Real-time Motion Planning with Dynamic Obstacles

Jarad Cannon and **Kevin Rose** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
jjr27, kja24 and ruml at cs.unh.edu

Abstract

Robust robot motion planning in dynamic environments requires that actions be selected under real-time constraints. Existing heuristic search methods that can plan high-speed motions do not guarantee real-time performance in dynamic environments. Existing heuristic search methods for real-time planning in dynamic environments fail in the high-dimensional state space required to plan high-speed actions. In this paper, we present extensions to a leading planner for high-dimensional spaces, R*, that allow it to guarantee real-time performance, and extensions to a leading real-time planner, LSS-LRTA*, that allow it to succeed in dynamic motion planning. In an extensive empirical comparison, we show that the new methods are superior to the originals, providing new state-of-the-art search performance on this challenging problem.

Introduction

As autonomous robots increasingly become incorporated in everyday human activities, they will need to move reliably among humans and other dynamic objects. When dynamic obstacles are present, a robot must plan around their present and predicted future trajectories, updating its plan in real time at a high enough frequency to remain reactive to its surroundings. Current search-based methods do not directly address this problem. In this paper, we present two new algorithms for this problem and give an empirical evaluation comparing them to several of the leading real-time and motion planning algorithms from AI and robotics. Our first algorithm, Real-time R* (RTR*), is a real-time adaptation of the motion planning algorithm R* (Likhachev and Stentz 2008), which has been shown to work well for high-dimensional motion planning problems. Our second algorithm, Partitioned Learning Real-time A* (PLRTA*), is an adaptation of the state-of-the-art real-time search algorithm LSS-LRTA (Koenig and Sun 2009) that allows it to handle motion planning with dynamic obstacles. Our empirical evaluation shows that RTR* and PLRTA* are significant improvements over the original algorithms, performing as well and often better than current motion planning and real-time search algorithms, with PLRTA* offering state-of-the-art performance.

Background

The problem of motion planning can be formulated in several different ways. In this paper, we want to find plans that are fast to execute, so we consider kinodynamic motion planning in which actions must obey the acceleration and deceleration constraints of the specific robot being used. This means that the state representation of the planner must include the robot's current direction and speed to ensure it doesn't, for example, try to turn sharply at high speed. In this paper, we model the robot as a non-holonomic differential drive vehicle. The presence of moving obstacles raises additional issues. The easiest approach is to treat moving obstacles as stationary. This has the advantage that time need not be part of the state space, but can result in highly suboptimal plans or even render problems unsolvable (Kushleyev and Likhachev 2009). Following Kushleyev and Likhachev (2009), we incorporate time as part of the state space. This is because the current and future locations of dynamic obstacles are dependent on time. We assume that the current locations of dynamic obstacles are known but that their future locations are unknown and are represented as a time-parameterized probability distribution.

The Planning Problem

A planning problem $\mathcal{P}$ is defined as a tuple $\{S, s_{start}, G, A, \alpha, O, D\}$ where

- S is the set of states, where a state is the tuple $\langle x, y, \theta, v, t\rangle$ corresponding to location, heading, speed, and the current time.
- s_{start} is the starting state, $s_{start} \in S$.
- G is the set of goal states, where each state $g \in G$ is underspecified as $\langle x, y, \theta, v\rangle$ because it is unknown when the robot will be able to arrive there.
- A is the set of motion primitives available to the robot. A primitive action is a function $a : S \rightarrow S$ that maps states to states and has a duration of t_a. The function $\alpha : S \rightarrow Q,\ Q \subseteq A$ maps states in S to the subset of actions in A that can safely be applied from that state.
- O is the set of static obstacles whose locations are known and do not change.
- D is the set of dynamic obstacles, each represented as a function $d : t \rightarrow \mathcal{N}$ from time to a bivariate Gaussian dis-

tribution representing the object's location. These functions can change as the robot acquires more observations of an obstacle.

A real-time planning algorithm must always return an action $a \in \alpha(s_{start})$ for a given problem $\mathcal{P}$ within the planning time-bound t_p. t_p is the maximum amount of time allowed per planning step. The value t_p must be less than or equal to the duration of the motion primitives, t_a, so that the robot will always have the next action to execute by the time it completes its current action.

The planner attempts to minimize the cost of the agent's trajectory over a fixed time horizon. The total cost of a path is simply the sum of the costs of each action taken along the path. The cost of an action is based on two components, the cost of time passing, C_{time}, and the cost of a collision, C_{col}. Given an action that transitions between two states, the total cost of the action, C, is defined as:

$$C \equiv P(col) \cdot C_{col} + status \cdot C_{time} \tag{1}$$

where $P(col)$ is the probability of a collision occurring with any of the dynamic obstacles and $status$ is 0 if the start state of the action is a goal and 1 otherwise. The probability $P(col)$ is computed assuming that the events of not colliding are independent, following Kushleyev and Likhachev (2009). The cost function is made up of both dynamic costs $(P(col) \cdot C_{col})$ and static costs $(status \cdot C_{time})$.

Previous Work

Previous approaches to this problem in AI and robotics can be classified into four major categories:

Potential Fields Potential field approaches treat the robot as a point charge and the world as a potential field. The goal attracts the robot while obstacles repel. The robot takes the action that lowers its potential the most. This approach works well in environments with few obstacles and is very fast to compute. It suffers from the fatal flaw that the robot can become trapped in local minima (Koren and Borenstein 1991).

Random Sampling Randomized sampling techniques attempt to gain speed at the cost of optimality by greatly reducing the number of states that need be explored. Rapidly-exploring Random Trees (RRT) (Lavalle 1998) are a popular planning technique that works by growing a tree randomly outwards from an initial state. The tree is biased towards unexplored regions of the state space. While RRTs are sometimes able to solve very hard problems quickly, their main drawback is that no guarantees are made on solution quality and in practice, their solutions are often poor. Since RRTs are not real-time, a modified version was used in our experiments where the number of tree expansions is limited to a constant. The action along the path to the node with the lowest heuristic value is then chosen (Lee, Pippin, and Balch 2008).

Heuristic Search D* Lite (Koenig and Likhachev 2002) is an incremental search algorithm developed for path planning in dynamic environments. It repeatedly plans backwards from the goal to the current state of the robot, allowing it to reuse work from previous iterations, greatly speeding up planning. However, it is not obvious how to apply this algorithm to kinodynamic motion planning where time is part of the state, as it is unknown what time the robot will actually arrive at the goal.

The Time-Bounded Lattice algorithm (TBL) (Kushleyev and Likhachev 2009) was designed for the problem representation we consider in this paper. The idea is to do weighted A* search (Pohl 1970) in the full state space out to a specific time bound. After that, the search proceeds in the two dimensional (x, y) space, greatly reducing the number of states that need to be explored. With this approach, dynamic obstacles "disappear" after the time bound cutoff is reached. Because weighted A* is not real-time, TBL is not real-time either. However, because it takes a search-based approach similar to our work, we include it in our experimental evaluation. We modified TBL to always plan at least to the time bound to ensure that it remains reactive to dynamic obstacles while on or near the goal. This resulted in a considerable performance improvement.

Real-time Heuristic Search Real-time algorithms must use specialized techniques to avoid getting stuck in local minima, since there is often not enough time available to plan a complete path from the start to the goal. Real-time A* (RTA*) (Korf 1990) forms the basis of many other real-time algorithms. It works by generating the successors of the agent's current state and doing some form of limited lookahead search to determine which of these successors to move to. The key step is to update the search's heuristic function after picking the best successor to move to. The cached h value of the current state is set to the f value of the second-best successor. The intuition here is that if the algorithm ever returns to that state, its h value would have to be at least the f value of the second best successor since it had already moved to the best successor and returned. In the limit of search iterations, this guarantees completeness in domains where there exists a path to the goal from every state. This means it is able to overcome admissible yet misleading heuristic functions that may lead the agent into local minima. However, this may take a very long time as only one state's h value is updated per search iteration.

A Sampling-based Approach: Real-time R*

In this paper, we investigate two approaches to solving the real-time robot motion planning problem: modifying a leading motion planning algorithm to be real-time, and modifying a leading real-time algorithm to be better suited for motion planning. In the first case, we use the R* algorithm because it has been shown to work well on hard motion planning problems involving high-dimensional state spaces (Likhachev and Stentz 2008).

R* attempts to quickly solve problems in high dimensional state spaces and avoid heuristic local minima by using random sampling paired with heuristic search. R* performs an interleaved two-level weighted A* search where the higher level states are generated randomly and sparsely over the state space and low-level searches are performed in the original state/action space to connect these higher level

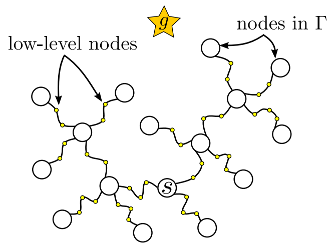

Figure 1: A depiction of an R* search with $k = 3$.

states. This has the advantage of splitting the problem up into smaller, easier to solve subproblems, while not forfeiting the guaranteed feasibility provided at the low-level search space.

When expanding a node s at the top level, R* selects a random set of k states that are within some distance Δ of s. These states will form a sparse graph Γ that is searched for a solution. The edges computed between nodes in Γ represent actual paths in the underlying state space. To find the cost between two nodes s and s' in Γ, R* does a weighted A* search from s to s' in the underlying state space (see Figure 1). If the low-level weighted A* search does not find a solution within a given node expansion limit, it gives up, labeling the node as AVOID, and allowing R* to focus the search elsewhere. R* will only return to these hard to solve subproblems if there are no non-AVOID labeled nodes left. R* solves the planning problem by carrying out searches that are much smaller than the original problem, and easier to solve. Note that R* finds complete paths to the goal on every search, and the time that this takes is not bounded.

Making R* Real-time

We made five major changes to transform R* into real-time R* (RTR*). We will discuss each in turn.

Limiting Expansions To meet the real-time constraints, we begin with the traditional approach of limiting the number of node expansions to a constant. In R*, there are two types of node expansions: nodes in the sparse graph are expanded by generating a set of random successors, while nodes in the low-level state space undergo regular expansion using α from $\mathcal{P}$. The former occur relatively infrequently, but take more CPU time due to the cost of setting up the weighted A* search. The latter occur much more frequently, but each expansion is much faster. In our implementation, we count each high level expansion as equivalent to thirty low-level expansions to account for this difference. Once the expansion limit is reached, the best action to execute is returned, as explained below. Also different from R*, RTR* does not terminate when a goal state is found. It only terminates when the expansion limit has been reached. This is because it isn't sufficient to just reach the goal state. In domains with moving obstacles, it may be necessary to move off the goal state at some future time to get out of the way of an obstacle.

Action Selection After each iteration of RTR*, an action to perform must be selected. As in other real-time searches (Korf 1990), RTR* picks the first action along the most promising path that has been generated. In traditional real-time searches, this corresponds to the best node on the open list. This approach cannot be taken directly in RTR*, because the nodes on the open list in the sparse graph may not all have low-level paths to them. We prefer nodes with complete paths to them, and of these, we prefer nodes with smaller weighted f values. If there are no nodes in the sparse graph with complete paths to them, then nodes with partial paths to them are considered. These are nodes for which the weighted A* search failed to find a complete path due to the node expansion limit. Again, nodes with smaller weighted f values are preferred.

Geometrically Increasing Expansion Limits In R*, if the path to a node is not found due to the node expansion limit, that node is labeled as AVOID and it is inserted back onto the open list. If the node is ever popped off of the open list again, another attempt is made at computing the path, this time with no node expansion limit. This subproblem could be very hard to solve, violating our real-time constraint. In RTR*, each time a search fails due to the expansion limit, the limit is doubled for that node the next time it is removed from the open list. This way, RTR* will not focus all of its effort on computing paths to hard subproblems unless completely necessary, and even then, the paths to the easier of these hard problems will be computed first. Since the expansion limit for computing the path to a node is doubled each time, the total amount of extra searching that may need to be done is bounded by a constant factor in the worse case. In practice, it should actually cause the search to expand many fewer nodes.

Theorem 1 *The total number of extra node expansions that must be done by RTR* because of doubling the expansion limit of a sparse node instead of solving the problem outright is bounded by a constant factor.*

Proof: Analogous to IDA* (Korf 1985) - see Rose (2011) for proof details. □

Path Reuse Because RTR* plans the robot's path over multiple iterations, we cache information after each iteration. The only issue is that the costs of the edges in the search graph can change between search iterations due to the unpredictability of the moving obstacles. RTR* only saves the nodes in the sparse graph that are on the best path found. This allows the RTR* search to seed the sparse graph Γ with nodes that appeared promising on the previous iteration. All other nodes in Γ not on the most promising path are discarded prior to the next planning iteration. The low-level paths between the saved sparse nodes are recomputed as necessary during the next planning cycle. If the costs of the graph have not changed much, then these nodes will most likely still be favorable and will be used by RTR*. If costs have changed, then RTR* is free to recompute a better path, or possibly not even use the cached sparse nodes at all.

Making Easily Solvable Subproblems One of the key insights of the R* algorithm is that dividing the original problem up into many smaller subproblems makes it generally

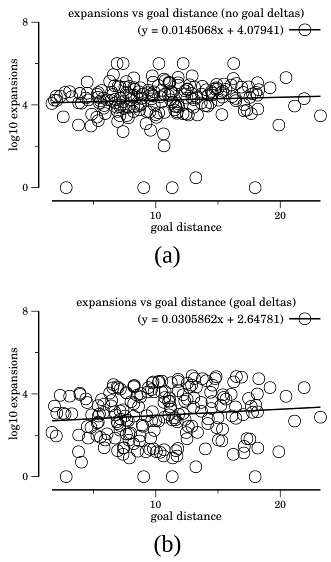

Figure 2: The $\log_{10}$ of the number of nodes needed by weighted A* to solve problems problems with and without a goal radius allowed. The x axis is the distance between the start and goal locations in meters.

easier to solve than solving the original. During node expansion, R* generates successors by randomly sampling the state space at some specified distance Δ away from the node being expanded. Likhachev and Stentz (2008) do not mandate a certain distance metric, although Euclidean distance or heuristic difference are often used. In certain domains, such as robot motion planning, shorter distance does not necessarily correspond to easier problems. Due to the constraints of the vehicle, it could actually be quite difficult to move to a state that is only a small Euclidean distance away (consider parallel parking). We found that requiring RTR* to plan paths to the exact nodes in the sparse graph was prohibiting the search from exploring further into the search space. The reason is that, although the start and goal nodes of these subproblems were close, it was often very hard to maneuver the robot precisely onto a given state. To illustrate this, we ran an experiment in a small world without any static or dynamic obstacles. Despite these ideal conditions, these problems were still quite difficult to solve, with only minuscule correlation between the distance from the start to the goal node and how many node expansions were required to solve the problem. To make these problems easier, the goal condition used for the low-level weighted A* searches was relaxed to allow states within 0.5 meters of the actual goal state and with any heading and any speed to be considered a goal. Figure 2b shows the log of the number of nodes taken to solve a collection of small problems with random start and goal states and no obstacles both with and without the relaxed goal condition. The relaxed goal condition reduced the mean number of nodes expanded to solve the problems by over a factor of 7, allowing RTR* to solve subproblems much more quickly.

Algorithm 1 Partitioned Learning Real-Time A*

```
PLRTA($s_{start}$, $lookahead$)
 1: open = $\{s_{start}\}$
 2: closed = $\{\}$
 3: ASTAR(open, closed, $lookahead$)
 4: $g' \leftarrow$ peek(open)
 5: LEARN_STATIC(open, closed)
 6: LEARN_DYNAMIC(open, closed)
 7: return first action along path from $s_{start}$ to $g'$
```

A Real-time Search Approach: PLRTA*

We developed RTR* by altering a leading motion planning algorithm to be real-time. We also pursued the opposite approach: adapting a state-of-the-art real-time algorithm, LSS-LRTA* (Koenig and Sun 2009), to the problem of robot motion planning with dynamic obstacles. Before presenting the resulting algorithm, Partitioned Learning Real-time A* (PLRTA*), we first describe LSS-LRTA*.

Local Search Space Learning Real Time A* (LSS-LRTA*) is the leading real-time search algorithm. It works by first performing a node-limited A* search (Hart, Nilsson, and Raphael 1968) from the current state towards the goal. Once the node expansion limit is reached, the first action along the path to the lowest f node on the open list is returned. Next, a variant of Dijkstra's algorithm is performed from the nodes on the open list back to all the nodes on the closed list to update all their h values. This is in contrast with RTA*, which only updates one h value per iteration.

LSS-LRTA* has been tested on simple grid world goal finding tasks, however, we believe real-time search should also find applicability in more realistic motion planning. There are two problems with LSS-LRTA* that prevent it from being effectively applied to our problem. First, after an iteration of search, LSS-LRTA* will cause an agent to move along the path to the best node found, until that node is reached or until costs along the path rise. Only then will LSS-LRTA* run another iteration of search. This means that LSS-LRTA* will be incapable of recognizing when shorter paths become available, e.g. from a dynamic obstacle moving out of the way (Bond et al. 2010). Second, the h values learned for nodes will never decrease (Koenig and Sun 2009). This means that if LSS-LRTA* learns that a node has a high h value by backing up a high g value due to a dynamic obstacle, it is unable to later discover that the node has low h cost if the dynamic obstacle were to move away. In this way, the g values in this domain can be seen as inadmissible, breaking a fundamental assumption of previous heuristic search algorithms.

Partitioned Learning Real-Time A* (PLRTA*)

As we show in the evaluation below, LSS-LRTA* can struggle in motion planning. Our central modification to the algorithm is to separate components of the cost function to make learning more effective in this domain. In order to more effectively learn heuristic costs, PLRTA* partitions g and h values into *static* and *dynamic* portions. The static portion refers to only those things that are not time dependent. The

Algorithm 2 Dijkstra's Algorithm for learning h_s values

```
Dijkstra(Closed, Open)
 1: Closed, Open ← InitDijktra(Closed, Open)
 2: while Closed $\neq \emptyset$ AND Open $\neq \emptyset$ do
 3:   delete a state $s$ with the smallest $h_s$ value from Open
 4:   if $s \in$ Closed then
 5:     Closed ← Closed $\setminus \{s\}$
 6:   end if
 7:   for $p \in predecessors(s)$ do
 8:     if $p \in$ Closed AND $h_s(p) > c_s(p,s)+h_s(s)$ then
 9:       $h_s(p) \leftarrow c_s(p,s) + h_s(s)$
10:       if $p \notin$ Open then
11:         Open ← Open $\cup \{p\}$
12:       end if
13:     end if
14:   end for
15: end while
```

Algorithm 3 Initialize Static Dijkstra

```
InitDijkstra(Closed, Open)
1: Closed′ = {}
2: for $n \in$ Closed do
3:   if $n \notin$ Closed′ then
4:     $h_s(n) \leftarrow \infty$
5:     Closed′ = Closed′ $\cup$ { $n$ }
6:   end if
7: end for
8: $\forall n \in$ Open, if $n \in$ Closed′ then Open = Open - { $n$ }
9: return Closed′, Open
```

dynamic portion refers to only those things that are dependent on time, such as the locations of the dynamic obstacles. Because the locations of the static obstacles do not depend on time, we can cache these static h values and use them for any search node representing the same pose $\langle x, y, \theta, v\rangle$, regardless of time. Dynamic values by their definition will change with time and thus can only be cached for specific time-stamped states $\langle x, y, \theta, v, t\rangle$.

For each search node encountered, we track the static costs (g_s), dynamic costs (g_d), static cost-to-go (h_s) and dynamic cost-to-go (h_d). The evaluation function is now:

$$f(n) = g_s(n) + g_d(n) + h_s(n) + h_d(n)$$

Tie-breaking prefers higher g_s values.

Partitioned Learning Real-time A* (PLRTA*) like LSS-LRTA*, performs a limited lookahead A* search forward from the agent. It then selects the minimum f node in the open list and labels it g'. We then perform the heuristic learning described in the following section. The planning iteration ends by taking the first action along the path from s to g'. An overview of PLRTA* is shown in Algorithm 1.

We now described the partitioned learning techniques as well as a method for keeping our algorithm complete in certain situations called Heuristic Decay.

Partitioned Learning

For simplicity, we divide heuristic learning into separate steps for h_s and h_d. The main difference between the two learning steps is in the setup phase. We sort the open list by lowest h_s for the static learning phase. The pseudocode for this is shown in Algorithm 2. We set the h_s of all nodes in the closed list to ∞. States that are duplicates when ignoring time will, by definition, share the same h_s, and are therefore reduced to a single representative node by combining their parent pointers to a single list. We then perform the learning step of LSS-LRTA*, using the h_s of a state and the static cost (g_s) incurred by moving from one state to its successors. Unlike in LSS-LRTA*, the Dijkstra procedure can be terminated when either the open or closed list is exhausted, not simply when the closed list has been emptied.

Theorem 2 *If the static learning step terminates due to an empty open list, it is because the remaining nodes in the closed list are those nodes whose successors lead to dead ends.*

Proof:

The proof is by contradiction. Assume there is some node n in the closed list when the algorithm terminates that has some successor that does not exclusively lead to a dead-end. The open list must be empty since the algorithm only terminates due to an empty open or closed list. This means that n must have had a descendant on the open list at some point during the search. If there did not exist a descendent of n on the open list then we have a contradiction that n's descendents do not lead exclusively to dead ends. Let us call this descendent that was on the open list m. Because of the termination condition, we know that m was removed from the open list as the smallest h_s value at some point during the learning step. Once removed from the open list, m is removed from the closed list if it appears in it, signifying that we have updated its $h_s$value. Then, all of m's predecessors are generated, and those which appear on closed list and have h_s values greater than the cost of moving to m plus $h_s(m)$ are inserted into open. The condition:

$$h_s(predecessors(m)) < c_s(predecesors(m), m)+h_s(m)$$

will hold for any of the $predecessors(m)$ on the closed list at least once, as all nodes on the closed list have their h_s values set to ∞ in Algorithm 3. Therefore, m must have at least one predecessor in the closed list which gets inserted into the open list, otherwise, m would not be in the open list. It then follows that at some point in the future that predecessor of m would be removed from the open list and removed from the closed list, its predecessors generated and placed on the closed list. Ultimately, because n is an ancestor of m, n would have to be inserted onto the open list and sometime in the future removed from both the open list and the closed list. But this is a contradiction, because we stated that n was on the closed list at termination. Thus, it cannot be true that n has some descendent who does not lead exclusively to to a dead-end. Therefore, n must exclusively lead to a dead-end.
□

We also proved that our h_s values will never decrease during the successive searches.

Theorem 3 *The h_s value of the same pose is monotonically nondecreasing over time and thus remains constant or becomes more informed over time.*

Proof: We rely on the proof of Theorem 1 shown by Koenig and Sun (2009). One simply substitutes their use of h with h_s and their notion of a state with pose. We have assumed our h_s values to be consistent and we use the same Dijkstra style learning rule, which are the necessary assumptions for their proof. This means that all the preconditions for their proof have been met and as such, their proof follows trivially. □

This ensures that if using an admissible heuristic, our heuristic will remain admissible, yet become more informed as subsequent search iterations are performed.

Dynamic Heuristic Learning Developing accurate heuristics for predicting the cost-to-go due to dynamic obstacles is a hard problem that, to our knowledge, has not been addressed in the literature. We use $h_d = 0$ for this reason. While this is very weak, we can improve it drastically during the search using the dynamic learning step described in the following section. This is another key advantage of this technique over the competing methods: because we track dynamic and static costs separately, we can learn h_d values through our g_d costs. This will allow future searches to to avoid areas of high cost caused by dynamic obstacles. These h_d values of a state can frequently change with respect to time. The learning rule for the h_d values is:

$$h'_d(n) = [\min_{n' \in succ} g_d(n') + h_d(n')] - g_d(n)$$

where h'_d is the new learned dynamic h value. The intuition here is that a node n's h_d value should be the best $g_d + h_d$ of its children, minus the cost to get to n. The h_d values are computed using a Dijkstra-style traversal of the local search space as in the static world learning step. We may only prune duplicates with identical times, as the time of the state is important in determining its h_d value. The termination condition, however, is the same as in the static learning step.

Heuristic Decay

As mentioned earlier, g_d costs, and hence h_d costs, increase or decrease depending on the movements of dynamic obstacles and their predicted future locations. As in LSS-LRTA*, the learning step of PLRTA* will always only raise the heuristic estimate of a state. However, there must be a way to lower a high cached heuristic value if the dynamic obstacle that caused the high value moves away. To accomplish this, we decay the cached dynamic heuristic values of all states over time. This allows the algorithm to "unlearn" dynamic heuristic values that turned out to be overestimates.

Whenever a h_d value is learned and cached, we note the current planning iteration p_i. Then at some future planning iteration p_j where $i < j$, the value of the cached h_d is decayed linearly so that after some constant $t_d \geq 0$ number of iterations, the value is back to zero and it is removed from the cache. This encourages the search algorithm to potentially re-evaluate the node when it is next generated in some future planning phase instead of using the possibly stale cached value.

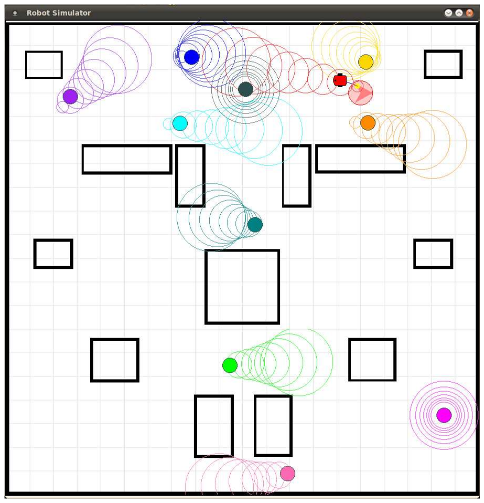

Figure 3: The simulator used showing the dynamic obstacles (circles), predicted trajectories (rings), and the robot (red).

It is important to note that PLRTA* can only guarantee completeness when there are no dynamic obstacles in the world. In a static world, PLRTA* inherits the same completeness guarantees of LSS-LRTA* (which inherited the completeness guarantee of LRTA* (Korf 1990)).

Theorem 4 *In a finite problem space with positive edge costs and finite heuristic values, in which a goal state is reachable from every state, PLRTA* will find a solution if one exists in the discretized space.*

Proof: We rely on the proof of Theorem 1 shown by Korf (1990). One simply substitutes his use of h with h_s and their notion of a state with pose. When dealing with poses, time has been removed from the state and so our problem space becomes finite. We have no negative edge costs. This means that all the preconditions for his proof have been met and as such, his proof follows trivially. □

It is impossible for any algorithm that myopically optimizes our cost function using a naive opponent model to be able to guarantee completeness. For example, consider a situation in which a robot has two possible paths to reach a goal. Whenever it begins to take the shorter path, a dynamic obstacle blocks its path. As the robot reverts to the longer path, the obstacle moves away, luring the robot back to attempt the shorter path again. This endless cycle is inevitable whenever a optimizing planner lacks the ability to estimate future obstacles accurately. In this way, no algorithm can guarantee completeness in the presence of adversarial obstacles. However, this does not diminish the usefulness of developing methods that are effective in practical situations.

Experimental Evaluation

Given the many relevant approaches to motion planning, we evaluate our new algorithms in a simulated environment against five previous proposals. To our knowledge, this is the first time that these diverse algorithms have been empirically compared. We used a custom simulator capable of supporting multiple, physically realistic robots. The simulator had a distributed architecture, allowing planners to run in parallel on remote machines. By having each planner use its own machine, we realistically simulated each robot actively planning while the simulator carried out their previous actions. In our tests, the planners controlled simulated differential drive robots from start to goal locations while avoiding static and dynamic obstacles in trials lasting one minute. Their objective was to minimize the cost function, balancing both their need to avoid obstacles while attempting to stay on their goal position as much as possible.

We ran two different kinds of problems. The first used 36 different random pairings of start and goal positions for the algorithm under test. The number of dynamic obstacles was varied from zero to ten opponents for each of the different start and goal combinations. The paths that the opponents follow are arbitrary paths traced by a human with a pointer device and then stored for reuse. The heuristic used was the 2D Dijkstra heuristic (Likhachev and Ferguson 2009). The maps were discretized into 4cm square cells. The size of the map was 500 by 500 cells, corresponding to a 20 by 20 meter map. The cost of a time step passing was 5 while not on the goal and 0 while on the goal. The cost of a collision was 1000. An example run with one planning bot and 10 dynamic obstacles is shown in Figure 3.

We compared RTR* and PLRTA* with LSS-LRTA* (Koenig and Sun 2009), RTA* (Korf 1990), our modified version of Time-Bounded Lattice (Kushleyev and Likhachev 2009) described above, and our real-time version of the RRT algorithm (Lavalle 1998). The parameters for each algorithm were chosen to offer the best performance based on pilot experiments. The lookaheads used were determined by empirically testing how many nodes the algorithms could expand within the given time limit. PLRTA* was run with a lookahead of 1000 nodes, and a decay steps value of 4. LSS-LRTA* was run with a lookahead of 1000 nodes. RTA* was run with a lookahead depth limit of 4. Time-Bounded lattice was run with a time-bound of 4 seconds and a weight of 1.1. RRT was run with a sampling limit of 500 samples. RTR* was run with an expansion limit of 5000 nodes and an avoid limit of 1000 nodes. The value of k was set to 10 and w was set to 3. The Δ parameter was set to 0.4 meters. RTR* and R* both need a heuristic from any arbitrary node to any other arbitrary node: the straight line heuristic was used because of its speed of computation.

Figure 4 shows a box plot for each algorithm tested when run with 0, 1, 4, 6, 8, and 10 dynamic obstacles. The y axis denotes the actual cost accrued by the agent running the algorithm as reported by the simulator. These box plots display the sample minimum, the lower quartile, median, upper quartile, sample maximum, and outliers. We see that PLRTA* is clearly the best across the board with the exception of instances with no or few opponents where many algorithms perform well. Its partitioned heuristic and learning scheme seem to offer it a great advantage when compared to LSS-LRTA*. RTR* performed much better than R* and comparably to LSS-LRTA* and TBL. RTR* performs relatively poorly on the experiment with no dynamic obstacles. From our observations, RTR* is able to avoid hitting moving obstacles fairly well, but it is unable to quickly get to the goal, even if there are no dynamic obstacles. R* and the real-time version of RRT perform the worst on all the experiments, not being able to plan low cost paths to the goal with no dynamic obstacles and having a high collision rate when dynamic obstacles are present. Time-Bounded Lattice (TBL) performs well when there are few dynamic obstacles, and performs the best of all algorithms when there were no dynamic obstacles. Not being real-time however, it is unable to reliably avoid collisions in the presence of many obstacles. Likewise, RTA* is able to perform well when there are no dynamic obstacles, but its learning does not scale and it is unable to avoid collisions as the number of dynamic obstacles increases, performing about as badly as RRT. We found that in the 10 dynamic obstacle case, the mean cost accrued by PLRTA* was 5.5 to 25 times less than the for other algorithms tested.

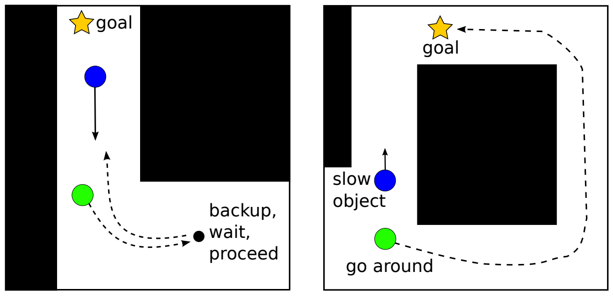

Figure 5: **Left:** The robot first backs up away from the oncoming robot, moving out of its way and allowing it to pass before travelling towards the goal. **Right:** Due to the obstacle moving very slowly, the planner realizes it would be much quicker to take the longer path around.

PLRTA* is the only algorithm to collide with less than 1 obstacles on average. In our challenging benchmarking suite, 1000 is the cost of a collision and PLRTA* stays lower than this over all configurations. We tested a variant in which heuristic decay was not used and found that it achieved the same cost. We believe this is because the learning performed during the dynamic learning stage will only raise the value for specific time stamped states that exclusively lead to states that have high dynamic cost. Because our h_d function is so weak, a node will maintain a h_d of 0 unless all the paths through its descendants have high dynamic cost. In an infinite state space, this is a very small number of nodes and has little effect on the search, despite its theoretical importance in ensuring completeness.

For our second set of experiments we ran the algorithms on 6 specific handcrafted challenge scenarios. These are detailed in more depth by Cannon (2011): two examples are shown in Figure 5. Qualitative ratings of good, ok, and bad were used to describe the performance of the algorithms. Overall, PLRTA* and Time-Bounded Lattice perform the best, with only two and three ratings of bad respectively.

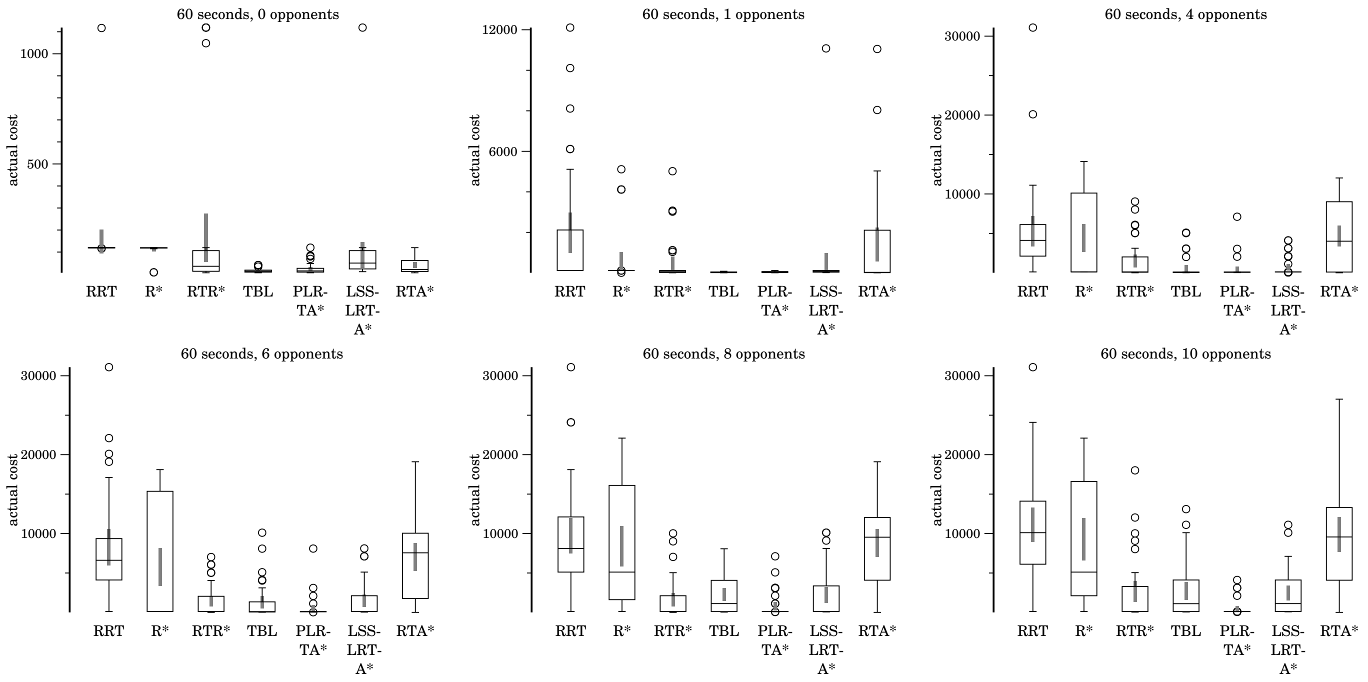

Figure 4: Total trajectory cost over 60 seconds when traveling to a goal with varying numbers of dynamic obstacles.

PLRTA* performs comparably or better than TBL on all but one scenario. However, Time-Bounded Lattice missed the real-time deadline by an average of 10.5 times per planning scenario. This gave it a stuttering behavior, which is not desirable because missing the deadline can easily result in collisions. All other algorithms had at least three ratings of bad and performed much worse overall.

Conclusion

We have presented the first two real-time heuristic search algorithms for kinodynamic motion planning with dynamic obstacles. We investigated two approaches, RTR* and PLRTA*, based on previous successful motion planning and real-time search algorithms, respectively. In the first comprehensive comparison of sampling-based and real-time heuristic search-based methods, the two new algorithms performed equal to or often better than their original progenitors, and PLRTA* surpassed all other algorithms tested. We hope this work furthers the applicability of real-time heuristic search-based methods for fully embodied agents working among humans.

References

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)*, 16–22.

Cannon, J. 2011. Robot motion planning using real-time heuristic search. Master's thesis, University of New Hampshire.

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

Koenig, S., and Likhachev, M. 2002. Improved fast replanning for robot navigation in unknown terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 968–975.

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

Koren, Y., and Borenstein, J. 1991. Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1398–1404.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of IJCAI-85*, 1034–1036.

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

Kushleyev, A., and Likhachev, M. 2009. Time-bounded lattice for efficient planning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 4303–4309. Piscataway, NJ, USA: IEEE Press.

Lavalle, S. M. 1998. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR 98-11, Computer Science Department at Iowa State University.

Lee, J.; Pippin, C.; and Balch, T. 2008. Cost based planning with rrt in outdoor environments. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, 684–689. IEEE.

Likhachev, M., and Ferguson, D. 2009. Planning long dynamically feasible maneuvers for autonomous vehicles. *International Journal of Robotic Research* 28:933–945.

Likhachev, M., and Stentz, A. 2008. R* search. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 1, 344–350. AAAI Press.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1(3-4):193–204.

Rose, K. 2011. Real-time sampling-based motion planning with dynamic obstacles. Master's thesis, University of New Hampshire.