AI Communications 27 (2014) 345–362
DOI 10.3233/AIC-140604
IOS Press

Real-time heuristic search for motion planning with dynamic obstacles

Jarad Cannon, Kevin Rose and Wheeler Ruml *
Department of Computer Science, University of New Hampshire, Durham, NH, USA
E-mails: {jarad.cannon, rose.kevin.jordan}@gmail.com, ruml@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 heuristic search performance on this challenging problem.

Keywords: Motion planning, dynamic obstacles, heuristic search, real-time search, randomized search

1. Introduction

As autonomous robots increasingly become incorporated into 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 perform 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* [15], 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* [7] that allows it to handle state spaces like those encountered in 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 several current motion planning and real-time search algorithms, with the relatively simple PLRTA* offering the best performance.

*Corresponding author: Wheeler Ruml, Department of Computer Science, University of New Hampshire, Durham, NH 03824, USA. E-mail: ruml@cs.unh.edu.

2. Background

The problem of motion planning can be formulated in several different ways [2,12]. 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 heading and speed to ensure that it doesn't, for example, try to turn sharply at high speed. 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 it can result in highly suboptimal plans or even render problems unsolvable. To avoid this, we follow Kushleyev and Likhachev [10] and 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.

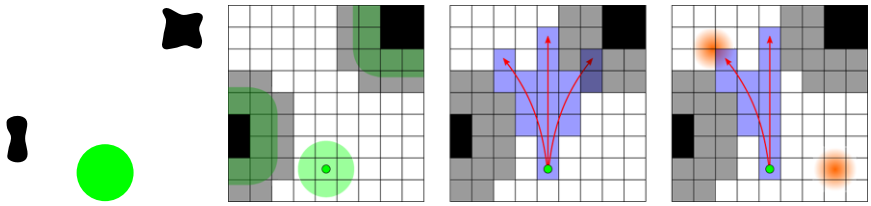

Fig. 1. Generating the search space using (b) discretization, (c) motion primitives and collision checking and (d) dynamic obstacles. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

2.1. The state space

In the example domain used in this paper, the robot is a non-holonomic differential drive vehicle. The capabilities of the robot are represented by a set of motion primitives, each of which specifies a possible change in the robot's state over a fixed time duration (in our work, 0.5 s). The environment is represented as a grid of fixed-size cells, with obstacles represented as blocked cells. Moving into a blocked cell is not permitted and, as detailed below, moving into a cell that intersects the predicted position of a dynamic obstacles incurs extra cost. Figure 1 illustrates how the motion primitives, the environment, and the dynamic obstacles give rise to the search space of the planner. Panel (a) shows an environment with the robot and two static obstacles. In Panel (b), the environment is discretized and the static obstacles are inflated by the radius of the robot, allowing us to treat the robot as a point. Panel (c) shows the motion primitives as arrows, along with the cells that they intersect, which must be checked for collisions. (In our implementation, the relative positions of these cells are pre-computed for each primitive to save time during planning.) Finally, in Panel (d), the remaining actions are assigned costs, with higher costs for those that intersect the predicted future location of a dynamic obstacle, depicted by the orange circular gradients.

2.2. 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 $a : S \to S$ that maps states to states and has a duration of t_a. In our work, all actions have the same t_a. The function $\alpha : S \to Q$, $Q \subseteq A$ 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 $d : t \to \mathcal{N}$ from time to a bivariate Gaussian distribution over x and y representing the object's location. These functions can change across planning episodes 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 currently executing motion primitives, t_a, so that the robot will always have the next action to execute by the time it completes its current action. Figure 2 illustrates the interaction in more detail. When the robot (or more precisely, the simulator) starts executing action a_t at time t, it sends the planner its prediction of the state of the world at $t+1$. The planner selects an appropriate action using whichever planning algorithm it implements, but must report its selection in time to be executed at $t+1$.

Our problem setting models a physically embodied robot that does not disappear once it reaches a goal. For example, it is undesirable to reach the goal safely but then undergo a collision a moment later. To capture this 'lifelong planning' setting, 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}.

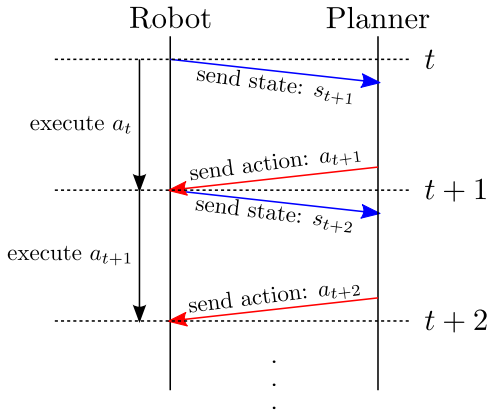

Fig. 2. The interaction between the robot simulator and each planner. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

Given an action that transitions between two states, the total cost of the action, C, is defined as:

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

where $P(col)$ is the probability of a collision occurring with any of the dynamic obstacles and *status* is zero if the start state of the action is a goal and one otherwise. Following Kushleyev and Likhachev [10], the probability $P(col)$ is computed assuming that the events of not colliding are independent. In this way, both dynamic costs ($P(col) \cdot C_{col}$) and static costs ($status \cdot C_{time}$) are considered in our cost function. To create the probability distributions representing the dynamic obstacles' trajectories, we simply used linear interpolation based on previous recorded states of each dynamic obstacle. This method is very fast to compute and performed adequately for our experiments. More advanced methods such as Kalman filters [10] could certainly be used.

3. Previous work

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

3.1. 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 [8].

3.2. 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) [11] 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.[1] 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 [13].

3.3. Heuristic search

Heuristic search methods solve planning problems by considering them as shortest-path problems in graphs. While Dijkstra's algorithm [3] can usually be used to solve shortest-path problem, motion planning is similar to other problems in AI in that the graph is too large to explicitly instantiate and is instead generated lazily. We call the process of generating a node's successors *expanding* the node. Furthermore, heuristic search uses a heuristic function $h(n)$ that provides a lower bound on the cost from node n to a goal. While Dijkstra's algorithm uses $g(n)$, the cost of arriving at n, to order its expansion of nodes, heuristic search algorithms such as A* [4] compute a lower bound $f(n)$ on the cost of any plan passing through node n, as $f(n) = g(n) + h(n)$, and order their node expansion on $f(n)$. A* finds optimal solutions and is not suitable for real-time or dynamic settings.

D* Lite [6] is an incremental heuristic 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

[1] The RRT* variant [5] eventually converges to optimality but requires expensive 're-wiring' steps after each sample.

the state, as it is unknown what time the robot will actually arrive at the goal.

The Safe Interval Path Planning algorithm (SIPP) [16] allows for bounding the number of distinct time steps seen by the search. It is a method for reducing the size of the search space by not searching over distinct values of time but instead distinct *safe intervals*. A safe interval for a given location is the period of time such that there is no dynamic obstacle in the location for the entire interval, however there is an obstacle in the location at one time step before the interval and one time step after the interval. A* search is used to generate plans over the state space discretized by time intervals and has been shown to visit many fewer nodes. There are two assumptions that this algorithm relies on that may or may not be true for a given robot. First, it is assumed that the robot is capable of waiting in place for an arbitrary amount of time. This may not be the case if the robot requires movement to remain in a stable state, such as a motorcycle or an airplane. Second, it is assumed that the acceleration of the robot is negligible, i.e. robot can speed up or slow down instantaneously. SIPP is not constrained to generate paths that are dynamically feasible for the robots simulated in this paper, so it is not considered in our empirical evaluation.

The Time-Bounded Lattice algorithm (TBL) [10] was designed for the problem representation we consider in this paper. The idea is to do weighted A* search [17] 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.

3.4. Real-time heuristic search

In this paper, we address the problem of 'hard real-time' planning, in which an action must always be selected within a prespecified amount of time, as opposed to 'soft real-time' planning, in which one is satisfied with an algorithm that often executes quickly but that may sometimes take longer, which may be sufficient in some contexts to give the appearance of responsive behavior. Thus, in this paper, a real-time search method can only perform a bounded number of node expansions before it must return a plan. This has the consequence that a complete path to a goal node may not have been computed. A real-time search returns only the single next action to take, rather than a complete plan to a goal in the way that A* does. Because they may lead the agent down a blind alley or into a cycle, real-time searches must be able to learn from their experience and improve their choices if they return to a previously visited state. One way to view this behavior is as 'filling in' heuristic values that are too low by learning more accurate values.

Real-time A* (RTA*) [9] forms the basis of many other real-time algorithms. It works by first initializing the search tree with the agent's current state as the root node. This root node is assigned a g value of zero. The algorithm then generates the successors of the root node and then does 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 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 that 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.

4. A Sampling-based approach: 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 [15].

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 consisting of a high-level graph and a low-

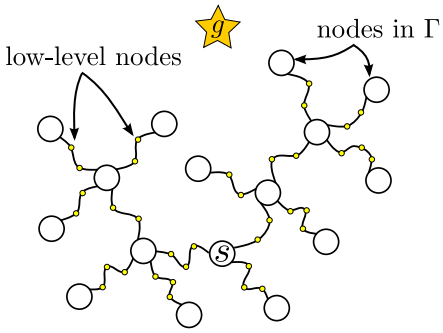

Fig. 3. The nodes in Γ (large, white) and the nodes in the low-level state space (small, yellow) that are explored by an R* search with $k = 3$. (The colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

level graph, where the higher level states are generated randomly and sparsely over the state space. To compute the cost and the actual path between two high-level states, a low-level search is performed in the original state/action space. 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 high-level, 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 Fig. 3). 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. In this way, 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.

Pseudocode of the R* algorithm is shown in Algorithm 1. The main loop of the R* algorithm is similar to a best-first search such as A*. First, the best node on the open list is removed (line 5). The ordering function for the open list first prefers nodes that have not been labeled AVOID. It then prefers nodes with lower f value, with ties broken on lower h value.

In R*, there are two types of nodes in Γ that can be popped off the open list: the node can either be lacking a low-level path from its parent or already have one computed. In the first case, in which a path hasn't been found, R* uses a bounded weighted A* search to find one (line 11). If the search succeeds, then the g cost of the node is updated to reflect the cost of the path that was found (line 15), the node is updated in OPEN, and

Algorithm 1. Pseudo-code for the R* algorithm

```
R*(s_start, s_goal)
1. OPEN ← ∅, CLOSED ← ∅
2. g(s_start) ← 0
3. insert s_start into OPEN
4. while OPEN ≠ ∅ and
        pri(s_goal) ⩾ arg min_{s'∈OPEN} (pri(s'))
5.     remove s with the smallest priority from OPEN
6.     if s ≠ s_start and path(pred(s), s) = null
7.         reevaluate(s)
8.     else
9.         expand(s)
10. return incumbent solution if found,
        impossible otherwise

Re-evaluate(s)
11. path(pred(s), s) ← wA*(pred(s), s)
12. if path = null or
        g(pred(s)) + path.cost) > w · h(s_start, s)
13.    avoid(s) ← true
14. pred(s) ←
        arg min_{s'|s∈SUCCS(s')} (g(s') + pathcost(s', s))
15. g(s) ← g(pred(s)) + pathcost(pred(s), s)
16. insert/update s in OPEN

Expand(s)
17. if is_goal(s) and g(s) < g(incumbent)
18.    incumbent ← s
19. insert s into CLOSED
20. SUCCS(s) ← k random states a distance Δ from s
21. if distance(s, s_goal) ⩽ Δ
22.    SUCCS(s) ← SUCCS(s) ∪ s_goal
23. SUCCS(s) ← SUCCS(s)−
                SUCCS(s) ∩ CLOSED
24. for all s' ∈ SUCCS(s)
25.    pathcost(s, s') ← h(s, s')
26.    if s' hasn't been generated before or
            g(s) + h(s, s') < g(s')
27.        pred(s') ← s
28.        g(s') ← g(s) + pathcost(s, s')
29.        insert/update s' on OPEN
```

the search continues. If a path is not found, this is because the node expansion limit was reached, indicating that this subproblem may be hard to compute (line 13). In this case, the weighted A* search will return the cost of the best node on the frontier. This cost is used to update the g value of the node with a better estimate of the true path cost. In the second case in the main loop (line 8), in which a low-level path already exists to the node, the node is just expanded. This involves randomly generating k successors that are a distance Δ away (line 20). The goal state is also added to the list of successors if it is within this distance (line 22). These nodes and edges are then added to the sparse graph Γ (line 29) and the search continues.

It is worth mentioning that in R*, the g values of the nodes in Γ are conceptually made up of two parts. Let n and n' be two nodes in Γ where n is the parent of n'. The first portion of the g value of n' consists of the g value from the start state to n. Since n is guaranteed to have a low-level path to it, this value represents a real cost, not an estimate. The second portion of the g value of n' is the portion that represents the cost of the edge from n to n' in the high-level graph. Initially, the true cost of this edge is unknown because a low-level path has not been computed. In this case, the cost is estimated by using a heuristic. Note that the heuristic used must be capable of admissibly estimating the cost of the path between any two nodes, not just from any node to the goal node. Once R* computes the low-level path between n and n', this estimated porion of the g value will be updated (line 15). If a low-level path is found, then the g value will be updated to be the cost of the path. If a path is not found, due to the expansion limit, then the g value will be updated to be the f value of the best low-level node that was on the open list when the low-level search was terminated, since this will be a lower-bound on the true cost of the complete low-level path.

5. Making R* real-time

R* has advantages over traditional real-time heuristic search algorithms that make it attractive as a foundation for a real-time planning algorithm. Real-time heuristic search algorithms deal with the problem of not being able to plan complete paths to the goal by using information gathered from previous search iterations to escape from heuristic local minima and find a path to the goal. In the case of the robot motion planning domain, where there is a large search space and a high branching factor, the lookahead performed by traditional A* style real-time search, such as LSS-LRTA*, may not by able to see far enough into the future to make informed decisions about what action to take. One way to increase the depth of the A* lookahead used would be to reduce the branching factor of the search space by limiting the number of actions available to the robot. This however reduces the quality of plans returned, since fewer actions may be used, possibly even making the problem unsolvable. The size of the state space may also be reduced by increasing the size of the discretization used, for example in the size of the static obstacle or cost grid. This also has the adverse effect of reducing the quality of the plans returned and again possibly making the problem unsolvable. R* is able to deal with high dimensional state spaces by splitting the problem up into smaller, easier to solve subproblems. This does not reduce the size of the action set available to the robot, nor does it increase the size of the problem discretization.

While R* has been shown to perform well in many hard domains, it is not a real-time algorithm. R* must find complete paths to the goal on every search, and the time that this takes is not bounded. We made five major changes to transform R* into real-time R* (RTR*). We will discuss each in turn.

5.1. Limiting expansions

To meet the real-time constraint, 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 after it has been achieved to get out of the way of a dynamic obstacle.

5.2. Action selection

After each iteration of RTR*, an action to perform must be selected. As in other real-time searches [9], RTR* chooses 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.

5.3. 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. (Note that this node-specific limit is different and secondary to the overall real-time search limit.) In 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 worst 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. Suppose there is a state s and its successor state s' in the sparse graph Γ. Suppose that R* must compute the path between s and s' to reach the goal. Let the number of low-level nodes that must be expanded by weighed A* to compute this path be n. In the worst case, the series of weighted A* searches that uses a node expansion limit that doubles will expand $n - 1$ nodes on its second to last iteration before expanding n nodes on its last iteration. In addition to these last two searches, the total number of nodes expanded by weighted A* on all previous iterations will be:

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots \approx n.$$

So in total, $n + (n - 1) + n \approx 3n$ nodes will be expanded. Since a weighted A* search that does not use an expansion limit will visit n nodes, the overhead of using the doubling technique is bounded by a constant factor of approximately three in the worst case. Now suppose that there are k of these paths that must be computed in the whole problem. The number of nodes expanded using doubling will be $3nk$ in the worst case, versus nk when not using doubling, so the constant factor remains the same. □

5.4. Path reuse

Because real-time search interleaves planning and execution, RTR* attempts to leverage previous planning effort by caching 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. Figure 4 shows an example of how the path saving mechanism works across iterations. On the left, RTR* has reached its expansion limit and calculated the best node on the frontier and the corresponding action to take along the path (shown in red). The nodes in Γ that exist along the path (green) are added to the initial sparse graph in the next iteration of searching, shown on the right. The gray dashed lines between the nodes on the right show the edges in the sparse graph and indicate that a low-level path between them has not yet been computed. The low-level paths will be 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. If costs have changed, then RTR* is free to recompute a better path, possibly not even using those nodes or edges in the sparse graph at all.

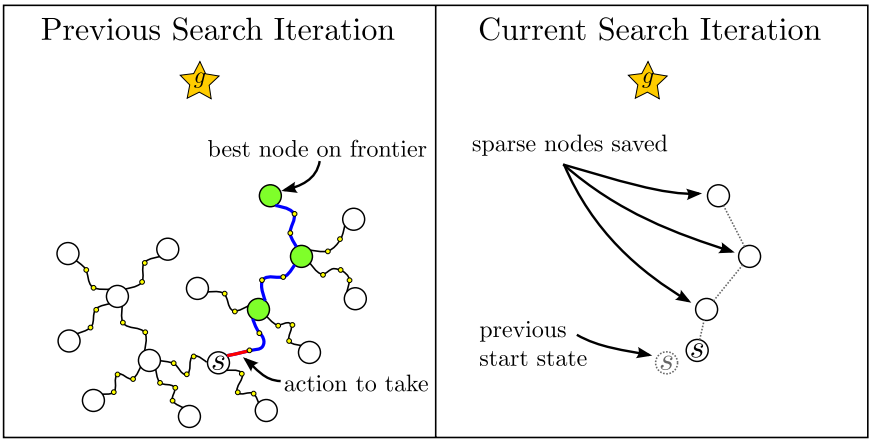

Fig. 4. Path saving across iterations of RTR*. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

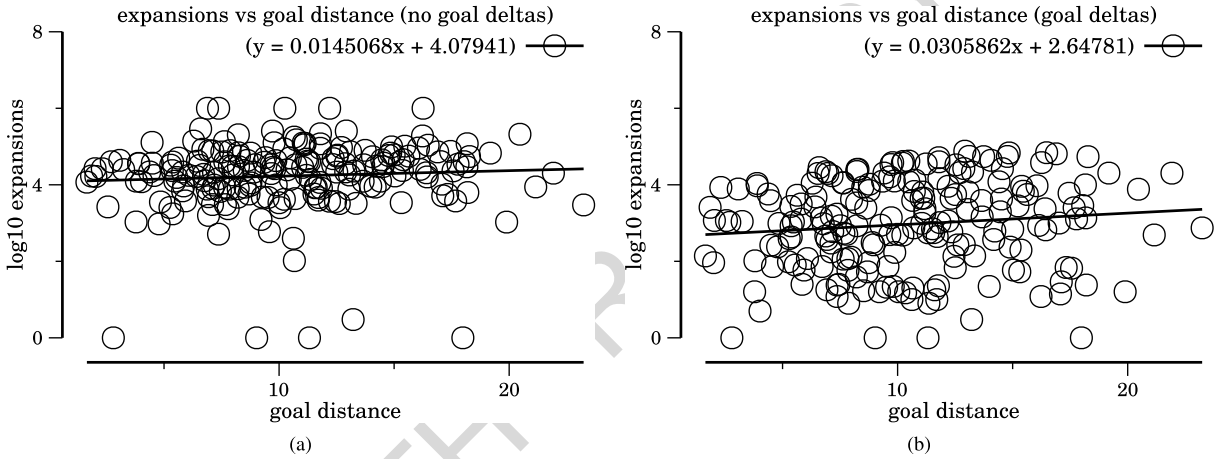

Fig. 5. The log of the number of nodes needed by weighted A* to solve problems with and without a goal radius allowed. The x axis is the distance between the start and goal locations.

5.5. *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 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 [15] 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. (An intuitive example of this phenomenon is the task of parallel parking a car.) 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 (Fig. 5(a)). To make these problems easier, the goal condition used for the low-level weighted A* searches was relaxed to allow any state within some distance

0.5 meters of the actual goal state and with any heading and any speed to be considered a goal. Figure 5(b) 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. This relaxed goal condition allows RTR* to solve subproblems much more quickly.

6. A real-time search approach: LSS-LRTA*

In the previous section, we developed RTR* by altering a leading motion planning search algorithm to be real-time. In this section, we pursue the opposite approach: adapting a state-of-the-art real-time search algorithm, LSS-LRTA* [7], to the problem of robot motion planning with dynamic obstacles.

Recall the basic RTA* algorithm from Section 3.4: it performs a depth-limited lookahead to assess the value of each next action, then updates the heuristic value of the current state as it transitions to the next state. Local Search Space Learning Real Time A* (LSS-LRTA*) extends these ideas into a state-of-the-art real-time heuristic search algorithm. Pseudo-code for the algorithm is sketched in Algorithm 2. It works by first performing a node-limited A* search [4] from the current state towards the goal, in contrast to RTA*'s depth-bounded lookahead. This is illustrated graphically in Fig. 6. The search frontier contains all nodes that have been generated but not yet expanded. Once the node expansion limit has been 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 the part of the algorithm responsible for "learning"

Algorithm 2. Local search space learning real-time A*

LSS-LRTA*(s_{start}, *lookahead*)
1. open = $\{s_{start}\}$
2. closed = $\{\}$
3. ASTAR(open, closed, *lookahead*)
4. $g' \leftarrow$ peek(open)
5. LEARN_H_VALUES(open, closed)
6. return first action along path from s_{start} to g'

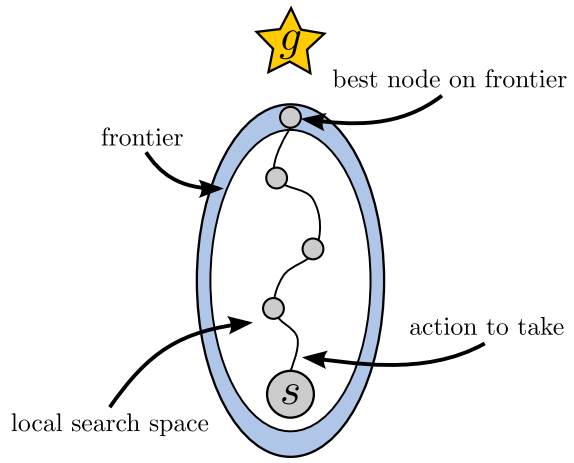

Fig. 6. The local search space and frontier of an iteration of LSS-LRTA*. The best node on the frontier and the corresponding action to take are shown. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

the improved h values. This is in contrast with RTA*, which only updates one h value per iteration.

To our knowledge, LSS-LRTA* had previously been tested only on simple grid world path 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* moves the agent 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 [1]. Second, the h values learned for nodes will never decrease [7]. 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. To see this, note that the g values along a path are estimates of the costs for future actions, which can be overestimates if they include costs due to predicted collisions with dynamic obstacles that might move away before the agent reaches their location. This makes this problem domain fundamentally different than the domains these algorithms have been applied to in the past. As we show in the evaluation below, LSS-LRTA* can struggle in motion planning.

7. Partitioned learning real-time A*

Our central modification to LSS-LRTA* is to separate components of the cost function to make learning more effective in this domain. In order to more effectively learn heuristic costs, we partition g and h values into *static* and *dynamic* portions. The static portion refers to only those costs that are not time dependent, that is, the cost incurred due to static obstacles in the world, e.g. walls. The dynamic portion refers to only those costs 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:

$$f(n) = g_\mathrm{s}(n) + g_\mathrm{d}(n) + h_\mathrm{s}(n) + h_\mathrm{d}(n).$$

Tie-breaking prefers higher g_s values. The static costs (both g_s and h_s) are those that assume a world with no dynamic obstacles, only static obstacles. The dynamic costs (both g_d and h_d) assume a world with no static obstacles, only dynamic obstacles.

We call the resulting algorithm Partitioned Learning Real-time A* (PLRTA*). PLRTA* generalizes static costs across states with the same pose. Although one could certainly design generalization policies that propagate dynamic costs to nearby times, in this study we simply keep dynamic costs specific to individual time-stamped states. Like LSS-LRTA*, PLRTA* 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 update heuristic values in a manner much like LSS-LRTA*, described in detail in the next 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 3.

Algorithm 3. Partitioned learning real-time A*: main loop and h_s learning

```
PLRTA(s_start, lookahead)
1. open = {s_start}
2. closed = {}
3. ASTAR(open, closed, lookahead)
4. g' ← peek(open)
5. LEARN_STATIC(open, closed)
6. LEARN_DYNAMIC(open, closed)
7. return first action along path from s_start to g'

Dijkstra_hs(Closed, Open)
8. Closed, Open ← InitDijkstra_hs(Closed, Open)
9. while Closed ≠ ∅ and Open ≠ ∅
10.   remove a state s with minimal h_s from Open
11.   if s ∈ Closed
12.       Closed ← Closed \ {s}
13.   for p ∈ predecessors(s)
14.       if p ∈ Closed and h_s(p) > c_s(p, s) + h_s(s)
15.           h_s(p) ← c_s(p, s) + h_s(s)
16.           if p ∉ Open
17.               Open ← Open ∪ {p}

InitDijkstra_hs(Open, Closed)
18. Closed' = {}
19. for n ∈ Closed
20.   if n ∉ Closed'
21.       h_s(n) ← ∞
22.       Closed' = Closed' ∪ {n}
23. for all n ∈ Open
24.   if n ∈ Closed'
25.       remove n from Open
26. return Closed', Open
```

7.1. Partitioned learning

We now describe the partitioned learning technique in more detail. For simplicity, we divide heuristic learning into separate steps for h_s and h_d. Both steps are closely modeled on the heuristic update procedure of LSS-LRTA*. The main difference between the two learning steps is in the setup phase. The static learning phase (function LEARN_STATIC) is used to learn the h_s values of the world. To do this, it makes use of the Dijkstra$_{hs}$ function. We start by sorting the open list by lowest h_s. The pseudocode for this is shown in Algorithm 3 (lines 18–26). 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. This is shown in the Dijkstra$_{hs}$ function (lines 9–17). 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. The following Theorem explains why.

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 learning algorithm terminates that has some successor that does not exclusively lead to a dead-end. The algorithm must have terminated due to an empty open list as the algorithm only terminates when either the open list or the closed list becomes empty. This means that n must have had a descendant (either a direct successor or a node along the path going through a direct successor) 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 descendants do not lead exclusively to dead ends. Therefore, 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:

$$h_s\big(\mathit{pred}(m)\big) < c_s\big(\mathit{pred}(m), m\big) + h_s(m)$$

will hold for any of the predecessors, *pred*(m), on the closed list at least once, as all nodes on the closed list have their h_{s} values set to ∞ in line 21 of 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 a dead-end. Therefore, n must exclusively lead to dead-ends. □

In fact, the h_{s} learning step is so similar to the learning step of LSS-LRTA* that we inherit their result that 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 [7]. 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 so their proof follows directly. □

Theorem 4. *The h_{s} values remain consistent and thus also admissible.*

Proof. We rely on the proof of Theorem 2 shown by Koenig and Sun [7]. Once again the substitutions in our proof for Theorem 3 are used and meet all the necessary assumptions for their proof. This means that all the preconditions for their proof have been met and so their proof follows directly. □

Theorems 3 and 4 together ensures that if using an admissible heuristic, our heuristic will remain admissible, yet become more informed as subsequent search iterations are performed.

7.1.1. 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. For this reason, we use $h_d = 0$. While this is very weak, our partitioned values can improve it during the search using the dynamic learning step. This is a key advantage of PLRTA*: 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 avoid areas of high cost caused by dynamic obstacles. Thus the h_{d} values of a state can change over time as predicted obstacle trajectories change. The

learning rule for the h_{d} values is:

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

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 analogous to that used 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.

7.2. Heuristic decay

The g_{d} costs in the search space, 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 turn out to be overestimates.

Whenever an 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 $t_d \geqslant 0$ number of iterations, the value is back to zero. 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.

This has a few obvious drawbacks. First, we do not decay based on any additional information about the dynamic obstacle; we simply linearly decay the learned value. Second, we are at the mercy of the obstacle prediction model. In this paper we use a simple linear projection. This can be inaccurate and can cause us to learn high h_{d} values for states that are in reality much safer. Third, we only learn these values for specific timestamped states. These updated states will represent a very small fraction of the overall space and thus do not help guide the search as much as they would if their values were generalized to more neighboring states. It is certainly likely that more sophisticated heuristic decay techniques could be leveraged to improve the performance of the search.

It is important to note that, like any real-time search, PLRTA* can only guarantee completeness when there are no dynamic obstacles in the world. It is impossible for any algorithm that myopically optimizes our cost function using a naive opponent model to be able to guarantee completeness. To see this, 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.

In a static world, PLRTA* inherits the same completeness guarantees of LSS-LRTA*, which in turn inherited the completeness guarantee of LRTA* [9].

Theorem 5. *In a finite problem space with no dynamic obstacles, 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 search space.*

Proof. We rely on the proof of Theorem 1 shown by Korf [9]. Note that, with no dynamic obstacles, $g_d = 0$ for all states and thus $g = g_s$ and $h = h_s$. One simply substitutes Korf's 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 thus his proof follows directly. □

Of course, the algorithms discussed in this paper are intended for use in worlds with dynamic obstacles, and so we now turn to an experimental evaluation of their performance.

8. Experimental evaluation

Given the many relevant approaches to motion planning, we evaluate our two new algorithms in a simulated environment against five previous proposals. We

compared RTR* and PLRTA* with LSS-LRTA* [7], R* [15], RTA* [9], our modified version of Time-Bounded Lattice [10] described above, and our real-time version of the RRT algorithm [11]. 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 has 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 (as depicted earlier in Fig. 2). 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. 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. Note that robots will incur collision penalties even while in the goal state. Algorithms that are capable of doing look-ahead past the goal state should strive to leave the goal to let an obstacle pass and then return to it. Algorithms that are incapable of this look-ahead may be penalized for collisions that occur on the goal.

The parameters for each algorithm were chosen to offer the best performance based on pilot experiments. The lookahead used was determined by empirically testing how many nodes the algorithms could expand within the given time limit of 0.5 s. 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 s 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. Specifically, the heuristic is the straight-line distance, divided by the maximum distance the robot can travel in a single time step, multiplied by the per-time-step penalty of not being on the goal.

We ran two different kinds of problems. The first were synthetic problems with random state and goal locations and varying numbers of dynamic obstacles. The second were small handcrafted scenarios designed to test whether the planners could find specific desired behaviors. We will cover each type in turn.

8.1. Randomized runs

The random synthetic experiments used a single fixed environment (Fig. 7). We used 36 different random pairings of start and goal positions. The number of dynamic obstacles was varied from zero to ten obstacles for each of the different start and goal combinations. The paths that the obstacles follow were arbitrary paths traced by a human with a pointer device and then stored for reuse. The robots have no knowledge about the future actions of the dynamic obstacles. They can only estimate their trajectories based on the obstacles' past observed states. The heuristic used was the 2D Dijkstra heuristic [14]. It is made admissible by dividing each value reported by the Dijkstra search by the maximum distance the robot can travel in a single time step, and then multiplying this value by the per-time-step penalty for not being on the goal. The map was discretized into 4 cm square cells. The size of the map was 500 by 500 cells, corresponding to a 20 by

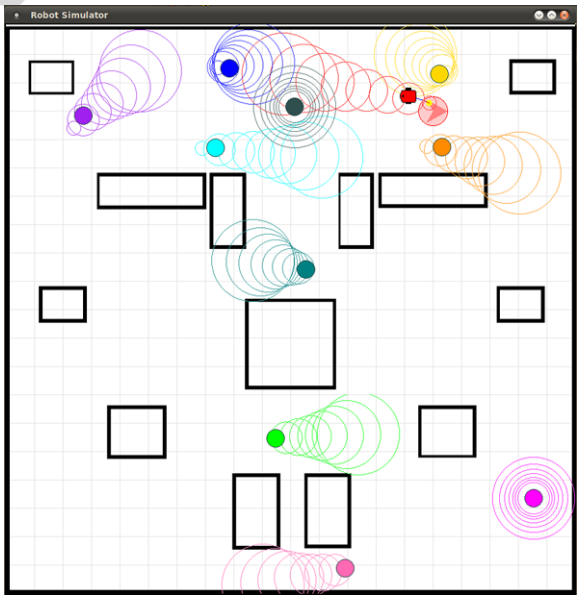

Fig. 7. The simulator used showing the dynamic obstacles (filled circles), the robot (red filled circle), and the goal (light red filled circle with arrowhead, near robot). Hollow circles indicate probability distributions over predicted future trajectories of obstacles. (The colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

20 meter map. An example run with one planning bot and 10 dynamic obstacles is shown in Fig. 7.

Figure 8 shows a line plot of the mean total cost accrued for each algorithm as the number of dynamic obstacles increases. Error bars show 95% confidence intervals around the mean (offset for legibility). Although all algorithms perform well when no dynamic obstacles are present, clear differences emerge as they are added. RTR* clearly surpasses the original R* in these tests, and PLRTA* surpasses the original LSS-LRTA*. Overall, PLRTA* performs the best, with RTR* and LSS-LRTA* performing comparably and second best.

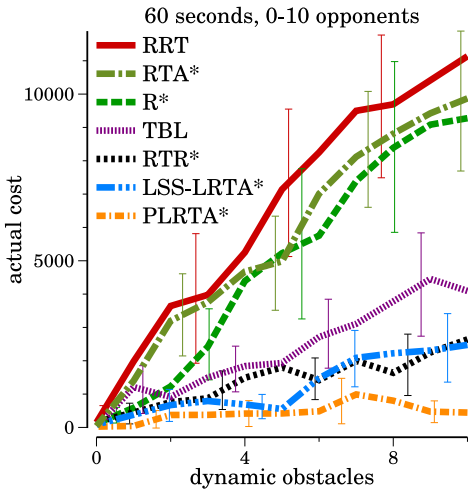

Fig. 8. Experiments of length 60 s. Shows actual cost accrued versus number of dynamic obstacles for each algorithm with 95% confidence intervals. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

Figure 9 shows a subset of the data (0, 1, 4, 6, 8 and 10 dynamic obstacles) in a more detailed box plot format. The y axis denotes the actual cost accrued by the agent running the algorithm. Each box encloses the middle 50% of the data, with a line at the median, whiskers extending to the sample minimum and maximum, and outliers plotted with circles. Gray rectangles (sometimes too small to discern) indicate 95% confidence intervals on the mean. From these plots, we again see that PLRTA* is clearly the best across the board, with the exception of instances with no or few obstacles 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 ob-

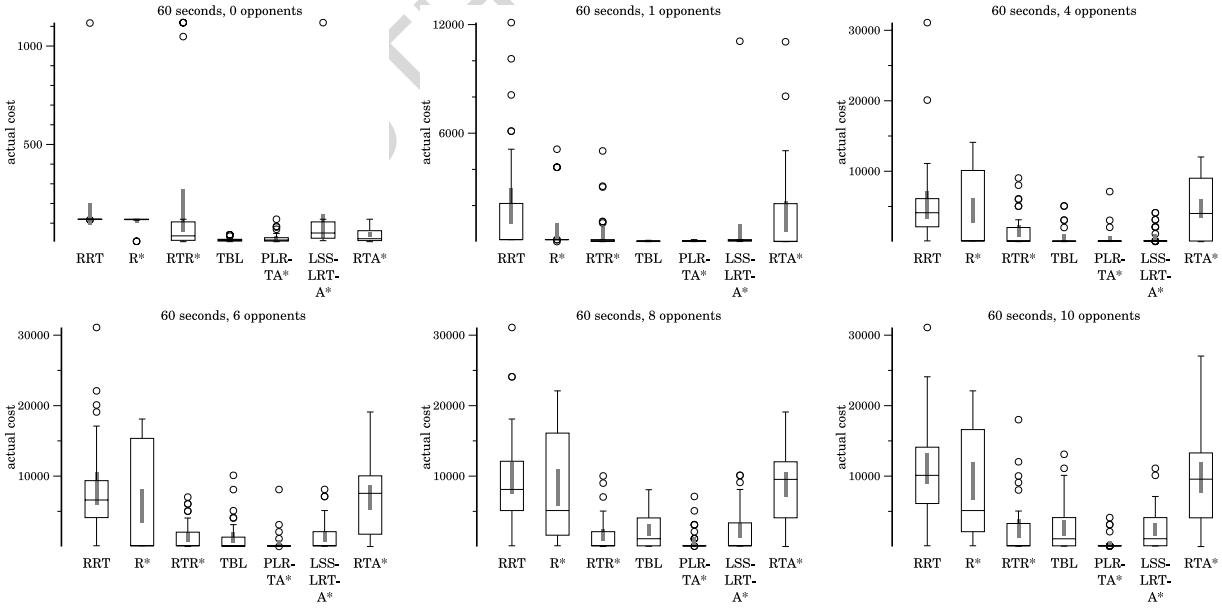

Fig. 9. Total trajectory cost over 60 s when traveling to a goal with varying numbers of dynamic obstacles.

stacles 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 5.5–25 times less than that of the other algorithms tested.

PLRTA* is the only algorithm to collide with less than 1 obstacle on average. In our challenging benchmarking suite, 1000 is the cost of a collision and PLRTA* stays lower than this over all configurations. To help determine which aspect of the algorithm most contributed to its performance, 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 lead exclusively 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.

8.2. Handcrafted scenarios

We also compared our best algorithm, PLRTA*, against the two best competitors, LSS-LRTA* and TBL, on six handcrafted challenge scenarios. The scenarios are shown in Fig. 10, numbered from 1 to 6, left-to-right starting in the upper left corner. Each scenario lasts only 30 s. The static environments tested in the scenarios are smaller than those of the random runs in order to isolate performance on these single scenarios.

Table 1 shows the results of all these runs. The table shows the actual cost incurred, the number of expansions performed, and a qualitative assessment of how each agent behaved while acting in each specific scenario. This is qualified with three possible assessments: good, OK and bad. As we can see in Table 1, not all those plans that have low cost necessarily look good. As a human observer, it can sometimes be hard to understand why the agent is behaving in a certain way. For example, in Scenario 1, the Time-Bounded Lattice algorithm freezes numerous times, as it takes too long

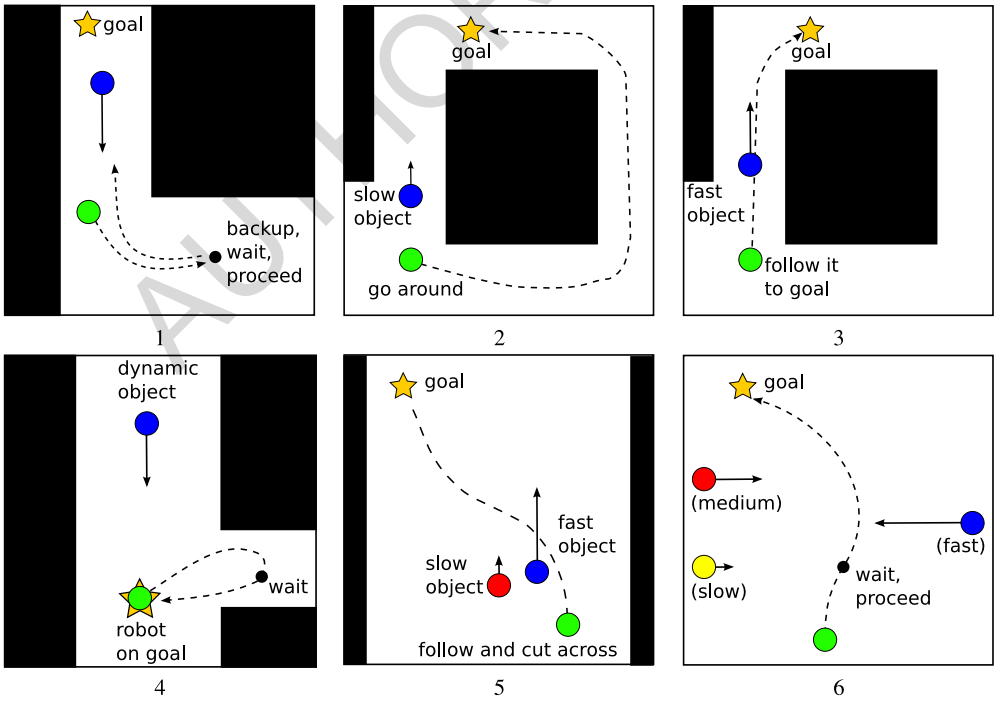

Fig. 10. Handcrafted challenge scenarios, showing movement of the obstacles and the optimal behavior for the agent. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140604.)

Table 1
Experimental results on the handcrafted challenge scenarios

Algorithm	Actual cost	Nodes expanded	Look
	Scenario 1		
tb_lattice tb:4000 w:1.0	29	272,349	ok
tb_lattice tb:4000 w:1.1	30	290,356	ok
plrta lh: 1000	37	60,000	ok
lsslrta lh: 1000	60	60,000	bad
	Scenario 2		
tb_lattice tb:4000 w:1.0	30	1,420,197	good
tb_lattice tb:4000 w:1.1	41	137,395	ok
plrta lh: 1000	60	60,000	ok
lsslrta lh: 1000	60	60,000	bad
	Scenario 3		
tb_lattice tb:4000 w:1.0	60	797,715	bad
tb_lattice tb:4000 w:1.1	60	669,209	bad
plrta lh: 1000	32	60,000	good
lsslrta lh: 1000	60	54,003	bad
	Scenario 4		
tb_lattice tb:4000 w:1.0	3000	60	bad
tb_lattice tb:4000 w:1.1	3000	60	bad
plrta lh: 1000	26	60,000	good
lsslrta lh: 1000	54	60,000	ok
	Scenario 5		
tb_lattice tb:4000 w:1.0	22	115,827	good
tb_lattice tb:4000 w:1.1	22	123,933	good
plrta lh: 1000	60	60,000	bad
lsslrta lh: 1000	60	60,000	bad
	Scenario 6		
tb_lattice tb:4000 w:1.0	60	651,882	bad
tb_lattice tb:4000 w:1.1	60	600,860	bad
plrta lh: 1000	41	60,000	ok
lsslrta lh: 1000	46	60,000	ok

to compute the action to take. Even once unimpeded paths to the goal are present, it sometimes takes multiple planning phases to pass before an action to take is returned. Also, PLRTA* oscillates back and forth between plans while moving to the goal, giving it a look of indecisiveness.

In Scenario 2, the Time-Bounded Lattice finds the long path around the static obstacle and reaches the goal fairly quickly, although it does freeze a few times along the way. LSS-LRTA* never makes it around the static obstacle and instead moves indecisively around the starting area. PLRTA* finds the path around the static obstacle and reaches the goal quickly, yet struggles in trying to arrange itself perfectly on the goal state.

Table 2
Totals over all the handcrafted challenge scenarios

Algorithm	Actual cost	Nodes expanded	Look
tb_lattice tb:4000 w:1.0	3201	3,258,030	ok
tb_lattice tb:4000 w:1.1	3213	1,821,813	ok
plrta lh: 1000	256	360,000	good
lsslrta lh: 1000	340	354,003	bad

PLRTA* really shines in Scenario 4, as it waits on the goal as long as it can before moving out of the way, letting the dynamic obstacle pass, and then returning back to the goal. LSS-LRTA* moves out of the way on this scenario as well, yet never returns to the goal afterward.

In Scenario 4, the Time-Bounded Lattice agent fails to move off of the goal, even though a dynamic obstacle was known to be coming towards it. This is again because of the fact that Time-Bounded Lattice was designed to run until it expands the goal during the search. Thus, it was not entirely clear how to convert this into an algorithm which plans beyond the goal. As explained earlier, we simply expand one node if the agent begins on the goal state and move to the child with the lowest f. This is clearly not enough lookahead for the agent to escape and as such, it decides to continue sitting on the goal. Clearly, this is not a desirable result.

To summarize the performance in these scenarios, Table 2 shows the accumulated total cost. Obviously, the cost of the Time-Bounded Lattice's performance in Scenario 4 skews these results. Ignoring them, however, you can see it did not outperform PLRTA* or LSS-LRTA*by a significant margin. Also notable is the significant amount of additional work Time-Bounded Lattice has to perform in terms of nodes expanded to achieve these costs. The Time-Bounded Lattice with a weight of 1.0 does nearly 10 times as many expansions as PLRTA*, even though it only does one expansion per planning iteration once it reaches the goal.

Table 2 also gives an aggregate qualitative assessment of each algorithm's performance. The Time-Bounded Lattice agents work well in most cases, yet cannot deal with the situation of needing to leave their goal location. This resulted in a collision in Scenario 4. LSS-LRTA* performs the worst overall despite never colliding with any obstacles. This is because it made a large number of seemingly unintelligent moves is most scenarios. PLRTA* performs the best, never colliding with dynamic obstacles, and coming up with reasonable looking plans.

This is a positive result as even though PLRTA* is only doing a limited amount of lookahead search, it is still able to react well to the dynamic obstacles around it and find intelligent looking plans to reach the goal.

9. Future directions

Some real-time algorithms for static environments have been proven to be complete, in that they are guaranteed to eventually reach a goal. This requires that, to compensate for their limited-lookahead, the algorithms update (or 'learn') cost-to-go estimates to states they have visited. This allows them to avoid falling into cyclic behavior. While, as discussed previously, dynamic environments are inherently incomplete, it would be interesting to develop a complete version of RTR* for use in static worlds.

PLRTA*and RTR*both performed comparably or better than previous state-of-the-art algorithms on this domain. However, one deficiency of all the algorithms tested is the lack of a heuristic that can effectively predict costs due to dynamic obstacles. Currently, the only way that these costs are discovered by the algorithms is through search. In the very dense state space that is being explored, a heuristic that could model the expected costs due to dynamic obstacles could greatly improve the efficiency of all the algorithms tested. One possible way to accomplish this would be to abstract the state space to only three dimensions, x location, y location, and time. These would have to be discretized to be coarse enough to allow the heuristic to be calculated quickly. Each time the predictions of the dynamic obstacles changes, the heuristic could be computed as follows: Create a 3D matrix representing the three dimensions x, y and t where the cost in each cell is the cost associated with the dynamic obstacles in that location and that time. Then perform Dijkstra's algorithm starting at the goal location. This will compute minimum cost paths to all other locations at all other times. The cost of these paths could then be used as a heuristic during search. While this heuristic would of course be inadmissible, it might well increase algorithm performance. However, developing such heuristics lies outside the scope of this paper.

10. Conclusion

We have presented the first two real-time heuristic search algorithms for kinodynamic motion planning with dynamic obstacles. We introduced 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.

Acknowledgements

We gratefully acknowledge support from the DARPA CSSG program (grant D11AP00242) and the NSF RI program (grant IIS-0812141).

References

[1] D.M. Bond, N.A. Widger, W. Ruml and X. Sun, Real-time search in dynamic worlds, in: *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 2010, pp. 16–22.

[2] H. Choset, K.M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L.E. Kavraki and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*, MIT Press, 2005.

[3] E.W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik* **1** (1959), 269–271.

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

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

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

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

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

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

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

[11] S.M. Lavalle, Rapidly-exploring random trees: A new tool for path planning, TR 98-11, Computer Science Department, Iowa State University, 1998.
[12] S.M. LaValle, *Planning Algorithms*, Cambridge Univ. Press, 2006.
[13] J. Lee, C. Pippin and T. Balch, Cost based planning with RRT in outdoor environments, in: *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008*, IEEE, 2008, pp. 684–689.
[14] M. Likhachev and D. Ferguson, Planning long dynamically feasible maneuvers for autonomous vehicles, *International Journal of Robotic Research* **28** (2009), 933–945.
[15] M. Likhachev and A. Stentz, R* search, in: *Proceedings of the 23rd National Conference on Artificial Intelligence*, Vol. 1, AAAI Press, 2008, pp. 344–350.
[16] M. Phillips and M. Likhachev, SIPP: Safe interval path planning for dynamic environments, in: *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, 2011.
[17] I. Pohl, Heuristic search viewed as path finding in a graph, *Artificial Intelligence* **1**(3,4) (1970), 193–204.