Avoiding Dead Ends in Real-Time Heuristic Search

Bence Cserna, William J. Doyle, Jordan S. Ramsdell, Wheeler Ruml

Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
bence, doyle, ruml at cs.unh.edu

Abstract

Many systems, such as mobile robots, need to be controlled in real time. Real-time heuristic search is a popular on-line planning paradigm that supports concurrent planning and execution. However, existing methods do not incorporate a notion of safety and we show that they can perform poorly in domains that contain dead-end states from which a goal cannot be reached. We introduce new real-time heuristic search methods that can guarantee safety if the domain obeys certain properties. We test these new methods on two different simulated domains that contain dead ends, one that obeys the properties and one that does not. We find that empirically the new methods provide good performance. We hope this work encourages further efforts to widen the applicability of real-time planning.

Introduction

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 paper, 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 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.

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 prac-

tice, 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 paper, 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.

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 and Sun 2009). (Note that incremental approaches, such as D* (Koenig and Likhachev 2002), or anytime approaches, such as ARA* (Lik ha chev, Gordon, and Thrun 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.

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

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
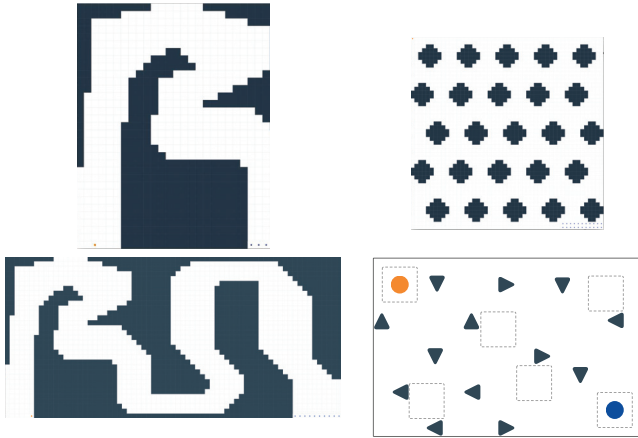7 $s_{root} \leftarrow s$

Figure 1: Top: (left) Barto racetrack; (right) Handmade cluttered track. Bottom: (left) Hansen-Barto combined track; (right) Traffic: white cells are bunkers—see text.

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 states are discovered during planning, the agent commits to the best path to a goal. 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 and Sun 2009). 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. It can handle planning in unknown and dynamic environments. Many other algorithms build on LSS-LRTA*, including RTAA* (Koenig and Likhachev 2006) and daLSS-LRTA* (Hernández and Baier 2012).

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, Burns, and Ruml (2015): reminiscent of the video game Frogger, the agent must navigate a grid from the upper-left to the lower-right using four-way movement while avoiding moving obstacles. A cartoon sketch is shown at the bottom right of Figure 1 (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, Bradtke, and Singh 1995). The agent moves in a grid attempting to reach one of a set of goal locations while avoiding static obstacles. Figure 1 shows the three maps used in our experiments. The top left track was created by Hansen and Zilberstein (2001) and the bottom left track was made by combining it with a track by Barto, Bradtke, and Singh (1995). We created the cluttered track in the top right to provide variety. 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
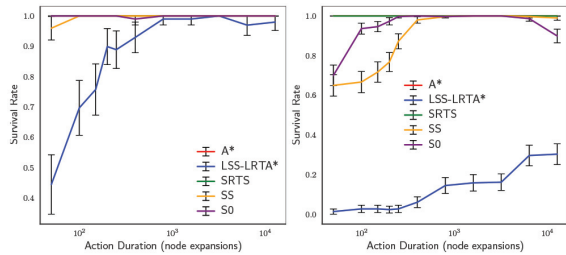
Figure 2: Success rates in (left) traffic and (right) racetrack.

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, Burns, and Ruml 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 shows 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.

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 paper. 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.

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. 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, S0 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 exam-

ple, 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 shows that simple safe search (yellow line denoted SS) is able to keep the agent much safer than LSS-LRTA* in both of our 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 our 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. □

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

Algorithm 2: SafeRTS

Input: s_{root}, *bound*

1 **while** $s_{root} \neq s_{goal}$ **do**
2      $C \leftarrow \emptyset$
3      $b \leftarrow 10 \vartriangleleft$ initialize expansion budget
4   **while** *expansion limit* bound *is not reached* **do**
5   perform ASTAR for b expansions
6   perform BEST-FIRST SEARCH on d_{safe}
               from the top node t of *open*
               until comfortable node c is found
               or until b expansions
7   **if** *such c found* **then**
8   cache comfort of nodes in path from t to c
9              $b \leftarrow 10 \vartriangleleft$ reset budget
10             $C \leftarrow C \cup \{t\}$
11   **else**
12             $b \leftarrow 2 * b \vartriangleleft$ increase budget
13   **for** $c \in C$ **do**
14   propagate comfort to ancestors of c
15   choose node s in *open* with lowest f value
               that has s_{safe} safe predecessor
16   **if** *such s and* s_{safe} *exists* **then**
17         $s_{target} \leftarrow s_{safe}$
18   **else if** *identity action is available at* s_{root} **then**
19         $s_{target} \leftarrow s_{root} \vartriangleleft$ apply identity action
20   **else**
21   TERMINATE $\vartriangleleft$ no safe path is available
22   use DIJSKTRA to update h values of the nodes
23   move the agent along the path from s_{root} to
       s_{target}
24     $s_{root} \leftarrow s_{target}$

---

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, S0 dominates LSS-LRTA* and surpasses simple safe search in racetrack. In the traffic domain, 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.

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
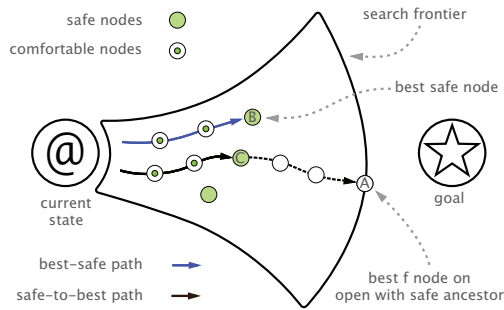
Figure 3: Safe action selection strategies.

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 ∞.

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 and attempting to prove that the currently most promising frontier node is safe via a best-first search on d_{safe}. The node with lowest f (breaking ties toward low h) at the end of the exploration stage is chosen as the promising node for the subsequent proving stage. The proving stage succeeds when the chosen node is determined to be a predecessor of a comfortable node (line 7).

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 10) and the nodes used for the proof are marked as comfortable and this information is cached for future use (line 9). If no comfortable descendant is found, the stage budget is doubled, effective with the next exploration stage (line 13). 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 on promising states. When the overall time bound is exhausted, SafeRTS performs learning as in LSS-LRTA* and backs up comfort through predecessors as in S0.

We experimented with two action selection strategies: safe-toward-best, as used in simple safe and S0 and shown in Algorithm 2 (line 15), 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 3 illustrates their differ-

ences, with safe-toward-best working back from promising nodes on open and best-safe considering all safe nodes. 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. In this case, SafeRTS preserves its current search tree and augments it during the next planning phase.

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) the 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 our heuristic values are initially admissible and consistent (A1) and action costs are positive (A2), the same proof as used for LSS-LRTA* (Koenig and Sun 2009, 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 for 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 our proofs 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. □

Empirical Performance

In our experiments, SafeRTS seemed to perform slightly better with the safe-toward-best action selection strategy, so that is what we present here. Figure 2 shows 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 our benchmark set. In our 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.

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, Burns, and Ruml 2015). This measure is also well-defined for off-line methods like A*, allowing us to compare them to real-time search. The left and center panels of Figure 4 presents goal achievement time for the racetrack and 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. Our 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 right panel of Figure 4 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.

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
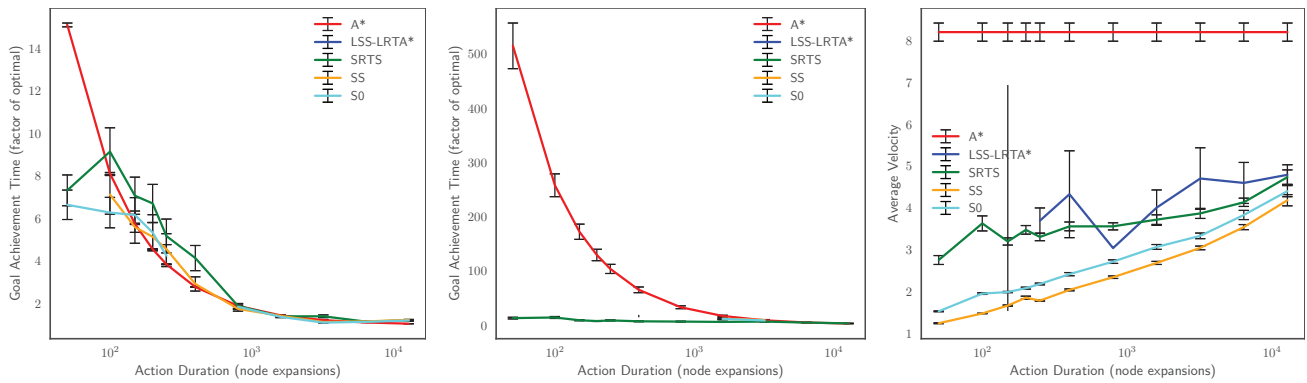
Figure 4: Goal achievement times in (left) traffic and (center) racetrack. Average velocity (right) in the uniform racetrack.

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 and Bulitko 2016), as it updates its h estimates. (Existing methods that avoid this behavior, such as EDA* (Sharon, Felner, and 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, and 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 and Bulitko 2011; Sharon, Sturtevant, and 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, and Weld 2010) and avoiding them in off-line planning (Camacho, Muise, and 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 re-taskable 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

Full job description below. 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 and Likhachev 2006) and adapted for dynamic environments (Bond et al. 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).

Conclusion

As our 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. We 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 efficiently find safe states. We 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. We hope this work encourages further efforts in widening the applicability of on-line planning.

Acknowledgments

References

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

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

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

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

Camacho, A.; Muise, C.; and McIlraith, S. A. 2016. From fond to robust probabilistic planning: Computing compact policies that bypass avoidable deadends. In *Proceedings of ICAPS*.

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

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

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

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

Kiesel, S.; Burns, E.; and 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., and Likhachev, M. 2002. D* lite. In *AAAI*, 476–483. American Association for Artificial Intelligence.

Koenig, S., and Likhachev, M. 2006. Real-time adaptive a*. In *Proceedings of AAMAS*.

Koenig, S., and Sun, X. 2009. Comparing real-time and incremental heuristic search for real-time situated agents. *Journal of Autonomous Agents and Multi-Agent Systems* 18(3):313–341.

Kolobov, A.; Mausam; and Weld, D. S. 2010. SixthSense: Fast and reliable recognition of dead ends in MDPs. In *Proceedings of AAAI*.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42:189–211.

Likhachev, M.; Gordon, G.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems 16*.

Lipovetzky, N.; Muise, C.; and Geffner, H. 2016. Traps, invariants, and dead-ends. In *Proceedings of ICAPS*.

Moldovan, T. M., and Abeel, P. 2012. Safe exploration in Markov decision processes. In *Proceedings of ICML*.

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

Schmerling, E., and Pavone, M. 2013. Evaluating trajectory collision probability through adaptive importance sampling for safe motion planning. In *Proceedings of RSS-13*.

Sharon, G.; Felner, A.; and Sturtevant, N. R. 2014. Exponential deepening A* for real-time agent-centered search. In *Proceedings of AAAI-14*.

Sharon, G.; Sturtevant, N. R.; and Felner, A. 2013. Online detection of dead states in real-time agent-centered search. In *Proceedings of SoCS*.

Sturtevant, N. R., and 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)*, 365–370.

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

Zilberstein, S. 2015. Building strong semi-autonomouse systems. In *Proceedings of AAAI*, 4088–4092.