

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

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.

Copyright © 2008, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

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.

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 $\text{succ}(s, a) \in S$ 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 $H(s, s') \leq c(s, a) + H(\text{succ}(s, a), s')$ 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 $h(s) \leq c(s, a) + h(\text{succ}(s, a))$ 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 $\text{tree}(s)$ (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 preceding h-values at least weakly [7]. Thus, they are no less informed than the immediately preceding h-values and an A* search with the updated h-values thus cannot expand more states than an A* search with the immediately preceding 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 preceding 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, $h(s) \leq c(s, a) + h(\text{succ}(s, a))$ for $s \in$

$S \setminus \{s_{\text{goal}}\}$ implies that $h(s) \leq c(s, a) + h(\text{succ}(s, a)) \leq c'(s, a) + h(\text{succ}(s, a))$ 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 x th execution of `ComputePath()`, that is, the x th A* search.
- The value of *search(s)* is x if state s was generated last by the x th 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 x th 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 ($\text{search}(s) = 0$) [Line 9] or has not yet been generated by the current A* search ($\text{search}(s) \neq \text{counter}$) but was generated by some A* search ($\text{search}(s) \neq 0$) [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 ($\text{search}(s) = 0$) [Line 10]

```

1 procedure InitializeState(s)
2   if search(s)  $\neq$  counter AND search(s)  $\neq$  0
3     if  $g(s) + h(s) < \text{pathcost}(\text{search}(s))$ 
4        $h(s) := \text{pathcost}(\text{search}(s)) - g(s)$ ;
5        $h(s) := h(s) - (\text{deltah}(\text{counter}) - \text{deltah}(\text{search}(s)))$ ;
6        $h(s) := \max(h(s), H(s, s_{\text{goal}}))$ ;
7        $g(s) := \infty$ ;
8     else if search(s) = 0
9        $g(s) := \infty$ ;
10       $h(s) := H(s, s_{\text{goal}})$ ;
11      search(s) := counter;
12 procedure ComputePath()
13   while  $g(s_{\text{goal}}) > \min_{s' \in \text{OPEN}} (g(s') + h(s'))$ 
14     delete a state  $s$ 
15     with the smallest f-value  $g(s) + h(s)$  from OPEN;
16     for all actions  $a \in A(s)$ 
17       InitializeState(succ(s, a));
18       if  $g(\text{succ}(s, a)) > g(s) + c(s, a)$ 
19          $g(\text{succ}(s, a)) := g(s) + c(s, a)$ ;
20         tree(succ(s, a)) :=  $s$ ;
21         if succ(s, a) is in OPEN then delete it from OPEN;
22         insert succ(s, a) into OPEN
23         with f-value  $g(\text{succ}(s, a)) + h(\text{succ}(s, a))$ ;
24 procedure Main()
25   counter := 1;
26   deltah(1) := 0;
27   for all states  $s \in S$ 
28     search(s) := 0;
29   while  $s_{\text{start}} \neq s_{\text{goal}}$ 
30     InitializeState( $s_{\text{start}}$ );
31     InitializeState( $s_{\text{goal}}$ );
32      $g(s_{\text{start}}) := 0$ ;
33     OPEN :=  $\emptyset$ ;
34     insert  $s_{\text{start}}$  into OPEN with f-value  $g(s_{\text{start}}) + h(s_{\text{start}})$ ;
35     ComputePath();
36     if OPEN =  $\emptyset$ 
37       pathcost(counter) :=  $\infty$ ;
38     else
39       pathcost(counter) :=  $g(s_{\text{goal}})$ ;
40     change the start and/or goal states, if desired;
41     set  $s_{\text{start}}$  to the start state;
42     set  $s_{\text{newgoal}}$  to the goal state;
43     if  $s_{\text{goal}} \neq s_{\text{newgoal}}$ 
44       InitializeState( $s_{\text{newgoal}}$ );
45       if  $g(s_{\text{newgoal}}) + h(s_{\text{newgoal}}) < \text{pathcost}(\text{counter})$ 
46          $h(s_{\text{newgoal}}) := \text{pathcost}(\text{counter}) - g(s_{\text{newgoal}})$ ;
47         deltah(counter + 1) := deltah(counter) +  $h(s_{\text{newgoal}})$ ;
48          $s_{\text{goal}} := s_{\text{newgoal}}$ ;
49     else
50       deltah(counter + 1) := deltah(counter);
51       counter := counter + 1;
52     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 ($\text{search}(s) \neq \text{counter}$). Second, the state was generated by a previous A* search ($\text{search}(s) \neq 0$). Third, the state was expanded by the A* search that generated it last ($g(s) + h(s) < \text{pathcost}(\text{search}(s))$). 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 ($\text{pathcost}(\text{search}(s))$) 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 ≠ sgoal 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 \ {sgoal} 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 x th A* search is the running sum of all corrections up to the beginning of the x th 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, $h(s_{goal}) = 0$ 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

¹Line 3' can be simplified to “for all state-action pairs (s, a) whose $c(s, a)$ decreased” since $h(s_{goal}) = 0$ for consistent h-values with respect to the goal state and the condition on Line 6' is thus never satisfied for $s = s_{goal}$. 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.

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 $s \neq s_{goal}$ 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

