Generalized Adaptive A*

Xiaoxun Sun
USC
Computer Science
Los Angeles, California
xiaoxuns@usc.edu

Sven Koenig
USC
Computer Science
Los Angeles, California
skoenig@usc.edu

William Yeoh
USC
Computer Science
Los Angeles, California
wyeoh@usc.edu

ABSTRACT

Agents often have to solve series of similar search problems. Adaptive A* is a recent incremental heuristic search algorithm that solves series of similar search problems faster than A* because it updates the h-values using information from previous searches. It basically transforms consistent h-values into more informed consistent h-values. This allows it to find shortest paths in state spaces where the action costs can increase over time since consistent h-values remain consistent after action cost increases. However, it is not guaranteed to find shortest paths in state spaces where the action costs can decrease over time because consistent h-values do not necessarily remain consistent after action cost decreases. Thus, the h-values need to get corrected after action cost decreases. In this paper, we show how to do that, resulting in Generalized Adaptive A* (GAA*) that finds shortest paths in state spaces where the action costs can increase or decrease over time. Our experiments demonstrate that Generalized Adaptive A* outperforms breadth-first search, A* and D* Lite for moving-target search, where D* Lite is an alternative state-of-the-art incremental heuristic search algorithm that finds shortest paths in state spaces where the action costs can increase or decrease over time.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Problem Solving

General Terms

Algorithms

Keywords

A*; D* Lite; Heuristic Search; Incremental Search; Shortest Paths, Moving-Target Search

1. INTRODUCTION

Most research on heuristic search has addressed one-time search problems. However, agents often have to solve series of similar search problems as their state spaces or their knowledge of the state spaces changes. Adaptive A* [7] is a recent incremental heuristic search algorithm that solves series of similar search problems faster than A* because it updates the h-values using information from previous searches. Adaptive A* is simple to understand and easy to implement because it basically transforms consistent h-values into more informed consistent h-values. Adaptive A* can easily be generalized to moving-target search [5], where the goal state changes over time. Consistent h-values do not necessarily remain consistent when the goal state changes. Adaptive A* therefore makes the h-values consistent again when the goal state changes. Adaptive A* is able to find shortest paths in state spaces where the start state changes over time since consistent h-values remain consistent when the start state changes. It is able to find shortest paths in state spaces where the action costs can increase over time since consistent h-values remain consistent after action cost increases. However, they are not guaranteed to find shortest paths in state spaces where the action costs can decrease over time because consistent h-values do not necessarily remain consistent after action cost decreases, which limits their applicability. Thus, the h-values need to get corrected after action cost decreases. In this paper, we show how to do that, resulting in Generalized Adaptive A* (GAA*) that finds shortest paths in state spaces where the action costs can increase or decrease over time.

Cite as: Generalized Adaptive A*, Xiaoxun Sun, Sven Koenig and William Yeoh, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. XXX-XXX.

2. NOTATION

We use the following notation: S denotes the finite set of states. $s_{\text{start}} \in S$ denotes the start state, and $s_{\text{goal}} \in S$ denotes the goal state. $A(s)$ denotes the finite set of actions that can be executed in state $s \in S$. $c(s, a) > 0$ denotes the action cost of executing action $a \in A(s)$ in state $s \in S$, and $succ(s, a) \in S$ denotes the resulting successor state. We refer to an action sequence as path. The search problem is to find a shortest path from the start state to the goal state, knowing the state space.

3. HEURISTICS

Heuristic search algorithms use h-values (= heuristics) to focus their search. The h-values are derived from user-supplied H-values $H(s, s')$, that estimate the distance from any state s to any state s'. The user-supplied H-values $H(s, s')$ have to satisfy the triangle inequality (= be consistent), namely satisfy $H(s', s') = 0$ and $H(s, s') \leq c(s, a) + H(succ(s, a), s')$ for all states s and s' with $s \neq s'$ and all actions a that can be executed in state s. If the user-supplied H-values $H(s, s')$ are consistent, then the h-values $h(s) = H(s, s_{\text{goal}})$ are consistent with respect to the goal

state s_{goal} (no matter what the goal state is), namely satisfy $h(s_{\text{goal}}) = 0$ and $h(s) \leq c(s,a) + h(succ(s,a))$ for all states s with $s \neq s_{\text{goal}}$ and all actions a that can be executed in state s [11].

4. A*

Adaptive A* is based on a version of A* [2] that uses consistent h-values with respect to the goal state to focus its search. We therefore assume in the following that the h-values are consistent with respect to the goal state.

4.1 Variables

A* maintains four values for all states s that it encounters during the search: a g-value $g(s)$ (which is infinity initially), which is the length of the shortest path from the start state to state s found by the A* search and thus an upper bound on the distance from the start state to state s; an h-value $h(s)$ (which is $H(s, s_{\text{goal}})$ initially and does not change during the A* search), which estimates the distance of state s to the goal state; 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 A* search.

4.2 Datastructures and Algorithm

A* maintains an *OPEN* list (a priority queue which contains only the start state initially). A* identifies a state s with the smallest f-value in the *OPEN* list [Line 14 from Figure 1]. A* terminates once the f-value of state s is no smaller than the f-value or, equivalently, the g-value of the goal state. (It holds that $f(s_{\text{goal}}) = g(s_{\text{goal}}) + h(s_{\text{goal}}) = g(s_{\text{goal}})$ since $h(s_{\text{goal}}) = 0$ for consistent h-values with respect to the goal state.) Otherwise, A* removes state s from the *OPEN* list and expands it, meaning that it performs the following operations for all actions that can be executed in state s and result in a successor state whose g-value is larger than the g-value of state s plus the action cost [Lines 18-21]: First, it sets the g-value of the successor state to the g-value of state s plus the action cost [Line 18]. Second, it sets the tree-pointer of the successor state to (point to) state s [Line 19]. Finally, it inserts the successor state into the *OPEN* list or, if it was there already, changes its priority [Line 20-21]. (We refer to it generating a state when it inserts the state for the first time into the *OPEN* list.) It then repeats the procedure.

4.3 Properties

We use the following known properties of A* searches [11]. Let $g(s)$, $h(s)$ and $f(s)$ denote the g-values, h-values and f-values, respectively, after the A* search: First, the g-values of all expanded states and the goal state after the A* search are equal to the distances from the start state to these states. Following the tree-pointers from these states to the start state identifies shortest paths from the start state to these states in reverse. Second, an A* search expands no more states than an (otherwise identical) other A* search for the same search problem if the h-values used by the first A* search are no smaller for any state than the corresponding h-values used by the second A* search (= the former h-values dominate the latter h-values at least weakly).

5. ADAPTIVE A*

Adaptive A* [7] is a recent incremental heuristic search algorithm that solves series of similar search problems faster than A* because it updates the h-values using information from previous searches. Adaptive A* is simple to understand and easy to implement because it basically transforms consistent h-values with respect to the goal state into more informed consistent h-values with respect to the goal state. We describe Lazy Adaptive Moving-Target Adaptive A* (short: Adaptive A*) in the following.

5.1 Improving the Heuristics

Adaptive A* updates (= overwrites) the consistent h-values with respect to the goal state of all expanded state s after an A* search by executing

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

This principle was first used in [4] and later resulted in the independent development of Adaptive A*. The updated h-values are again consistent with respect to the goal state [7]. They also dominate the immediately preceeding h-values at least weakly [7]. Thus, they are no less informed than the immediately preceeding h-values and an A* search with the updated h-values thus cannot expand more states than an A* search with the immediately preceeding h-values (up to tie breaking).

5.2 Maintaining Consistency of the Heuristics

Adaptive A* solves a series of similar but not necessarily identical search problems. However, it is important that the h-values remain consistent with respect to the goal state from search problem to search problem. The following changes can occur from search problem to search problem:

- The start state changes. In this case, Adaptive A* does not need to do anything since the h-values remain consistent with respect to the goal state.
- The goal state changes. In this case, Adaptive A* needs to correct the h-values. Assume that the goal state changes from s_{goal} to s'_{goal}. Adaptive A* then updates (= overwrites) the h-values of all states s by assigning

  $$h(s) := \max(H(s, s'_{\text{goal}}), h(s) - h(s'_{\text{goal}})). \quad (2)$$

  The updated h-values are consistent with respect to the new goal state [9]. However, they are potentially less informed than the immediately preceeding h-values. Taking the maximum of $h(s) - h(s'_{\text{goal}})$ and the user-supplied H-value $H(s, s'_{\text{goal}})$ with respect to the new goal state ensures that the h-values used by Adaptive A* dominate the user-supplied H-values with respect to the new goal state at least weakly.
- At least one action cost changes. If no action cost decreases, Adaptive A* does not need to do anything since the h-values remain consistent with respect to the goal state. This is easy to see. Let c denote the original action costs and c' the new action costs after the changes. Then, $h(s) \leq c(s,a) + h(succ(s,a))$ for $s \in$

$S \setminus \{s_{\text{goal}}\}$ implies that $h(s) \leq c(s,a) + h(succ(s,a)) \leq c'(s,a) + h(succ(s,a))$ for $s \in S \setminus \{s_{\text{goal}}\}$. However, if at least one action cost decreases then the h-values are not guaranteed to remain consistent with respect to the goal state. This is easy to see by example. Consider a state space with two states, a non-goal state s and a goal state s', and one action whose execution in the non-goal state incurs action cost two and results in a transition to the goal state. Then, an h-value of two for the non-goal state and an h-value of zero for the goal state are consistent with respect to the goal state but do not remain consistent with respect to the goal state after the action cost decreases to one. Thus, Adaptive A* needs to correct the h-values, which is the issue addressed in this paper.

5.3 Updating the Heuristics Lazily

So far, the h-values have been updated in an eager way, that is, right away. However, Adaptive A* updates the h-values in a lazy way, which means that it spends effort on the update of an h-value only when it is needed during a search, thus avoiding wasted effort. It remembers some information during the A* searches (such as the g-values of states) [Lines 18 and 30] and some information after the A* searches (such as the g-value of the goal state, that is, the distance from the start state to the goal state) [Lines 35 and 37] and then uses this information to calculate the h-value of a state only when it is needed by a future A* search [9].

5.4 Pseudocode

Figure 1 contains the pseudocode of Adaptive A* [9]. Adaptive A* does not initialize all g-values and h-values up front but uses the variables *counter*, *search*(s) and *pathcost*(x) to decide when to initialize them:

- The value of *counter* is x during the xth execution of ComputePath(), that is, the xth A* search.
- The value of *search*(s) is x if state s was generated last by the xth A* search (or is the goal state). Adaptive A* initializes these values to zero [Lines 25-26].
- The value of *pathcost*(x) is the length of the shortest path from the start state to the goal state found by the xth A* search, that is, the distance from the start state to the goal state.

Adaptive A* executes ComputePath() to perform an A* search [Line 33]. (The minimum over an empty set is infinity on Line 13.) Adaptive A* executes InitializeState(s) when the g-value or h-value of a state s is needed [Lines 16, 28, 29 and 42].

- Adaptive A* initializes the g-value of state s (to infinity) if state s either has not yet been generated by some A* search (*search*(s) = 0) [Line 9] or has not yet been generated by the current A* search (*search*(s) $\neq$ *counter*) but was generated by some A* search (*search*(s) $\neq$ 0) [Line 7].
- Adaptive A* initializes the h-value of state s with its user-supplied H-value with respect to the goal state if the state has not yet been generated by any A* search (*search*(s) = 0) [Line 10]

```
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_goal));
7     g(s) := ∞;
8  else if search(s) = 0
9     g(s) := ∞;
10    h(s) := H(s, s_goal);
11 search(s) := counter;
12 procedure ComputePath()
13 while g(s_goal) > min_{s'∈OPEN}(g(s') + h(s'))
14    delete a state s
      with the smallest f-value g(s) + h(s) from OPEN;
15    for all actions a ∈ A(s)
16       InitializeState(succ(s, a));
17       if g(succ(s, a)) > g(s) + c(s, a)
18          g(succ(s, a)) := g(s) + c(s, a);
19          tree(succ(s, a)) := s;
20          if succ(s, a) is in OPEN then delete it from OPEN;
21          insert succ(s, a) into OPEN
            with f-value g(succ(s, a)) + h(succ(s, a));
22 procedure Main()
23 counter := 1;
24 deltah(1) := 0;
25 for all states s ∈ S
26    search(s) := 0;
27 while s_start ≠ s_goal
28    InitializeState(s_start);
29    InitializeState(s_goal);
30    g(s_start) := 0;
31    OPEN := ∅;
32    insert s_start into OPEN with f-value g(s_start) + h(s_start);
33    ComputePath();
34    if OPEN = ∅
35       pathcost(counter) := ∞;
36    else
37       pathcost(counter) := g(s_goal);
38    change the start and/or goal states, if desired;
39    set s_start to the start state;
40    set s_newgoal to the goal state;
41    if s_goal ≠ s_newgoal
42       InitializeState(s_newgoal);
43       if g(s_newgoal) + h(s_newgoal) < pathcost(counter)
44          h(s_newgoal) := pathcost(counter) − g(s_newgoal);
45       deltah(counter + 1) := deltah(counter) + h(s_newgoal);
46       s_goal := s_newgoal;
47    else
48       deltah(counter + 1) := deltah(counter);
49    counter := counter + 1;
50    update the increased action costs (if any);
```

Figure 1: Adaptive A*

- Adaptive A* updates the h-value of state s according to Assignment 1 [Line 4] if all of the following conditions are satisfied: First, the state has not yet been generated by the current A* search (*search*(s) $\neq$ *counter*). Second, the state was generated by a previous A* search (*search*(s) $\neq$ 0). Third, the state was expanded by the A* search that generated it last ($g(s)+h(s) <$ *pathcost*(*search*(s))). Thus, Adaptive A* updates the h-value of a state when an A* search needs its g-value or h-value for the first time during the current A* search and a previous A* search has expanded the state already. Adaptive A* sets the h-value of state s to the difference of the distance from the start state to the goal state during the last A* search that generated and, according to the conditions, also expanded the state (*pathcost*(*search*(s))) and the g-value of the state after the same A* search ($g(s)$ since the g-value has not changed since then).
- Adaptive A* corrects the h-value of state s according to Assignment 2 for the new goal state. The update for the new goal state decreases the h-values of

```
1'   update the increased and decreased action costs (if any);
2'   OPEN := ∅;
3'   for all state-action pairs (s, a)
     with s ≠ s_goal whose c(s, a) decreased
4'     InitializeState(s);
5'     InitializeState(succ(s,a));
6'     if (h(s) > c(s, a) + h(succ(s, a)))
7'       h(s) := c(s, a) + h(succ(s, a));
8'       if s is in OPEN then delete it from OPEN;
9'       insert s into OPEN with h-value h(s);
10'  while OPEN ≠ ∅
11'    delete a state s' with the smallest h-value h(s) from OPEN;
12'    for all states s ∈ S \ {s_goal} and actions a ∈ A(s)
       with succ(s, a) = s'
13'      InitializeState(s);
14'      if h(s) > c(s, a) + h(succ(s, a))
15'        h(s) := c(s, a) + h(succ(s, a));
16'        if s is in OPEN then delete it from OPEN;
17'        insert s into OPEN with h-value h(s);
```

Figure 2: Consistency Procedure

all states by the h-value of the new goal state. This h-value is added to a running sum of all corrections [Line 45]. In particular, the value of *deltah*(x) during the xth A* search is the running sum of all corrections up to the beginning of the xth A* search. If a state s was generated by a previous A* search (*search*(s) $\neq 0$) but not yet generated by the current A* search (*search*(s) $\neq$ *counter*), then Adaptive A* updates its h-value by the sum of all corrections between the A* search when state s was generated last and the current A* search, which is the same as the difference of the value of *deltah* during the current A* search (*deltah*(*counter*)) and the A* search that generated state s last (*deltah*(*search*(s))) [Line 5]. It then takes the maximum of this value and the user-supplied H-value with respect to the new goal state [Line 6].

6. GENERALIZED ADAPTIVE A*

We generalize Adaptive A* to the case where action costs can increase and decrease. Figure 2 contains the pseudocode of a consistency procedure that eagerly updates consistent h-values with respect to the goal state with a version of Dijkstra's algorithm [1] so that they remain consistent with respect to the goal state after action cost increases and decreases.[1] The pseudocode is written so that it replaces Line 50 of Adaptive A* in Figure 1, resulting in Generalized Adaptive A* (GAA*). InitializeState(s) performs all updates of the h-value of state s and can otherwise be ignored.

THEOREM 1. *The h-values remain consistent with respect to the goal state after action cost increases and decreases if the consistency procedure is run.*

Proof: Let c denote the action costs before the changes, and c' denote the action costs after the changes. Let $h(s)$ denote the h-values before running the consistency procedure and $h'(s)$ the h-values after its termination. Thus, $h(s_{\text{goal}}) = 0$ and $h(s) \leq c(s,a) + h(succ(s,a))$ for all non-goal states s and all actions a that can be executed in state s since the h-values are consistent with respect to the goal state for the action costs before the changes. The h-value of the goal state remains zero since it is never updated. The h-value of any non-goal state is monotonically non-increasing over time since it only gets updated to an h-value that is smaller than its current h-value. Thus, we distinguish three cases for a non-goal state s and all actions a that can be executed in state s:

- First, the h-value of state $succ(s,a)$ never decreased (and thus $h(succ(s,a)) = h'(succ(s,a))$) and $c(s,a) \leq c'(s,a)$. Then, $h'(s) \leq h(s) \leq c(s,a) + h(succ(s,a)) = c(s,a) + h'(succ(s,a)) \leq c'(s,a) + h'(succ(s,a))$.
- Second, the h-value of state $succ(s,a)$ never decreased (and thus $h(succ(s,a)) = h'(succ(s,a))$) and $c(s,a) > c'(s,a)$. Then, $c(s,a)$ decreased and $s \neq s_{\text{goal}}$ and Lines 4'-9' were just executed. Let $\bar{h}(s)$ be the h-value of state s after the execution of Lines 4'-9'. Then, $h'(s) \leq \bar{h}(s) \leq c'(s,a) + h'(succ(s,a))$.
- Third, the h-value of state $succ(s,a)$ decreased and thus $h(succ(s,a)) > h'(succ(s,a))$. Then, the state was inserted into the priority queue and later retrieved on Line 11'. Consider the last time that it was retrieved on Line 11'. Then, Lines 12'-17' were executed. Let $\bar{h}(s)$ be the h-value of state s after the execution of Lines 12'-17'. Then, $h'(s) \leq \bar{h}(s) \leq c'(s,a) + h'(succ(s,a))$.

Thus, $h'(s) \leq c'(s,a) + h'(succ(s,a))$ in all three cases, and the h-values remain consistent with respect to the goal state. ■

Adaptive A* updates the h-values in a lazy way. However, the consistency procedure currently updates the h-values in an eager way, which means that it is run whenever action costs decrease and can thus update a large number of h-values that are not needed during future searches. It is therefore important to evaluate experimentally whether Generalized Adaptive A* is faster than A*.

7. EXAMPLE

We use search problems in four-neighbor gridworlds of square cells (such as, for example, gridworlds used in video games) as examples. All cells are either blocked (= black) or unblocked (= white). The agent always knows its current cell, the current cell of the target and which cells are blocked. It can always move from its current cell to one of the four neighboring cells. The action cost for moving from an unblocked cell to an unblocked cell is one. All other action costs are infinity. The agent has to move so as to occupy the same cell as a stationary target (resulting in stationary-target search) or a moving target (resulting in moving-target search). The agent always identifies a shortest path from its current cell to the current cell of the target after its current path might no longer be a shortest path because the target left the path or action costs changed. The agent then begins to move along the path given by the tree-pointers. We use the Manhattan distances as consistent user-supplied H-values.

Figure 3 shows an example. After the A* search from the current cell of the agent (cell E3, denoted by "Start") to the current cell of the target (cell C5, denoted by "Goal"), the

[1]Line 3' can be simplified to "for all state-action pairs (s,a) whose $c(s,a)$ decreased" since $h(s_{\text{goal}}) = 0$ for consistent h-values with respect to the goal state and the condition on Line 6' is thus never satisfied for $s = s_{\text{goal}}$. Similarly, Line 12' can be simplified to "for all states $s \in S$ and actions $a \in A(s)$ with $succ(s,a) = s'$" for the same reason.

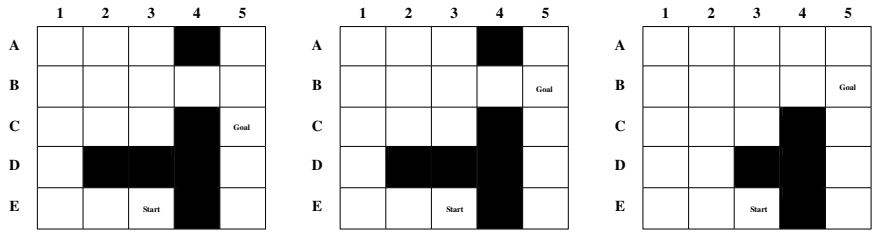

Figure 3: Example

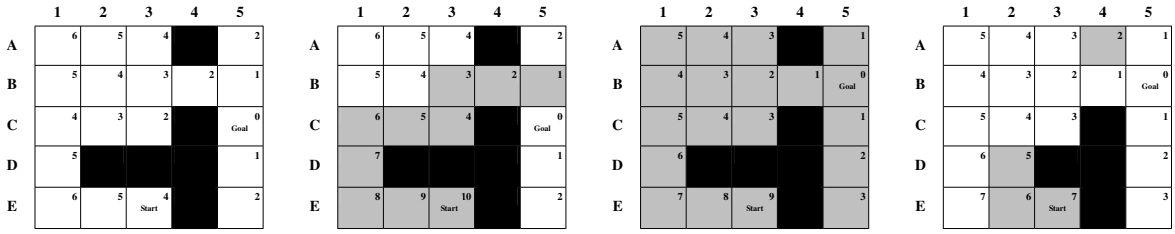

Figure 4: Ideas behind Generalized Adaptive A*

target moves to cell B5 and then cells A4 and D2 become unblocked (which decreases the action costs between these cells and their neighboring unblocked cells from infinity to one). Figure 4 shows the h-values (in the upper right corners of the cells) using the ideas from Sections 5.1, 5.2 and 6, that all update the h-values in an eager way. The first gridworld shows the Manhattan distances as user-supplied H-values, the second gridworld shows the improved h-values after the A* search, the third gridworld shows the updated h-values after the current cell of the target changed and the fourth gridworld shows the updated h-values determined by the consistency procedure after cells A4 and D2 became unblocked. (When a cell becomes unblocked, we set initialize its h-value to infinity before running the consistency procedure.) Grey cells contain h-values that were updated. Figure 5 shows the g-values (in the upper left corners of the cells), h-values (in the upper right corners) and *search*-values (in the lower left corners) of Generalized Adaptive A*, that updates the h-values in a lazy way except for the consistency procedure. The first gridworld shows the values before the A* search, the second gridworld shows the values after the A* search, the third gridworld shows the values after the current cell of the target changed and the fourth gridworld shows the values determined by the consistency procedure after cells A4 and D2 became unblocked. Grey cells are arguments of calls to InitializeState(s). The h-value of a cell s in the fourth gridworld of Figure 5 is equal to the h-value of the same cell in the fourth gridworld of Figure 4 if $search(s) = 2$. An example is cell C2. The h-value of a cell s in the fourth gridworld of Figure 5 is uninitialized if $search(s) = 0$. In this case, the Manhattan distance with respect to the current cell of the target is equal to the h-value of the same cell in the fourth gridworld of Figure 4. An example is cell A2. Otherwise, the h-value of a cell s in the fourth gridworld of Figure 5 needs to get updated. It needs to get updated according to Sections 5.1 and 5.2 to be equal to the h-value of the same cell in the fourth gridworld of Figure 4 if $g(s) + h(s) < pathcost(search(s))$. An example is cell C3. It needs to get updated according to Section 5.2 to be equal to the h-value of the same cell in the fourth gridworld of Figure 4 if $g(s) + h(s) \geq pathcost(search(s))$. An example is cell B3.

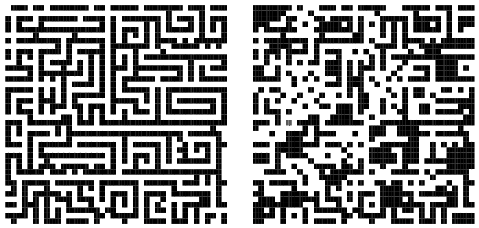

Figure 6: Example Test Gridworlds

8. EXPERIMENTAL EVALUATION

We perform experiments in the same kind of four-neighbor gridworlds, one independent stationary-target or moving-target search problem per gridworld. The agent and target move on 100 randomly generated four-connected gridworlds of size 300 × 300 that are shaped like a torus for ease of implementation (that is, moving north in the northern-most row results in being in the southern-most row and vice versa, and moving west in the western-most column results in being in the eastern-most column and vice versa). Figure 6 (left) shows an example of a smaller size. We generate their corridor structures with depth-first searches. We choose the initial cells of the agent and target randomly among the unblocked cells. We unblock k blocked cells and block k unblocked cells every tenth time step in a way so that there always remains a path from the current cell of the agent to the current cell of the target, where k is a parameter whose value we vary from one to fifty. Figure 6 (right) shows that these changes create shortcuts and decrease the distances

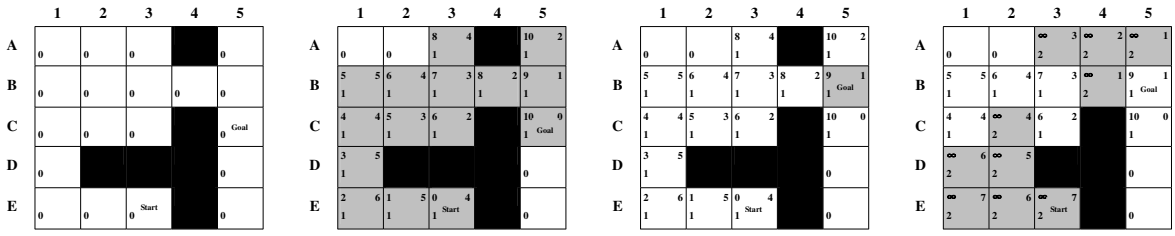

Figure 5: Generalized Adaptive A*

between cells over time. For moving-target search, the target moves randomly but does not return to its immediately preceeding cell unless this is the only possible move. It does not move every tenth time step, which guarantees that the agent can reach the target.

We compare Generalized Adaptive A* (GAA*) against breadth-first search (BFS), A* and a heavily optimized D* Lite (Target-Centric Map) [9]. Breadth-first search, A* and Generalized Adaptive A* can search either from the current cell of the agent to the current cell of the target (= forward) or from the current cell of the target to the current cell of the agent (= backward). Forward searches incur a runtime overhead over backward searches since the paths given by the tree-pointers go from the goal states to the start states for forward searches and thus need to be reversed. D* Lite is an alternative incremental heuristic search algorithm that finds shortest paths in state spaces where the action costs can increase or decrease over time. D* Lite can be understood as changing the immediately preceeding A* search tree into the current A* search tree, which requires the root node of the A* search tree to remain unchanged and is fast only if the number of action cost changes close to the root node of the A* search tree is small and the immediately preceeding and current A* search trees are thus similar. D* Lite thus searches from the current cell of the target to the current cell of the agent for stationary-target search but is not fast for large k since the number of action cost changes close to the root node of the A* search tree is then large. D* Lite can move the map to keep the target centered for moving-target search and then again searches from the current cell of the target to the current cell of the agent but is not fast since moving the map causes the number of action cost changes close to the root node to be large [9]. Thus, D* Lite is fast only for stationary-target search with small k. We use comparable implementations for all heuristic search algorithms. For example, A*, D* Lite and Generalized Adaptive A* all use binary heaps as priority queues and 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.

We run our experiments on a Pentium D 3.0 GHz PC with 2 GBytes of RAM. We report two measures for the difficulty of the search problems, namely the number of moves of the agent until it reaches the target and the number of searches run to determine these moves. All heuristic search algorithms determine the same paths and their number of moves and searches are thus approximately the same. They differ slightly since the agent can follow different trajectories due to tie breaking. We report two measures for the efficiency of the heuristic search algorithms, namely the number of expanded cells (= expansions) per search and the runtime per search in microseconds. We calculate the runtime per search by dividing the total runtime by the number of A* searches. For Generalized Adaptive A*, we also report the number of h-value updates (= propagations) per search by the consistency procedure as third measure since the runtime per search depends on both its number of expansions and propagations per search. We calculate the number of propagations per search by dividing the total number of h-value updates by the consistency procedure by the number of A* searches. We also report the standard deviation of the mean for the number of expansions per search (in parentheses) to demonstrate the statistical significance of our results. We compare the heuristic search algorithms using their runtime per search. Unfortunately, the runtime per search depends on low-level machine and implementation details, such as the instruction set of the processor, the optimizations performed by the compiler and coding decisions. This point is especially important since the gridworlds fit into memory and the resulting state spaces are thus small. We do not know of any better method for evaluating heuristic search methods than to implement them as well as possible, publish their runtimes, and let other researchers validate them with their own and thus potentially slightly different implementations. For example, it is difficult to compare them using proxies, such as their number of expansions per search, since they perform different basic operations and thus differ in their runtime per expansion. Breadth-first search has the smallest runtime per expansion, followed by A*, Generalized Adaptive A* and D* Lite (in this order). This is not surprising: Breadth-first search leads due to the fact that it does not use a priority queue. D* Lite and Generalized Adaptive A* trail due to their runtime overhead as incremental heuristic search algorithms and need to make up for it by decreasing their number of expansions per search sufficiently to result in smaller runtimes per search. Table 1 shows the following trends:

- D* Lite is fast only for stationary-target search with small k. In the other cases, its number of expansions per search is too large (as explained already) and dominates some of the effects discussed below, which is why we do not list D* Lite in the following.
- The number of searches and moves until the target is reached decreases as k increases since the distance

	Stationary Target						Moving Target					
	searches until target reached	moves until target reached	expansions per search		runtime per search	propagations per search	searches until target reached	moves until target reached	expansions per search		runtime per search	propagations per search
						$k = 1$						
BFS (Forward)	298	2984	13454	(77.8)	2009	N/A	1482	2673	13255	(34.4)	1992	N/A
BFS (Backward)	298	2984	10078	(58.2)	1510	N/A	1482	2673	9885	(25.6)	1474	N/A
A* (Forward)	298	2984	12674	(73.3)	2670	N/A	1482	2673	12428	(32.2)	2596	N/A
A* (Backward)	298	2984	9593	(55.4)	1982	N/A	1482	2673	9402	(24.4)	1899	N/A
D* Lite	298	2984	558	(15.7)	397	N/A	1211	2705	15500	(39.6)	57704	N/A
GAA* (Forward)	299	2987	3686	(21.3)	1163	1311	1479	2660	5308	(13.7)	1283	191
GAA* (Backward)	310	3097	3581	(20.3)	1035	746	1510	2679	3504	(9.0)	838	150
						$k = 5$						
BFS (Forward)	177	1770	11479	(86.1)	1781	N/A	945	1707	10634	(34.6)	1588	N/A
BFS (Backward)	176	1765	8933	(67.1)	1390	N/A	943	1705	8528	(27.7)	1252	N/A
A* (Forward)	177	1775	10491	(78.6)	2328	N/A	946	1709	9673	(31.4)	2036	N/A
A* (Backward)	176	1759	8324	(62.6)	1810	N/A	942	1704	7876	(25.6)	1613	N/A
D* Lite	176	1760	1534	(32.5)	798	N/A	787	1754	13103	(41.6)	59270	N/A
GAA* (Forward)	176	1757	3197	(24.1)	1548	3607	950	1710	4235	(13.7)	1052	494
GAA* (Backward)	180	1801	3103	(23.1)	1284	2437	1008	1776	2645	(8.3)	662	412
						$k = 10$						
BFS (Forward)	131	1307	10228	(89.3)	1650	N/A	768	1366	10084	(36.3)	1437	N/A
BFS (Backward)	130	1305	7712	(67.3)	1266	N/A	763	1357	7701	(27.8)	1071	N/A
A* (Forward)	130	1305	9050	(79.1)	2091	N/A	759	1354	8889	(32.2)	1864	N/A
A* (Backward)	130	1303	6990	(61.1)	1605	N/A	759	1353	6950	(25.2)	1402	N/A
D* Lite	130	1306	2048	(44.3)	1053	N/A	619	1376	11478	(44.8)	60916	N/A
GAA* (Forward)	131	1308	2991	(26.1)	1903	5232	777	1372	4038	(14.4)	1058	688
GAA* (Backward)	139	1388	2813	(23.8)	1530	3708	813	1400	2289	(8.0)	626	603
						$k = 20$						
BFS (Forward)	114	1143	9895	(92.3)	1657	N/A	589	1064	9018	(37.1)	1305	N/A
BFS (Backward)	115	1149	7548	(70.2)	1283	N/A	588	1067	7482	(30.8)	1058	N/A
A* (Forward)	115	1154	8552	(79.4)	2063	N/A	589	1069	7777	(32.0)	1666	N/A
A* (Backward)	115	1149	6670	(62.1)	1602	N/A	586	1066	6582	(27.1)	1369	N/A
D* Lite	115	1151	2700	(54.0)	1380	N/A	480	1092	10784	(49.3)	66649	N/A
GAA* (Forward)	115	1147	3097	(28.8)	2251	6241	598	1070	3679	(15.0)	1040	888
GAA* (Backward)	117	1171	2922	(26.9)	1782	4426	632	1113	2163	(8.6)	660	817
						$k = 50$						
BFS (Forward)	82	819	8891	(97.9)	1633	N/A	448	789	8559	(40.4)	1268	N/A
BFS (Backward)	82	821	7169	(78.8)	1360	N/A	450	791	6537	(30.8)	942	N/A
A* (Forward)	82	819	7221	(79.5)	1927	N/A	454	795	6849	(32.1)	1530	N/A
A* (Backward)	82	822	5963	(65.5)	1611	N/A	448	790	5453	(25.7)	1173	N/A
D* Lite	82	821	3531	(72.8)	1949	N/A	354	789	8917	(51.3)	79896	N/A
GAA* (Forward)	83	832	3265	(35.6)	3001	8288	452	788	3545	(16.6)	1108	1139
GAA* (Backward)	83	831	3039	(33.2)	2395	6151	473	814	1975	(9.1)	713	1066

Table 1: Experimental Results

between two cells then decreases more quickly (as explained already), which decreases more quickly the length of the path from the current cell of the agent to the current cell of the target. This decreases the number of moves and also the number of searches since the action costs change every tenth time step and a new search thus needs to be performed (at least) every tenth time step. The number of searches until the target is reached is larger for moving-target search than stationary-target search since additional searches need to be performed every time the target leaves the current path.

- The number of propagations per search of Generalized Adaptive A* increases as k increases since a larger number of action costs then decrease. The number of propagations per search of Generalized Adaptive A* is smaller for moving-target search than stationary-target search for two reasons: First, the agent performs an A* search for stationary target search whenever the action costs change. Thus, the number of searches equals the number of runs of the consistency procedure. The agent performs an A* search for moving-target search whenever the action costs change or the target leaves the current path. Thus, the number of searches is larger than the number of runs of the consistency procedure and the propagations are now amortized over several searches. Second, the additional updates of the h-values from Section 5.2 keep the h-values smaller. Thus, the h-values of more cells are equal to their Manhattan distances, and cells whose h-values are equal to the Manhattan distances stop the updates of the h-values by the consistency procedure. (This effect also explains why the number of propagations per search of Generalized Adaptive A* is smaller for backward search than forward search.)

- The number of expansions per search of Generalized Adaptive A* is smaller than the one of A* since its h-values cannot be less informed than the ones of A*. Similarly, the number of expansions per search of A* is smaller than the one of breadth-first search since its h-values cannot be less informed than uninformed.

- The runtime per search of breadth-first search, A* and Generalized Adaptive A* is smaller for moving-target search than stationary-target search, which is an artifact of calculating the runtime per search by dividing the total runtime by the number of searches since the runtime for generating the initial gridworlds and initializing the data structures is then amortized over a larger number of searches.

We say in the following that one heuristic search algorithm is better than another one if its runtime per search is smaller for the same search direction, no matter what the search direction is. We included breadth-first search in our comparison since our gridworlds have few cells and small branching factors, which makes breadth-first search better than A*. Generalized Adaptive A* mixes A* with

Dijkstra's algorithm for the consistency procedure, and Dijkstra's algorithm has a larger runtime per expansion than breadth-first search. The number of propagations per search of Generalized Adaptive A* increases as k increases. Generalized Adaptive A* cannot be better than breadth-first search when this number is about as large as the number of expansions of breadth-first search, which happens only for stationary-target search with large k. Overall, for stationary-target search, D* Lite is best for small k and breadth-first search is best for large k. Generalized Adaptive A* is always worse than D* Lite. It is better than breadth-first search and A* for small k but worse then them for large k since its number of propagations is then larger and the improved h-values remain informed for shorter periods of time. On the other hand, for moving-target search, Generalized Adaptive A* is best (which is why it advances the state of the art in heuristic search), followed by breadth-first search, A* and D* Lite (in this order).

9. RELATED WORK

Real-time heuristic search [10] limits A* searches to part of the state space around the current state (= local search space) and then follows the resulting path. It repeats this procedure once it leaves the local search space (or, if desired, earlier), until it reaches the goal state. After each search, it uses a consistency procedure to update some or all of the h-values in the local search space to avoid cycling without reaching the goal state [12, 6, 3]. Real-time heuristic search thus typically solves a search problem by searching repeatedly in the same state space and following a trajectory from the start state to the goal state that is not a shortest path. Adaptive A* and Generalized Adaptive A* (GAA*), on the other hand, solve a series of similar but not necessarily identical search problems. For each search problem, they search once and determine a shortest path from the start state to the goal state. This prevents the consistency procedure of Generalized Adaptive A* (GAA*) from restricting the updates of the h-values to small parts of the state space because all of its updates are necessary to keep the h-values consistent with respect to the goal state and thus find a shortest path from the start state to the goal state. However, Adaptive A* has recently been modified to perform real-time heuristic search [8].

10. CONCLUSIONS

In this paper, we showed how to generalize Adaptive A* to find shortest paths in state spaces where the action costs can increase or decrease over time. Our experiments demonstrated that Generalized Adaptive A* outperforms both breadth-first search, A* and D* Lite for moving-target search. It is future work to extend our experimental results to understand better when Generalized Adaptive A* (GAA*) runs faster than breadth-first search, A* and D* Lite. It is also future work to combine the principles behind Generalized Adaptive A* (GAA*) and D* Lite to speed it up even more.

11. 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.

12. REFERENCES

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

[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] C. Hernández and P. Maseguer. LRTA*(k). In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1238–1243, 2005.

[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 the International Joint Conference on Artificial Intelligence*, pages 204–210, 1991.

[6] S. Koenig. A comparison of fast search methods for real-time situated agents. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, pages 864–871, 2004.

[7] 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 402–405, 2006.

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

[9] S. Koenig, M. Likhachev, and X. Sun. Speeding up moving-target search. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 2007.

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

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

[12] J. Pemberton and R. Korf. Incremental path planning on graphs with cycles. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 179–188, 1992.