Moving Target D* Lite*

Xiaoxun Sun   William Yeoh   Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{xiaoxuns, wyeoh, skoenig}@usc.edu

ABSTRACT

Incremental search algorithms, such as Generalized Fringe-Retrieving A* and D* Lite, reuse search trees from previous searches to speed up the current search and thus often find cost-minimal paths for series of similar search problems faster than by solving each search problem from scratch. However, existing incremental search algorithms have limitations. For example, D* Lite is slow on moving target search problems, where both the start and goal states can change over time. In this paper, we therefore introduce Moving Target D* Lite, an extension of D* Lite that uses the principle behind Generalized Fringe-Retrieving A* to repeatedly calculate a cost-minimal path from the hunter to the target in environments whose blockages can change over time. We demonstrate experimentally that Moving Target D* Lite is four to five times faster than Generalized Adaptive A*, which so far was believed to be the fastest incremental search algorithm for solving moving target search problems in dynamic environments.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Graph and tree search strategies

General Terms

Algorithms, Experimentation, Performance, Theory

Keywords

D* Lite, Dynamic Environment, Generalized Fringe-Retrieving A*, Hunter, Incremental Search, Moving Target Search, Path Planning, Video Games

1. INTRODUCTION

A moving target search problem is a planning problem where a hunter has to catch a moving target [5]. One application of this area of research is computer games, where computer characters (= hunters) has to chase or catch up with other characters (= targets), typically in gridworlds whose blockages can change over time [5, 13, 9]. The moving target search problem is solved once the hunter and target are in the same state (= cell). The hunter has to determine quickly how to move. The computer game company Bioware, for example, recently imposed a limit of 1-3 milliseconds on the search time [1]. There are generally two classes of approaches for solving moving target search problems with search algorithms:

- Offline approaches take into account all possible contingencies (namely, movements of the target and changes in the environment) to find the best plan for the hunter, for example, using minimax or alpha-beta search [9]. Unfortunately, offline approaches do not scale to large environments due to the large number of contingencies.

- Online approaches sacrifice optimality for smaller computation times by interleaving planning and movement, for example, by finding the best plan for the hunter with the information currently available and allowing the hunter to gather additional information while moving. For example, they can determine a cost-minimal path (or a prefix of it) from the current state of the hunter to the current state of the target in the current environment. This allows them to find plans for the hunter efficiently while being reactive to movements of the target and changes in the environment. Examples include real-time search algorithms [5] and incremental search algorithms [13].

Most real-time search algorithms limit the lookahead of each search and thus find only a prefix of a path from the hunter to the target before the hunter starts to move. Their advantage is that the hunter starts to move in constant time, independent of the number of states. Their disadvantage is that the trajectory of the hunter can be highly suboptimal and that it is difficult to determine whether there exists a path from the hunter to the target, although there are recent exceptions [16]. Furthermore, they do not scale to large environments due to their memory requirements that typically grow quadratically in the number of states [2]. Incremental search algorithms, on the other hand, find a path from the

hunter to the target before the hunter starts to move. If they run sufficiently fast, they provide alternatives to real-time search algorithms that avoid their disadvantages. We therefore investigate incremental search algorithms in this paper. There are generally two classes of incremental search algorithms:

- Incremental search algorithms of the first kind use information from previous searches to update the h-values to make them more informed over time and focus the (complete) searches better. Examples include MT-Adaptive A* [8] and its generalization Generalized Adaptive A* (GAA*) [12], which build on an idea in [4].

- Incremental search algorithms of the second kind transform the previous search tree to the current search tree. The current search then starts with the current search tree instead of from scratch. Examples include Differential A* [15], D* [11], D* Lite [7], Fringe-Retrieving A* (FRA*) [13] and its generalization Generalized FRA* (G-FRA*) [14].

Most incremental search algorithms of the second kind were designed for search problems with stationary start states or in static environments (whose blockages do not change over time). Thus, they are not efficient for moving target search problems in dynamic environments (whose blockages can change over time) since both the start and goal states can change over time for moving target search problems. For example, D* Lite can shift the map to keep the start state stationary but then cannot reuse much of the information from previous searches and can be slower than running A* from scratch [8]. In this paper, we therefore introduce Moving Target D* Lite (MT-D* Lite), an extension of D* Lite that uses the principle behind G-FRA* to repeatedly calculate a cost-minimal path from the hunter to the target in dynamic environments. MT-D* Lite does not shift the map to keep the start state stationary and is four to five times faster than GAA*, which so far was believed to be the fastest incremental search algorithm for solving moving target search problems in dynamic environments.

2. PROBLEM DEFINITION

Although all proposed search algorithms can operate on arbitrary directed graphs, for ease of illustration, we describe their operation on known four-neighbor gridworlds with blocked and unblocked states. The hunter can move from its current unblocked state to any unblocked neighboring state with cost one and to any blocked neighboring state with cost infinity. It always follows a cost-minimal path from its current state to the current state of the target until it reaches the current state of the target, which is often a reasonable strategy for the hunter that can be computed efficiently. We make no assumptions about how the target moves.

We use the following notation: S denotes the finite set of all (blocked and unblocked) states, $s_{start} \in S$ denotes the current state of the hunter and the start state of the search, and $s_{goal} \in S$ denotes the current state of the target and the goal state of the search. $c(s, s')$ denotes the cost of a cost-minimal path from state $s \in S$ to state $s' \in S$. $Succ(s) \subseteq S$ denotes the set of successors of state $s \in S$, and $Pred(s) \subseteq S$ denotes the set of predecessors of state $s \in S$. The cost of

moving from state s to its successors or from its predecessors to state s can be infinity. In gridworlds, the predecessors and successors of a state are thus its (blocked and unblocked) neighbors.

3. BACKGROUND

We now provide the background on A*, Generalized Fringe-Retrieving A* and D* Lite that is necessary to understand the proposed search algorithms, closely following the descriptions in [13, 14].

3.1 A*

A* [3] is probably the most popular search algorithm in artificial intelligence and the basis of all search algorithms described in this paper. A* maintains four values for every state $s \in S$: (1) The h-value $h(s, s_{goal})$ is a user-provided approximation of $c(s, s_{goal})$. The h-values have to be consistent [10]. (2) The g-value $g(s)$ is an approximation of $c(s_{start}, s)$. Initially, it is infinity. (3) The f-value $f(s) := g(s) + h(s, s_{goal})$ is an approximation of the smallest cost of moving from the start state via state s to the goal state. (4) The parent pointer $par(s) \in Pred(s)$ points to the parent of state s in the search tree. Initially, it is $NULL$. The parent pointers are used to extract a cost-minimal path from the start state to the goal state after the search terminates. A* also maintains two data structures: (1) The $OPEN$ list contains all states to be considered for expansion. Initially, it contains only the start state with g-value zero. (2) The $CLOSED$ list contains all states that have been expanded. Initially, it is empty. Actually, none of the search algorithms described in this paper require a $CLOSED$ list but, for ease of description, we pretend they do. A* repeatedly deletes a state s with the smallest f-value from the $OPEN$ list, inserts it into the $CLOSED$ list and expands it by performing the following operations for each state $s' \in Succ(s)$. If $g(s) + c(s, s') < g(s')$, then A* generates s' by setting $g(s') := g(s) + c(s, s')$ and $par(s') := s$ and, if s' is not in the $OPEN$ list, inserting it into the $OPEN$ list. A* terminates when the $OPEN$ list is empty or when it expands the goal state. The former condition indicates that no path exists from the start state to the goal state, and the latter condition indicates that A* found a cost-minimal path. Therefore, one can solve moving target search problems in dynamic environments by finding a cost-minimal path with A* from the current state of the hunter to the current state of the target whenever the environment changes or the target moves off the known path.

3.2 G-FRA*

Generalized Fringe-Retrieving A* (G-FRA*) [14] is an incremental search algorithm that generalizes Fringe-Retrieving A* (FRA*) from gridworlds to arbitrary directed graphs. We use the principle behind G-FRA* in one of the proposed search algorithms. G-FRA* cannot solve moving target search problems in dynamic environments but it solves moving target search problems in static environments by finding a cost-minimal path with A* from the current state of the hunter to the current state of the target whenever the target moves off the known path. Each A* run is called a search iteration. In each search iteration, G-FRA* first transforms the previous search tree to the initial search tree, which consists of the initial $OPEN$ and $CLOSED$ lists. It then starts A* with the initial $OPEN$ and $CLOSED$ lists

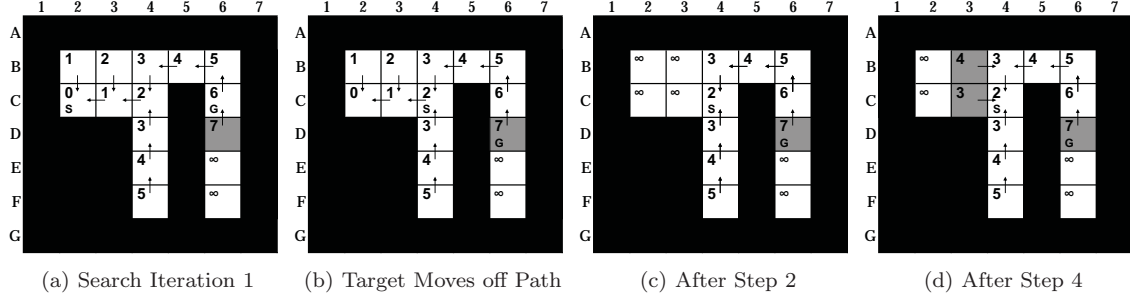| (a) Search Iteration 1 | (b) Target Moves off Path | (c) After Step 2 | (d) After Step 4 |

Figure 1: Trace of G-FRA*

instead of from scratch. Thus, G-FRA* inherits the properties of A*, for example, it finds cost-minimal paths from the current state of the hunter to the current state of the target.

The initial search tree is the subtree of the previous search tree rooted at the current start state. Thus, the initial and previous search trees are different if the hunter moved. To determine the initial *OPEN* and *CLOSED* lists, G-FRA* maintains a *DELETED* list, which contains all states that are in the previous search tree but not the initial search tree. Then, the initial *CLOSED* list contains those states that are in the previous *CLOSED* list but not the *DELETED* list, and the initial *OPEN* list contains both (O1) those states that are in the previous *OPEN* list but not the *DELETED* list and (O2) those states that are in the *DELETED* list and are successors of states in the initial *CLOSED* list.

Figure 1 illustrates the steps of G-FRA*. Blocked states are black, and unblocked states are white. For ease of illustration, we use h-values that are zero for all states. The first search iteration of G-FRA* runs A* from the current state C2 of the hunter, labeled S, to the current state C6 of the target, labeled G (see Figure 1(a)). The g-value of a state is shown in its upper left corner. The arrow leaving a state points to its parent. The *CLOSED* list contains B2, B3, B4, B5, B6, C2, C3, C4, C6, D4, E4 and F4, and the *OPEN* list contains D6. A state is shaded iff it is in the *OPEN* list. The cost-minimal path is C2, C3, C4, B4, B5, B6 and C6. The hunter moves along this path to C4, at which point in time the target moves off the path to D6 (see Figure 1(b)). Since the target moved off the path, G-FRA* finds a cost-minimal path from the current state C4 of the hunter to the current state D6 of the target using the following steps, where the previous start state is C2 and the previous goal state is C6.

- **Step 1 (Starting A* Immediately):** G-FRA* performs the following operations in this step if it did not terminate in Step 3 (Terminating Early) in the previous search iteration. If the current start state is the same as the previous start state and the current goal state is not in the *CLOSED* list, G-FRA* runs A* with the *OPEN* and *CLOSED* lists, uses the parent pointers to extract a cost-minimal path from the current start state to the current goal state and terminates the current search iteration. If the current start state is the same as the previous start state and the current goal state is in the *CLOSED* list, G-FRA* uses the parent pointers to extract a cost-minimal path from the current start state to the current goal state and terminates the current search iteration. In our example, G-FRA*

executes the next steps since the current start state is different from the previous start state.

- **Step 2 (Deleting States):** G-FRA* sets the parent pointer of the current start state to *NULL*, determines the *DELETED* list and deletes all states in the *DELETED* list from the *CLOSED* and *OPEN* lists. It then sets the parent pointers of all states in the *DELETED* list to *NULL* and their g-values to infinity. In our example, G-FRA* sets the parent pointer of C4 to *NULL*, determines the *DELETED* list to contain B2, B3, C2 and C3 and deletes these states from the *OPEN* and *CLOSED* lists. The *CLOSED* list now contains B4, B5, B6, C4, C6, D4, E4 and F4, and the *OPEN* list contains D6. The *OPEN* list is still incomplete since it contains only the states satisfying (O1) so far. G-FRA* then sets the g-values of B2, B3, C2 and C3 to infinity and their parent pointers to *NULL* (see Figure 1(c)).

- **Step 3 (Terminating Early):** If the current goal state is in the *CLOSED* list, G-FRA* uses the parent pointers to extract a cost-minimal path from the current start state to the current goal state and terminates the current search iteration. In our example, G-FRA* executes the next steps since the current goal state is not in the *CLOSED* list.

- **Step 4 (Inserting States):** G-FRA* adds those states to the *OPEN* list that satisfy (O2), which makes the *OPEN* list complete. G-FRA* then sets the parent pointers of all states s added to the *OPEN* list to the state s' in the *CLOSED* list that minimizes $g(s') + c(s', s)$ and sets their g-values to $g(s') + c(s', s)$ for this state s'. In our example, G-FRA* adds B3 and C3 to the *OPEN* list, which now contains B3, C3 and D6. It sets the parent pointers of B3 and C3 to B4 and C4, respectively, and their g-values to 4 and 3, respectively (see Figure 1(d)).

- **Step 5 (Starting A*):** G-FRA* runs A* with the *OPEN* and *CLOSED* lists, uses the parent pointers to extract a cost-minimal path from the current start state to the current goal state and terminates the current search iteration.

3.3 D* Lite

D* Lite [6] is an incremental search algorithm that forms the basis of the proposed search algorithms. D* Lite

```
01  function CalculateKey(s)
02  return [min(g(s), rhs(s)) + h(s, s_goal) + k_m; min(g(s), rhs(s))];
03  procedure Initialize()
04  OPEN := ∅;
05  k_m := 0;
06  for all s ∈ S
07    rhs(s) := g(s) := ∞;
08    par(s) := NULL;
09  s_start := the current state of the hunter;
10  s_goal := the current state of the target;
11  rhs(s_start) := 0;
12  OPEN.Insert(s_start, CalculateKey(s_start));
13  procedure UpdateState(u)
14  if (g(u) ≠ rhs(u) AND u ∈ OPEN)
15    OPEN.Update(u, CalculateKey(u));
16  else if (g(u) ≠ rhs(u) AND u ∉ OPEN)
17    OPEN.Insert(u, CalculateKey(u));
18  else if (g(u) = rhs(u) AND u ∈ OPEN)
19    OPEN.Delete(u);
20  procedure ComputeCostMinimalPath()
21  while (OPEN.TopKey() < CalculateKey(s_goal)
        OR rhs(s_goal) > g(s_goal))
22    u := OPEN.Top();
23    k_old := OPEN.TopKey();
24    k_new := CalculateKey(u);
25    if (k_old < k_new)
26      OPEN.Update(u, k_new);
27    else if (g(u) > rhs(u))
28      g(u) := rhs(u);
29      OPEN.Delete(u);
30      for all s ∈ Succ(u)
31        if (s ≠ s_start AND (rhs(s) > g(u) + c(u, s)))
32          par(s) := u;
33          rhs(s) := g(u) + c(u, s);
34          UpdateState(s);
35    else
36      g(u) := ∞;
37      for all s ∈ Succ(u) ∪ {u}
38        if (s ≠ s_start AND par(s) = u)
39          rhs(s) := min_{s'∈Pred(s)}(g(s') + c(s', s));
40          if (rhs(s) = ∞)
41            par(s) := NULL;
42          else
43            par(s) := arg min_{s'∈Pred(s)}(g(s') + c(s', s));
44          UpdateState(s);
45  function Main()
46  Initialize();
47  while (s_start ≠ s_goal)
48    s_oldstart := s_start;
49    s_oldgoal := s_goal;
50    ComputeCostMinimalPath();
51    if (rhs(s_goal) = ∞) /*no path exists*/
52      return false;
53    identify a path from s_start to s_goal using the parent pointers;
54    while (target not caught AND target on path from s_start to s_goal
          AND no edge costs changed)
55      hunter follows path from s_start to s_goal;
56    if hunter caught target
57      return true;
58    s_start := the current state of the hunter;
59    s_goal := the current state of the target;
60    k_m := k_m + h(s_oldgoal, s_goal);
61    if (s_oldstart ≠ s_start)
62      shift the map appropriately (which changes s_start and s_goal);
63    for all directed edges (u, v) with changed edge costs
64      c_old := c(u, v);
65      update the edge cost c(u, v);
66      if (c_old > c(u, v))
67        if (v ≠ s_start AND rhs(v) > g(u) + c(u, v))
68          par(v) := u;
69          rhs(v) := g(u) + c(u, v);
70          UpdateState(v);
71      else
72        if (v ≠ s_start AND par(v) = u)
73          rhs(v) := min_{s'∈Pred(v)}(g(s') + c(s', v));
74          if (rhs(v) = ∞)
75            par(v) := NULL;
76          else
77            par(v) := arg min_{s'∈Pred(v)}(g(s') + c(s', v));
78          UpdateState(v);
79  return true;
```

Figure 2: D* Lite for Moving Target Search

```
80  procedure BasicDeletion()
81  par(s_start) := NULL;
82  rhs(s_oldstart) := min_{s'∈Pred(s_oldstart)}(g(s') + c(s', s_oldstart));
83  if (rhs(s_oldstart) = ∞)
84    par(s_oldstart) := NULL;
85  else
86    par(s_oldstart) := arg min_{s'∈Pred(s_oldstart)}(g(s') + c(s', s_oldstart));
87  UpdateState(s_oldstart);
88  procedure OptimizedDeletion()
89  DELETED := ∅;
90  par(s_start) := NULL;
91  for all s ∈ S in the search tree but not the
        subtree rooted at s_start
92    par(s) := NULL;
93    rhs(s) := g(s) := ∞;
94    if (s ∈ OPEN)
95      OPEN.Delete(s);
96    DELETED := DELETED ∪ {s};
97  for all s ∈ DELETED
98    for all s' ∈ Pred(s)
99      if (rhs(s) > g(s') + c(s', s))
100        rhs(s) := g(s') + c(s', s);
101        par(s) := s';
102    if (rhs(s) < ∞)
103      OPEN.Insert(s, CalculateKey(s));
```

Figure 3: (Basic) MT-D* Lite

solves sequences of search problems in dynamic environments where the start state does not change over time by repeatedly transforming the previous search tree to the current search tree. Researchers have extended it in a straight-forward way to solve moving target search problems in dynamic environments by finding a cost-minimal path from the current state of the hunter to the current state of the target whenever the environment changes or the target moves off the known path. In each search iteration, D* Lite first shifts the map to keep the start state stationary (for example, it shifts the map one unit south if the hunter moved one unit north) and then updates the g-values and parent pointers of states as necessary [8] until it has found a cost-minimal path from the current state of the hunter to the current state of the target. Figure 2 shows the pseudocode of such a version of D* Lite.[1] We explain only those parts of D* Lite that are sufficient for understanding how to extend it to Moving Target D* Lite. D* Lite maintains an h-value, g-value, f-value and parent pointer for every state s with similar meanings as used by A* but it also maintain an rhs-value. The rhs-value is defined to be

$$rhs(s) = \begin{cases} c & \text{if } s = s_{start} \quad \text{(Eq. 1)} \\ \min_{s' \in Pred(s)} (g(s') + c(s', s)) & \text{otherwise} \quad \text{(Eq. 2)} \end{cases}$$

where $c = 0$. Thus, the rhs-value is basically is a one-step lookahead g-value. State s is called locally inconsistent iff $g(s) \neq rhs(s)$. The f-value of state s is defined to be $\min(g(s), rhs(s)) + h(s, s_{goal})$. Finally, the parent pointer of state s is defined to be

---

[1] The pseudocode uses the following functions to manage the $OPEN$ list: $OPEN$.Top() returns a state with the smallest priority of all states in the $OPEN$ list. $OPEN$.TopKey() returns the smallest priority of all states in the $OPEN$ list. (If the $OPEN$ list is empty, then $OPEN$.TopKey() returns $[\infty; \infty]$.) $OPEN$.Insert(s, k) inserts state s into the $OPEN$ list with priority k. $OPEN$.Update(s, k) changes the priority of state s in the $OPEN$ list to k. $OPEN$.Delete(s) deletes state s from the $OPEN$ list. Priorities are compared lexicographically. The minimum of an empty set is infinity.

$$par(s) = \begin{cases} NULL & \text{if } s = s_{start} \text{ OR} \\ & \quad rhs(s) = \infty \qquad \text{(Eq. 3)} \\ \underset{s' \in Pred(s)}{\operatorname{argmin}} (g(s') + c(s',s)) & \text{otherwise} \qquad \text{(Eq. 4)} \end{cases}$$

which is exactly the definition used by A*.[2] D* Lite also maintains an *OPEN* list with a similar meaning as used by A*. ComputeCostMinimalPath() determines a cost-minimal path from the start state to the goal state. Before it is called, states can have arbitrary *g*-values but their *rhs*-values and parent pointers have to satisfy Eqs. 1-4 and the *OPEN* list has to contain all locally inconsistent states. The runtime of ComputeCostMinimalPath() is typically the higher the more *g*-values it updates.

4. CONTRIBUTIONS

The blockage statuses of many states typically change when D* Lite shifts the map to keep the start state stationary, which in turn changes the *g*-values of many states. For example, B3 inherits the blockage status of A3 when D* Lite shifts the map one unit south. Then, D* Lite updates the *g*-values of many states, which makes it often slower than running A* from scratch [8]. We therefore introduce Basic Moving Target D* Lite (Basic MT-D* Lite) and Moving Target D* Lite (MT-D* Lite), two extensions of D* Lite that solve moving target search problems in dynamic environments without having to shift the map. We first describe Basic MT-D* Lite and then MT-D* Lite, an optimization of Basic MT-D* Lite that uses the principle behind G-FRA* to speed it up.

4.1 Basic MT-D* Lite

Basic MT-D* Lite is an incremental search algorithm that speeds up the version of D* Lite that shifts the map by changing it slightly to avoid having to shift the map. Basic MT-D* Lite solves moving target search problems in dynamic environments by finding a cost-minimal path with ComputeCostMinimalPath() from the current state of the hunter to the current state of the target whenever the environment changes or the target moves off the known path. Figure 3 shows the necessary changes to the pseudocode from Figure 2. Basic MT-D* Lite calls BasicDeletion() on Line 62 instead of shifting the map. At this point in time, the hunter has moved from the previous start state $s_{oldstart}$ to the current start state s_{start}. Before ComputeCostMinimal-Path() is called again, the *rhs*-values and parent pointers of all states have to satisfy Eqs. 1-4 and the *OPEN* list has to contain all locally inconsistent states. Fortunately, the *rhs*-values and parent pointers of all states already satisfy these invariants, with the possible exception of the previous and current start states. Basic MT-D* Lite therefore calculates the *rhs*-value of the previous start state (according to Eq. 2 since it is no longer the current start state), its parent pointer (according to Eqs. 3-4) and its membership in the *OPEN* list on Lines 82-87. The correctness proofs of D* Lite continue to hold if *c* is an arbitrary finite value in Eq. 1. instead of zero. Thus, the *rhs*-value of the current start

state can be an arbitrary finite value, including its current *rhs*-value since its current *rhs*-value is finite.[3] Basic MT-D* Lite therefore does not calculate the *rhs*-value of the current start state and its membership in the *OPEN* list and only sets its parent pointer to *NULL* (according to Eq. 3) on Line 81.

Figure 4 illustrates the steps of Basic MT-D* Lite using the setup from Figure 1. The first search iteration of Basic MT-D* Lite runs ComputeCostMinimalPath() from the current state C2 of the hunter to the current state C6 of the target (see Figure 4(a)). The *rhs*-value of a state is shown in its upper right corner. The cost-minimal path is C2, C3, C4, B4, B5, B6 and C6. The hunter then moves along the path to C4, the target moves off the path to D6 and C5 becomes unblocked, which changes the costs *c*(C4,C5), *c*(C5,C4), *c*(C5,C6), *c*(C6,C5), *c*(C5,B5) and *c*(B5,C5) from infinity to one (see Figure 1(b)). Since the target moved off the path and the environment changed, MT-D* Lite finds a cost-minimal path from the current state C4 of the hunter to the current state D6 of the target using the following steps. Basic MT-D* Lite sets the parent pointer of the current start state C4 to *NULL*, updates the *rhs*-value and parent pointer of the previous start state C2 to 2 and C3, respectively, and inserts C2 into the *OPEN* list (see Figure 1(c)). It then processes the edges with changed edge costs, like D* Lite, and runs ComputeCostMinimalPath(), which expands C2, B2, C3, B3, C3, C5, B3, C2 and C6 (see Figures 4(d-l)). The cost-minimal path is C4, C5, C6 and D6.

Basic MT-D* Lite can be optimized. When it runs ComputeCostMinimalPath(), the *g*-values of all states in the subtree of the previous search tree rooted at the current start state are based on the *g*-value of the current start state and thus correct. The *g*-values of all other states in the previous search tree could be incorrect, in which case they are too small. Basic MT-D* Lite updates the *g*-values of these states by expanding them. When it expands one of them for the first time, it sets its *g*-value to infinity. When it expands the state a second time, it sets its *g*-value to the correct value. In our example, the *g*-values of B2, B3, C3 and C4 could be incorrect. Basic MT-D* Lite expands all of them to set their *g*-values to infinity and then expands all of them again (except for B2) to set their *g*-values to the correct value. Each state expansion is slow since Basic MT-D* Lite iterates over all successors of the expanded state to update their *rhs*-values (according to Eqs. 1-2), parent pointers (according to Eqs. 3-4) and memberships in the *OPEN* list on Lines 30-34 or 37-44. Basic MT-D* Lite also iterates over all predecessors of each successor. Thus, each state expansion can require n^2 operations on *n*-neighbor gridworlds. Furthermore, each state expansion manipulates the *OPEN* list, which, if it is implemented as a binary heap, requires the execution of heap operations to keep it sorted each time Basic MT-D* Lite expands a state since it needs to determine the next state to expand, which is a state with the smallest *f*-value in the *OPEN* list. We therefore optimize Basic MT-D* Lite in the following.

4.2 MT-D* Lite

---

[2]D* Lite typically defines the parent pointers of both the start state and states with *rhs*-values that are infinity differently because they are not really needed. We set them to *NULL* in preparation for Moving Target D* Lite that requires them to have this value.

[3]The *rhs*-value of the current start state is finite because it is on the cost-minimal path from the previous start state to the previous goal state. The *rhs*-values of all states on this path are no larger than the *rhs*-value of the previous goal state, which is finite due to Line 51.
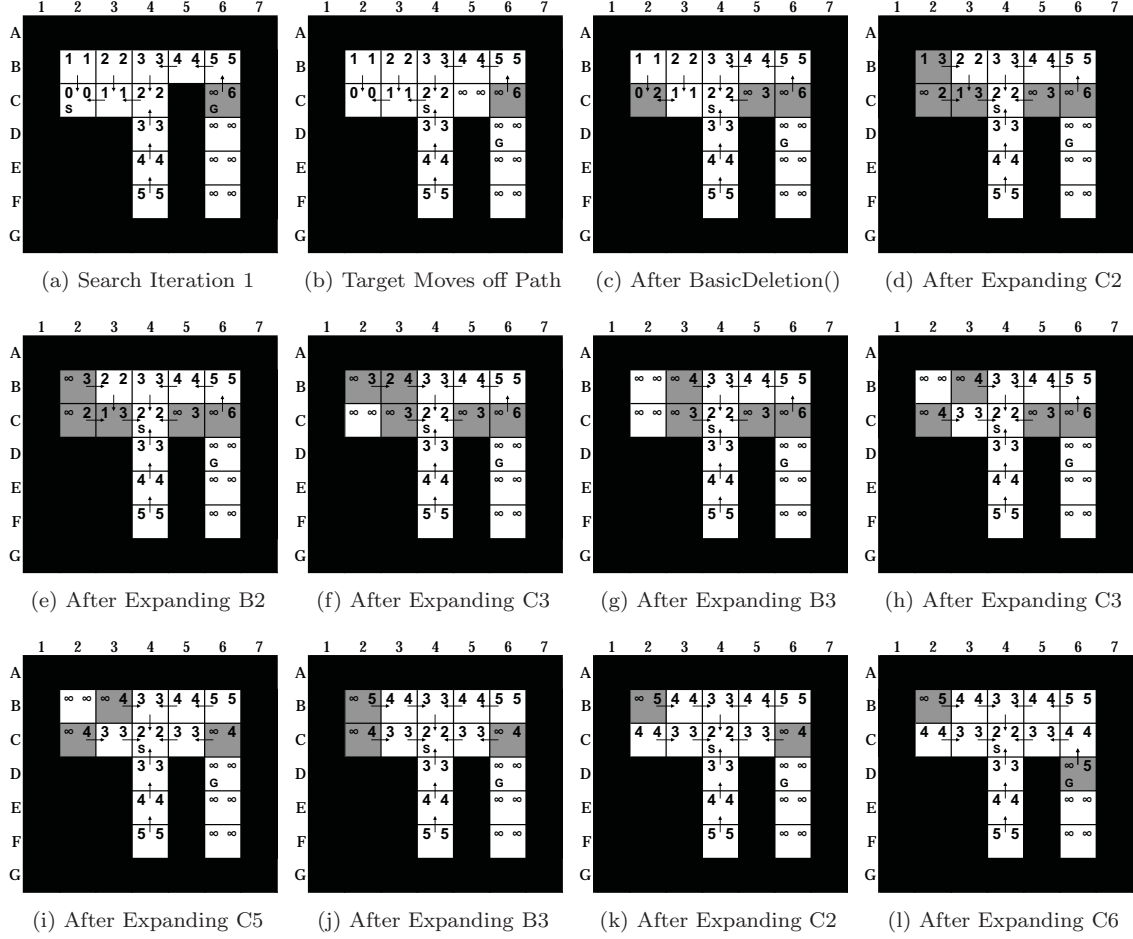
(a) Search Iteration 1   (b) Target Moves off Path   (c) After BasicDeletion()   (d) After Expanding C2

(e) After Expanding B2   (f) After Expanding C3   (g) After Expanding B3   (h) After Expanding C3

(i) After Expanding C5   (j) After Expanding B3   (k) After Expanding C2   (l) After Expanding C6

Figure 4: Trace of (Basic) MT-D* Lite

MT-D* Lite is an incremental search algorithm that is an optimized version of Basic MT-D* Lite. MT-D* Lite operates in the same way as Basic MT-D* Lite except that it replaces BasicDeletion() with a more sophisticated procedure. Figure 3 shows the necessary changes to the pseudocode from Figure 2. Basic MT-D* Lite calls OptimizedDeletion() on Line 62 instead of shifting the map. Basic MT-D* Lite lazily expands the states in the previous search tree that are not in the subtree rooted at the current start state one at a time to set their g-values to infinity when needed. Whenever it sets the g-value of a state to infinity, it needs to update the rhs-values, parent pointers and memberships in the OPEN list for all successors of the state. MT-D* Lite, on the other hand, eagerly uses a specialized procedure that operates in two phases. MT-D* Lite maintains a DELETED list, which contains all states in the previous search tree that are not in the subtree rooted at the current start state, which is exactly the definition that G-FRA* uses. First, MT-D* Lite sets the parent pointer of the current start state to NULL on Line 90. Then, Phase 1 on Lines 91-96 eagerly sets the g-values of all states in the DELETED list to infinity in one pass. Finally, Phase 2 on Lines 97-103 updates the rhs-values, parent pointers and memberships in the OPEN list of all potentially affected states in one pass, which are the states in the DELETED list. Their rhs-values, parent point-

ers and memberships in the OPEN list can be updated in one pass since they depend only on the g-values of their predecessors, which do not change in Phase 2. If the g-values of all predecessors are infinity, which is likely the case for many of these states due to Phase 1, their rhs-values have to be updated to infinity, their parent pointers have to be updated to NULL and they have to be deleted from the OPEN list since they are not locally inconsistent. Phase 1 therefore sets the rhs-values of all states in the DELETED list to infinity, their parent pointers to NULL and removes them from the OPEN list. Phase 2 then only updates the rhs-values (according to Eqs. 1-2) and parent pointers (according to Eqs. 3-4) of the few exceptions and inserts them into the OPEN list if they are locally inconsistent. Overall, Phase 1 is thus similar to Step 2 (Deleting States) of G-FRA*, and Phase 2 is similar to Step 4 (Inserting States) of G-FRA*. The runtimes of both phases are small. Phases 1 and 2 iterate over all states in the DELETED list. Phase 2 also iterates over all predecessors of the states in the DELETED list. Thus, each state in the DELETED list can require n operations on n-neighbor gridworlds. Both phases manipulate the OPEN list, which, if it is implemented as a binary heap, requires the execution of heap operations to keep it sorted only once, namely at the end of Phase 2, since it can remain unsorted until then.

	searches per test case	moves per test case	expanded states per search		deleted states per search		runtime per search
A*	387	688	14013	(31.2)			7250
Differential A*	387	688	14013	(40.1)			9533
GAA*	387	688	12351	(28.2)			6002
FRA*	391	688	515	(13.0)	362	(11.6)	397
G-FRA*	392	688	716	(10.6)	534	(16.1)	501
Basic MT-D* Lite	391	688	2443	(13.7)			2625
MT-D* Lite	392	688	1377	(10.8)	1380	(37.6)	1559

Table 1: Static Environments

Figure 4 illustrates the steps of MT-D* Lite using the setup from Figure 1. The only difference from the steps of Basic MT-D* Lite is that MT-D* Lite does not perform the state expansions from Figures 4(c-g). Instead, Optimized-Deletion() updates the *rhs*-values, parent pointers and memberships in the *OPEN* list of all states in the *DELETED* list, which contains B2, B3, C2 and C3 (see Figure 4(g)). The remaining state expansions are the same as the ones of Basic MT-D* Lite (see Figures 4(h-l)).

5. EXPERIMENTAL RESULTS

We now compare Basic Moving Target D* Lite (Basic MT-D* Lite) and Moving Target D* Lite (MT-D* Lite) against A*, Differential A*, Generalized Adaptive A* (GAA*), Fringe-Retrieving A* (FRA*) and Generalized FRA* (G-FRA*) for solving moving target search problems. For fairness, we use comparable implementations. For example, all search algorithms implement the *OPEN* list as binary heap and find a cost-minimal path from the current state of the hunter to the current state of the target whenever the environment changes or the target moves off the known path.

We perform our experiments in four-neighbor gridworlds of size 1000×1000 with 25 percent randomly blocked states and randomly chosen start and goal states. We average our experimental results over the same 1000 test cases for each search algorithm. The hunter always knows the blockage statuses of all states. The target always follows a cost-minimal path from its current state to a randomly selected unblocked state and repeats the process whenever it reaches that state or cannot move due to its path being blocked. The target skips every tenth move to ensure that the hunter catches it. We perform the experiments in two different environments. In static environments, the blockage statuses of states does not change. In dynamic environments, we randomly block k unblocked states and unblock k blocked states after every move of the hunter in a way that ensures that there always exists a path from the current state of the hunter to the current state of the target. We vary k from 1 to 1000. We use the Manhattan distances as consistent *h*-values.

We report two measures for the difficulty of the moving target search problems, namely the average number of searches and the average number of moves of the hunter until it catches the target. These values vary slightly among the search algorithms due to their different ways of breaking ties among several cost-minimal paths. We report two measures for the efficiency of the search algorithms, namely the average number of expanded states per search and the average runtime per search in microseconds on a Pentium D 3.0 Ghz PC with 2 GByte of RAM. We also report the average number of deleted states (from the search tree) per search for FRA*, G-FRA* and MT-D* Lite. Finally, we report the standard deviation of the mean for the number of expanded and deleted states per search (in parentheses) to demonstrate the statistical significance of our results.

	searches per test case	moves per test case	expanded states per search		deleted states per search		runtime per search
A*	682	682	16961	(91.0)			7621
Differential A*	682	682	16961	(91.0)			10524
GAA*	694	694	12774	(75.0)			4911
Basic MT-D* Lite	682	682	1708	(26.2)			1856
MT-D* Lite	682	682	915	(19.5)	925	(9.1)	990

$$k = 1$$

A*	679	679	16814	(90.8)			7544
Differential A*	679	679	16814	(90.8)			10432
GAA*	692	692	13123	(78.9)			5163
Basic MT-D* Lite	678	678	1716	(26.2)			1872
MT-D* Lite	678	678	923	(19.6)	929	(9.2)	997

$$k = 10$$

A*	677	677	16594	(89.5)			7444
Differential A*	677	677	16594	(89.5)			10293
GAA*	691	691	13174	(78.3)			5342
Basic MT-D* Lite	677	677	1730	(26.8)			1921
MT-D* Lite	677	677	960	(20.5)	916	(9.1)	1075

$$k = 100$$

A*	681	681	16652	(89.1)			7520
Differential A*	681	681	16652	(89.1)			10318
GAA*	693	693	13389	(81.8)			6370
Basic MT-D* Lite	680	680	2073	(31.7)			2494
MT-D* Lite	681	681	1348	(28.7)	899	(9.1)	1592

$$k = 1000$$

Table 2: Dynamic Environments

Tables 1 and 2 show our experimental results in static and dynamic environments, respectively. The version of D* Lite that shifts the map is not included in the tables because it has been shown to be an order of magnitude slower than A* and is thus not competitive [13]. D* is not included in the tables because it has been shown to be about as efficient as D* Lite [7]. FRA* and G-FRA* are not included in Table 2 because they cannot solve moving target search problems in dynamic environments. We make the following observations:

- In dynamic environments, A* and Differential A* have runtimes that are independent of k because they construct their search trees from scratch and thus expand about the same number of states per search independent of k.

- In dynamic environments, GAA* has a runtime that increases with k because it uses a consistency procedure to update the *h*-values of those states whose *h*-values become inconsistent due to states becoming unblocked and the number of such states increases with k [12].

- In dynamic environments, Basic MT-D* Lite and D* Lite have runtimes that increase with k because they update more *g*-values as k increases and hence expand more states per search.

- In both static and dynamic environments, A* has a smaller runtime than Differential A* because Differential A* constructs its current search tree from scratch and thus expands the same number of states per search as A* but also deletes all states from the previous search tree.

- In both static and dynamic environments, GAA* has a smaller runtime than A* because GAA* updates the *h*-values to make them more informed over time and hence expands fewer states per search.

- In static environments, G-FRA* has a smaller runtime than A* because G-FRA* does not expand the states in the subtree of the previous search tree rooted at the current start state. A*, on the other hand, expands some of these states.

- In both static and dynamic environments, Basic MT-D* Lite and MT-D* Lite have smaller runtimes than GAA* because GAA* constructs its current search tree from scratch. Basic MT-D* Lite and D* Lite, on the other hand, reuse the previous search tree and hence expand fewer states per search.

- In both static and dynamic environments, MT-D* Lite has a smaller runtime than Basic MT-D* Lite because Basic MT-D* Lite expands the states in the previous search tree that are not in the subtree rooted at the current start state to set their g-values to infinity. MT-D* Lite, on the other hand, uses OptimizedDeletion() instead, which runs faster and results in only a slightly larger number of deleted and expanded states.

- In static environments, FRA* and G-FRA* have smaller runtimes than Basic MT-D* Lite and MT-D* Lite because of two reasons: (1) FRA* and G-FRA* have a smaller runtime per state expansion than Basic MT-D* Lite and MT-D* Lite. For example, the approximate overhead per state expansion (calculated by dividing the runtime per search by the number of expanded states per search) is 0.77 and 0.71 microseconds for FRA* and G-FRA*, respectively, while the approximate overhead per state expansion is 1.07 and 1.13 microseconds for Basic MT-D* Lite and MT-D* Lite, respectively. (2) FRA* and G-FRA* expand fewer states than Basic MT-D* Lite and MT-D* Lite in the following case: If the target moves to a state in the subtree of the previous search tree that rooted at the current start state, FRA* and G-FRA* terminate without expanding states due to Step 3 (Terminating Early). Basic MT-D* Lite and MT-D* Lite, on the other hand, expand all locally inconsistent states whose f-values are smaller than the f-value of the goal state.

- In static environments, FRA* has a smaller runtime than G-FRA* because G-FRA* reuses only the subtree of the previous search tree rooted at the current start state. FRA*, on the other hand, uses an optimization step for gridworlds, described in [13], that allows it to reuse more of the previous search tree. Thus, FRA* deletes and expands fewer states per search.

Overall, FRA* has the smallest runtime in static environments, and MT-D* Lite has the smallest runtime in dynamic environments, where it is four to five times faster than GAA*.

6. CONCLUSIONS

Existing incremental search algorithms are slow on moving target search problems in dynamic environments. In this paper, we therefore introduced MT-D* Lite, an extension of D* Lite that uses the principle behind Generalized Fringe-Retrieving A* to solve moving target search problems in dynamic environments fast. We demonstrated experimentally that MT-D* Lite is four to five times faster

than Generalized Adaptive A*, which so far was believed to be the fastest incremental search algorithm for solving moving target search problems in dynamic environments. It is future work to investigate how to speed up the computation of more sophisticated strategies for the hunter [17].

7. REFERENCES

[1] V. Bulitko, Y. Bjornsson, M. Luvstrek, J. Schaeffer, and S. Sigmundarson. Dynamic control in path-planning with real-time heuristic search. In *Proceedings of ICAPS*, pages 49–56, 2007.

[2] V. Bulitko, N. Sturtevant, J. Lu, and T. Yau. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research*, 30(1):51–100, 2007.

[3] 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*, 2:100–107, 1968.

[4] R. Holte, T. Mkadmi, R. Zimmer, and A. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1–2):321–361, 1996.

[5] T. Ishida and R. Korf. Moving target search. In *Proceedings of IJCAI*, pages 204–211, 1991.

[6] S. Koenig and M. Likhachev. D* Lite. In *Proceedings of AAAI*, pages 476–483, 2002.

[7] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *Transaction on Robotics*, 21(3):354–363, 2005.

[8] S. Koenig, M. Likhachev, and X. Sun. Speeding up moving-target search. In *Proceedings of AAMAS*, pages 1136–1143, 2007.

[9] C. Moldenhauer and N. Sturtevant. Optimal solutions for moving target search (Extended Abstract). In *Proceedings of AAMAS*, pages 1249–1250, 2009.

[10] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[11] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of IJCAI*, pages 1652–1659, 1995.

[12] X. Sun, S. Koenig, and W. Yeoh. Generalized Adaptive A*. In *Proceedings of AAMAS*, pages 469–476, 2008.

[13] X. Sun, W. Yeoh, and S. Koenig. Efficient incremental search for moving target search. In *Proceedings of IJCAI*, pages 615–620, 2009.

[14] X. Sun, W. Yeoh, and S. Koenig. Generalized Fringe-Retrieving A*: Faster moving-target search on state lattices. In *Proceedings of AAMAS*, 2010.

[15] K. Trovato and L. Dorst. Differential A*. *IEEE Transactions on Knowledge and Data Engineering*, 14(6):1218–1229, 2002.

[16] C. Undeger and F. Polat. Real-time edge follow: A real-time path search approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 37(5):860–872, 2007.

[17] C. Undeger and F. Polat. Multi-agent real-time pursuit. *Autonomous Agents and Multi-Agent Systems*, pages 1–39, 2009.