Generalized Fringe-Retrieving A*: Faster Moving Target Search on State Lattices

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

Moving target search is important for robotics applications where unmanned ground vehicles (UGVs) have to follow other friendly or hostile UGVs. Artificial intelligence researchers have recently used incremental search to speed up the computation of a simple strategy for the hunter. The fastest incremental search algorithm, Fringe-Retrieving A*, solves moving target search problems only on two-dimensional grids, which are rather unrealistic models for robotics applications. We therefore generalize it to Generalized Fringe-Retrieving A*, which solves moving target search problems on arbitrary graphs, including the state lattices used for UGV navigation.

Categories and Subject Descriptors

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

General Terms

Algorithm; Experimentation; Performance

Keywords

A*; Fringe-Retrieving A*; Incremental Search; Moving Target Search; Path Planning; State Lattices; Unmanned Ground Vehicles

1. INTRODUCTION

Incremental search algorithms reuse information from previous searches to speed up the current search and are thus often able to find cost-minimal paths for series of similar search problems faster than by solving each search problem with A* [1] from scratch [5]. As a result, artificial intelligence researchers have recently used them to speed up the computation of a simple strategy for the hunter in the context of moving target search problems where the hunter uses the following strategy to catch a moving target in a known environment: The hunter always follows a cost-minimal path from its current state to the current state of the target and replans a new path whenever the target moves off the previous path. It repeats this process until it is in the same state as the target, at which point the target is caught [6, 13, 14]. Moving target search is important for robotics applications where unmanned ground vehicles (UGVs) have to (a) follow other friendly or hostile UGVs, (b) move from one surveillance position to the next one as their surveillance target moves or (c) find an empty parking spot in a parking lot, where a different parking spot becomes their destination in case their parking spot is taken by a different vehicle before they get to it. Fringe-Retrieving A* (FRA*) is the fastest incremental search algorithm for recomputing paths for the hunter on two-dimensional grids according to a first study in [14]. It starts A* with initial *OPEN* and *CLOSED* lists obtained from the previous search rather than from scratch. Although two-dimensional grids are the standard test domains for moving target search in artificial intelligence [4, 10], they are rather unrealistic models for UGV navigation since they are unable to model motion constraints. Unfortunately, FRA* uses geometric properties that are specific to two-dimensional grids and thus does not apply to arbitrary graphs. We therefore generalize it to Generalized FRA* (G-FRA*), which solves moving target search problems on arbitrary graphs, including the state lattices used for UGV navigation [15].

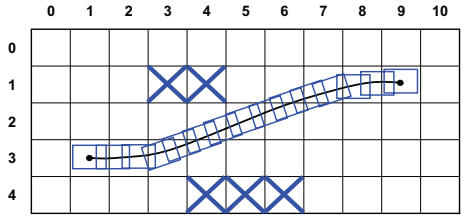


Figure 1: Example of Feasible Motion Primitive

2. STATE LATTICES

Two-dimensional grids are the standard test domains for moving target search in artificial intelligence. They consist of blocked and unblocked cells. The states are the unblocked cells, and both the hunter and target are able to move from their current cell to any adjacent unblocked cell [4, 6, 13, 14, 10]. State lattices are extensions of two-dimensional grids that are able to model motion constraints [12, 9, 7] and are therefore well suited to planning for non-holonomic and highly constrained robotic systems with limited maneuver-

Cite as: Generalized Fringe-Retrieving A*: Faster Moving Target Search on State Lattices, X. Sun, W. Yeoh and S. Koenig, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 1081-1088

ability, such as unmanned ground vehicles (UGVs) [9, 15]. A state lattice is constructed by discretizing the configuration space into a high-dimensional grid and connecting the cells of the grid with motion primitives, which are the building blocks for more complicated motions. A state in a state lattice is a tuple (x, y, θ), where x and y are the location of the center of the UGV and θ is its orientation. A motion primitive is feasible in a state iff the UGV does not collide with obstacles when executing it in that state. Ideally, an edge from state u to state v in a state lattice exists iff there is a feasible motion primitive in u whose execution results in v. State lattices often include only a subset of edges to make path planning fast [3, 9]. Figure 1 shows a feasible motion primitive in state $(1, 3, 90^\circ)$, whose execution results in state $(9, 1, 90^\circ)$, where 90° is to the right. Blue crosses are obstacles. The black line represents the path of the center of the UGV, and the blue rectangles are the perimeters of the UGV as it executes the motion primitive.

3. INCREMENTAL SEARCH

Fringe-Retrieving A* (FRA*) is an incremental search algorithm that is based on A* but solves moving target search problems only on two-dimensional grids. We therefore generalize it to Generalized FRA* (G-FRA*), which solves moving target search problems on arbitrary graphs, including state lattices. We first describe A*, then FRA* and finally G-FRA*.

3.1 Notation

We use the following notation: (1) S denotes the finite set of states, (2) $s_{start} \in S$ denotes the current state of the hunter and the start state of the search, (3) $s_{goal} \in S$ denotes the current state of the target and the goal state of the search, (4) $Succ(s) \subseteq S$ denotes the set of successor states of state $s \in S$, (5) $Pred(s) \subseteq S$ denotes the set of predecessor states of state $s \in S$, and (6) $c(s, s')$ denotes the cost of a cost-minimal path from state $s \in S$ to state $s' \in S$.

3.2 A*

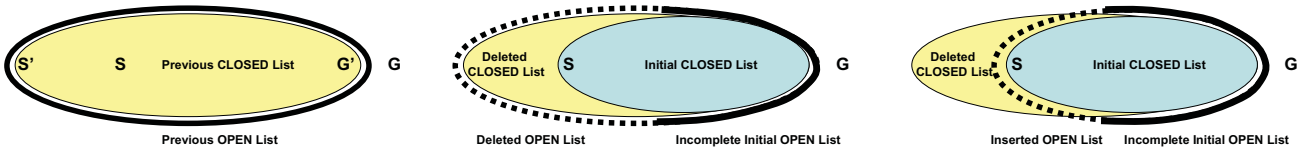
A* [1] is a search algorithm that provides the foundation for FRA* and G-FRA*. Our description closely follows [13, 14]. A* uses user-provided h-values to focus its search, where the h-value $h(s, s')$ is an approximation of $c(s, s')$. The h-values have to be consistent [11]. A* maintains two values for every state $s \in S$: (1) The g-value $g(s)$ is an approximation of $c(s_{start}, s)$. Initially, it is ∞. (2) The parent pointer $parent(s)$ points to the parent of state s in the search tree. Initially, it is *NULL*. A* also maintains the *OPEN* and *CLOSED* lists, whose states together form the search tree. At the start of a search for a cost-minimal path from the start state to the goal state, A* sets the g-value of the start state to zero and initializes the *OPEN* list to contain the start state and the *CLOSED* list to be empty. It then repeats the following procedure: It deletes a state s with the smallest $g(s) + h(s, s_{goal})$ from the *OPEN* list, inserts it into the *CLOSED* list and expands it by performing the following operations for each successor $s' \in Succ(s)$: If s' is neither in the *OPEN* nor *CLOSED* lists, then A* generates it by setting $g(s') := g(s) + c(s, s')$, setting $parent(s') := s$ and inserting it into the *OPEN* list. If s' is already in the *OPEN* list and $g(s') > g(s) + c(s, s')$, then A* re-generates it by setting $g(s') := g(s) + c(s, s')$ and $parent(s') := s$. A* terminates when its *OPEN* list is empty, which indicates that no path exists from the start state to the goal state, or when it expands the goal state, which indicates that A* found a cost-minimal path from the start state to the goal state. A* has the following properties. **Property 1:** The *OPEN* list contains all states that are not in the *CLOSED* list but have some predecessor in the *CLOSED* list, except at the start of the search when it contains only the start state. The parent pointer of a state s in the *OPEN* list (with the exception of the start state) points to the predecessor s' in the *CLOSED* list that minimizes $g(s') + c(s', s)$. The g-value of state s is $g(s') + c(s', s)$ for this predecessor s'. **Property 2:** For any two states $s, s' \in S$ in the *CLOSED* list so that s' is in the subtree of the search tree rooted at s, a cost-minimal path from s to s' is obtained when following the parent pointers in reverse and its cost is $g(s') - g(s)$.

3.3 FRA*

Fringe-Retrieving A* (FRA*) [14] is an incremental search algorithm that G-FRA* extends. Its pseudocode in Figure 3 and our description closely follow [14].[1] At the start of a moving target search, FRA* runs an A* search to find a cost-minimal path from the current state of the hunter to the current state of the target. The hunter then moves along the path until it catches the target or the target moves off the path. In the latter case, FRA* could run an A* search from scratch to find a new cost-minimal path from the current start state (= the current state of the hunter) to the current goal state (= the current state of the target). However, FRA* runs an A* search with initial *OPEN* and *CLOSED* lists obtained from the previous search rather than from scratch. Each A* run is called a search iteration. A search iteration finds a cost-minimal path from the current start state to the current goal state since it maintains Properties 1 and 2 of A*. It can run faster than an A* search from scratch since it does not expand the states in the initial *CLOSED* list. Figure 2 illustrates the operations during a search iteration. In the beginning of a search iteration, the initial *OPEN* and *CLOSED* lists are the same as the previous *OPEN* and *CLOSED* lists (= the *OPEN* and *CLOSED* lists at the end of the previous search iteration), respectively. Figure 2(a) visualizes the previous *OPEN* and *CLOSED* lists. **S'** and **S** represent the previous and current start states, respectively, and **G'** and **G** represent the previous and current goal states, respectively.

- **Step 1 (Starting A* Immediately):** FRA* executes this step if it did not terminate in Step 3 in the previous search iteration. If the previous and current start states are identical, then FRA* executes the remainder of this step and then skips Steps 2 to 5. If the current goal state is in the initial *CLOSED* list (Line 54), then the previous search already determined a cost-minimal path from the current start state to the current goal state. FRA* thus determines a cost-minimal path from the current start state to the current goal state by following the parent pointers in reverse. If the current goal state is not in the initial *CLOSED* list (Line 54), then the previous search can be continued to determine a cost-minimal path

[1]We omit the optional Changing Parents step of FRA* since it uses geometric properties that are specific to two-dimensional grids. G-FRA* applies to arbitrary graphs and thus cannot use this step.



(a) Before Step 1 (b) After Step 2 (c) After Step 4

Figure 2: Illustration of the Operations of FRA*

from the current start state to the current goal state. FRA* thus runs an A* search with the initial *OPEN* and *CLOSED* lists (Line 51) and then determines a cost-minimal path from the current start state to the current goal state by following the parent pointers in reverse.

- **Step 2 (Deleting States):** FRA* deletes all states in the search tree that are not in the subtree rooted at the current start state by deleting them from the initial *OPEN* and *CLOSED* lists (Lines 26-29). Figure 2(b) visualizes the initial *OPEN* and *CLOSED* lists after Step 2. The dotted line represents the states deleted from the initial *OPEN* list (called "deleted *OPEN* list"), the yellow area represents the states deleted from the initial *CLOSED* list (called "deleted *CLOSED* list"), the solid line represents the remaining initial *OPEN* list (called "incomplete initial *OPEN* list"), and the blue area represents the remaining initial *CLOSED* list (called "initial *CLOSED* list").

- **Step 3 (Terminating Early):** If the current goal state is in the initial *CLOSED* list (Line 54), then the previous search already determined a cost-minimal path from the current start state to the current goal state. FRA* thus determines a cost-minimal path from the current start state to the current goal state by following the parent pointers in reverse and skips Steps 4 and 5.

- **Step 4 (Inserting States):** The initial *OPEN* list can be incomplete since, according to Property 1, it needs to contain all states that are not in the initial *CLOSED* list but have some predecessor in the initial *CLOSED* list. Since the states in the initial *CLOSED* list form a contiguous area on two-dimensional grids, FRA* completes the initial *OPEN* list by traversing (the relevant part of) the outer perimeter of this contiguous area and performing the following check for all encountered states: If the state is not in the initial *OPEN* list yet but one of its predecessors is in the initial *CLOSED* list (Lines 37-38), then FRA* sets its parent pointer and g-value according to Property 1 and inserts it into the initial *OPEN* list (Lines 33 and 39-40). Figure 2(c) visualizes the initial *OPEN* and *CLOSED* lists after Step 4. The dotted line represents the inserted states that complete the initial *OPEN* list (called "inserted *OPEN* list"). The solid and dotted lines together represent the initial *OPEN* list.

- **Step 5 (Starting A*):** FRA* runs an A* search with the initial *OPEN* and *CLOSED* lists (Line 51) and then determines a cost-minimal path from the current start state to the current goal state by following the parent pointers in reverse.

3.4 G-FRA*

FRA* solves moving target search problems only on two-dimensional grids since Step 4 uses geometric properties that are specific to them. Figure 5(a) shows an example eight-neighbor grid where **C** represents the states in the initial *CLOSED* list and **O** represents the states that have to be inserted into the initial *OPEN* list to complete it. Step 4 of FRA* identifies the states that have to be inserted into the initial *OPEN* list by traversing the outer perimeter of the initial *CLOSED* list, which is possible since the states on the outer perimeter of the initial *CLOSED* list are connected on two-dimensional grids. However, Figure 5(c) shows that this is not necessarily the case on arbitrary graphs. Thus, Step 4 has to identify the states that have to be inserted into the initial *OPEN* list in a different way. Step 4 could iterate over the states in the initial *CLOSED* list and insert their successors that are not in the initial *CLOSED* list into the initial *OPEN* list. However, iterating over the states in the initial *CLOSED* list can be slow since the initial *CLOSED* list can be large. Instead, Step 4 iterates over the states deleted from the initial *OPEN* and *CLOSED* lists in Step 2 and inserts those states that have a predecessor in the initial *CLOSED* list into the initial *OPEN* list. G-FRA*, the resulting version of FRA*, solves moving target search problems on arbitrary graphs. Figure 4 shows the necessary changes to the pseudocode from Figure 3. G-FRA* maintains a *DELETED* list, that contains all states deleted from the initial *OPEN* and *CLOSED* lists in all executions of Step 2 since G-FRA* ran the last A* search. G-FRA* initializes the *DELETED* list to empty between Lines 49 and 50, calls GeneralizedStep2() on Line 63 instead of Step2() and calls GeneralizedStep4() on Line 67 instead of Step4(). In Step 2, G-FRA* now also inserts all states deleted from the initial *OPEN* and *CLOSED* lists (that is, all states in the search tree that are not in the subtree rooted at the current start state) into the *DELETED* list (Line 72). In Step 4, G-FRA* completes the initial *OPEN* list by performing the following check for all states in the *DELETED* list before it sets the *DELETED* list to empty again (Line 87): If the state has a predecessor in the initial *CLOSED* list (Line 80), then G-FRA* sets its parent pointer and g-value according to Property 1 and inserts it into the initial *OPEN* list (Lines 84-86). According to Property 1, the initial *OPEN* list has to contain all states that are not in the initial *CLOSED* list but have some predecessor in the initial *CLOSED* list. The correctness of G-FRA* follows from the fact that every state that has a predecessor in the initial *CLOSED* list when G-

```
01 procedure InitializeState(s)
02 if (generatediteration(s) ≠ iteration)
03    g(s) := ∞;
04    generatediteration(s) := iteration;
05    expanded(s) := false;

06 function TestClosedList(s)
07 return (s = s_start OR (expanded(s) AND parent(s) ≠ NULL));

08 function ComputeCostMinimalPath()
09 while (OPEN ≠ ∅)
10    s := arg min_{s∈OPEN}(g(s) + h(s, s_goal));
11    OPEN := OPEN \ {s};
12    expanded(s) := true;
13    forall s' ∈ Succ(s)
14       if (NOT TestClosedList(s'))
15          InitializeState(s');
16          if (g(s') > g(s) + c(s, s'))
17             g(s') := g(s) + c(s, s');
18             parent(s') := s;
19             if (s' ∉ OPEN)
20                OPEN := OPEN ∪ {s'};
21    if (s = s_goal)
22       return true;
23 return false;

24 procedure Step2()
25 parent(s_start) := NULL;
26 forall s ∈ S in the search tree rooted at previous_s_start
      but not the subtree rooted at s_start
27    parent(s) := NULL;
28    if (s ∈ OPEN)
29       OPEN := OPEN \ {s};

30 procedure Step4()
31 forall s ∈ S on relevant part of outer perimeter of CLOSED list[a]
32    if (s ∉ OPEN AND ∃s' ∈ Pred(s) : TestClosedList(s'))
33       OPEN := OPEN ∪ {s};
34 forall s ∈ OPEN
35    InitializeState(s);
36 forall s ∈ OPEN
37    forall s' ∈ Pred(s)
38       if (TestClosedList(s') AND g(s) > g(s') + c(s', s))
39          g(s) := g(s') + c(s', s);
40          parent(s) := s';

41 function Main()
42 forall s ∈ S
43    generatediteration(s) := 0;
44    expanded(s) := false;
45    parent(s) := NULL;
46 iteration := 1;
47 InitializeState(s_start);
48 g(s_start) := 0;
49 OPEN := {s_start};
50 while (s_start ≠ s_goal)
51    if (NOT ComputeCostMinimalPath())
52       return false;
53    openlist_incomplete := false;
54    while (TestClosedList(s_goal))
55       while (target is on path from s_start to s_goal and not caught)
56          follow cost-minimal path from s_start to s_goal;
57       if (target is caught)
58          return true;
59       previous_s_start := s_start;
60       s_start := the current state of the hunter;
61       s_goal := the current state of the target;
62       if (s_start ≠ previous_s_start)
63          Step2();
64          openlist_incomplete := true;
65    if (openlist_incomplete)
66       iteration := iteration + 1;
67       Step4();
68 return true;
```

[a] The *CLOSED* list contains all states $s' \in S$ with TestClosedList(s').

Figure 3: FRA*

```
69 procedure GeneralizedStep2()
70 parent(s_start) := NULL;
71 forall s ∈ S in the search tree rooted at previous_s_start
      but not the subtree rooted at s_start
72    DELETED := DELETED ∪ {s};
73    parent(s) := NULL;
74    if (s ∈ OPEN)
75       OPEN := OPEN \ {s};

76 procedure GeneralizedStep4()
77 forall s ∈ OPEN
78    generatediteration(s) := iteration;
79 forall s ∈ DELETED
80    if (∃s' ∈ Pred(s) : TestClosedList(s'))
81       InitializeState(s);
82       forall s' ∈ Pred(s)
83          if (TestClosedList(s') AND g(s) > g(s') + c(s', s))
84             g(s) := g(s') + c(s', s);
85             parent(s) := s';
86       OPEN := OPEN ∪ {s};
87 DELETED := ∅;
```

Figure 4: Generalized Fringe-Retrieving A*

FRA* executes Step 4 was in the *OPEN* or *CLOSED* lists after the last A* search and thus is in the initial *CLOSED* list, the initial *OPEN* list or the *DELETED* list when G-FRA* executes Step 4. We distinguish three cases:

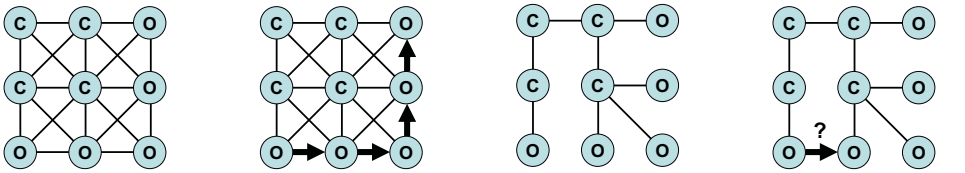
- If a state is in the initial *CLOSED* list when G-FRA* executes Step 4, then it was in the *CLOSED* list and thus not in the *OPEN* list after the last A* search and still is not in the initial *OPEN* list when G-FRA* executes Step 4. G-FRA* does nothing since it should indeed not be in the initial *OPEN* list when G-FRA* executes Step 4.

- If a state is in the initial *OPEN* list when G-FRA* executes Step 4, then it was in the *OPEN* list and thus was not in the *CLOSED* list but had some predecessor in the *CLOSED* list after the last A* search and this still holds when G-FRA* executes Step 4 (otherwise it would have been deleted from the initial *OPEN* list in Step 2). G-FRA* does nothing since it should indeed be in the initial *OPEN* list when G-FRA* executes Step 4 and its parent pointer and g-value still satisfy Property 1.

- If a state is in the *DELETED* list, then it was deleted from the initial *OPEN* and *CLOSED* lists in Step 2. Thus, if it has some predecessor in the initial *CLOSED* list, G-FRA* inserts it into the initial *OPEN* list in Step 4 and sets its parent pointer and g-value according to Property 1.

4. EXPERIMENTAL EVALUATION

FRA* is the fastest incremental search algorithm for moving target search on two-dimensional grids according to a first study in [14]. Thus, G-FRA* could be fast for moving target search on arbitrary graphs, including state lattices. Generalized Adaptive A* (GAA*, sometimes also called Generalized MT-Adaptive A*) [13] is the fastest incremental search algorithm so far for moving target search on arbitrary graphs according to a first study in [13]. GAA* is a generalization of Moving-Target Adaptive A* [6], which build on an idea in [2]. Different from FRA* and G-FRA*, GAA* does not speed up the current search by reusing parts



(a) Example Grid (b) Traversing the Outer Perimeter (c) Example Graph (d) Inability to Traverse the Outer Perimeter

Figure 5: Limitations of FRA*

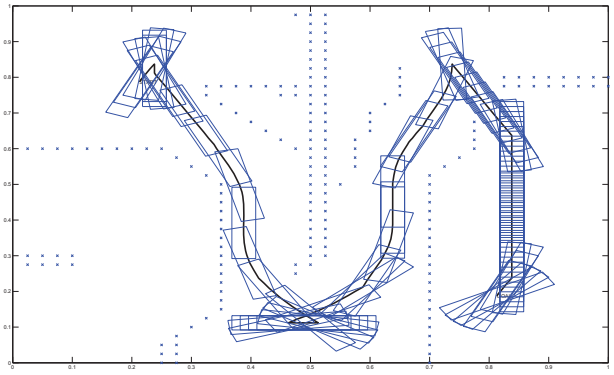
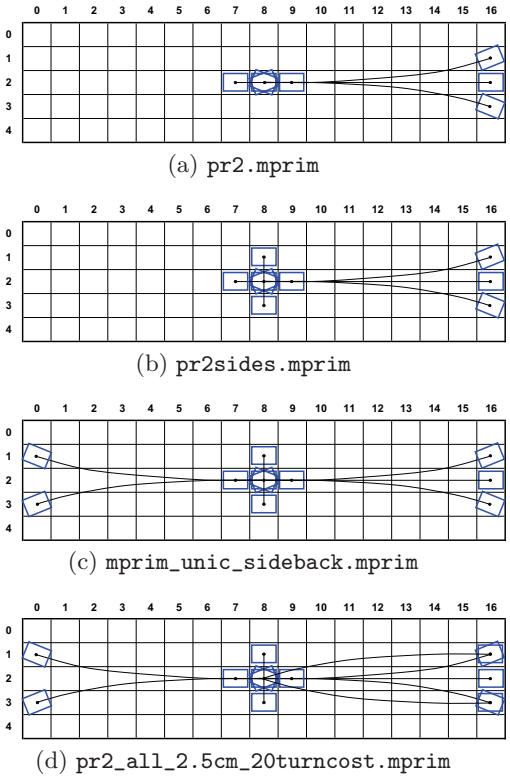


Figure 6: Example UGV Path



(a) `pr2.mprim`

(b) `pr2sides.mprim`

(c) `mprim_unic_sideback.mprim`

(d) `pr2_all_2.5cm_20turncost.mprim`

Figure 7: Motion Primitives

of the previous search tree but by making the previous h-values more informed. We therefore perform preliminary experiments that compare G-FRA* against running A* from scratch and GAA* on known state lattices. We use similar experimental settings as earlier experiments that compared FRA* against running A* from scratch and GAA* on two-dimensional grids [14]. They allow us to determine the influence of the graph topology on the runtime of the incremental search algorithms and to use SBPL both for planning and execution (instead of using a motion simulator) but do not result in a realistic simulation of moving target search with UGVs since the UGV and target take turns executing uninterruptible actions with potentially different execution times. We implemented all search algorithms in similar ways. For example, they all find cost-minimal paths from the current state of the UGV to the current state of the target and re-plan when the target moves off the path, and they all use a binary heap to implement the *OPEN* list.

4.1 SBPL

We implement all search algorithms in the Search-Based Planning Library (SBPL) [8], a publicly available library for UGV navigation with state lattices that is written in C++. SBPL discretizes the orientation θ into multiples of 22.5° and constructs the state lattices from three inputs, namely the size of the UGV, an environment definition file and a motion primitive definition file. An environment definition file defines the size of the grid, the size of each cell, the locations of obstacles, the translational and rotational velocities of the UGV and the start and goal states of the UGV. A motion primitive definition file defines the motion primitives in each state. The cost of a motion primitive is its execution time. Figure 6 shows an example UGV path for a state lattice constructed by SBPL. We use the default size of the UGV, namely 20 centimeters $\times$ 4 centimeters. We use one of the example environment definition files (`env2.cfg`), which defines the size of the grid to be 1200 cells $\times$ 100 cells, the size of each cell to be 2.5 centimeters $\times$ 2.5 centimeters, the translational velocity of the UGV to be 1 meter per second and the rotational velocity of the UGV to be 22.5° per second. We use four motion primitive definition files together with the default h-values, namely the amount of time needed for the UGV to move to the goal state at maximal speed in a straight line in the absence of obstacles and motion constraints. Figure 7 shows the motion primitives defined in the four files for state $(8, 2, 90^\circ)$. The target moves randomly with the same translational and rotational velocities as the UGV but skips every tenth move.

	searches per test case	costs per test case	states expanded per search	states deleted per search	states reused per search	runtime per search
A*	157	37,741	83,303			984
GAA*	154	39,978	57,864			655
G-FRA*	151	40,056	12,511	4,062	117,901	61

`pr2.mprim`
(7 Motion Primitives per State)

A*	161	32,240	118,965			1,570
GAA*	145	32,466	68,674			773
G-FRA*	176	32,132	13,361	5,408	125,144	82

`pr2sides.mprim`
(9 Motion Primitives per State)

A*	156	32,437	147,172			2,017
GAA*	153	32,136	86,770			1,122
G-FRA*	174	32,749	15,463	5,714	124,098	116

`mprim_unic_sideback.mprim`
(11 Motion Primitives per State)

A*	172	39,820	166,321			2,554
GAA*	160	40,562	103,326			1,313
G-FRA*	166	42,941	15,942	5,718	129,093	130

`pr2_all_2.5cm_20turncost.mprim`
(13 Motion Primitives per State)

Table 1: Experimental Results

4.2 Results

Table 1 reports two measures for the difficulty of the moving target search problems, namely the average number of searches and the average cost of the UGV path until it catches the target. These values vary slightly among the different moving target search algorithms due to them breaking ties differently among several cost-minimal paths. The table reports two measures for the efficiency of the moving target search algorithms, namely the average number of expanded states per search and the average runtime per search in milliseconds on a Pentium D 3.0 Ghz PC with 2 GByte of RAM. For G-FRA*, the table also reports the average numbers of deleted states per search (= number of states deleted from the search tree in Step 2) and reused states per search (= number of states remaining in the search tree after Step 2). We make the following observations:

- A*, GAA* and G-FRA* have runtimes per search that generally increase with the number of motion primitives per state. As the number of motion primitives per state increases, the number of successors per state increases, which in turn increases the runtime per state expansion.
- GAA* has a smaller runtime per search than A* because it makes the h-values more informed over time and hence expands fewer states per search.
- G-FRA* has a smaller runtime than A* and GAA* because it reuses the previous search tree and hence expand fewer states per search than A* and GAA*. The runtime per search of G-FRA* is about 5.3 to 29.9 times smaller than the one of A* and about 3.0 to 23.6 times smaller than the one of GAA*.

Overall, these results for G-FRA* on state lattices are similar to earlier results that compared FRA* against running A* from scratch and GAA* on two-dimensional grids using similar experimental settings [14]. They show that G-FRA* can be up to one order of magnitude faster than GAA*.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we used incremental search to speed up the computation of a simple strategy for the hunter in the context of moving target search problems where the hunter has to catch a moving target in a known environment. Our main technical contribution was the development G-FRA*, a generalization of the incremental search algorithm FRA* from two-dimensional grids to arbitrary graphs. Our experiments showed that G-FRA* can be up to one order of magnitude faster than GAA*, the fastest incremental search algorithm so far for solving moving target search problems on arbitrary graphs according to a first study in [13]. G-FRA* thus seems to be a promising building block for applying incremental search to moving target search with UGVs. However, we did not attempt to build on a realistic simulation of moving target search with UGVs because we wanted to use similar experimental settings as earlier experiments. The resulting limitations of our experiments are as follows: We used a coarse-grained discrete simulation on the motion-primitive level. The UGV and target had symmetrical motion capabilities and took turns executing uninterruptible actions with potentially different execution times. The motion strategy of the target was also simplistic. It is therefore future work to evaluate G-FRA* as part of a realistic simulation of moving target search with UGVs, where (a) a motion simulator provides a continuous simulation, (b) the UGV and target are able move in parallel, (c) the UGV and target are able to interrupt their execution of motion primitives, (d) the UGV and target have different motion capabilities, and (e) the target uses a more sophisticated motion strategy. It is also future work to exploit the structure of graphs to speed up the computation of simple strategies for the UGV even more and, more importantly, to speed up the computation of more complex strategies for the UGV, such as minimax strategies [10], and to generalize the results to moving target search in unknown environments.

Acknowledgments

We thank Maxim Likhachev, the creator of SBPL, for his help and the anonymous reviewers for their helpful comments. This material is based upon work supported by, or in part by, NSF under contract/grant number 0413196, ARL/ARO under contract/grant number W911NF-08-1-0468 and ONR in form of a MURI under contract/grant number N00014-09-1-1031. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

6. REFERENCES

[1] 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.

[2] 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.

[3] T. Howard and A. Kelly. Optimal rough terrain trajectory generation for wheeled mobile robots.

International Journal of Robotics Research, 26(1):141–166, 2007.

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

[5] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy. Incremental heuristic search in artificial intelligence. *AI Magazine*, 25(2):99–112, 2004.

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

[7] A. Kushleyev and M. Likhachev. Time-bounded lattice for efficient planning in dynamic environments. In *Proceedings of ICRA*, pages 1662–1668, 2009.

[8] M. Likhachev. SBPL graph search library, 2009. `www.seas.upenn.edu/~maximl/software.html`.

[9] M. Likhachev and D. Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *International Journal of Robotics Research*, 28(8):933–945, 2009.

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

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

[12] M. Pivtoraiko and A. Kelly. Generating near minimal spanning control sets for constrained motion planning in discrete state spaces. In *Proceedings of IROS*, pages 3231–3237, 2005.

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

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

[15] C. Urmson et al. Autonomous driving in urban environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 25(1):425–466, 2008.