem 19 *TBA* eventually provides optimal goal achievement time as the time bound per itera-*

tion increases.

Proof: If the action duration is long enough that the first A* search finds an optimal path to a goal, TBA* will execute only those actions. □

Theorem 20 *TBwA* and TB-GBFS are inherently suboptimal regardless of the time bound.*

Proof: The underlying search tree of TBwA* weighted A* can be suboptimal even if it is precomputed, thus even with an infinite time bound, optimality cannot be guaranteed. □

Theorem 21 *The worst case complexity of TBA* backtracing and short-cutting increases linearly with the domain size.*

Proof: Finding the shortest path between the agent and the current best node on the open-list might require the expansion of all states previously discovered by the search. Consider a domain where the root has two children A and B and every other node has only one successor. Let's assume all expansion happens under A and the agent is a constant number of nodes α away from end of the A node chain. When the first node is expanded under B, finding the shortest path between the agent and the node expanded under B requires tracing back through all nodes that were previously expanded minus α. □

5.3 Cluster Real-time Search

Cluster Real-time Search (Cluster-RTS) adheres to the same essential search phases as TBA*: exploration, backtracing, and commitment. Cluster-RTS explores the state space by incrementally constructing a graph of connected clusters. The cluster graph is used to facilitate finding a path from the agent to the target node on the frontier during backtracing.

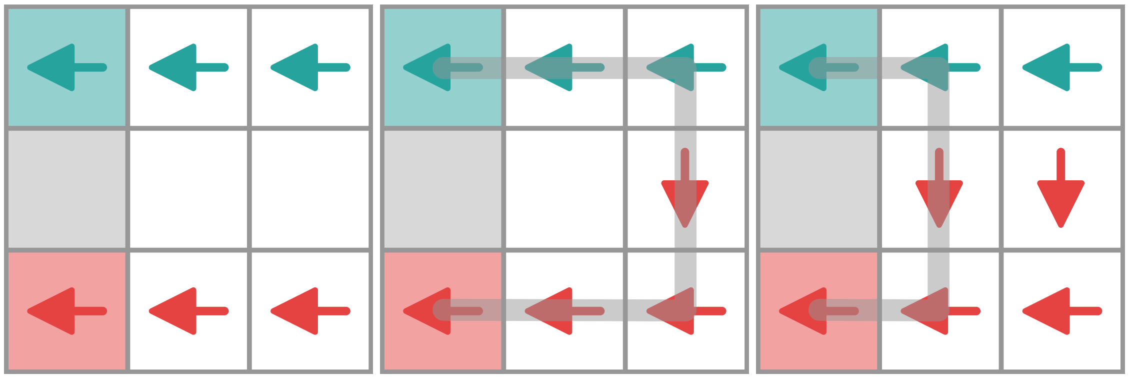
5.3.1 Exploration

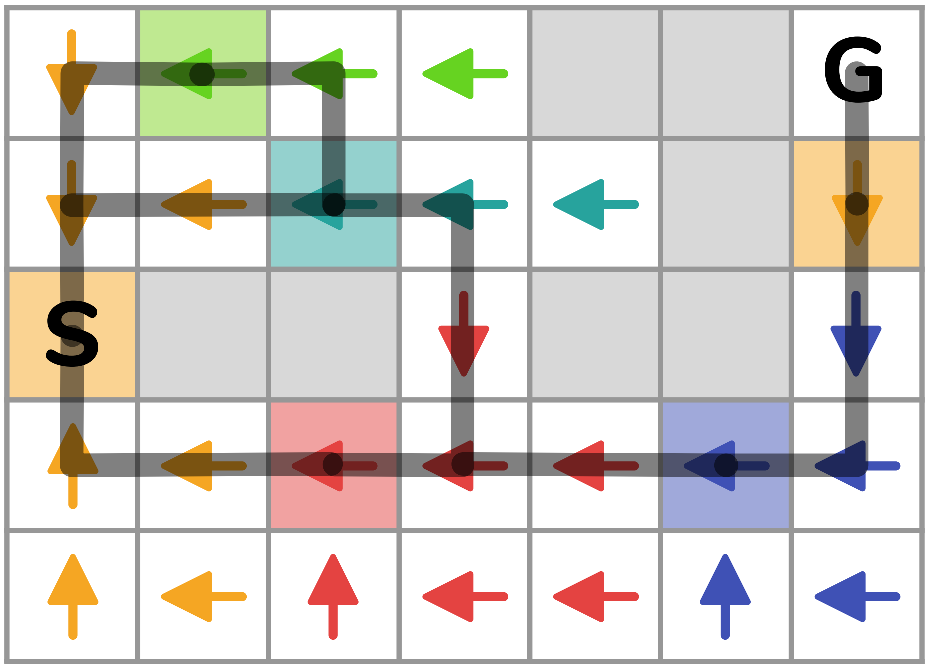
A *cluster* is a set of nodes in a tree structure rooted at a node denoted as the *cluster seed*. The cluster size is limited by an upper bound on the distance from the cluster seed, which we call *depth*.

Each cluster has its own open list sorted on increasing $f = g + h$, where g is calculated from the cluster seed and h is the 'regular' global heuristic function. Clusters themselves are organized into a minimum priority queue sorted on the h value of the first node in each cluster's open list. To expand a node, Cluster-RTS selects the first cluster in the queue and then expands the first node in its open list. A newly generated node is added to its parent's cluster unless this would violate the depth bound, in which case a new cluster is created with the node as its seed. Each cluster behaves as a mini self-contained A* search rooted at the seed, so the cluster's search tree stores the optimal path from each node within the cluster back to the seed. The two-tier strategy results in exploration that is locally optimal and globally greedy. Every node in the search space belongs to exactly one cluster. As new clusters are created, the cluster depth limit is increased by a constant factor, thus, the sizes of the clusters grow exponentially.

The clusters are represented by vertices in the cluster graph. Cluster-RTS constructs the cluster graph's edges in two ways: first, when a node is expanded and creates a new cluster, and second, when a cluster attempts to expand a node that is already claimed by another cluster. In both cases, a *cluster edge* is constructed that stores the two adjacent nodes and the shortest discovered distance between the two cluster's seeds. Calculating the length of the shortest path between two adjacent cluster seeds is a constant time operation as the nodes on both sides of the connection store the g value from their cluster's seed. Only the lowest cost connection is kept between two clusters. Figure 5-2 illustrates, with filled cells representing cluster seeds and arrows representing parent pointers. In b), after the red cluster tried to expand a state that is already part of the teal cluster, the first connection is created. In c), the previous connection is removed in the favor of a new shorter connection. Panel d) shows a fully clustered space with the final connections forming the cluster graph.

Note that although the adjacent nodes that make up a cluster edge each store the optimal path to their respective clusters' seeds, the path connecting the two seeds through this edge is potentially sub-optimal.


(a) (b) (c)



(d)

Figure 5-2: Connecting clusters into a graph.

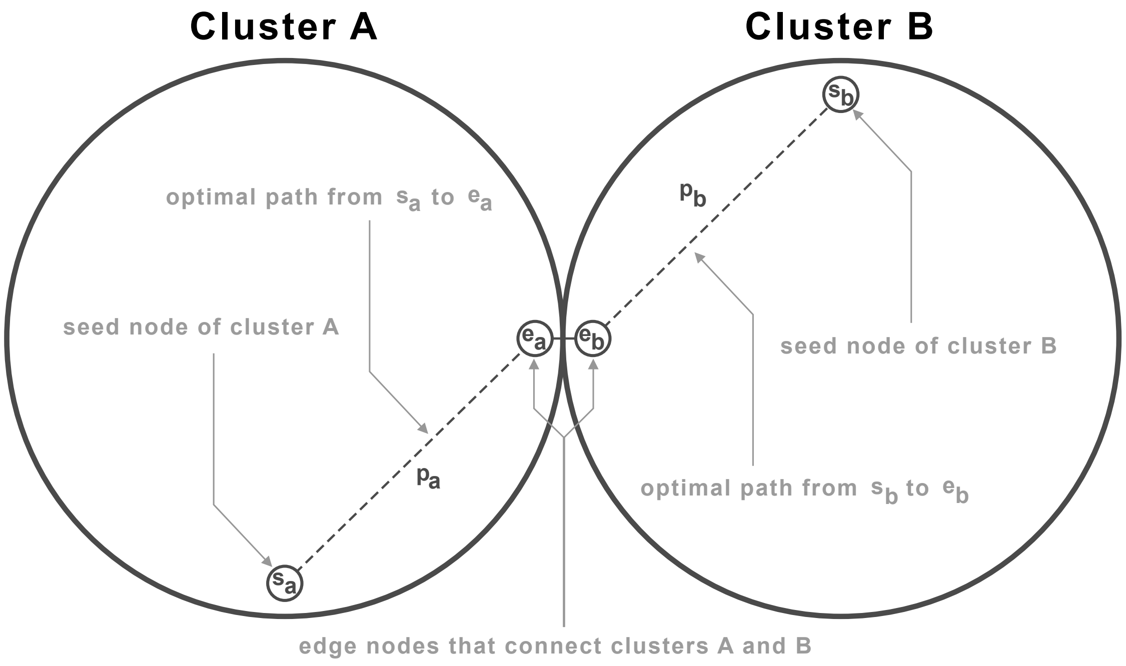

Cluster A
Cluster B
optimal path from s_a to e_a
seed node of cluster A
p_a
s_a
e_a
e_b
s_b
p_b
seed node of cluster B
optimal path from s_b to e_b
edge nodes that connect clusters A and B

Figure 5-3: Abstract cluster edge connection.

5.3.2 Path extraction

As in TBA*, because the search envelope is expanded independently from the agents current location, the agent has to be connected somehow with its intended target: the best node on the search frontier or the goal, once discovered. To connect the agent with the target, a backward search is initiated in the cluster graph from the target until the agent is found. The backward search identifies and stores the sequence of clusters and cluster edges the agent should traverse in order to reach the target. This sequence is then unpacked into a sequence of actions through these clusters that the agent should execute. The action sequence is cached until a new backward search is initiated. Figure 5-3 shows the cluster edge connection between two adjacent clusters. A cluster edge stores the two clusters connected A and B and the edge nodes on both side of the connection e_a and e_b. Each cluster is a locally optimal search tree, thus given any edge node (e.g. e_a) the optimal path (e.g. p_a) can be extracted between that node and the cluster seed (e.g. s_a). Thus unpacking a cluster edge connecting the clusters A and B (on Figure 5-3) into an action sequence would add the following subsequences:

1. Actions of p_a that connect seed node s_a with edge node e_a.

2. The action that leads to edge node of cluster B e_b from edge node of cluster A e_a.

3. The reverse actions of p_b from edge node e_b to seed node s_b.

The last segment of the sequence has to be reverted as the actions recorded in the search tree lead outwards from the root node not inwards.

To reduce the search time on the cluster graph, in subsequent iterations the backward search only continues until the agent or the currently cached path is reached. If the target node is in the same cluster as the agent's current state, the backward search immediately terminates. At the end of each search iteration the agent follows the latest cached path.

5.3.3 Behavior of Cluster-RTS

Theorem 22 *Cluster-RTS is complete in finite, undirected domains.*

Proof: The node expansion order differs from A* in Cluster-RTS. However, it only expands every node once, thus it eventually finds the goal in finite state spaces. Once the goal is found a path will be extracted from the cluster graph that guides the agent to the goal. This path always exists as undirected domains have no dead-ends and the explored state space contains the agent. □

Theorem 23 *The number of clusters is bounded by the log of the number of expanded nodes. Given:* $\mathit{growthFactor} > 1$

Proof: To generate k clusters the cluster depth limit should be reached k times. The minimum number of nodes required for this occurs when each cluster contains only one node except the first cluster that is pushing the limit. The initial cluster (that is responsible for the limit expansion) has a size larger than or equal to $\mathit{initialClusterSize} * \mathit{growthFactor}^k$ and smaller $\mathit{initialClusterSize} * \mathit{growthFactor}^{k+1}$ than after k depth limit increases. Thus the total number of nodes required to create k clusters is at least $\mathit{initialClusterSize} * \mathit{growthFactor}^k + (k - 1)$. □

Theorem 24 *The number of clusters "expanded" by the back-tracing search that connects the agent with the search frontier is bounded by the log of the number of expanded nodes.*

Proof: This is a direct consequence of Theorem 23 as each cluster is expanded at most once. □

In contrast, for TBA* and its variants, this relationship is linear since it might touch all discovered nodes during the back-tracing phase (Theorem 21).

Theorem 25 *The longest path through the cluster with depth d is* $2 * d - 1$ *and the number of nodes in a cluster with such path is at least* $2 * d - 1$.

Proof: Each path leading through a cluster touches the seed of the cluster. Thus the length of the path is the sum of the lengths of two path segments; one that is coming from an edge of the cluster to the seed and a second that is going from the seed to another edge of the cluster. The maximum root-leaf distance in a tree is the depth, thus the maximum total length of two such segments is $2 * d - 1$ as we only include the root once. Trivially, the cluster has at least an equal amount of nodes. Note that if the two paths segments overlap the actual path would be simplified as in TBA*. In case of overlapping segments the agent would go back and forth on the same path thus the states in the overlapping segments are omitted. □

5.4 Empirical Evaluation

Because Cluster-RTS expends time trying to exploit the internal structure of state spaces, its success will depend on the extent to which this structure is available and its exploitation necessary. We carefully implemented TB(w)A* and Cluster-RTS to study their behavior, using identical data structures where possible. Wall-clock time was used to enforce bounds on each planning iteration. Most data structures were pre-allocated to minimize variation due to memory management. If low-level effects caused an algorithm to overshoot its time bound, the extra time taken was recorded as a tiny 'no-op' action that did not change the agent's state. Each algorithm was tested 10 times on each test instances to capture timing variation. All experiments were carried out on identical machines (Core2 duo E8500 3.16 GHz with 8GB RAM). The implementation was single-threaded and only one experiment was performed at a time on each CPU to enhance accuracy. Following Björnsson et al. (2009), TBA* was configured to focus 90% of the available time on expansions and 10% on back-tracing. Cluster-RTS used an initial cluster size of 500 with a growth rate of 1‰. Grid maps

Algorithm 6: Cluster Real-time Search

```
Explore
while $expansionTime$ not exhausted do
    $c \leftarrow clusterOpen.top()$
    if $c.open.isEmpty()$ then
        $clusterOpen.remove(c)$
        continue
    $n \leftarrow c.open.pop()$
    if $n$ is already assigned to a cluster $c'$ then
        connect clusters $c$ and $c'$
        continue
    if depth of $n$ < $depthBound$ then
        assign $n$ to $c$
        add successors of $n$ to $c.open$
    else
        create new cluster $c_{new}$
        $c_{new}.seed = n$
        add successors of $n$ to $c_{new}.open$
        $clusterOpen.push(c_{new})$
        $depthBound = depthBound * growthFactor$
Main
while $agentState \neq goalState$ do
    Explore
    ExtractPath
    MoveAgent
```

were generated in a variety of sizes. Following the advice of Sturtevant (2012), each grid cell is blocked with probability 0.38.

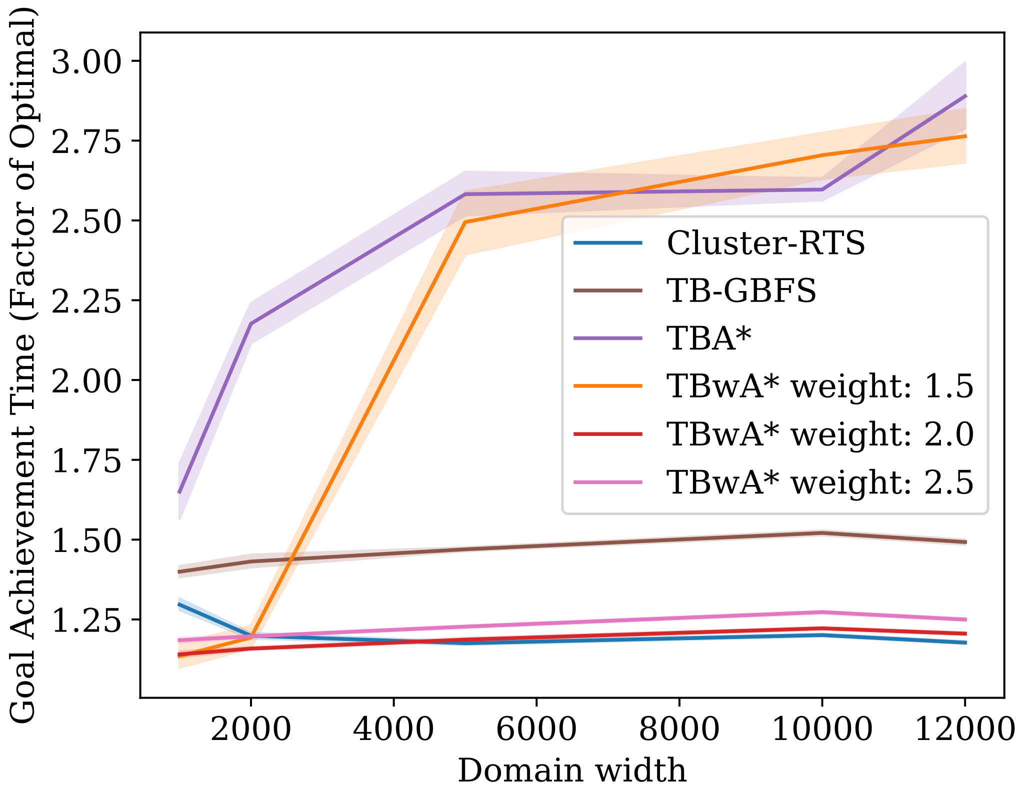
Our first experiment tests the algorithms' ability to scale to larger problems. We generated 50 grid worlds instances with height 1,000 and width from 1,000 to 10,000. The start was at the middle of the left side, the goal at the middle of the right. A time bound of 0.1ms was used.

The *a* panel of Figure 5-4 shows the results. The y axis shows GAT as a factor of the GAT of an oracle that executes an optimal plan with zero planning time. Bands indicate 95% confidence intervals on the mean. We can see that plain TBA* fares poorly as problems become more difficult, as does TBwA* with $w = 1.5$. TBwA* with $w = 2$ and Cluster-RTS both perform well for a wide variety of problem sizes, although Cluster-RTS appears to suffer from some overhead on small problems. Increasing TBwA*'s weight past 2 appears to weaken its performance, culminating in a mediocre performance by TB-GBFS. In these problems, a greedy offline search results in a GAT of roughly 1.4 times optimal.

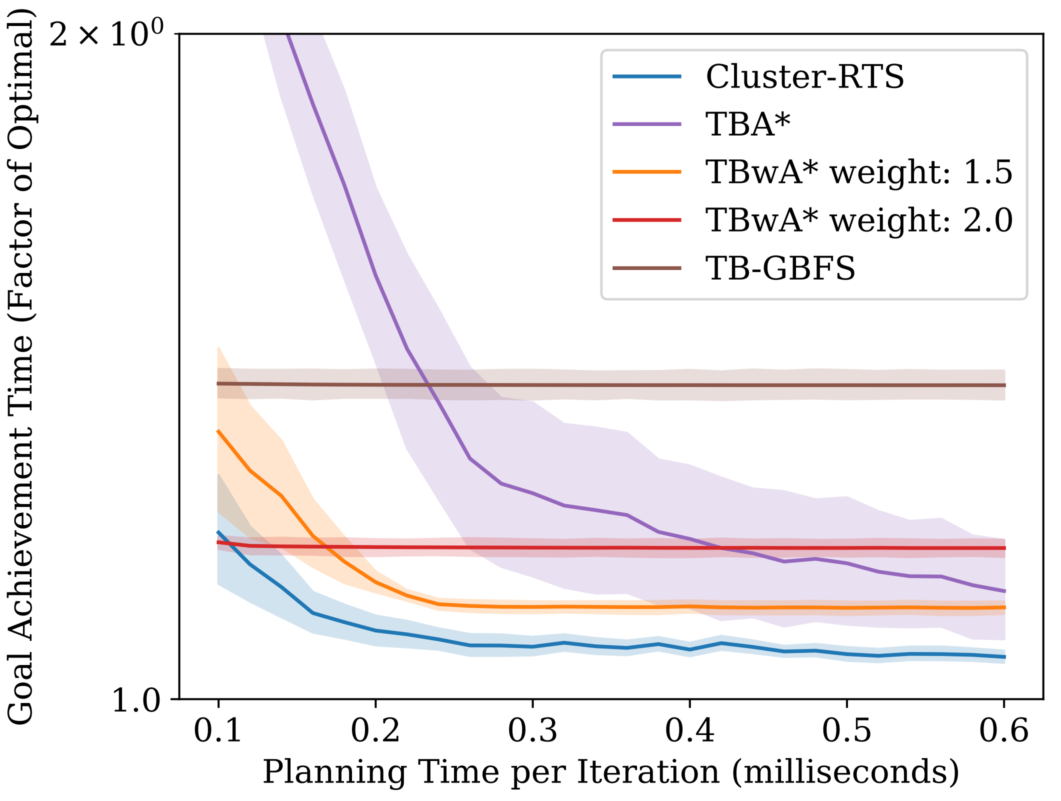
Our second experiment examines the versatility of the algorithms with different time bounds. We generated 50 instances with height 1,500 and width 3,000 with the start state at the top left and the goal state in the bottom right. We tested action durations from $0.1ms$ to $0.6ms$ in $0.02ms$ increments. The right panel of Figure 5-4 shows the results. The y axis is on a logarithmic scale to enhance detail at lower values. Cluster-RTS yields lower GAT than TBA* and TBwA* for a wide range of time bounds. The best performing weights for TBwA* were different for the domain instance we have tested. While in the action duration based experiment TBwA* performed the best with $w = 1.5$ (on Figure 5-4b), the same w were among the worst results when we were testing domains of different sizes. Cluster-RTS was able to provide good performance without parameter tuning in these domains.

5.4.1 The Behavior of TBwA*

We performed several experiments to understand the behavior of TB(w)A*. Our results suggest that its performance is influenced by three factors: the speed with which the search frontier reaches a goal, the length of the solution path found by that search, and the amount of backtracking that is

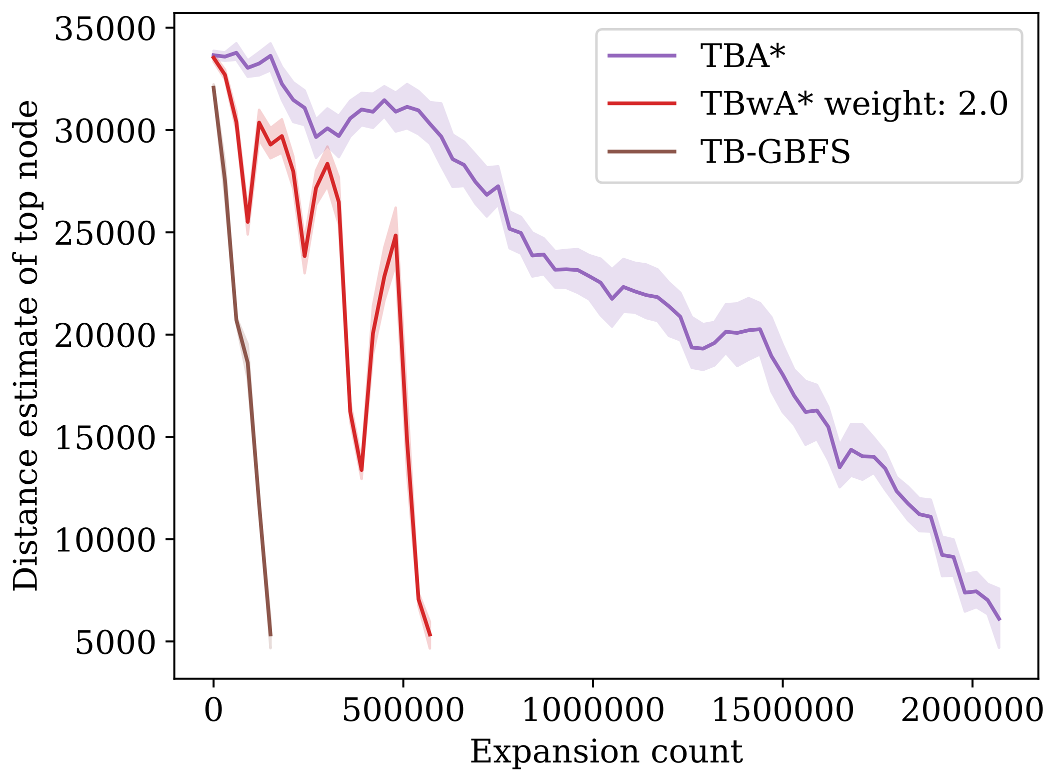


(a)

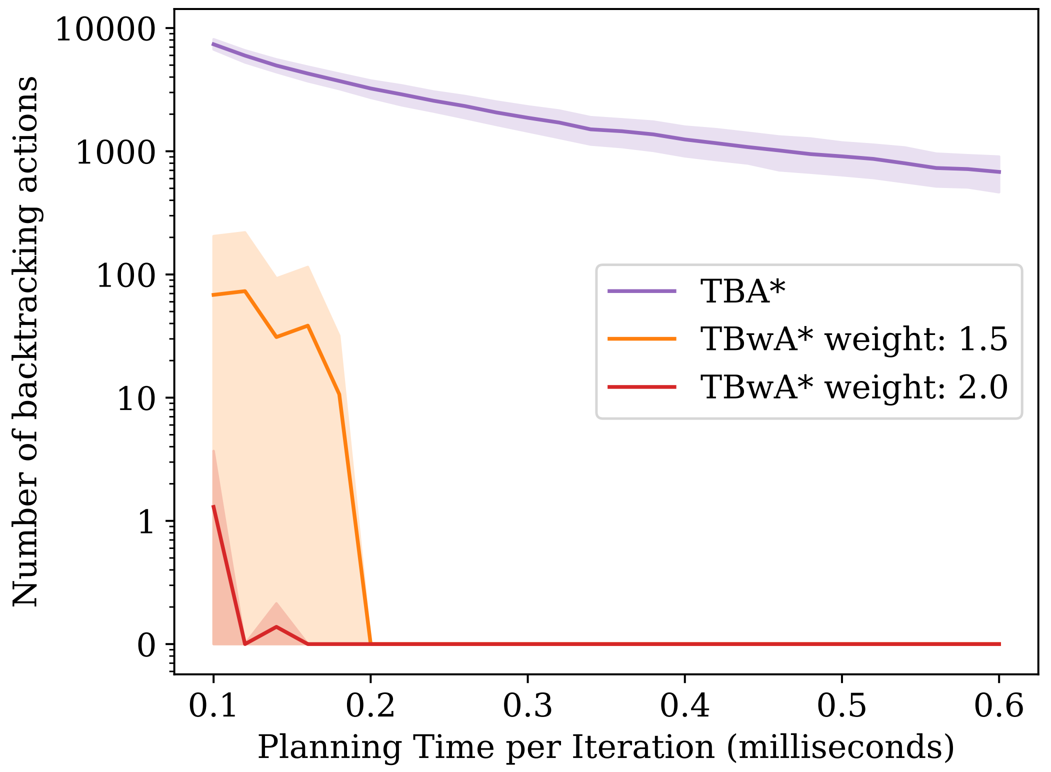


(b)

Figure 5-4: a: GAT as instance size increases. b: GAT as time bound increases.



(a)



(b)

Figure 5-5: a: The *h* value of the currently expanded node. b: TBA*'s backtracking.

required by the agent as it attempts to track and follow the path toward the most promising node on the frontier. First, we note that weighting is commonly used to reduce search time. The *a* panel of Figure 5-5 presents a trace of the *h* value of the expanded nodes over the course of a search on a single instance. (Due to the large number of expansions, the data is subsampled and thus the y axis does not touch 0.) Plain TBA* expands nodes in A* order and we see it taking a long time to reach a goal. In contrast, TB-GBFS finds a goal very quickly, and TBwA* falls in between.

However, wA* and GBFS sacrifice solution quality to reduce the number of expansions. The *b* panel in Figure 5-4 indicates that TB-GBFS is relatively insensitive to the time bound, likely because it finds a path to the goal very quickly. However, as Theorem 20 pointed out, this path can be quite suboptimal and hence the total GAT is lower bounded by the time required for the agent to traverse it. For high time bounds, TBA* converges toward the oracle's performance as the agent wastes few steps and follows the optimal path from the initial state to a goal. These two behaviors define the trade-off faced by TBwA*: as its weight is increased, its dependence on the time bound decreases, but the GAT it can achieve for high bounds is increased. In our benchmarks, a weight of 1.5 seems to balance these factors the best.

Finally, the third influence, and a significant reason why TBA* and TBwA* perform poorly for short time bounds, is the number of actions that the agent takes to backtrack in order to reach the traceback path. The *b* panel of Figure 5-5 shows the number of backtrack moves performed by TBA* and TBwA*. These increase rapidly (note the logarithmic scale) for TBA*, and for a lesser extent, TBwA* with a low weight, as the time bound decreases. Their behavior in the *b* panel of Figure 5-4 suggests that these backtracks play an important role in lowering performance. To the extent this is true, it justifies the approach of Cluster-RTS, which tries to ease navigation of the agent toward the target node on the frontier.

In summary, TB-GBFS and TBwA* can find a goal faster and require fewer backtracking steps, but this must be balanced against the increased length of the path found. In difficult domains, it takes longer for TBA* to find a goal, forcing the agent to commit to suboptimal actions and later backtrack.

5.5 Discussion

Finding the shortest path in the cluster graph has a worst-case time that grows with the number of clusters. The exponential growth of the cluster depth bound is intended to keep the number of clusters logarithmic with respect to the number of expanded nodes, but this is not the same as constant. The incremental construction of state space abstractions for real-time heuristic search deserves continued attention.

The original TBA* algorithm is asymptotically optimal and TBwA* is asymptotically ϵ-optimal with respect to the action duration. Further work is required to design a variant of Cluster-RTS with performance guarantees.

The clusterized state space structure allows for optimizations that could further improve the goal achievementtime. For example, in domains where several solution paths are available, it could be beneficial to select clusters for expansion based on a combination of their estimated goal distance and the agent's estimated distance from the cluster. This would allow the search to focus the expansion of nodes that are closer to the agent and as such reduce the time it takes the agent to reach the search frontier.

One of the bottle necks of parallel heuristic search is the overhead required for communication between threads, which may be traded against the overhead of node re-expansion (Burns, Lemons, Ruml, & Zhou, 2010; Fukunaga, Botea, Jinnai, & Kishimoto, 2017; Hatem, Burns, & Ruml, 2018). In Cluster-RTS, the cluster structure may be useful in defining duplicate detection scopes and avoiding communication, as clusters whose seeds are further from each other than the cluster depth limit can be concurrently expanded without synchronization. Due the the depth limit two such clusters could not attempt to claim the same state, thus no synchronization is necessary between them. Finding independent clusters (in addition to the primary cluster) can be reduced to the vertex cover problem.

Cluster search may have benefits for repeated execution in environments with dynamic start goal pairs, as the underlying graph could allow quickly extracting a path between any two nodes in the expanded state space.

5.5.1 Related Work

Exponential deepening (Sharon et al., 2014) avoids becoming trapped by local minima by simulating iterative deepening with an exponentially-growing cost bound. However, it is not obvious how to use the algorithm with significant lookahead, limiting its applicability to large problems.

The FRIT algorithm (Rivera, Illanes, Baier, & Hernández, 2014) does not rely on LSS expansion as the only means to guide the agent. Instead a relaxed search graph (i.e. one with no obstacles) is constructed either implicitly or via pre-processing and is then updated as the agent discovers that assumed edges do not exist in the true search graph. This approach shows improvements when compared against other LSS algorithms, but still restricts its learning to areas local to the agent. It is not limited to expanding states at most once.

Abstraction and hierarchical path-finding are common techniques for fast grid pathfinding (Botea, Müller, & Schaeffer, 2004). However most previous work assumes that the state space is fully available prior to the start of the search. These techniques require a preprocessing stage where a grid is split into sectors that cache information about the state space. When the online search begins, this information is thus already available to the agent. In contrast, Cluster-RTS generates its clusters online and requires no a priori knowledge of the state space. Thus, it should be applicable to initially-unknown domains. Cluster-RTS applies to any undirected domain, as opposed to grid-specific abstraction techniques that rely on fitting rectangular sectors.

Eifler, Fickert, Hoffman, and Ruml (2019) present methods for refining abstractions online during real-time search. However, their Cartesian abstractions assume a factored finite domain representation of the state space.

5.6 Conclusion

In this chapter, we introduced Cluster-RTS a real-time heuristic search algorithm designed for large state spaces with local minima. Cluster-RTS appears to be a promising approach to non-agent-centered real-time heuristic search. It introduces a novel way to explore large state spaces by incrementally growing a graph of small search trees called clusters that provide a decentralized structure for efficient agent navigation. Cluster-RTS addresses the two most pressing issues that limit the exploration and navigation capabilities of the state-of-the-art method, TBA*. First, Cluster-RTS relies on a state space structuring that is not limited by the original start state, thus it can scale to efficiently explore large state spaces. Second, its underlying state space structuring, the cluster graph, allows the agent to navigate efficiently within the explored state space, avoiding extensive backtracking. Our results indicate that Cluster-RTS matches the performance of TBA* in small benchmark problems and is superior in large ones.

CHAPTER 6

Optimization for Goal Achievement Time

This chapter addresses the characteristic truly unique to on-line planning – that time passes during planning. In an on-line planning setting there is no distinction between time spent on planning, execution, or both at the same time. This fundamental characteristic can be exploited to reduce the goal achievement time of real-time heuristic search methods.

When minimizing makespan during off-line planning, the fastest action sequence to reach a particular state is, by definition, preferred. When trying to reach a goal quickly in on-line planning, previous work has inherited that assumption: the faster of two paths that both reach the same state is usually considered to dominate the slower one. In this chapter, we point out that, when planning happens concurrently with execution, selecting a slower action can allow additional time for planning, leading to better plans. We present Slo'RTS, a metareasoning planning algorithm that estimates whether the expected improvement in future decision-making from this increased planning time is enough to make up for the increased duration of the selected action. Using simple benchmarks, we show that Slo'RTS can yield shorter time-to-goal than a conventional planner. This generalizes previous work on metareasoning in on-line planning and highlights the inherent uncertainty present in an on-line setting. This work appeared in the Proceedings of the Twenty-eighth International Conference on Automated Planning and Scheduling (ICAPS-17) (Cserna et al., 2017).

6.1 Introduction

Traditionally, planning has been considered from an off-line perspective, in which plan synthesis is completed before plan execution begins. In that setting, if the objective is to minimize plan makespan, then it is clearly advantageous to return a faster plan to achieve the goal. For example, in a heuristic search-based approach to planning, if the planner discovers two alternative plans for achieving the same state, it only needs to retain the faster of the two. In A* search, this corresponds to the usual practice of retaining only the copy of a duplicate state that has the lower g value.

However, many applications of planning demand an on-line approach, in which the objective is to achieve a goal as quickly as possible, and planning takes place concurrently with execution. For example, while the agent is transitioning from state s_1 to state s_2, the planner can decide on the action to execute at s_2. In this way, the agent's choice of trajectory unfolds over time during execution, rather than being completely pre-planned before execution begins. While this may result in a trajectory that is longer than an off-line optimal one, it can result in achieving the goal faster than off-line planning because the planning and execution are concurrent (Kiesel et al., 2015; Cserna et al., 2016). This setting also models situations in which an agent's goals can be updated during execution, requiring on-line replanning.

The first contribution of this chapter is to point out that the on-line setting differs from the off-line one in that it may be advantageous for the planner to select a slower action to execute at s_2 even when a faster one is known to reach the same resulting state s_3. This is because the longer action will give the agent more time to plan before reaching s_3. This might result in a better decision at s_3, allowing the agent to reach a goal sooner. If the decision is substantially better, the difference may even be large enough to offset the delay due to the slower action. Anyone who has slowed down while driving on a highway in order to have more time to study a map before passing a crucial exit is intuitively familiar with this scenario. We generalize this reasoning to cover actions that do not immediately lead to the same state s_3.

The second contribution of this chapter is a practical on-line planning algorithm, Slo'RTS (pronounced *Slow-are-tee-ess*), that takes this observation into account. We work in the paradigm of forward state-space search, using real-time heuristic search algorithms that perform limited lookahead search and then use the lookahead frontier to inform action selection. When operating in a domain that has durative actions, whose execution times can be different, Slo'RTS takes actions' durations into account, estimating the effect on decision-making at future states. In this way, Slo'RTS reasons about its own behavior; in other words, it engages in metareasoning. We implement and test Slo'RTS in some simple gridworld benchmarks, finding that its metareasoning can indeed result in better agent behavior. To our knowledge, this is the first example of a planning algorithm that can dynamically plan to give itself more time to think without assuming that the world is static. More generally, this work is part of a recent resurgence of interest in metareasoning in heuristic search, illustrating how this beautiful idea can yield practical benefits.

6.2 Previous Work

We briefly review the real-time heuristic search and metareasoning algorithms that Slo'RTS is based on. Given our objective to minimize time to goal, we will assume that plan cost represents makespan.

6.2.1 Real-Time Search

As a quick reminder, Local Search Space-Learning Real-Time A* (LSS-LRTA*) is a leading general-purpose agent centered real-time heuristic search algorithm (Koenig & Sun, 2008). Each iteration of LSS-LRTA* operates in two phases: exploration and learning. First, the exploration phase uses an A* search to expand a local search space around the agent until an expansion bound is reached, at which point the agent will commit to the action leading to the best node at the edge of the lookahead frontier. Second, in the learning phase, the heuristic values of expanded states are updated by propagating information backwards from the lookahead frontier using a Dijsktra-like propagation algorithm. The next iteration then uses these updated h values, generating a fresh lookahead search frontier. For more information on LSS-LRTA* please refer back to Algorithm 1 in Chapter 2.

Dynamic $\hat{f}$ is a variant of LSS-LRTA* (Kiesel et al., 2015). It employs an inadmissible heuristic, notated $\hat{h}$. This value is an unbiased estimate of a node's true cost-to-goal, rather than an admissible lower bound. Just as $f(s) = g(s) + h(s)$, we will write $\hat{f} = g(s) + \hat{h}(s)$. During lookahead search, Dynamic $\hat{f}$ sorts the frontier on $\hat{f}$ instead of f, and afterwards, the action selected for execution is the one leading to the frontier node with the best $\hat{f}$ value.

While any $\hat{h}$ could be used, in experiments reported below, we use a version of the standard admissible h that is debiased online using the 'single-step path-based error' method of Thayer, Dionne, and Ruml (2011). During search, the error ϵ in the admissible h is estimated at every expansion by the difference between the f value of the parent node and the f value of its best successor (these will be the same for a perfect h). If $\bar{\epsilon}_s$ is the average error over the nodes along the path from the root to a node s and $d(s)$ is an estimate of the remaining search distance (number of edges along the path) from a state s to the nearest goal, then $\hat{h}(s) = h(s) + \bar{\epsilon}_s \cdot d(s)$.

During the learning phase, if node a inherits its f value from a node b on the search frontier, then Dynamic $\hat{f}$ will update $h(a)$ using the path cost between a and b (the difference in their g values) and b's h value. The remaining error in the estimate of $f(a)$ will then derive from the error in $h(b)$. So for each updated h value, Dynamic $\hat{f}$ also records the d value of the node it was inherited from, and thus $\hat{h}(a) = h(a) + \bar{\epsilon}_b \cdot d(b)$.

6.2.2 Metareasoning Search

Metareasoning On-line Real-time Search (Mo'RTS, pronounced Moe-are-tee-ess) addresses domains that have identity actions, which function as a no-op or idle action by taking time but not changing the state of the world (O'Ceallaigh & Ruml, 2015). Mo'RTS uses metareasoning to decide when to execute identity actions. When no identity actions are taken, Mo'RTS works just like dynamic $\hat{f}$. When an identity action is taken, Mo'RTS can preserve its lookahead frontier from its previous search iteration and extend it, allowing deeper lookahead and more accurate decision-making. In an extreme case, Mo'RTS can decide to take so many identity actions that it is able to plan all the way to a goal, thereby imitating the behavior of off-line A* search.

The central decision of Mo'RTS is whether the delay in reaching the goal caused by the duration of an identity action $t_{identity}$ is outweighed by the expected benefit B of additional search:

$$B > t_{identity}. \tag{6.1}$$

Search will provide benefit if, instead of the current best action α, additional search causes the planner to select some other action β instead. Abusing notation by referring to states by the actions that lead to them, Mo'RTS represents the value of an action α as a belief distribution over possible values, with $p_\alpha(x)$ representing the probability density that α's true f^* value is x. This distribution is assumed to be a Gaussian centered at $\hat{f}(\alpha)$ with variance $\sigma^2 = (\bar{\epsilon}_b \cdot d(b))^2$, where b is the frontier node from which α inherits its value. Just as in Dynamic $\hat{f}$, Mo'RTS allows further search to decrease $d(b)$ and hence sharpen our belief about α.

Because actions are chosen based on their $\hat{f}$ values, to estimate benefit, we need to estimate what α and β's $\hat{f}$ values might be after we have performed more search. This is represented as a Gaussian belief $p'_{\overline{\alpha}}(x)$ centered at $\hat{f}(\alpha)$ with variance

$$\sigma^2_{p'_{\overline{\alpha}}} = \sigma^2_{p_\alpha} \cdot \min(1, \frac{d_s}{d(b)}) \tag{6.2}$$

where d_s is the distance, in search steps, along the path to a goal that we expect to cover during the extra search. The intuition is that, if no further search were done, the variance of $p'_{\overline{\alpha}}$ is zero since $\hat{f}(\alpha)$ wouldn't change, and if we searched all the way to a goal, we would expect $\hat{f}(\alpha)$ to be distributed like p_α, since that is the definition of p_α. d_s is estimated by dividing the number of expansions that will be done during the search by the expected number of expansions required to make progress along a search path (estimated by the average number of expansions from when a node is generated until it is expanded) (Dionne, Thayer, & Ruml, 2011).

The benefit of search, if the value of the most promising action α were to become x_α and the value of some competing action β were to become x_β, would be

$$b(x_\alpha, x_\beta) = \begin{cases} 0 & \text{if } x_\alpha \leq x_\beta \\ x_\alpha - x_\beta & \text{otherwise} \end{cases} \tag{6.3}$$

because we would have done α if we had not searched. The expected benefit is the expected value over the estimates of $p'_{\overline{\alpha}}$ and $p'_{\overline{\beta}}$:

$$B = \int_{x_\alpha} p'_{\overline{\alpha}}(x_\alpha) \int_{x_\beta} p'_{\overline{\beta}}(x_\beta) b(x_\alpha, x_\beta) dx_\beta dx_\alpha. \tag{6.4}$$

Using these estimates, Mo'RTS can decide whether an identity action is worthwhile. However, in domains in which there is no identity action that allows the agent to stop the world and think, Mo'RTS does not apply.

6.3 Metareasoning for Durative Actions

Slo'RTS generalizes the ideas behind Mo'RTS. Instead of a special comparison of the expected benefit of search against the time cost of an identity action, we add consideration of time when

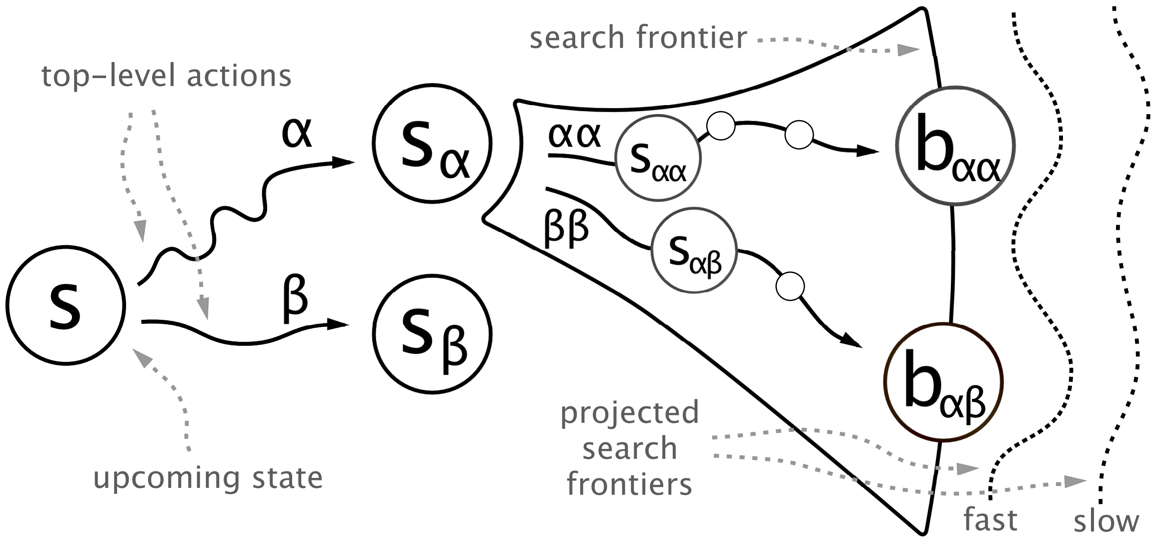


Figure 6-1: An example search tree considered by Slo'RTS.

computing the expected cost of every top-level action. Figure 6-1 illustrates a situation in which the agent is currently transitioning to state s and the planner is deciding whether to take action α or β once the agent arrives there. Ordinarily, an action like α would receive a backed up value equal to the minimum of its children, $s_{\alpha\alpha}$ and $s_{\alpha\beta}$, whose values are inherited from frontier nodes $b_{\alpha\alpha}$ and $b_{\alpha\beta}$. But this assumes that we have perfect information about these values and that no further information will be gained. In a real-time search context, this assumption does not hold; there may be insufficient time to fully explore the space, especially early on.

Slo'RTS recognizes that, if we choose to do α at s, by the time we actually reach s_α, we will have refined our beliefs about $s_{\alpha\alpha}$ and $s_{\alpha\beta}$, depending on how long α takes to execute. This leaves open the possibility that, by then, $\alpha\beta$ might be more attractive than $\alpha\alpha$. So when estimating the value of s_α, Slo'RTS computes the expected minimum over what it thinks its beliefs about $\hat{f}(s_{\alpha\alpha})$ and $\hat{f}(s_{\alpha\beta})$ might be after searching during α. In this way, when comparing s_α and s_β, the durations of α and β are taken into account.

More formally, for every child b of every top-level action a, Slo'RTS models its belief about $f^*(s_{ab})$ as a distribution as in Mo'RTS

$$p_{ab} \sim \mathcal{N}(\hat{f}(s_{ab}), (\bar{\epsilon}_{b_{ab}} \cdot d(b_{ab}))^2) \tag{6.5}$$

and its belief about the location of $\hat{f}(s_{ab})$ after search as

$$p'_{\overline{ab}} \sim \mathcal{N}(\hat{f}(s_{ab}), (\bar{\epsilon}_{b_{ab}} \cdot d(b_{ab}))^2 \cdot \min(1, \frac{d_s}{d(b_{ab})}) \tag{6.6}$$

As a reminder, d_s is lower bound on the number of nodes between node s and the closest goal state. Note that d_s will vary depending on the duration of a, causing the belief to be a spike at $\hat{f}(s_{ab})$ if a is very fast and to spread out to mimic p_{ab} with full search to the estimated goal depth. This means that deeper search implies more possibility for the estimate of $\hat{f}(s_{ab})$ to change and for ab to perhaps change its ranking with respect to other actions, possibly becoming the best action at s_a.

To estimate the value of a top-level successor state s_a, we take the expected minimum over the estimates of the beliefs about the children of s_a after planning during a. If the children were α and β, this would be

$$E_a = \int_{x_{a\alpha}} \int_{x_{a\beta}} p'_{\overline{a\alpha}}(x_{a\alpha}) p'_{\overline{a\beta}}(x_{a\beta}) \min(x_{a\alpha}, x_{a\beta}) dx_{a\beta} dx_{a\alpha}. \tag{6.7}$$

This may be lower than $\min(\hat{f}(s_{a\alpha}), \hat{f}(s_{a\beta}))$ if it appears that search might change the values significantly. In the implementation tested below, we integrate numerically, and we limit our consideration to the best two actions under each top-level action. After every search iteration, Slo'RTS chooses the top-level action that has the smallest expected value.

6.4 Evaluation

We now turn to a brief analysis of the behavior of Slo'RTS in comparison to conventional on-line planners.

Theorem 26 *Conventional real-time search algorithms always choose the shortest sequence of actions towards any expanded state within the lookahead search space, assuming the heuristic is consistent.*

Proof: Conventional algorithms such as LSS-LRTA* use A* for their lookahead phase. When A* generates a node representing the same state as a previously-generated node, only the one with the lowest g value is retained. This reflects the dynamic programming optimal substructure property that

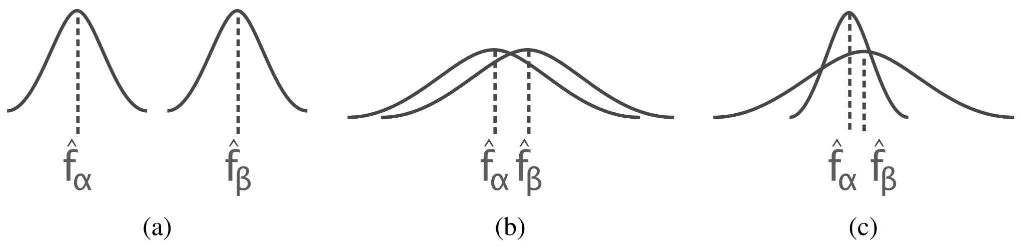


Figure 6-2: Three scenarios with beliefs about α and β.

subpaths of shortest paths are themselves shortest paths. Furthermore, if the heuristic is consistent, the g value of any node expanded by A* (and thus within the lookahead space) is optimal. Thus, real-time searches select only shortest paths to expanded nodes in the lookahead space. □

Corollary 1 *When two sequences of actions lead to the same state, conventional real-time search algorithms will always take the faster path over the slower.*

This illustrates the uniqueness of Slo'RTS, which has the ability to select a slower path when it deems it beneficial.

Theorem 27 *If the belief distributions of Slo'RTS are accurate, the algorithm makes the optimal rational decision.*

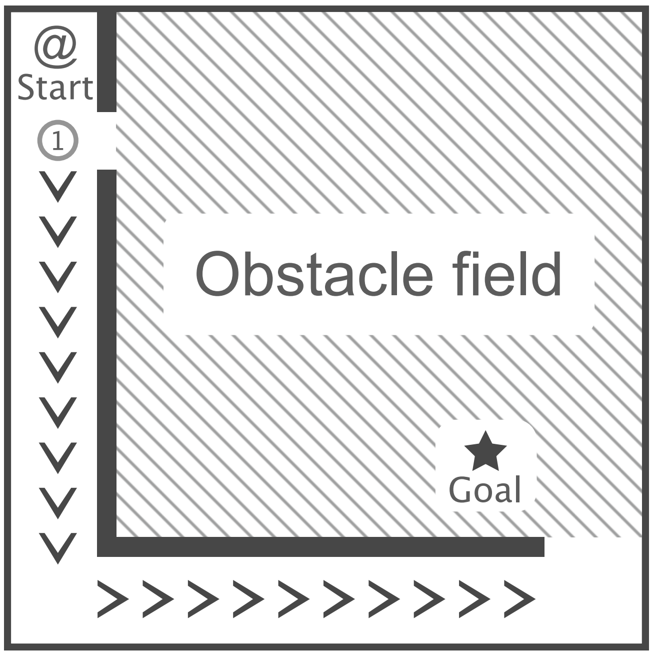
Proof: The heart of Slo'RTS, equation 7, estimates expected cost. Under the definition of rationality as maximizing expected utility, which, given the objective of Slo'RTS corresponds to minimizing cost, this is the optimal rational decision given the state of knowledge of the planner. □

The added time complexity of Slo'RTS over LSS-LRTA* is $O(b)$ where b is the branching factor. At the end of every planning iteration, when the algorithm makes a decision, it calculates the expected value of taking each top-level action using equation 7. If, as in our implementation, this is limited to two actions per top-level action, then it is constant time for each top-level action. Another approach would be to compare each applicable action against the best, leading to $O(b)$ time per top-level action and $O(b^2)$ overall.

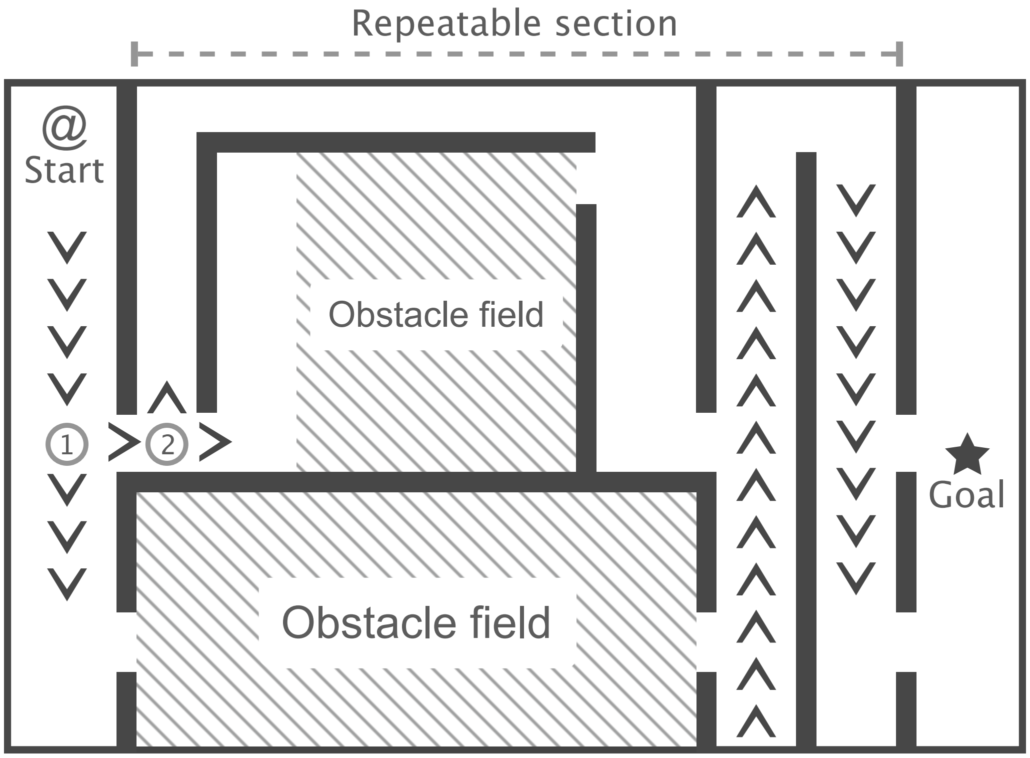
To gain a concrete sense of Slo'RTS' behavior, we first tested the core decision procedure on three classic but completely synthetic scenarios, illustrated in Figure 6-2. Each scenario involved two top-level actions, one fast and one slow, both leading to the same state at which two actions, α and β, were applicable. In scenario (a), we defined costs in the search space such that Slo'RTS was quite confident about the values of α and β. Additional search was not predicted to have a significant chance of causing β to appear better than α, and Slo'RTS correctly chose the fast action. In scenario (b), the beliefs about α and β were very uncertain, but so similar that it didn't matter much which was chosen. Slo'RTS recognized this and chose fast. In scenario (c), the belief about α is quite certain but β is uncertain and has a chance of leading to a significantly better outcome than α. In this case, Slo'RTS correctly chose the slow action to allow additional search.

We then implemented a Slo'RTS agent in a grid pathfinding domain using the Manhattan distance heuristic and compared it with LSS-LRTA*. In each cardinal direction, both fast and slow actions were available. Figure 6-3a shows a simple instance with an important early decision marked by 1. The agent has to decide between crossing the obstacle field that looks slightly worse from the outside or going around the L shaped obstacle. The optimal path goes through the obstacle field. Due to the limited lookahead, conventional real-time planners like LSS-LRTA* always take the better looking path and go around the obstacle. Slo'RTS recognizes that the path leading into the obstacle field has potentially better cost and slows down before the decision point. The slow down results in a lookahead that is sufficient to find a better path through the obstacle field.

Lastly, we implemented a class of benchmark problems featuring grid cells with expensive irreversible decisions, shown in Figure 6-3b. This simulates the example of deciding which exit to take on a highway. The middle section of the map is repeatable, yielding a sequence of decisions. Each time the agent passes through this section, it passes one or two decision points surrounded with irreversible cells, thus the agent cannot turn back to choose another path. Conventional algorithms like LSS-LRTA* always choose fast actions and are thus highly dependent on their lookahead parameter. Slo'RTS correctly identifies the decision points and slows down before reaching them, allowing better decision-making. Since Slo'RTS plans while traveling to the goal, node expansions are a perfect measure of time-to-goal; Figure 6-4 shows the total time-to-goal (in number of node expansions) as the number of middle segments is increased. Clearly, in these instances the benefits of slowing down to make a better decision outweigh increased execution time.



(a) Simple



(b) Highway

Figure 6-3: Schematics of the grid benchmarks.

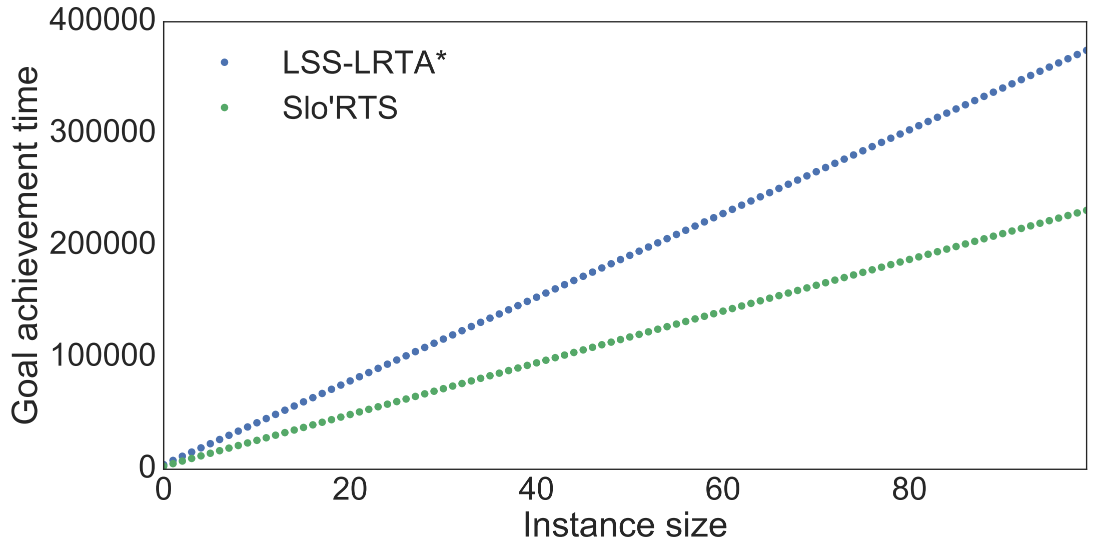


Figure 6-4: Planner performance on highway instances.

6.5 Discussion

There are several promising ways in which Slo'RTS could be extended. For example, it currently considers only the effects of time on the decisions at the successors of the current state, but this could in principle apply to any future anticipated states of the agent. We also believe that it should be possible to derive the ideal amount of time to spend planning, if the domain allows flexibility in action durations.

There has been extensive prior work on deliberation scheduling, in which decisions must be made about when to plan. Most of this work separates the planner from a metareasoning executive that initiate replanning or policy improvement meta-actions (Dean, Kaelbling, Kirman, & Nicholson, 1993; Musliner, Goldman, & Krebsbach, 2003; Krebsbach, 2009). Some work computes the value of deliberation off-line (Wu & Durfee, 2006). In contrast, Slo'RTS embeds metareasoning in the planner itself, where it has on-line access to detailed information regarding the benefit of planning. Our objective of minimum time-to-goal allows Slo'RTS to directly compare time spent planning to estimated improvement in plan cost.

The decision faced by a multi-armed bandit algorithm is similar to that faced by Slo'RTS, in that it involves a trade-off between exploring to gather more information versus exploiting the estimates that the agent has already gathered. Metareasoning itself has been formulated as an MDP (Lin, Kolobov, Kamar, & Horvitz, 2015). Work on control of anytime algorithms is also related, in that one must dynamically decide whether to continue planning (Hansen & Zilberstein, 2001b; Póczos, Abbasi-Yadkori, Szepesvári, Greiner, & Sturtevant, 2009).

There are many real-time search algorithms and it would be interesting to adapt Slo'RTS to them. daRTAA* attempts to avoid heuristic depressions which cause revisiting the same state multiple times (Hernández & Baier, 2012). EDA* provably avoids revisiting states many times, but is limited to undirected domains in which every action can be immediately and exactly undone and is nontrivial to use with lookahead (Sharon et al., 2014).

6.6 Conclusion

This chapter has focused on the fundamental issue unique to on-line planning (that time passes while planning), proposed a principled algorithm for harnessing it, and shown that the algorithm can provide benefit in simple benchmarks. While not every domain will see benefit from metareasoning about planning time, it is useful to recognize that planning can concern more than which actions to do, or even whether to think more, but also how to provide the time to do so.

6.7 Acknowledgements

This work was inspired by discussions with Dylan Hadfield-Menell, starting in the King's College London cafeteria during ICAPS 2016.

CHAPTER 7

Conclusion

Real-time heuristic search can be a viable planning framework for concurrent planning and execution in complex state spaces. This dissertation has shown that real-time heuristic search can guarantee the safety of a controlled agent in domains with dead-ends states, that it can operate in large domains with heuristic depressions, and that it can exploit meta-information specific to the on-line setting in partial search trees to achieve the goal faster.

First, we demonstrated that real-time heuristic search can be a potential alternative to anytime algorithms while it also imposes a bound on the time it takes to make a decision. Thereafter, the focus was to mitigate and address the major drawbacks of real-time heuristic search that limits its applicability while optimizing for goal achievement time.

The first drawback that this dissertation addressed, was the incompleteness of real-time heuristic search methods in directed domains with dead-end states. Chapter 2 introduced a general framework that provides online safety guarantees to ensure the agent does not fall into dead-end states during execution along with an efficient implementation SafeRTS. Chapter 3 demonstrated that the idea of safety is not only applicable to spatiotemporal planning but optimizing for safety can lead to higher performance. Planning in domains with dead-ends requires additional algorithmic step over the traditional methods that do not consider safety. Techniques to minimize this overhead were presented in Chapter 4. These chapters together demonstrated that safe real-time heuristic search can be applied to safety critical systems.

Unlike offline or anytime heuristic search algorithms, real-time heuristic search methods are prone to getting lost in local-minima. Chapter 5 presented, the second major contribution of this dissertation, methods that alleviate this deficiency by introducing a new way to explore and structure spaces of future decisions, Cluster-RTS.

The final contribution of the dissertation provided a metareasoning method to exploit the fundamental characteristic truly unique to on-line planning – that time passes during planning. Chapter 6 introduced a real-time metareasoning technique Slo'RTS that considers actions that do not lead to the best discovered node, according to the commonly used f metric, in order to provide more time to the agent to plan the upcoming iteration. We demonstrated that having more time to deliberate over a difficult decision could reduce the time it takes the agent to reach the goal.

The hope of this work is to encourage further efforts in widening the applicability of real-time heuristic search for online planning. I believe that real-time heuristic search as a framework for online planning has great potential and it is currently limited by its applications. The true power of real-time heuristic search can be exposed only by inspiring settings in which bounded time decision making is essential.

Bibliography

Bareiss, D., & van den Berg, J. (2015). Generalized reciprocal collision avoidance. *International Journal of Robotics Research*, *34*, 1501–1514.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*(1), 81–138.

Bekris, K. E., & Kavraki, L. E. (2007). Greedy but safe replanning under kinodynamic constraints.. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-07)*, pp. 704–710.

Björnsson, Y., Bulitko, V., & Sturtevant, N. (2009). TBA*: time-bounded A*. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp. 431–436.

Bond, D. M., Widger, N. A., Ruml, W., & Sun, X. (2010). Real-time search in dynamic worlds. In *Proceedings of the Symposium on Combinatorial Search (SoCS-10)*.

Boone, G. (1997). Minimum-time control of the acrobot. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, Vol. 4, pp. 3281–3287. IEEE.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, *1*, 7–28.

Burns, E., Kiesel, S., & Ruml, W. (2013). Experimental real-time heuristic search results in a video game. In *Proceedings of the Sixth International Symposium on Combinatorial Search (SoCS-13)*.

Burns, E., Lemons, S., Ruml, W., & Zhou, R. (2010). Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, *39*, 689–743.

Burns, E., Ruml, W., & Do, M. B. (2013). Heuristic search when time matters. *Journal of Artificial Intelligence Research*, *47*, 697–740.

Camacho, A., Muise, C., & McIlraith, S. A. (2016). From fond to robust probabilistic planning: Computing compact policies that bypass avoidable deadends. In *Proceeding of the Twenty-Sixth International Conference on Automated Planning and Scheduling, (ICAPS-16)*.

Cserna, B., Bogochow, M., Chambers, S., Tremblay, M., Katt, S., & Ruml, W. (2016). Anytime versus real-time heuristic search for on-line planning. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS, 2016.*, pp. 131–132.

Cserna, B., Doyle, W., Ramsdell, J., & Ruml, W. (2018). Avoiding dead ends in real-time heuristic search. In *Proceedings of the twenty second AAAI Conference on Artificial Intelligence (AAAI-18)*.

Cserna, B., Doyle, W. J., Gu, T., & Ruml, W. (2019). Safe temporal planning for urban driving. In *Workshop on Artificial Intelligence Safety 2019 co-located with the Thirty-Third AAAI Conference on Artificial Intelligence 2019 (AAAI-19), 2019*.

Cserna, B., Ruml, W., & Frank, J. (2017). Planning time to think: Metareasoning for on-line planning with durative actions. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS, 2017.*, pp. 56–60.

Dean, T., & Boddy, M. (1988). An analysis of time dependent planning. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence (AAAI-88)*.

Dean, T., Kaelbling, L. P., Kirman, J., & Nicholson, A. (1993). Deliberation scheduling for time critical decision making. In *Proceedings of the Ninth Annual Conference on Uncertainty in Artificial Intelligence, (UAI-93)*.

Dey, D., Sadigh, D., & Kapoor, A. (2016). Fast safe mission plans for autonomous vehicles. In *Proceedings of Robotics: Science and Systems Workshop*.

Dionne, A. J., Thayer, J. T., & Ruml, W. (2011). Deadline-aware search using on-line measures of behavior. In *Proceedings of the Symposium on Combinatorial Search (SoCS-11)*.

Eifler, R., Fickert, M., Hoffman, J., & Ruml, W. (2019). Refining abstraction heuristics during real-time planning. In *Proceedings of the Thirty-third AAAI Conference on Artificial Intelligence (AAAI-19), 2019*.

Ferguson, D., Howard, T. M., & Likhachev, M. (2008). Motion planning in urban environments. *Journal of Field Robotics*, *25*(11–12), 939–960.

Fukunaga, A., Botea, A., Jinnai, Y., & Kishimoto, A. (2017). A survey of parallel A*. *Computing Research Repository*, *abs/1708.05296*.

Gilon, D., Felner, A., & Stern, R. (2016). Dynamic potential search - A new bounded suboptimal search. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016.*, pp. 36–44.

Hansen, E. A., & Zilberstein, S. (2001a). Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, *129*, 35–62.

Hansen, E. A., & Zilberstein, S. (2001b). Monitoring and control of anytime algorithms: a dynamic approach. *Artificial Intelligence*, *126*, 139âĂŞ157.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, *SSC-4*(2), 100–107.

Hatem, M., Burns, E., & Ruml, W. (2018). Solving large problems with heuristic search: General-purpose parallel external-memory search. *Journal of Artificial Intelligence Research*, *62*, 233–268.

Hernández, C., Asín, R., & Baier, J. A. (2014). Time-bounded best-first search. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS, 2014*.

Hernández, C., & Baier, J. A. (2012). Avoiding and escaping depressions in real-time heuristic search. *Journal of Artificial Intelligence Research*, *43*, 523–570.

Hoberock, L. L. (1977). A survey of longitudinal acceleration comfort studies in ground transportation vehicles. *Journal of Dynamic Systems, Measurement, and Control*, *99*(2), 76–84.

JetBrains (2017). Kotlin programming language 1.2m2. https://kotlinlang.org/. Accessed: 2017-09-11.

Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research (IJRR)*, *30*(7), 846–894.

Kiesel, S., Burns, E., & Ruml, W. (2015). Achieving goals quickly using real-time search: Experimental results in video games. *Journal of Artificial Intelligence Research*, *54*, 123–158.

Koenig, S. (2001). Agent-centered search. *AI Magazine*, *22*(4).

Koenig, S., & Likhachev, M. (2002). D* lite. In *AAAI*, pp. 476–483. Proceedings of The Eighteenth National Conference on Artificial Intelligence, (AAAI-02).

Koenig, S., & Likhachev, M. (2006). Real-time adaptive A*. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-06)*, pp. 281–288.

Koenig, S., & Sun, X. (2008). Comparing real-time and in for real-time situated agents. *Journal of Autonomous Agents and Multi-Agent Systems*, *18*(3), 313–341.

Kolobov, A., Mausam, & Weld, D. S. (2010). SixthSense: Fast and reliable recognition of dead ends in MDPs. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, (AAAI-10)*.

Korf, R. (1990). Real-time heuristic search. *Artificial intelligence*, *42*(2-3), 189–211.

Krebsbach, K. D. (2009). Deliberation scheduling using gsmdps in stochastic asynchronous domains. *International Journal of Approximate Reasoning*, *50*(9), 1347–1359.

Kuwata, Y., Teo, J., Fiore, G., Karaman, S., Frazzoli, E., & How, J. P. (2009). Real-time motion planning with applications to autonomous urban driving. *IEEE Transactions on Control Systems Technology*, *17*(5), 1105–1118.

Likhachev, M. (2016). Personal communication..

Likhachev, M., & Ferguson, D. (2009). Planning long dynamically feasible maneuvers for autonomous vehicles. *International Journal of Robotics Research (IJRR)*, *28*(8), 933–945.

Likhachev, M., Gordon, G., & Thrun, S. (2004). ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of Advances in Neural Information Processing Systems, (ANIPS).*

Lin, C. H., Kolobov, A., Kamar, E., & Horvitz, E. (2015). Metareasoning for planning under uncertainty. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, (IJCAI-15).*

Lipovetzky, N., Muise, C., & Geffner, H. (2016). Traps, invariants, and dead-ends. In *Proceeding of the Twenty-Sixth International Conference on Automated Planning and Scheduling, (ICAPS-16).*

Martin, D., & Litwhiler, D. (2008). An investigation of acceleration and jerk profiles of public transportation vehicles. In *Proceedings of American Society of Engineering Education Conference, (ASEE).*

McNaughton, M., Urmson, C., Dolan, J. M., & Lee, J.-W. (2011). Motion planning for autonomous driving with a conformal spatiotemporal lattice. In *ICRA-2011*, pp. 4889–4895.

Moldovan, T. M., & Abeel, P. (2012). Safe exploration in Markov decision processes. In *Proceedings of the 29th International Conference on Machine Learning, (ICML-12).*

Murray, R. M., & Hauser, J. E. (1991). *A case study in approximate linearization: The acrobat example*. Electronics Research Laboratory, College of Engineering, University of California.

Musliner, D. J., Durfee, E. H., & Shin, K. G. (1995). World modeling fo the dynamic construction of real-time control plans. *Artificial Intelligence*, *74*, 83–127.

Musliner, D. J., Goldman, R. P., & Krebsbach, K. D. (2003). Deliberation scheduling strategies for adaptive mission planning in real-time environments. In *Proceedings of the Third International Workshop on Self Adaptive Software*.

O'Ceallaigh, D., & Ruml, W. (2015). Metareasoning in real-time heuristic search. In *Proceedings of the Symposium on Combinatorial Search (SoCS-15).*

Paden, B., Čáp, M., Yong, S. Z., Yershov, D., & Frazzoli, E. (2016). A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, *1*(1), 33–55.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Petrik, M., & Zilberstein, S. (2006). Learning parallel portfolios of algorithms. *Annals of Mathematics and Artificial Intelligence*, *48*(1-2), 85–106.

Pivtoraiko, M., Knepper, R. A., & Kelly, A. (2009). Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, *26*(3), 308–333.

Póczos, B., Abbasi-Yadkori, Y., Szepesvári, C., Greiner, R., & Sturtevant, N. R. (2009). Learning when to stop thinking and do something!. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009*, pp. 825–832.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, *1*, 193–204.

Powell, J., & Palacín, R. (2015). Passenger stability within moving railway vehicles: limits on maximum longitudinal acceleration. *Urban Rail Transit*, *1*(2), 95–103.

Rivera, N., Baier, J. A., & Hernández, C. (2015). Incorporating weights into real-time heuristic search. *Artificial Intelligence*, *225*, 1 – 23.

Rivera, N., Illanes, L., Baier, J. A., & Hernández, C. (2014). Reconnection with the ideal tree: A new approach to real-time search. *Journal of Artificial Intelligence Research*, *50*, 235–264.

Ruml, W., Do, M. B., Zhou, R., & Fromherz, M. P. J. (2011). On-line planning and scheduling: An application to controlling modular printers. *Journal of Artificial Intelligence Research*, *40*, 415–468.

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (Third edition). Prentice Hall.

Schmerling, E., & Pavone, M. (2013). Evaluating trajectory collision probability through adaptive importance sampling for safe motion planning. In *Proceedings of Robotics: Science and Systems, (RSS-13)*.

Shalev-Shwartz, S., Shammah, S., & Shashua, A. (2017). On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374*.

Sharon, G., Felner, A., & Sturtevant, N. R. (2014). Exponential deepening A* for real-time agent-centered search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, (AAAI-14)*, pp. 871–877.

Sharon, G., Sturtevant, N. R., & Felner, A. (2013). Online detection of dead states in real-time agent-centered search. In *Proceedings of the Sixth International Symposium on Combinatorial Search, (SoCS-13)*.

Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). *Introduction to Autonomous Mobile Robots* (2nd edition). The MIT Press.

Sturtevant, N., & Bulitko, V. (2011). Learning where you are going and from whence you came: h-and g-cost learning in real-time heuristic search. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI-11)*, pp. 365–370.

Sturtevant, N., & Bulitko, V. (2014). Reaching the goal in real-time heuristic search: Scrubbing behavior is unavoidable. In *Proceedings of the Seventh International Symposium on Combinatorial Search, (SoCS-14)*, pp. 166–174.

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(2), 144–148.

Sturtevant, N. R., & Bulitko, V. (2016). Scrubbing during learning in real-time heuristic search. *Journal of Artificial Intelligence Research*, *57*, 307–343.

Thayer, J. T., Dionne, A., & Ruml, W. (2011). Learning inadmissible heuristics during search. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Thayer, J. T., & Ruml, W. (2011). Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI-11)*.

Wilt, C. M., & Ruml, W. (2012). When does weighted A* fail?. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS, 2012*.

Wu, J., & Durfee, E. H. (2006). Mathematical programming for deliberation scheduling in time-limited domains. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, (AAMAS-06)*, pp. 874–881.

Ziegler, J., & Stiller, C. (2009). Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios. In *Procceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS-2009)*, pp. 1879–1884.

Zilberstein, S. (2015). Building strong semi-autonomouse systems. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, (AAAI-15)*, pp. 4088–4092.