Real-Time Search in Dynamic Worlds

David M. Bond, Niels A. Widger, Wheeler Ruml
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
david.bond at iol.unh.edu
niels.widger, ruml at cs.unh.edu

Xiaoxun Sun
Computer Science Department
University of Southern California
Los Angeles, CA 90089 USA
xiaoxuns at usc.edu

Abstract

For problems such as pathfinding in video games and robotics, a search algorithm must be real-time (return the next move within a fixed time bound) and dynamic (accommodate edge costs that can increase and decrease before the goal is reached). Existing real-time search algorithms, such as LSS-LRTA*, can handle edge cost increases but do not handle edge cost decreases. Existing dynamic search algorithms, such as D* Lite, are not real-time. We show how these two families of algorithms can be combined using bidirectional search, producing Real-Time D* (RTD*), the first real-time search algorithm designed for dynamic worlds. Our empirical evaluation shows that, for dynamic grid pathfinding, RTD* results in significantly shorter trajectories than either LSS-LRTA* or naive real-time adaptations of D* Lite because of its ability to opportunistically exploit shortcuts.

Introduction

Many applications of heuristic search impose constraints on the search algorithm. In video game pathfinding and robot motion planning, the search is often required to be *real-time*, that is, return the next move for the agent within a strict time bound, so that the agent can continue acting in the world. To accommodate this constraint, real-time search algorithms interleave planning and moving. Because the agent must make a choice of which move to execute next before finding a complete path to a goal, real-time search algorithms achieve fast response times at the cost of sub-optimality of the resulting trajectories. There has been much research on real-time search algorithms, starting with LRTA* (Korf 1990).

Video game pathfinding and robot motion planning often include an additional feature not captured by the real-time constraint: the world can change before the agent reaches the goal. This means that the search must be *dynamic*, that is, be able to recognize when edges in the state space change their costs. If a door opens, this in effect causes an edge between two states to decrease in cost from infinity to a small positive value. There has been much work on dynamic search algorithms, starting with D* (Stentz 1994). Replanning is initiated when the agent is informed of changed edge costs. D* is an incremental search algorithm that saves information between planning episodes so that re-planning can be faster.

Existing real-time search algorithms are designed for known static graphs. However, most real-time search algorithms only explore the part of the state space near the agent. When used in conjunction with the freespace assumption, they can be made to work in situations where the world is initially unknown and is discovered by the agent during its travel. They will typically tolerate discovering that an edge is more expensive than originally thought, as when an obstacle is discovered or when the world changes such that an edge cost is increased. However, because these algorithms are not designed to handle dynamic worlds, they do not include any mechanism for reacting when previously seen edges decrease in cost. This makes them ill-suited to handle path finding in dynamic worlds such as video games.

In this paper, we show how real-time and dynamic search algorithms can be combined, yielding the first real-time dynamic search algorithm, which we call Real-Time D* (RTD*). Our approach is conceptually simple: we perform a standard dynamic search but interrupt it periodically to move the agent according to a real-time search. In this paper, we use D* Lite (Koenig and Likhachev 2002) or Anytime D* (Likhachev et al. 2005) as the dynamic search algorithm and LRTA* (Korf 1990) or LSS-LRTA* (Koenig and Sun 2009) as the real-time search algorithm. D* Lite and Anytime D* perform a global search backwards from the goal, trying to reach the agent's current location. LRTA* and LSS-LRTA* perform a local search forward from the agent's current location, trying to reach the goal. Therefore, RTD* is a form of bidirectional search that combines two kinds of search, real-time and dynamic, dividing the real-time per-move computation limit between them. After explaining the algorithm in more detail, we empirically demonstrate its performance in comparison to state-of-the-art real-time and dynamic search algorithms on grid pathfinding problems. We find that RTD* is competitive with the state-of-the-art in initially unknown static worlds and superior in dynamically changing worlds.

Previous Work

We are not aware of previous work combining real-time and incremental search for dynamic worlds, although there has been much work in each area individually. Koenig and Sun (2009) compare real-time and incremental search algorithms in static worlds with initially unknown worlds. Before presenting our approach, we will first briefly discuss algorithms in each area, as they form the ingredients of our method.

LRTA* chooses moves by performing a bounded lookahead search from the current state towards the goal state at each iteration (Korf 1990). To prevent the agent from becoming stuck in local minima, it updates the h value of the current state before moving by backing up the minimum h value of the search frontier. Korf focuses on using breadth-first search for the lookahead, although he mentions a variant that uses A*, called Time Limited A* (TLA*) (Korf 1990).

Local Search Space LRTA* (LSS-LRTA*) is a state-of-the-art variant of LRTA* that uses an A*-based lookahead and updates the h values of all states in its lookahead space using Dijkstra's algorithm after the lookahead phase (Koenig and Sun 2009). Because it updates all values in the local search space, it learns more quickly than LRTA*. It then moves the agent to the state on the search frontier with the lowest f value before undertaking further lookahead, thereby expanding fewer nodes per agent move than LRTA*.

D* Lite is a state-of-the-art incremental search algorithm that computes a shortest path between the agent and the goal by searching backwards from the goal towards the current state of the agent (Koenig and Likhachev 2002). D* Lite's first search is exactly an A* search. Because it searches towards the current location of the agent, D* Lite can reuse much of its previous search computation even if the agent moves. When edge costs change, D* Lite efficiently replans a new shortest path. We refer to this as a planning episode. We explain D* Lite in some detail, as we will refer to these details later.

For each state, D* Lite maintains a g value (distance to the goal) and an *rhs* value (a one-step lookahead distance estimate). The *rhs* value of a state always satisfies the following relationship (Invariant 1):

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ min_{s' \in Succs(s)}(g(s') + c(s', s)) & \text{otherwise} \end{cases}$$

A state s is considered 'inconsistent' if $g(s) \neq rhs(s)$.

D* Lite performs its search by maintaining a priority queue similar to A*'s open list. The priority queue always contains exactly the set of inconsistent states (Invariant 2). D* Lite sorts the priority queue in ascending order by the key:

$$\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))$$

The priority of a state in the priority queue is always the same as its key (Invariant 3). Thus the queue is sorted similarly to A*'s open list.

The algorithm begins by initializing the g and *rhs* values of all states to ∞, with the exception that the *rhs* value of the goal state is set to zero. It then adds the goal state to the priority queue with a key of $[h(goal); 0]$. Thus, when the algorithm begins, the goal state is the only inconsistent state. The main loop of the algorithm continually removes the top state from the priority queue (the one with the smallest key), updates its g value based on its *rhs* value, propagates the new g value to its successors and adds any newly inconsistent states to the priority queue and deletes newly consistent states from the priority queue. D* Lite terminates when two conditions hold: the key of the top state in the priority queue is greater than or equal to the key of the start state and the start state is consistent.

Anytime D* (AD*) is an anytime version of D* Lite that returns increasingly shorter solutions by starting with a high weight and lowering it over time, eventually converging to the optimal solution (Likhachev et al. 2005). AD* caches work across searches with different weights to speed up planning.

Real-Time D*

Real-Time D* (RTD*) is a bidirectional search algorithm that combines a local search and a global search. The local search is performed around the current location of the agent and the global search is performed from the goal towards the agent's current location. The local search is used to make what we term a *partially informed* step choice when the global search has not yet reached the agent's location. When the agent is following a complete path to the goal as found by the global search we say that it is making a *fully informed* step choice. RTD* is different from other real-time search algorithms due to the addition of a global search. Performing a global search along with a local search allows RTD* to process changed edge costs that occur outside of the local search's lookahead space and therefore allows it to find shortcuts. Performing the local search along with a global search allows RTD* to always return an action within bounded time, even if a complete solution path has not yet been found.

In this paper, we consider LRTA* or LSS-LRTA* as our local search algorithm and D* Lite or AD* as our global search algorithm. It is easiest to understand RTD* as a modification of the global search algorithm, which is made real-time by two changes. First, we add computation limits, specified in terms of state expansions. RTD* takes a parameter, *LocalRatio*, that specifies the fraction of the per-step computation limit to reserve for the local search. The rest is used by the global search. Second, we change action selection to depend on whether the global search phase completed naturally or was terminated by the computation limit. If it completed naturally, it computed a complete path from the agent to the goal and we can therefore move the agent by taking the first action from the path. Otherwise, we perform the local search and move the agent based on its computed action. These two changes do not destroy the desirable properties of D* Lite or AD*, as we will discuss below.

Figure 1 shows the pseudocode of RTD* using D* Lite as its global search algorithm. s_{start} is the agent's current location and s_{goal} is the goal location. U is the priority queue, g(s) is the current distance estimate from s to the agent's current location, rhs(s) is the one-step lookahead distance estimate from s to the agent's current location (as in D* Lite) and c(s, s') is the cost from s to s' (s and s' are arbitrary states). We have excluded the TopKey, CalculateKey, and

```
1. function ComputeShortestPath(GlobalLimit, s_start)
2.     while ( U.TopKey() < CalculateKey(s_start)
           OR rhs(s_start) > g(s_start ) )
3.         if (GlobalLimit = 0) then
               return EXPANSION_LIMIT_REACHED
4.         GlobalLimit ← GlobalLimit − 1
5.         k_old ← U.TopKey()
6.         u ← U.Pop()
7.         if (k_old < CalculateKey(u)) then
8.             U.Insert(u, CalculateKey(u))
9.         else if (g(u) > rhs(u)) then
10.            g(u) ← rhs(u)
11.            for all s ∈ Pred(u) UpdateVertex(s)
12.        else g(u) ← ∞
13.            for all s ∈ Pred(u) ∪{u} UpdateVertex(s)
14.    if(rhs(s_start) = ∞) then return NO_PATH_EXISTS
15.    else return COMPLETE_PATH_FOUND

16. function ChooseStep(s_start, LocalLimit, status)
17.    if (status = EXPANSION_LIMIT_REACHED) then
           return LocalSearch(LocalLimit)
18.    else if (status = COMPLETE_PATH_FOUND) then
           return min_{s' ϵ Succ(s)}(c(s_start, s') + g(s'))
19.    else if (status = NO_PATH_EXISTS) then return s_start

20. function Main(ExpandLimit, s_start, s_goal, LocalRatio )
21.    LocalLimit ← LocalRatio × ExpandLimit
22.    GlobalLimit ← ExpandLimit − LocalLimit
23.    s_last ← s_start
24.    Initialize()
25.    status ← ComputeShortestPath(GlobalLimit, s_start)
26.    while (s_start ≠ s_goal)
27.        s_start ← ChooseStep(s_start, LocalLimit, status)
28.        k_m ← k_m+ h(s_last, s_start)
29.        S_last ← s_start
30.        if changed edge costs then
31.            for all u in changed vertices
32.                for all v in Succ(u)
33.                    UpdateEdgeCost(u, v)
34.                    UpdateVertex(u)
35.        status ← ComputeShortestPath(GlobalLimit, s_start )
36.    return GOAL_LOCATION_REACHED

37. function UpdateVertex( Vertex )
38.    if (u eqs_goal )
39.        rhs(u) ← min_{s' ϵ Succ(Vertex)}(c(Vertex, s') + g(s'))
40.    if (u ϵ U )
41.        U.Remove(Vertex)
42.    if g(Vertex) neq rhs(Vertex)
43.        U.Insert(Vertex, CalculateKey(Vertex))

44. function CalculateKey( Vertex )
45.    return [min(g(s), rhs(s)) + h(s_start, s) + k_m;
              min(g(s)), rhs(s))]
```

Figure 1: Pseudocode for RTD* using D* Lite.

Initialize functions from the pseudocode—they remain unchanged from the D* Lite pseudocode. TopKey returns the key $[\infty, \infty]$ if U is the empty set. k_m is a bound on key values that D* Lite uses to avoid frequent heap reordering.

The most important changes to note from the original D* Lite algorithm are on line 3 where we altered the ComputeShortestPath termination condition and on line 27 where we added the step choice. On line 21 and 22 we divide the total computation limit per iteration (expansion limit) between the local (*LocalLimit*) and global (*GlobalLimit*) search algorithms using the *LocalRatio* parameter. Line 16 shows the action selection function. On line 18, we make our step choice in the case where ComputeShortestPath terminated normally. We take the first action from the complete solution by tracing the start state's best successor according to the computed g values. Line 17 handles the case where ComputeShortestPath has terminated early due to the computation limit. At this point we run the local search algorithm with its computation limit and take its returned step. One can plug LRTA*, LSS-LRTA* or any other local search algorithm in here. On line 19 we handle the case where no path currently exists between the agent's current location and the goal. In this case we do not move the agent since the world may be dynamic and the agent may be able to reach the goal at a later time. This could be altered to return a 'path not found' error depending on the domain.

On line 35, ComputeShortestPath is called even if no changes in edge costs have been detected. This is because if ComputeShortestPath previously terminated due to the computation limit, the global search must continue. If ComputeShortestPath terminated normally this will have no effect unless there have been changes in the world since ComputeShortestPath will terminate immediately.

Properties of RTD*

Completeness Because RTD* follows D* Lite closely, it can inherit many of the same properties. In static finite worlds in which the goal is reachable from all states, RTD* is complete, meaning it will return a valid solution resulting in the agent reaching the goal state, provided that the computation limit is high enough. To show this, we start by establishing the similarity with D* Lite:

Theorem 1 *Invariants 1–3 of D* Lite also hold in RTD*.*

Proof: There are several changes made to D* Lite in adapting it for use with RTD*: 1) action selection may be chosen according to the local real-time search (line 17), 2) the planning phase may now terminate due to the expansion limit (line 3), and 3) k_m is updated and ComputeShortestPath is called even if changes have not occurred. No part of the proofs of D* Lite's invariants rely on how the agent's moves are chosen, so change 1 is irrelevant. Change 2 means that the agent may move during what would have been, in plain D* Lite, a single call to ComputeShortestPath. We need to confirm that the k_m update of change 3 is enough to ensure that D* Lite's invariants are restored after the agent's movement and before we re-enter ComputeShortestPath. Now, the agent's movement does not change the g

or rhs values or the contents of the priority queue, so invariants 1 and 2 are preserved. Invariant 3 depends on the priority queue keys. The first element of each key is analogous to A*'s f value and refers to $g(s)(distancefromgoal)$ and $h(s_{start}, s)(remainingdistancetostart)$. Movement of the agent due to change 2 changes s_{start}, so the keys become obsolete. Change 3 addresses this using the same mechanism of D* Lite, namely k_m, an upper bound on how much the first key element can have decreased. There is nothing in the proof of correctness of the k_m mechanism that depends on ComputeShortestPath having found a complete path. Hence change 3 ensures that we preserve invariant 3 in the face of change 2. □

Under certain assumptions, the completeness of D* Lite is preserved in RTD*:

Theorem 2 *In a static and finite world in which the goal is reachable from all states, if the computation limit is sufficiently large, repeated calls to ComputeShortestPath will eventually compute g values allowing a complete path from s_{start} to s_{goal} to be traced if one exists.*

Proof: (sketch) By Theorem 1, the invariants hold, therefore in the absence of an expansion limit, RTD* performs exactly like the original D* Lite and the original proofs also go through.

With an expansion limit, the difference from D* Lite is that the agent may move, obsoleting key values in the priority queues and necessitating additional expansions to update those key values (lines 7–8). If the expansion limit is larger than the number of necessary update expansions, then the additional expansions performed by RTD* will behave like ordinary D* Lite expansions and serve to enforce consistency (lines 9–13). When the world is static, both g and rhs values will strictly decrease (lines 10 and 39). This means that every node will be expanded a finite number of times. Because there will always be a node on a path to s_{start} in the queue, this implies that s_{start} will eventually be reached. □

Theorem 2 implies that, in a static finite world, the global search will eventually reach the agent, resulting in a path to the goal. Interestingly, the properties of the local search algorithm in RTD* are not vital to its completeness in this case, permitting the use of arbitrary domain-dependent enhancements in the local search. Completeness is more difficult to prove in an infinite world without making an assumption that the agent is moving towards the search frontier.

RTD* is not complete in dynamic worlds; however, this is an unavoidable property of dynamic search algorithms. Consider a situation where the agent is trapped in a room. While a door to the north always provides an exit from the room, there are two southern doors that are much closer to the agent. Each is equipped with a sensor such that, whenever the agent approaches, the door shuts temporarily. While the agent always has a way out by going north, any conventional dynamic search algorithm will guide the agent to toggle between the two southern doors forever. In general, without a model of the changes to the world it appears difficult to see how a dynamic algorithm could guarantee completeness.

Optimality If an algorithm must recommend a step based on incomplete information, it cannot make any guarantees on single trial solution optimality. In the limit of an infinite computation bound, the global search phase will always terminate naturally and RTD* will behave like D* Lite, always taking the next step along a path to the goal that is optimal given its current knowledge of the world.

Time Complexity Assuming the number of successors of a state is bounded, the time complexity of RTD* in relation to a single step is constant, meaning RTD* returns an action within a fixed expansion limit independent of the size of the world. RTD* limits its computation by imposing a limit on the number of expansions per planning phase. With a priority queue implemented as a heap, RTD* fails to be real-time. While a logarithmic growth rate may be acceptable for many applications, to ensure the real-time property, one can replace the heap with a bucketed hash table based on f values. This amortizes the time for insertions (Björnsson, Bulitko, and Sturtevant 2009; Sun, Koenig, and Yeoh 2008; Bulitko et al. 2008).

The overall time complexity of RTD* is directly proportional to the number of planning phases until the global search frontier intersects the agent's current state. Once this occurs, the algorithm has a complete path and can execute actions without further search until the world changes. An important property of D* Lite is that it finds shortest paths and that its first search is exactly an A* search. Therefore, in a static world with an infinite computation limit, RTD* converges to A*, expanding the minimum number of nodes while finding the shortest path.

Space Complexity The space complexity of RTD* is equivalent to that of D* Lite and A*, namely linear in the size of the state space. (Note that in many pathfinding problems, the state space is the same size as the map, which is usually stored explicitly.)

Empirical Evaluation

To evaluate whether RTD* performs well in practice, we compared it against the real-time search algorithms LRTA* and LSS-LRTA*.[1] We also compared against two naive baseline algorithms based on D* Lite. The first, called RTD* NoOp, trivially obeys the real-time constraint by issuing 'no-op' actions as necessary until the global D* Lite search terminates with a complete solution. The degree to which RTD* outperforms this NoOp variant illustrates the benefit gained from using a local search to interleave reactive action with deliberative planning. The second naive baseline, called Non-RTD*, is not real-time and is always allowed unlimited computation per step. Its performance illustrates the theoretical best-possible planning given the agent's evolving state of knowledge if the freespace assumption holds.

We evaluate these algorithms in two settings: pathfinding in worlds where the world is initially unknown to the agent but does not change over time and pathfinding in worlds that

[1] Just before press time, we noticed a small bug in our RTD* implementation: k_m was only updated when edge costs changed (eg, line 30 of Figure 1 appeared before line 28). Limited experiments with a corrected implementation showed only very minor performance changes in our test domains.

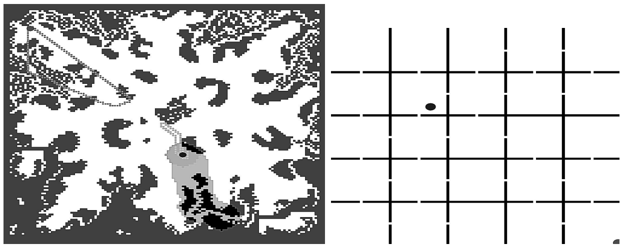

Figure 2: Example grids used for pathfinding in initially unknown static (left) and dynamic (right) worlds.

are known to the agent but that change over time. The worlds are modeled as grids with eight-way movement and the algorithms use octile distance as the heuristic. Movement along the four cardinal directions has unit cost and the diagonals cost $\sqrt{2}$. All actions are reversible. The grid itself consists of cells that are either passable or impassable.

In our experiments we used a number of values for RTD*'s *LocalRatio* parameter *LocalRatio*: 25%, 50%, and 75%. We display the best performing ratio in our results. The differences in the performance of the ones not shown were small.

Unknown Static Worlds

In these worlds, the grid is initially unknown to the agent, but it can observe obstacles in a radius of 7 cells around itself (an arbitrarily chosen value). As it moves around the grid, it learns which cells are impassable. The agent assumes a cell is passable until it discovers otherwise. Once the agent sees the true state of a cell, the cell will not change. Because any changes the agent detects are inside its field of vision and therefore close to it, they are likely to have an immediate effect on its current step.

We used three 161×161 cell grid maps from the popular game Warcraft (Figure 2, left). The start and end locations were randomly selected such that a valid path is guaranteed to exist, resulting in 519 problems. We tested at computation limits of 1, 8, 16, 32, 64, 128, 256 and 512 expansions per step.

The top graph in Figure 3 shows the performance of LRTA* (with TLA* lookahead), RTD* (using D* Lite and LRTA* with a *LocalRatio* of 25%), and the two baseline algorithms RTD* NoOp and Non-RTD*. The y axis represents the time steps used by the agent, where a time step represents one opportunity for the agent to move. All algorithms except RTD*NoOp move at every time step; for those algorithms this is equal to the distance traveled by the agent. Time is expressed relative to the trajectory of an agent guided by A* with full knowledge of the world and no computation limit. The x axis represents the total per-step computation limit, expressed in node expansions, and is on a logarithmic scale. Note that Non-RTD* does not obey the computation limit and thus appears as a straight line in the plot.

In the top graph, RTD* shows strong performance, outperforming the other real-time algorithms. LRTA* has decent performance for small computation limits but its slow

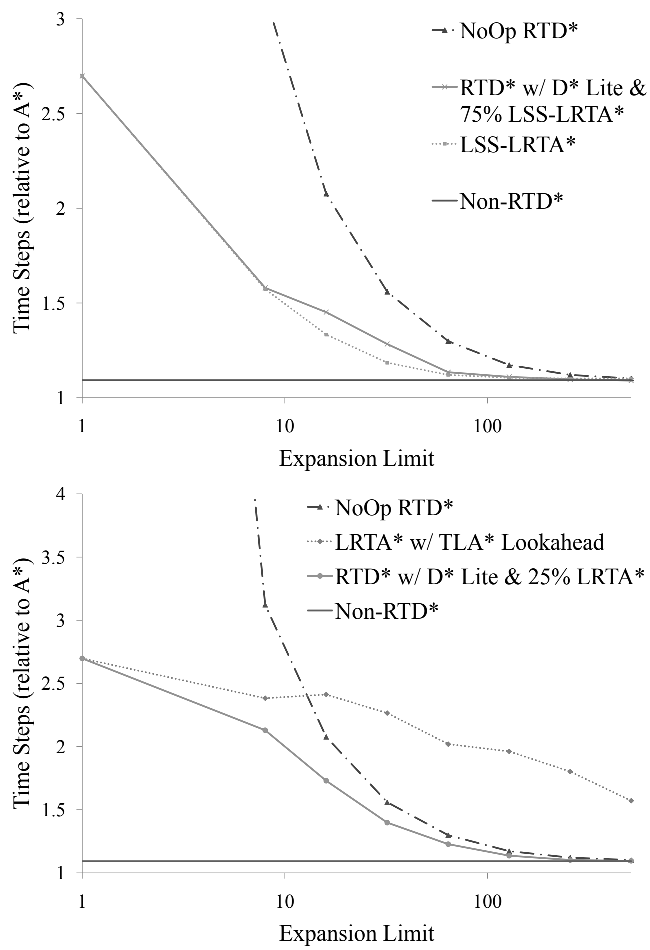

Figure 3: Results on pathfinding in initially unknown world, showing sub-optimality versus per-step expansion limit.

learning causes it to improve more slowly than the other algorithms when given more time. RTD*NoOp shows enormous improvement as the expansion limit is increased because it can move more frequently, however it still performs very poorly. RTD* outperforms RTD* NoOp, confirming that taking a partially informed step is better than not moving at all during deliberation since not moving is implicitly increasing the time to reach the goal. As expected, in the limit of infinite computation time RTD* approaches the performance of Non-RTD*.

The bottom graph in Figure 3 presents the results of a similar experiment using a more modern real-time search. It shows LSS-LRTA*, RTD* (using D* Lite and LSS-LRTA* with a *LocalRatio* of 75%), and the two baseline algorithms RTD* NoOp and Non-RTD*. LSS-LRTA* performs much better than LRTA*, but RTD* yields almost identical performance, even though the world is not truly dynamic.

Dynamic Worlds

To test the algorithms in dynamic worlds, we created a simple benchmark domain using a 100×100 cell grid divided into rectangular rooms (Figure 2, right). The start location was fixed at the upper left hand corner and the goal was fixed in the lower right hand corner. Each room has up to four open 'doors' that lead to an adjacent room. The doors open and close randomly as time passes, causing paths to become available or invalidated as the agent traverses the map. The openings and closings were randomly generated while ensuring that there was always a path from any location to the goal and then recorded to ensure the same dynamism for each algorithm. 100 different sequences of changes were used. Note that the world is fully observable, so agents are notified when changes occur. The movement costs and heuristic used were the same as in our experiments on unknown worlds, and we tested the same computation limits.

As we did with unknown worlds, we ran a baseline Non-RTD* algorithm that does not obey the real-time constraint, in order to gain a sense of the best achievable performance. We do not show results for our other naive baseline, RTD* NoOp, because it doesn't make sense for dynamic worlds. For many computation limits, RTD* NoOp never moves the agent because the global search is too busy tracking updates to the world and never reaches the agent's start state.

The top graph of Figure 4 shows the performance of LRTA* (with TLA* lookahead) and RTD* (using D* Lite and LRTA* with a *LocalRatio* of 25%). For the y axis, which again shows the distance traversed, we normalized against the optimal trajectory length computed off-line by A*, assuming that all doors were open.

As can be seen in the graph, RTD* outperforms LRTA* on all but the very lowest computation limits. As the computation limit increases, the difference in performance becomes larger. The ability to react to new shortcuts offers strong benefits in this domain.

The bottom graph of Figure 4 shows LSS-LRTA* and two versions of RTD*, one using D* Lite and LSS-LRTA* and the other using Anytime D* (AD*) and LSS-LRTA*. Both used a *LocalRatio* of 50%. AD* used a starting weight of 3, decreasing when possible by 0.5, and increasing when necessary by 0.1. The graph shows that RTD* using D* Lite and LSS-LRTA* is slightly worse than LSS-LRTA* on low computation limits but outperforms it significantly for higher limits. This performance gap eventually becomes constant because, at high computation limits, LSS-LRTA* converges to the performance of an A* search that assumes a static world, and RTD* converges to the performance of D* Lite.

When used with AD*, RTD* exhibits improved performance at smaller computation limits, surpassing LSS-LRTA*.

Once LSS-LRTA* sees a wall, it learns the increased h values for the states around that wall. However, in a dynamic world with decreasing edge costs, when a shortcut opens up LSS-LRTA*'s learning may guide the agent away from a shortcut. In contrast, RTD* will notice the opening of a shortcut and replan accordingly. RTD* outperforms LSS-LRTA* on dynamic worlds because of its ability to take advantage of shortcuts. This will hold true as long as the changes in the world are infrequent enough to give the RTD* frontier sufficient time to reach the agent.

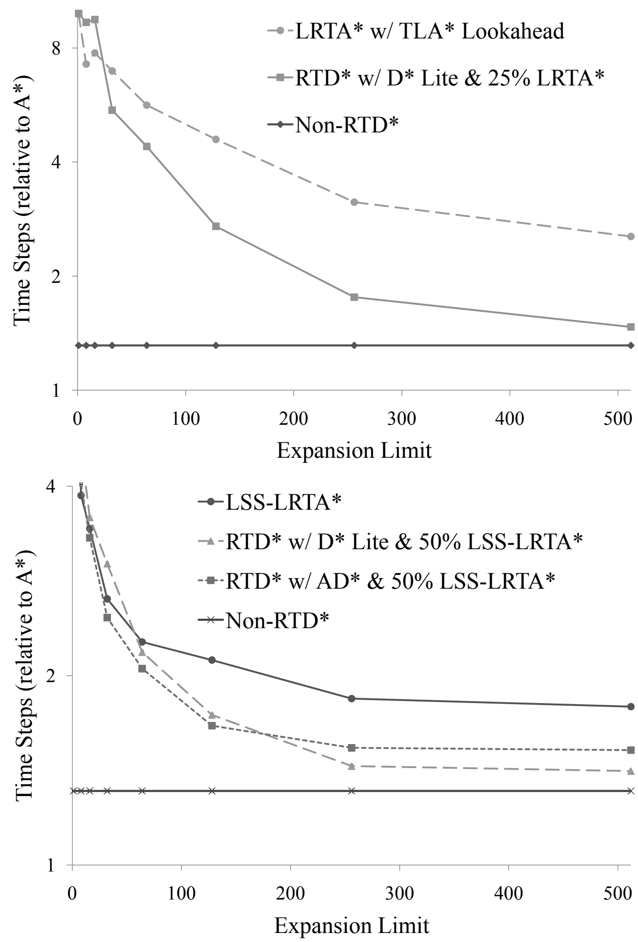

Figure 4: Results on pathfinding in dynamic worlds, showing sub-optimality versus per-step expansion limit.

Discussion

RTD* is the first real-time search algorithm specifically designed for dynamic worlds. RTD* outperforms existing real-time search algorithms, such as LRTA* and LSS-LRTA*, on dynamic worlds, but is also competitive on unknown worlds. Because the algorithm is relatively simple, it should be easy to adapt it to use new dynamic and real-time algorithms that are developed in the future.

For video games, having agents move in a believable manner is important. While we have not conducted quantitative tests of agent 'naturalness', we believe that RTD* should be able to produce more believable behavior than LSS-LRTA*. Shortcuts that open up outside of the local search lookahead space are not taken advantage of by existing real-time algorithms. The addition of a global search allows RTD* to direct the agent towards such shortcuts.

Most real-time searches are 'agent-centered,' in that they focus the search of the agent near its current location. Time Bounded A* (TBA*) is a recent proposal that separates the agent's search from its location (Björnsson, Bulitko, and Sturtevant 2009). It performs an A* search from the start to the goal, periodically moving the agent along the path to the best node on A*'s open list in order to comply with the real-time constraint. RTD* was inspired in part by this separation of search and agent movement, and we initially attempted to combine TBA* with Lifelong Planning A* (LPA*) (Koenig, Likhachev, and Furcy 2004) in order to handle dynamic worlds. However, we found the TBA* approach to be fundamentally limited because TBA*'s search proceeds from the initial start to the goal, ignoring the agent's current location. If the agent's path back to the start becomes impassable, it can become impossible for it to intersect with the global search. Thus we chose to search from the goal to the agent, as in D* Lite.

It is important to note that RTD* is not an anytime algorithm. Whereas an anytime algorithm determines complete trajectories that improve over time, RTD* must often make a partially informed step choice based on incomplete trajectories.

In our evaluation, we followed standard practice in the real-time search literature of measuring the per-step computation limit using the number of node expansions. While this provides some isolation from implementation- and machine-dependent code tuning, it does ignore the fact that different algorithms have different overhead. D* Lite and AD* (and thus RTD*) perform a global search using an open list data structure, that can be burdensome to maintain. LSS-LRTA* performs Dijkstra's algorithm from the lookahead fringe, which is not considered as node expansion. While these effects may be inconsequential in complex real-world domains where node expansion and heuristic calculation is the dominant cost, they may be significant in a simple domain such as video game pathfinding.

Conclusion

By combining real-time and dynamic search in a bidirectional framework, we have created the first real-time dynamic search algorithm, Real-Time D* (RTD*). We presented empirical results comparing it against LRTA*, LSS-LRTA*, and two naive baseline versions of D* Lite on both unknown and dynamic worlds. The results show that RTD* is comparable to the state-of-the-art in unknown worlds and it performs better than the state-of-the-art in dynamic worlds. This is because, in dynamic worlds, the ability to use shortcuts as they open up is crucial to good performance, and the global search used by RTD* is able to direct the agent towards these shortcuts when a local search would not.

Acknowledgments

We are indebted to Sven Koenig for very helpful suggestions on earlier drafts of this work. We also thank Nathan Sturtevant for providing the Warcraft game maps, Brad Larsen for the adversarial doors example, and NSF (grant IIS-0812141) and the DARPA CSSG program for support.

References

Björnsson, Y.; Bulitko, V.; and Sturtevant, N. R. 2009. TBA*: Time-Bounded A*. In *IJCAI*, 431–436.

Bulitko, V.; Lustrek, M.; Schaeffer, J.; Björnsson, Y.; and Sigmundarson, S. 2008. Dynamic Control in Real-Time Heuristic Search. *J. Artif. Intell. Res. (JAIR)* 32:419–452.

Koenig, S., and Likhachev, M. 2002. D* Lite. In *AAAI/IAAI*, 476–483.

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(3):313–341.

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong Planning A*. *Artif. Intell.* 155(1-2):93–146.

Korf, R. E. 1990. Real-Time Heuristic Search. *Artif. Intell.* 42(2-3):189–211.

Likhachev, M.; Ferguson, D. I.; Gordon, G. J.; Stentz, A.; and Thrun, S. 2005. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *ICAPS*, 262–271.

Stentz, A. 1994. Optimal and Efficient Path Planning for Partially-Known Environments. In *ICRA*, 3310–3317.

Sun, X.; Koenig, S.; and Yeoh, W. 2008. Generalized Adaptive A*. In *AAMAS*, 469–476.