Sven Koenig, Maxim Likhachev, and Xiaoxun Sun. 2007. Speeding up moving-target search. In Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems (AAMAS '07). ACM, New York, NY, USA, , Article 188 , 8 pages. DOI=10.1145/1329125.1329353 http://doi.acm.org/10.1145/1329125.1329353

Speeding up Moving-Target Search

Sven Koenig
USC
CS Department
skoenig@usc.edu

Maxim Likhachev
CMU
Robotics Institute
maxim+@cs.cmu.edu

Xiaoxun Sun
USC
CS Department
xiaoxuns@usc.edu

ABSTRACT

In this paper, we study moving-target search, where an agent (= hunter) has to catch a moving target (= prey). The agent does not necessarily know the terrain initially but can observe it within a certain sensor range around itself. It uses the strategy to always move on a shortest presumed unblocked path toward the target, which is a reasonable strategy for computer-controlled characters in video games. We study how the agent can find such paths faster by exploiting the fact that it performs A* searches repeatedly. To this end, we extend Adaptive A*, an incremental heuristic search method, to moving-target search and demonstrate experimentally that the resulting MT-Adaptive A* is faster than isolated A* searches and, in many situations, also D* Lite, a state-of-the-art incremental heuristic search method. In particular, it is faster than D* Lite by about one order of magnitude for moving-target search in known and initially unknown mazes if both search methods use the same informed heuristics.

Categories and Subject Descriptors

I.2 [**Artifical Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Performance, Experimentation, Theory

Keywords

D* Lite, Hunter, Incremental Heuristic Search, Moving-Target Search, MT-Adaptive A*, Planning with the Freespace Assumption, Target, Unknown Terrain, Video Games

1. INTRODUCTION

Moving-target search is the problem of catching a moving target in known or initially unknown terrain [1]. For example, computer-controlled characters in video games often need to chase other characters, including characters of the human players, despite having incomplete knowledge of the terrain. They always observe the terrain within their limited field of view and then remember it for future use but do not necessarily know the terrain initially or how it changes over time. The movement strategies of the computer have to satisfy several properties for moving-target search: 1) They should look believable. 2) They need to be challenging. 3) They should not be so good that the human players have no chance of escaping. 4) It must be possible to calculate them fast to enable the computer to move smoothly and without delay. A minimax strategy would allow the computer to corner its target, if possible, but might be too challenging for the human players and, most importantly, is time-consuming to compute. One possible strategy is the following one: The computer always moves on a shortest presumed unblocked path toward its target. This movement strategy looks believable, is challenging but not too challenging and can be calculated reasonably fast. It also applies to following (and catching up) with friendly characters in video games. Many computer-controlled characters can chase their targets at the same time in video games and each computer-controlled character needs to search repeatedly. It is therefore important for the searches to be as fast as possible since the computer needs to move smoothly and without delay even if it has a slow processor and the other game components use most of the available processor cycles. We study how the computer can find shortest paths faster by exploiting the fact that it performs A* searches repeatedly. We extend Adaptive A* [7], an incremental heuristic search method, to moving-target search and demonstrate experimentally that the resulting MT-Adaptive A* is faster than isolated A* searches and, in many situations, also D* Lite [5], a state-of-the-art incremental heuristic search method.

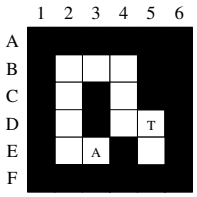

Figure 1: Gridworld

2. MOVING-TARGET SEARCH

We study a version of moving-target search in which an agent and a target move around in a gridworld. A gridworld is a discretization of terrain into square cells that are each either blocked or unblocked. The agent and the target can both move from their current cell to any one of the four adjacent cells, as long as that

AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.

978-81-904262-7-5 (RPS)

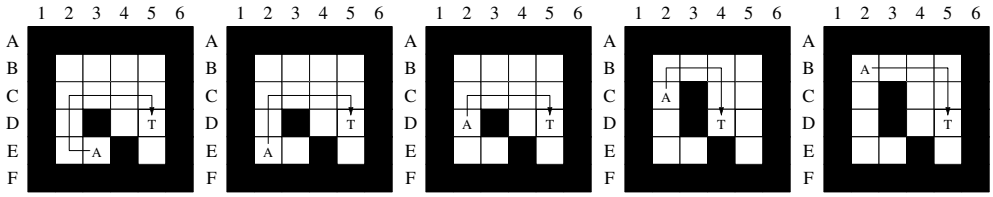

Figure 2: Moving-Target Search

cell is unblocked and still part of the gridworld. The agent always knows which (unblocked) cell it is in and which (unblocked) cell the target is in. Initially, the agent does not necessarily know which cells are blocked but it can always sense the blockage status of its four adjacent cells. Every move takes one time step for the agent and thus has cost one. The goal of the agent is to occupy the same cell as the target ("catch the target"). The agent always moves on a shortest presumed unblocked path (= a path that is not known to be blocked according to its current knowledge) from its current cell toward the current cell of the target (freespace assumption). If this path turns out to be blocked during execution or the target moves, then the agent stops following its path, finds another shortest presumed unblocked path from its current cell to the current cell of the target, taking into account the blockages that it knows about, and then moves along the new path, until it either catches the target or determines that it cannot catch the target because there is no shortest presumed unblocked path and it is thus separated from the target by blockages. An agent with this optimistic movement strategy is guaranteed to catch the target if it is not separated from it by blockages and the target either moves more slowly than the agent (for example, does not move from time to time) or moves suboptimally from time to time. Consider the gridworld from Figure 1. Black cells are blocked, and white cells are unblocked. The current cell of the agent is marked A, and the current cell of the target is marked T. Figure 2 illustrates the movement strategy of the agent in this gridworld. (The target does not move much in this example, different from our experiments.) Black cells are known to be blocked by the agent. White cells are either known to be unblocked or not known to be blocked and thus presumed to be unblocked. The arrow shows a shortest presumed unblocked path from the current cell of the agent (marked A) to the current cell of the target (marked T). In the first figure, the agent finds a shortest presumed unblocked path and then follows it for three movements. In the fourth figure, the agent finds a new shortest presumed unblocked path (since it observed its previous path to be blocked and the target to move) and then follows it for one movement. In the fifth figure, the agent finds another shortest presumed unblocked path (since it observed the target to move), and so on. We assume here for simplicity that the border of the gridworld is known to be blocked, and that the agent and the target have sensor range one, move in the four main compass directions, move by at most one cell and that every move has cost one, although these assumptions are not necessary. It is crucial, however, that blockages cannot change and the agent cannot forget about them.

3. EXISTING APPROACHES

Incremental heuristic search can be used to speed up the calculation of the paths. Incremental search methods reuse information from previous searches to find solutions to series of similar search problems potentially faster than isolated searches (which is important if the world or the agent's knowledge of the world change over time), while heuristic search methods - such as A* [13] - use heuristic knowledge in form of approximations of the goal distances to focus the search and solve search problems potentially faster than uninformed search methods. Incremental heuristic search methods combine these two approaches. The dominant line of incremental heuristic search methods, we call them LPA* variants, transforms the search tree of the previous (= immediately preceeding) A* search into the search tree of the current A* search, which requires the root of the search trees to remain unchanged in order for the A* search tree of the previous search to be re-usable. The overhead of LPA* variants over A* is substantial but they can decrease the number of state expansions dramatically and are then faster than A*. In particular, transforming the search tree of the previous A* search into the search tree of the current search can be faster than constructing the A* search tree of the current search from scratch if the two search trees are similar, which tends to be the case in our examples if only a small number of cells change their blockage status and these cells are close to the goal state. It can also be slower than constructing the search tree of the current search from scratch if the two search trees are different. In this case, LPA* variants should not be used since isolated A* searches are faster. A more detailed discussion of LPA* variants can be found in [10]. Some moving-target search methods use an amount of memory that is super-linear in the size of the gridworlds or implement navigation strategies different from ours, including navigation strategies that do not find complete paths [4]. We are not interested in them in this paper since they are based on different assumptions.

3.1 Stationary-Target Search

The first LPA* variant, D* [14], was originally developed in the context of a version of our moving-target search problem where the target does not move (stationary-target search). D* Lite [5] was later developed as an extension of LPA* [6] that results in a simple version of D* that is as fast as D* but easier to implement, understand and extend. We therefore use D* Lite in this paper. D* Lite repeatedly finds shortest paths in state spaces with the same start state but possibly different goal states whose action costs can increase or decrease between searches. It searches from the target to the current cell of the agent for stationary-target search since the agent moves but the target remains stationary. This way, the root of the search trees remains unchanged and the search tree of the previous search is re-usable. D* Lite is typically faster than isolated A* searches for stationary-target search since the agent observes blockages only around its current cell, which is the goal state. Thus, the search trees of the previous and current search are similar and the transformation of the search tree of the previous search into the search tree of the current search is fast.

3.2 Moving-Target Search

It is not straight-forward to apply D* Lite to moving-target search since both the agent and the target move but the root of the search trees needs to remain unchanged. D* Lite can search

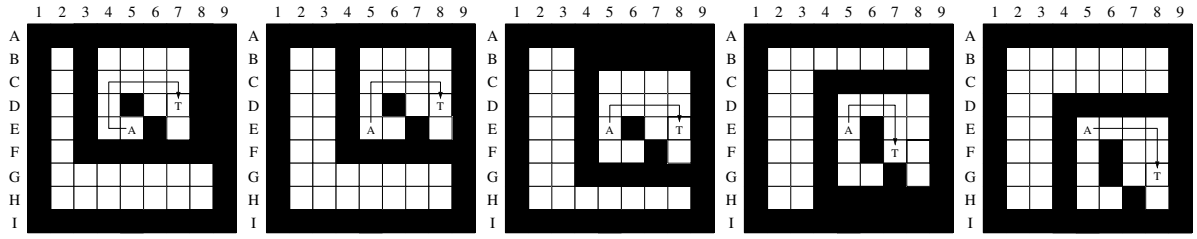

Figure 3: Agent-Centric Map of the Gridworld

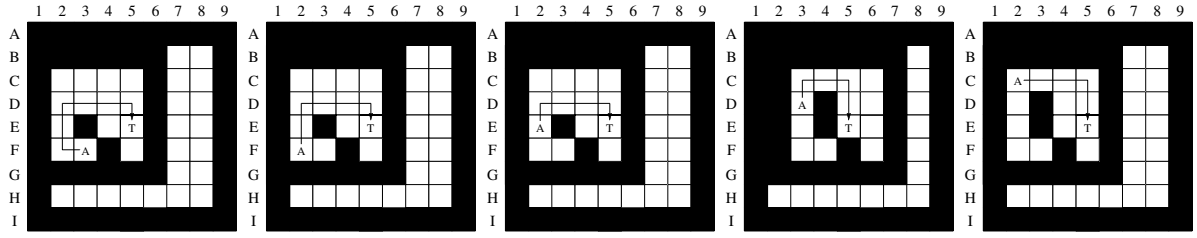

Figure 4: Target-Centric Map of the Gridworld

neither from the current cell of the target to the current cell of the agent nor from the current cell of the agent to the current cell of the target. However, it can solve moving-target search problems by using an agent-centric or target-centric map of the gridworld (as demonstrated by Anthony Stentz in his AAAI-02 Tutorial on Greedy On-Line Planning). An agent-centric map, for example, always has the agent in its center, as shown in Figure 3 for our example from Figure 2. D* Lite searches from the agent to the current cell of the target since the target moves but the agent remains stationary on the agent-centric map and thus needs to be the root of the search trees. Similarly, a target-centric map always has the target in its center, as shown in Figure 4 for our example from Figure 2. D* Lite searches from the target to the current cell of the agent since the agent moves but the target remains stationary on the target-centric map and thus needs to be the root of the search trees. We expect a target-centric version of D* Lite to be faster than an agent-centric version of D* Lite for stationary-target search, for two reasons: First, the map then does not need to be shifted repeatedly to keep the target in its center. Second, the changes in the map then occur only close to the current cell of the agent, which is the goal state. Thus, the search trees of the previous and current search are similar, and the transformation of the search tree of the previous search into the search tree of the current search is fast. However, we expect the runtime advantage of D* Lite over isolated A* searches to be smaller or even non-existent for moving-target search rather than stationary-target search because many cells can change their blockage status on a target-centric map when the target moves since the map then needs to be shifted repeatedly to keep the target in its center. In this case, many cells on the map can change their blockage status, including cells close to the start state. From the third to the fourth figure in Figure 4, for example, cells C6, D6, E3, E6, F4, and F6 become unblocked and cells B7, C2, C7, D2, D4, D7, E2, E4, E7, F2, F5, F7, and G7 become blocked. This slows down D* Lite since it needs to perform bookkeeping operations for each cell that changes its blockage status. Furthermore, this makes the search trees of the previous and current search different. The transformation of the search tree of the previous search into the search tree of the current one is thus slow. We therefore develop a different incremental heuristic search approach to moving-target search.

4. NEW APPROACH: MT-ADAPTIVE A*

Instead of building on LPA* variants, we build on Adaptive A*, an incremental heuristic search method that works according to a completely different principle. Adaptive A* [7] repeatedly finds shortest paths in state spaces with possibly different start states but the same goal state whose action costs can increase (but not decrease) between searches. It uses A* searches to find the shortest paths. However, it updates the h-values after each search to make them more informed and thus future searches more focused.

4.1 Notation

We use the following notation to describe Adaptive A* in arbitrary state spaces rather than the special case of gridworlds: S denotes the finite set of states. s_{start} denotes the start state, and s_{target} denotes the goal state. $A(s)$ denotes the finite set of actions that can be executed in state s. $c(s, a) > 0$ denotes the cost of executing action $a \in A(s)$ in state s, and $succ(s, a)$ denotes the resulting successor state. The start state can change from search to search, and the action costs can increase (but not decrease) from search to search. Each search of Adaptive A* finds a shortest path from the current start state to the goal state according to the current action costs. The user-supplied initial h-values $H(s, s')$ estimate the distance from any state s to any state s' and need to be consistent [13], that is, satisfy the triangle inequality: $H(s', s') = 0$ and $H(s, s') \leq c(s, s'') + H(s'', s')$ for every successor state s'' of state s. (We use the Manhattan distances in our examples.)

4.2 Review of A*

Adaptive A* performs A* searches. A* [2] is a heuristic search method for finding shortest paths in state spaces. For every state s, the user supplies a consistent h-value $h(s) := H(s, s_{\text{target}})$ that estimates the goal distance of state s. For every state s encountered during the search, A* maintains three values: a g-value $g(s)$ (which is infinity initially), which is the length of the shortest discovered path from the start state to state s; an f-value $f(s) := g(s) + h(s)$, which estimates the distance from the start state via state s to the

goal state; and a tree-pointer *tree*(s) (which is undefined initially), which is used to identify a shortest path after the search. A* then operates as follows: It maintains a priority queue, called *OPEN* list (which contains only the start state initially). A* removes a state s with the smallest f-value from the *OPEN* list. If the f-value of state s is no smaller than the f-value (or, equivalently, the g-value) of the goal state, it terminates. Otherwise, it expands state s, meaning that it inserts state s into the *CLOSED* list (which is empty initially) and then performs the following operations for each action that can be executed in state s and results in a successor state whose g-value is larger than the g-value of state s plus the action cost: First, it sets the g-value of the successor state to the g-value of state s plus the action cost. Second, it sets the tree-pointer of the successor state to point to state s. Finally, it inserts the successor state into the *OPEN* list if it was not there already. (We say that it generates a state when it inserts the state for the first time into the *OPEN* list.) It then repeats the process. After termination, the g-value of every expanded state (= state in the *CLOSED* list) is equal to the distance from the start state to the state, and following the tree-pointers from the goal state to the start state identifies a shortest path from the start state to the goal state in reverse.

4.3 Principle of Adaptive A*

The principle behind Adaptive A* is easy to describe. Let s denote any state that was expanded during an A* search and $g(s)$ denote its g-value after the search. Then, $g(s)$ is the distance from the start state to state s, and $g(s_{\text{target}})$ is the distance from the start state to the goal state, that is, the length of the shortest discovered path from the start state to the goal state. Adaptive A* updates the h-values of all states s that were expanded during the search as follows:

$$h(s) := g(s_{\text{target}}) - g(s). \tag{1}$$

It has to update the h-values after the search rather than during the search because it needs to know the length of the shortest discovered path, which is only known after the search. It turns out that the new h-values are consistent and dominate the previous h-values at least weakly [8], that is, cannot be smaller for any state. Thus, any A* search with the new h-values cannot expand more states than an otherwise identical A* search with any of the earlier h-values, including the user-supplied initial h-values. It therefore cannot be slower (except possibly for the small amount of overhead introduced by the bookkeeping actions), but will often expand fewer states and thus be faster. This principle has been used in [3] and later resulted in the independent development of Adaptive A*. The overhead of Adaptive A* over A* is small. In particular, it is smaller than the one of LPA* but Adaptive A* typically does not decrease the number of state expansions as much as LPA*. A more detailed discussion of Adaptive A* can be found in [8].

4.4 Stationary-Target Search

Adaptive A* applies in a straight-forward way to stationary-target search since it applies to search problems with changing start states and increasing action costs [7]. It searches from the current cell of the agent to the target since the agent moves but the target remains stationary and thus needs to be the goal since the goal needs to remain unchanged. Initially, all action costs are one. Whenever the agent discovers a blockage, it increases the costs of all actions that enter or leave the blocked cell from one to infinity.

4.5 Eager MT-Adaptive A*

Adaptive A* has not been applied to moving-target search since then both the agent and the target move but the goal needs to remain unchanged. We apply it to moving-target search by extending it to the case where the goal state changes between searches. This is not straight-forward since the h-values estimate the goal distances and thus need to be corrected when the goal state changes. Adaptive A* could attempt to calculate (over time) the distances between any two states but we only want to use an amount of memory that is linear in the size of the state space. We therefore continue to use a version of Adaptive A* that uses only the results of the previous search. MT-Adaptive A* (= Adaptive A* for moving-target search) continuously updates the h-values to make them more informed but now also corrects them to maintain their consistency with respect to the goal state whenever the goal state changes. To explain how it corrects the h-values when the goal state changes, assume that the h-values are consistent with respect to the previous goal state s_{target} after MT-Adaptive A* updated them after a search. Let $h(s)$ be the h-value of any state s at this point in time. MT-Adaptive A* then corrects the h-values for the new goal state s'_{target} with $s'_{\text{target}} \neq s_{\text{target}}$ by assigning

$$h'(s) := \max(H(s, s'_{\text{target}}), h(s) - h(s'_{\text{target}})) \tag{2}$$

for all states s.

THEOREM 1. *The h-values $h'(s)$ are consistent with respect to goal state s'_{target}.*

Proof: We consider three cases to prove that the triangle inequality holds with respect to goal state s'_{target} for the h-values $h'(s)$. In each case, we use the fact that the triangle inequality holds with respect to goal state s'_{target} for the user-supplied initial h-values $H(s, s'_{\text{target}})$ and with respect to goal state s_{target} for the h-values $h(s)$. First, if $s = s'_{\text{target}}$, then $h'(s'_{\text{target}}) = \max(H(s'_{\text{target}}, s'_{\text{target}}), h(s'_{\text{target}}) - h(s'_{\text{target}})) = \max(0, 0) = 0$. Second, if $s = s_{\text{target}}$ and thus $s \neq s'_{\text{target}}$, then $h'(s_{\text{target}}) = \max(H(s_{\text{target}}, s'_{\text{target}}), h(s_{\text{target}}) - h(s'_{\text{target}})) = \max(H(s_{\text{target}}, s'_{\text{target}}), 0 - h(s'_{\text{target}})) = H(s_{\text{target}}, s'_{\text{target}}) \leq H(succ(s_{\text{target}}, a), s'_{\text{target}}) + c(s_{\text{target}}, a) \leq \max(H(succ(s_{\text{target}}, a), s'_{\text{target}}), h(succ(s_{\text{target}}, a)) - h(s'_{\text{target}})) + c(s_{\text{target}}, a) = h'(succ(s_{\text{target}}, a)) + c(s_{\text{target}}, a)$ for all $a \in A(s_{\text{target}})$. Finally, if $s \neq s_{\text{target}}$ and $s \neq s'_{\text{target}}$, then $H(s, s'_{\text{target}}) \leq H(succ(s, a), s'_{\text{target}}) + c(s, a)$ and $h(s) \leq h(succ(s, a)) + c(s, a)$. Thus, $h'(s) = \max(H(s, s'_{\text{target}}), h(s) - h(s'_{\text{target}})) \leq \max(H(succ(s, a), s'_{\text{target}}), h(succ(s, a)) - h(s'_{\text{target}})) + c(s, a) = h'(succ(s, a)) + c(s, a)$ for all $a \in A(s)$. ■

MT-Adaptive A* uses this insight in a straight-forward way by performing an A* search, then *updating the h-values* of all states that were expanded during the search by executing Assignments (1), where s_{target} is the previous goal state, and finally *correcting the h-values* of *all* states (not just the expanded states) by executing Assignments (2), where s'_{target} is the new goal state. (In the following, we refer to "updating the h-values" and "correcting the h-values" to distinguish the two steps.) Taking the maximum of $h(s) - h(s'_{\text{target}})$ and the user-supplied initial h-values $H(s, s'_{\text{target}})$ ensures that the h-values used by MT-Adaptive A* dominate the user-supplied initial h-values at least weakly. Thus, it is still the case that MT-Adaptive A* with the new h-values cannot expand more states than an otherwise identical A* search with the user-supplied initial h-values. The main advantage of MT-Adaptive A* over D* Lite for moving-target search is that it does not need to shift the map repeatedly to keep the agent or target in its center.

MT-Adaptive A* can search from the current cell of the agent to the current cell of the target or from the current cell of the target

```
1  procedure InitializeState(s)
2  if search(s) ≠ counter AND search(s) ≠ 0
3      g(s) := ∞;
4  search(s) := counter;
5  procedure ComputePath()
6  while g(s_target) > min_{s'∈OPEN}(g(s') + h(s'))
7      delete a cell s with the smallest f-value g(s) + h(s) from OPEN;
8      CLOSED := CLOSED ∪ {s};
9      for each a ∈ A(s)
10         InitializeState(succ(s, a));
11         if g(succ(s, a)) > g(s) + c(s, a)
12             g(succ(s, a)) := g(s) + c(s, a);
13             tree(succ(s, a)) := s;
14             if succ(s, a) is in OPEN then delete it from OPEN;
15             insert succ(s, a) into OPEN with f-value g(succ(s, a)) + h(succ(s, a));
16 procedure Main()
17 counter := 0;
18 for every cell s ∈ S
19     h(s) := H(s, s_target);
20     search(s) := 0;
21     g(s) := ∞;
22 while s_start ≠ s_target
23     counter := counter + 1;
24     InitializeState(s_start);
25     InitializeState(s_target);
26     g(s_start) := 0;
27     OPEN := CLOSED := ∅;
28     insert s_start into OPEN with f-value g(s_start) + h(s_start);
29     ComputePath();
30     if OPEN = ∅
31         stop;
32     for each s ∈ CLOSED
33         h(s) := g(s_target) − g(s);
34     move the agent along the path identified by the tree-pointers
           until it reaches s_target, the current cell of the target changes or action costs increase;
35     set s_start to the current cell of the agent (if changed);
36     set s_newtarget to the current cell of the target;
37     update the increased action costs (if any);
38     if s_target ≠ s_newtarget
39         old := h(newtarget);
40         for each s ∈ S
41             h(s) := max(H(s, newtarget), h(s) − old);
42         s_target := s_newtarget;
```

Figure 5: Eager MT-Adaptive A*

to the current cell of the agent. Figure 5 contains the pseudo code of Eager MT-Adaptive A* that searches from the current cell of the agent to the current cell of the target. (We explain below why we consider Eager MT-Adaptive A* to be eager.) ComputePath() implements an A* search to find a shortest path from the current cell of the agent (s_{start}) to the current cell of the target (s_{target}). (The minimum of an empty set on Line 6 is infinity.) Lines 32-33 update the h-values of all expanded cells, and Lines 39-41 correct the h-values of all cells for the new cell of the target. The pseudo code is a bit clumsy but this allows us to transform Eager MT-Adaptive A* into Lazy MT-Adaptive A* (which is more efficient) in the next section. It is possible to optimize our pseudo code further. For example, the path from the agent to the target needs to be recomputed only if the previous path turns out to be blocked or the target moves and leaves this path because otherwise the previous path remains a shortest path. Also, the h-value of a cell does not need to be initialized up front on Line 19 but can be initialized when it is first needed. We use an optimized version of MT-Adaptive A* in our experiments that implements these and other optimizations.

Figure 6 shows how Eager MT-Adaptive A* updates the h-values for our example from Figure 2 when it searches from the current cell of the agent to the current cell of the target. The first figure shows the user-supplied initial h-values, namely the (consistent) Manhattan distances, to the current cell of the target. The second figure shows the first A* search. All cells have their h-value (from the first figure) in the lower left corner. Generated cells also

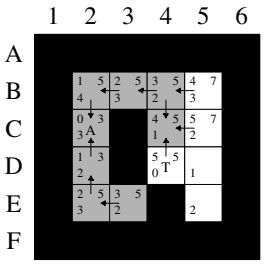

Figure 7: A*

have their g-value in the upper left corner, their f-value in the upper right corner and their tree-pointer visualized as arrow pointing to their parent in the search tree. Expanded cells are shown in grey. (The current cell of the target does not count as expanded since the search stops immediate before expanding it.) The length of the shortest discovered path is seven. The third figure shows the updated h-values after the h-values of all grey cells in the second figure have been updated to seven minus their g-values. The fourth figure shows the corrected h-values after the target moved. The h-values of all cells are corrected to the maximum of their Manhattan distances to the new cell of the target and their current h-values decreased by one. The fifth figure shows the second A* search. Note that cells D2, E2 and E3 have more informed h-values than their Manhattan distances to the current cell of the target. MT-Adaptive A* therefore expands two cells (namely, cells E2 and E3) less than an A* search with the Manhattan distances, as shown in Figure 7, demonstrating the advantage of MT-Adaptive A* over isolated A* searches. (We broke ties for both heuristic search methods in the same way.) Finally, the sixth figure shows the updated h-values after the search.

4.6 Lazy MT-Adaptive A*

Eager MT-Adaptive A* can update the h-values of states that are not needed during future searches, which explains its name. This can be time-consuming if the number of expanded states is large (and Eager MT-Adaptive A* then needs to update the h-values of all expanded states) or the target moves (and Eager MT-Adaptive A* then needs to correct the h-values of all states). Lazy MT-Adaptive A* remembers some information when a state is expanded during a search (such as its g-value) and some information after the search (such as the length of the shortest discovered path) and then uses that information to compute its h-value only when it is needed during a future search. Lazy and Eager MT-Adaptive A* perform the same searches and find the same paths. They differ only in when they calculate the h-values.

Figure 8 contains the pseudo code of Lazy MT-Adaptive A* that searches from the current cell of the agent to the current cell of the target. As before, ComputePath() implements an A* search to find a shortest path from the current cell of the agent (s_{start}) to the current cell of the target (s_{target}). Whenever a cell s is about to be generated for the first time on Line 20 and thus its h-value is needed for the first time, then InitializeState(s) is called on Line 15. If the cell was not generated during any previous search ($search(s) = 0$), then it initializes its h-value on Line 9. Otherwise, it first checks on Lines 3-4 whether it needs to update the h-value of the cell (Step 1) and then corrects the h-value of the cell on Lines 5-6 for the new cell of the target (Step 2). We now explain these two steps in detail:

- Step 1 (Updating the H-Values): The value of *counter* is x during the xth invocation of ComputePath(), that is, the xth

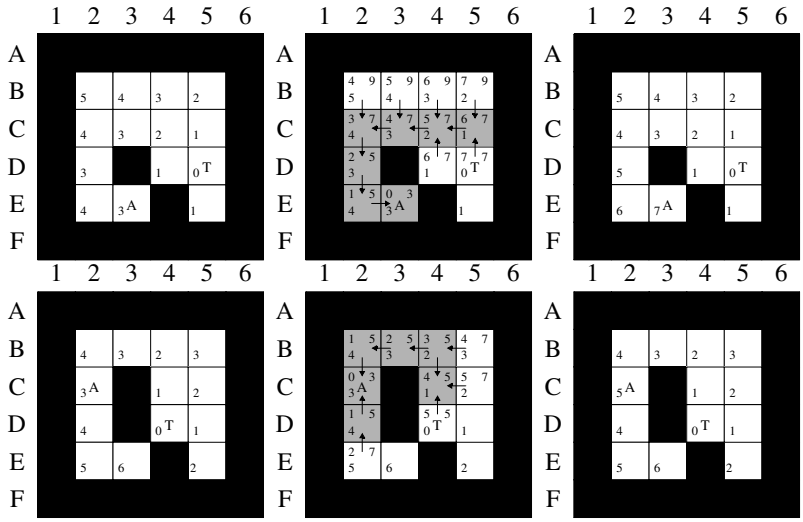

Figure 6: Eager MT-Adaptive A*

A* search. The value of *pathcost*(x) is the length of the shortest discovered path during the xth search. The value of *search*(s) is x if cell x was generated last during the xth search. If the cell was generated during a previous search (*search*$(s) \neq 0$) and then expanded during the same search ($g(s) + h(s) <$ *pathcost*(*search*(s))) but not yet generated during the current search (*search*$(s) \neq$ *counter*), then InitializeState(s) updates the h-value of the cell by setting it to the difference of the length of the shortest discovered path after the search when the cell was generated and expanded (*pathcost*(*search*(s))) and the g-value of the cell after the same search ($g(s)$).

- Step 2 (Correcting the H-Values): The correction for the new cell of the target decreases the h-values of all cells every time the target moves by the h-value of the new cell of the target with respect to the previous cell of the target. This h-value is computed on Lines 44-45 and then added to a running sum of all corrections on Line 46. In particular, *deltah*(x) is the running sum of all corrections up to the beginning of the xth search. If a cell s was generated during a previous search but not yet generated during the current search, then InitializeState(s) corrects its h-value by the sum of all corrections between the search when cell s was generated last and the current search, which is the same as the difference of the value of *deltah* during the current search (*deltah*(*counter*)) and the search when cell s was generated last (*deltah*(*search*(s))).

5. EXPERIMENTS

We now compare Lazy MT-Adaptive A* against A* and a heavily optimized version of D* Lite for moving-target search.

5.1 Test Cases

We perform our experiments in randomly generated four-neighbor torus-shaped mazes of size 100×100 in which the agent can catch the target, see Figure 9. Their corridor structure is generated with depth-first search. The initial cells of the agent and target are chosen randomly. We use the Manhattan distances as (consistent) informed user-supplied initial h-values and the zero h-values as (consistent) uninformed user-supplied initial h-values. We vary the experimental conditions by either keeping the target stationary or moving it randomly (with the restriction that it does not move every tenth time step and, if it moves, returns to the adjacent cell that it came from only if this is the only possible move in its current cell) and by either providing a-priori information to the agent about which cells are blocked or not, resulting in four different experimental conditions. A stationary target in an initially known maze does not require repeated searches. Consequently, this experimental condition is not interesting to us. A stationary target in an initially unknown maze corresponds to the search problems that D* Lite was developed for, namely search problems where the agent's knowledge of the world (namely, its knowledge about blockages) changes. A moving target in an initially unknown maze combines both reasons for re-planning since both the world (namely, the position of the target) and the agent's knowledge of the world change.

5.2 Search Methods

We compare Lazy MT-Adaptive A* (forward search), that searches forward from the current cell of the agent to the current cell of the target, and Lazy MT-Adaptive A* (backward search), that searches backward from the current cell of the target to the current cell of the agent, against A* (forward search), A* (backward search), D* Lite with an agent-centric map and D* Lite with a target-centric map. To be fair, we optimized D* Lite heavily for moving-target search, for example, by performing bookkeeping operations for only those cells on the map that change their blockage status when the map is shifted, which makes this version of D* Lite fast in initially unknown mazes as long as only a small number of the blockages have been discovered by the agent. Lazy MT-Adaptive A*, A* and D* Lite do not differ in how the agents move (if they could break ties in the exact same way) but only in their runtime, which depends on low-level implementation and machine details, such as the instruction set of the processor, the optimizations performed by the compiler, and the data structures used for the priority queues. This point is especially important since mazes for moving-target search typically fit into memory and thus are rel-

```
1  procedure InitializeState(s)
2  if search(s) ≠ counter AND search(s) ≠ 0
3      if g(s) + h(s) < pathcost(search(s))
4          h(s) := pathcost(search(s)) − g(s);
5      h(s) := h(s) − (deltah(counter) − deltah(search(s)));
6      h(s) := max(h(s), H(s, s_target));
7      g(s) := ∞;
8  else if search(s) = 0
9      h(s) := H(s, s_target);
10 search(s) := counter;
11 procedure ComputePath()
12 while g(s_target) > min_{s'∈OPEN}(g(s') + h(s'))
13     delete a cell s with the smallest f-value g(s) + h(s) from OPEN;
14     for each a ∈ A(s)
15         InitializeState(succ(s, a));
16         if g(succ(s, a)) > g(s) + c(s, a)
17             g(succ(s, a)) := g(s) + c(s, a);
18             tree(succ(s, a)) := s;
19             if succ(s, a) is in OPEN then delete it from OPEN;
20             insert succ(s, a) into OPEN with f-value g(succ(s, a)) + h(succ(s, a));
21 procedure Main()
22 counter := 0;
23 deltah(1) := 0;
24 for every cell s ∈ S
25     search(s) := 0;
26     g(s) := ∞;
27 while s_start ≠ s_target
28     counter := counter + 1;
29     InitializeState(s_start);
30     InitializeState(s_target);
31     g(s_start) := 0;
32     OPEN := ∅;
33     insert s_start into OPEN with f-value g(s_start) + h(s_start);
34     ComputePath();
35     if OPEN = ∅
36         stop;
37     pathcost(counter) := g(s_target);
38     move the agent along the path identified by the tree-pointers
           until it reaches s_target, the current cell of the target changes or action costs increase;
39     set s_start to the current cell of the agent (if changed);
40     set s_newtarget to the current cell of the target;
41     update the increased action costs (if any);
42     if s_target ≠ s_newtarget
43         InitializeState(s_newtarget);
44         if g(s_newtarget) + h(s_newtarget) < pathcost(counter)
45             h(s_newtarget) := pathcost(counter) − g(s_newtarget);
46         deltah(counter + 1) := deltah(counter) + h(s_newtarget);
47         s_target := s_newtarget,
```

Figure 8: Lazy MT-Adaptive A*

atively small search domains. However, we do not know of any better method for evaluating heuristic search methods than to implement them as best as possible, publish their runtimes, and let other researchers validate them with their own and thus potentially slightly different implementations. For fairness, we use comparable implementations. For example, Lazy MT-Adaptive A*, A* and D* Lite all use binary heaps as priority queues, break ties among cells with the same f-values in favor of cells with larger g-values (which is known to be a good tie-breaking strategy) and do not search again as long as the goal remains on the previous path.

5.3 Experimental Results

The performance measures in Table 1 are averaged over the same 1000 mazes. We report two measures for the difficulty of the moving-target search problems, namely the average number of moves of the agent until it catches the target and the average number of searches required to find these moves. We report two measures for the efficiency of the heuristic search methods, namely the number of expanded cells per search and the total runtime per search in microseconds on a Pentium D 3.0 GHz PC with 2 GByte of RAM. (We show the standard deviation of the mean for the number of expanded cells in parentheses to demonstrate the statistical significance of our results.) Table 1 shows the following relationships:

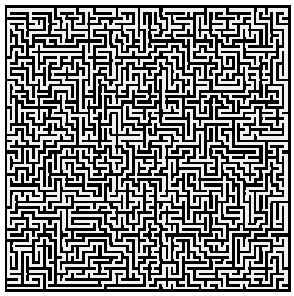

Figure 9: A Maze

- Lazy MT-Adaptive A* (forward) has sometimes a smaller and sometimes a larger runtime than Lazy MT-Adaptive A* (backward).
- Lazy MT-Adaptive A* always has a smaller runtime than A* with the same search direction for moving-target search.
- D* Lite with a target-centric map always has a smaller runtime than D* Lite with an agent-centric map.
- D* Lite with a target-centric map always has a smaller runtime than both Lazy MT-Adaptive A* (forward) and Lazy MT-Adaptive A* (backward) for stationary-target search (as already explained), but both Lazy MT-Adaptive A* (forward) and Lazy MT-Adaptive A* (backward) have smaller runtimes than D* Lite with a target-centric map (and thus also than D* Lite with an agent-centric map) for moving-target search.

In general, D* Lite has either uninformed h-values available (which do not focus its search) or informed h-values (which are rather misleading in our mazes). Lazy MT-Adaptive A*, on the other hand, updates the h-values to make them more informed or less misleading. In the case of known mazes, it is time-consuming for D* Lite to shift the map repeatedly because many blockages are known right from the beginning and thus need to be shifted. Lazy MT-Adaptive A*, on the other hand, does not need to shift the map at all. Overall, one can safely use Lazy MT-Adaptive A* for moving-target search. For example, Lazy MT-Adaptive A* (forward) and Lazy MT-Adaptive A* (backward) is faster than D* Lite with a target-centric map by about one order of magnitude for moving-target search in initially unknown mazes (factors of 13.5 and 6.5, respectively) and known mazes (factors of 31.6 and 35.0, respectively) if both search methods use the same informed h-values.

6. CONCLUSIONS

We studied a moving-target search strategy that always moves the agent on a shortest presumed unblocked path toward the target. Thus, the agent has to find shortest paths repeatedly until it catches the target. We studied how the agent can solve these deterministic search problems faster by exploiting the fact that it performs A* searches repeatedly. To this end, we extended Adaptive A*, an incremental heuristic search method, to moving-target search. We then demonstrated experimentally that the resulting (Lazy) MT-Adaptive A* is faster than isolated A* searches. It is also faster than D* Lite by about one order of magnitude for moving-target search

Moving-Target Search in Initially Unknown Mazes

	informed h-values					uninformed h-values				
	searches until target caught	moves until target caught	expansions per search		runtime per search	searches until target caught	moves until target caught	expansions per search		runtime per search
A* (Forward Search)	1997	2605	500	(0.7)	75	1966	2575	3703	(5.4)	570
A* (Backward Search)	1980	2582	1528	(1.5)	255	1894	2454	4519	(5.9)	722
Lazy MT-Adaptive A* (Forward Search)	1993	2598	342	(0.5)	63	1957	2551	2334	(4.5)	465
Lazy MT-Adaptive A* (Backward Search)	1908	2560	663	(0.7)	131	1814	2434	2025	(4.6)	411
D* Lite (Agent-Centric Map)	1951	2526	501	(0.7)	941	1974	2542	2229	(5.8)	1481
D* Lite (Target-Centric Map)	1974	2387	511	(1.2)	848	1840	2365	806	(3.0)	833

Stationary-Target Search in Initially Unknown Mazes

	informed h-values					uninformed h-values				
	searches until target caught	moves until target caught	expansions per search		runtime per search	searches until target caught	moves until target caught	expansions per search		runtime per search
A* (Forward Search)	886	2555	389	(6.1)	69	867	2525	3711	(8.4)	581
A* (Backward Search)	889	2527	830	(3.6)	146	847	2442	4104	(8.7)	644
Lazy MT-Adaptive A* (Forward Search)	878	2546	136	(2.6)	31	882	2580	391	(2.8)	81
Lazy MT-Adaptive A* (Backward Search)	879	2533	796	(3.4)	157	865	2494	3410	(8.8)	729
D* Lite (Agent-Centric Map)	877	2564	363	(3.4)	836	878	2516	2467	(9.8)	1655
D* Lite (Target-Centric Map)	842	2437	22	(0.6)	21	844	2448	31	(0.9)	15

Moving-Target Search in Known Mazes

	informed h-values					uninformed h-values				
	searches until target caught	moves until target caught	expansions per search		runtime per search	searches until target caught	moves until target caught	expansions per search		runtime per search
A* (Forward Search)	340	764	1978	(8.1)	221	340	764	2120	(7.3)	215
A* (Backward Search)	340	764	1640	(6.8)	170	340	764	1740	(6.8)	159
Lazy MT-Adaptive A* (Forward Search)	340	764	1182	(5.5)	143	340	764	1245	(5.6)	144
Lazy MT-Adaptive A* (Backward Search)	340	764	1087	(5.1)	129	340	764	1133	(5.2)	128
D* Lite (Agent-Centric Map)	341	764	3131	(13.0)	4834	341	764	2945	(11.5)	4243
D* Lite (Target-Centric Map)	341	764	2679	(11.4)	4516	341	764	2531	(10.1)	4036

Table 1: Experimental Results

in known or initially unknown mazes if both search methods use the same informed heuristics. We claimed in our previous papers that the principle behind Adaptive A* is simple, which makes it easy to extend Adaptive A* to new applications. MT-Adaptive A* backs up this claim, as did our earlier extension of Adaptive A* to real-time heuristic search [9]. In future work, we intend to explore different movement strategies for moving-target search, including one that tries to "corner" the target [11]. We intend to study how the agent can perform these minimax (AND/OR) searches faster by exploiting the fact that it performs minimax searches repeatedly, by adapting Adaptive A* to minimax searches, perhaps in a way similar to how we have already adapted LPA* to minimax searches [12].

Acknowledgments

All experimental results are the responsibility of Xiaoxun Sun. This research has been partly supported by an NSF award to Sven Koenig under contract IIS-0350584. 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, companies or the U.S. government.

7. REFERENCES

[1] M. Goldenberg, A. Kovarksy, X. Wu, and J. Schaeffer. Multiple agents moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1538–1538, 2003.

[2] 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.

[3] 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.

[4] T. Ishida and R. Korf. Moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 204–210, 1991.

[5] S. Koenig and M. Likhachev. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, pages 476–483, 2002.

[6] S. Koenig and M. Likhachev. Incremental A*. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 1539–1546, Cambridge, MA, 2002. MIT Press.

[7] S. Koenig and M. Likhachev. Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 1311–1312, 2005.

[8] S. Koenig and M. Likhachev. A new principle for incremental heuristic search: Theoretical results. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 410–413, 2006.

[9] S. Koenig and M. Likhachev. Real-Time Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 281–288, 2006.

[10] S. Koenig, M. Likhachev, and D. Furcy. Lifelong Planning A*. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[11] S. Koenig and R. Simmons. Real-time search in non-deterministic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1660–1667, 1995.

[12] M. Likhachev and S. Koenig. Speeding up the Parti-Game algorithm. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1563–1570, Cambridge, MA, 2002. MIT Press.

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

[14] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.