Simple Optimization Techniques for A*-Based Search[*]

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

ABSTRACT

In this paper, we present two simple optimizations that can reduce the number of priority queue operations for A* and its extensions. Basically, when the optimized search algorithms expand a state, they check whether they will expand a successor of the state next. If so, they do not first insert it into the priority queue and then immediately remove it again. These changes might appear to be trivial but are well suited for Generalized Adaptive A*, an extension of A*. Our experimental results indeed show that they speed up Generalized Adaptive A* by up to 30 percent if its priority queue is implemented as a binary heap.

Categories and Subject Descriptors

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

General Terms

Algorithm; Experimentation; Theory

Keywords

A* Search; Incremental Heuristic Search; Path Planning

1. INTRODUCTION

A* [1] is probably the most popular search algorithm in artificial intelligence and has been extended in various directions, resulting in search algorithms such as Generalized Adaptive A* [10], Learning Real-Time A* [6], Dynamic Weighting A* [9] and Anytime Replanning A* [7]. In this paper, we present two simple optimizations that can reduce the number of priority queue operations for A* and its extensions. Basically, when the optimized search algorithms expand a state, they check whether they will expand a successor of the state next. If so, they do not first insert it into the priority queue and then immediately remove it again.

These changes might appear to be trivial but are well suited for Generalized Adaptive A*, an extension of A*. Our experimental results indeed show that they speed up Generalized Adaptive A* by up to 30 percent if its priority queue is implemented as a binary heap.

2. NOTATION

Although A* and its extensions apply to general graphs, we illustrate them and our optimizations on four-neighbor grids with blocked and unblocked cells. S is the set of all cells, $s_{start} \in S$ is the start cell of the search, and $s_{goal} \in S$ is the goal cell of the search. If cell $s \in S$ is unblocked, then $succ(s) \subseteq S$ is the set of its neighboring cells that are unblocked. Otherwise, $succ(s) = \emptyset$. Moving from a cell $s \in S$ to its successor $s' \in succ(s)$ has cost $c(s, s') = 1$. Moving from a cell $s \in S$ to any other cell $s' \in S$ has cost $c(s, s') = \infty$.

3. A*

We describe a version of A* that we will later generalize to Generalized Adaptive A*. Its pseudocode is shown in Figure 1.

3.1 Heuristics

A* and its extensions use h-values ($=$ heuristics) to focus their search. The h-values are derived from user-supplied H-values $H(s, s')$ that estimate the cost of moving from cell $s \in S$ to cell $s' \in S$. The user-supplied H-values $H(s, s')$ have to satisfy the triangle inequality, namely satisfy $H(s', s') = 0$ and $0 \leq H(s, s') \leq c(s, s'') + H(s'', s')$ for all cells $s, s' \in S$ and $s'' \in succ(s)$. In this case, the h-values $h(s) = H(s, s_{goal})$ are consistent with respect to any goal cell $s_{goal} \in S$, namely satisfy $h(s_{goal}) = 0$ and $0 \leq h(s) \leq c(s, s') + h(s')$ for all cells $s \in S$ and $s' \in succ(s)$ (Consistency Property) [8]. We use the Manhattan distances as H-values, that is, the smallest cost of moving from one cell to another cell on a four-neighbor grid without blocked cells.

3.2 Values

A* maintains four values for every cell $s \in S$: **(1)** The h-value $h(s) := H(s, s_{goal})$ is an approximation of the smallest cost of moving from cell s to the goal cell, as described above. **(2)** The g-value $g(s)$ is the smallest cost of moving from the start cell to cell s found so far. **(3)** The f-value $f(s) := g(s) + h(s)$ is an estimate of the smallest cost of moving from the start cell via cell s to the goal cell. **(4)** The parent pointer $parent(s)$ points to the parent of cell s in the A*

```
01  function Main()
02  g(s_start) := 0;
03  parent(s_start) := NULL;
04  OPEN := ∅;
05  OPEN.Insert(s_start, g(s_start) + h(s_start));
06  CLOSED := ∅;
07  while OPEN ≠ ∅
08      s := OPEN.Pop();
09      if s = s_goal
10          return "path found";
11      CLOSED = CLOSED ∪ {s};
12      foreach s' ∈ succ(s)
13          if s' ∉ CLOSED
14              if s' ∉ OPEN
15                  g(s') = g(s) + c(s, s');
16                  parent(s') = s;
17                  OPEN.Insert(s', g(s') + h(s'));
18              else if g(s) + c(s, s') < g(s')
19                  g(s') = g(s) + c(s, s');
20                  parent(s') = s;
21                  OPEN.Remove(s');
22                  OPEN.Insert(s', g(s') + h(s'));
23  return "no path found";
```

Figure 1: A*.

search tree. The parent pointers are used to extract the path after the search terminates.

3.3 OPEN and CLOSED Lists

A* maintains two data structures: **(1)** The $OPEN$ list is a priority queue that contains all cells to be considered for expansion. $OPEN.Insert(s, x)$ inserts cell s with value x into the $OPEN$ list, $OPEN.Remove(s)$ removes cell s from the $OPEN$ list, and $OPEN.Pop()$ removes a cell with the smallest value from the $OPEN$ list and returns it. **(2)** The $CLOSED$ list is a set that contains all cells that have already been expanded.

3.4 Algorithm

A* repeats the following procedure until the $OPEN$ list is empty (Line 23) or it is about to expand the goal cell (Line 10): It removes a cell s with the smallest f-value from the $OPEN$ list (Line 08), inserts the cell into the $CLOSED$ list (Line 11) and *expands* it by performing the following operations for each successor $s' \in succ(s)$ of cell s. If cell s' is neither in the $OPEN$ nor $CLOSED$ list, then A* *generates* the cell by assigning $g(s') := g(s) + c(s, s')$, setting the parent pointer of cell s' to cell s, and then inserting cell s' into the $OPEN$ list with f-value $f(s') = g(s') + h(s')$ (Lines 15-17). If cell s' is in the $OPEN$ list and $g(s) + c(s, s') < g(s')$, then A* *updates* cell s' by assigning $g(s') := g(s) + c(s, s')$, setting the parent pointer of cell s' to cell s, and then updating the f-value of cell s' in the $OPEN$ list to f-value $f(s') = g(s') + h(s')$ (Lines 19-22).

3.5 Properties

A* has the following properties if its h-values are consistent [8].

- **A* Property 1:** The sequence of the f-values of the expanded cells is monotonically non-decreasing.

- **A* Property 2:** A path with the smallest cost of moving from the start cell to any expanded cell s can be identified in reverse by repeatedly following the parent pointers from cell s to the start cell. Similarly, a path with the smallest cost of moving from the start cell to the goal cell s can be identified in reverse by
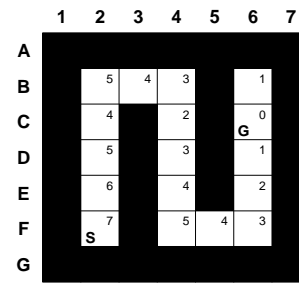

Figure 2: Example Grid.

repeatedly following the parent pointers from the goal cell to the start cell after the search terminates.

- **A* Property 3:** An A* search expands no more cells than an otherwise identical A* search for the same search problem if the h-values used by the first A* search are no smaller for any cell than the corresponding h-values used by the second A* search (= the former h-values dominate the latter h-values at least weakly).

3.6 Optimizations

Consider the following two simple optimizations that can reduce the number of priority queue operations (insertions and removals of cells) for A* and thus potentially speed it up in case the priority queue operations are expensive.

- **A* Variant 1:** Assume that A* expands a cell s with f-value $f(s)$ and, in the process, inserts a successor s' of the cell into the $OPEN$ list with f-value $f(s')$ (Line 17) so that (Expansion Property 1) $f(s') = f(s)$. Since the sequence of the f-values of the expanded cells is monotonically non-decreasing (A* Property 1), A* can expand cell s' next and then removes it again from the $OPEN$ list (Line 08). In this case, our optimized A* Variant 1 expands cell s' immediately after cell s. It leaves out both the insertion of cell s' into the $OPEN$ list and its immediate removal from the $OPEN$ list. We say that cell s' is expanded fast. In the experiments, our A* Variant 1 expands the first generated successor with Expansion Property 1 fast and all other successors regularly (= slowly). The resulting order of cell expansions in grids is consistent with breaking ties among cells with equal f-values in favor of cells with larger g-values, which is considered to be a good tie-breaking strategy.

- **A* Variant 2:** Assume that A* expands a cell s with f-value $f(s)$ and, in the process, inserts a successor s' of the cell into the $OPEN$ list with f-value $f(s')$ (Line 17) so that (Expansion Property 2) every successor of the cell that is also inserted into the $OPEN$ list is inserted into the $OPEN$ list with an f-value that is no smaller than $f(s')$ and (Expansion Property 3) either the $OPEN$ list is empty or the smallest f-value of any cell in the $OPEN$ list is no smaller than $f(s')$. Since the sequence of the f-values of the expanded cells is monotonically non-decreasing (A* Property 1), A* can expand cell s' next and then removes it again from the $OPEN$ list (Line 08). In the experiments,
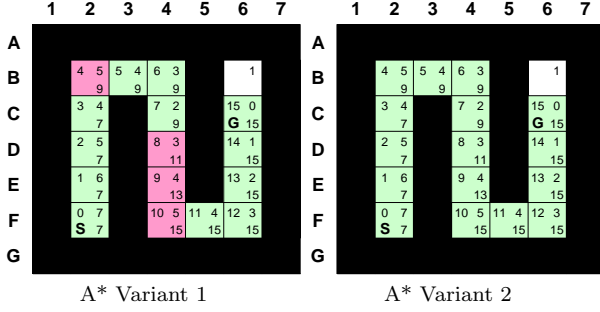
Figure 3: Execution Traces of A* Variants.

our A* Variant 2 expands the first generated successor with Expansion Property 1 fast and all other successors slowly. If there is no successor with Expansion Property 1, then it expands the first generated successor with Expansion Properties 2 and 3 fast and all other successors slowly.

A* Variants 1 and 2 can expand the start cell fast as well. A* Variant 1 will sometimes be able to expand cells fast due to the h-values used. Every cell $s \in S \setminus \{s_{goal}\}$ has one or two neighboring cells $s' \in S$ in grids with $h(s) = c(s, s') + h(s')$ if the Manhattan distances are used. If cells s and s' are unblocked, then $s' \in succ(s)$ and it holds that $f(s) = g(s) + h(s) = g(s) + c(s, s') + h(s') = g(s') + h(s') = f(s')$ when A* expands cell s and updates the g-value of its successor s' to $g(s') := g(s) + c(s, s')$. A* Variant 2 expands every cell fast that A* Variant 1 expands fast and thus reduces the number of priority queue operations of A* at least as much as A* Variant 1 but has additional overhead due to the bookkeeping operations required to keep track of the successor with the smallest f-value. Thus, the effect of A* Variants 1 and 2 on the runtime of A* needs to be determined experimentally.

3.7 Example

We use the grid from Figure 2 to illustrate A* Variants 1 and 2. Black cells are blocked, and white cells are unblocked. The start cell $F2$ is marked S, and the goal cell $C6$ is marked G. The h-values are shown in the top right corners of the cells. The execution traces of A* Variants 1 and 2 are shown in Figure 3. The g-values and f-values are shown in the top left and bottom right corners, respectively, of the cells. A* Variant 1 expands the red cells slowly and the green cells fast. A* Variant 2 expands all (green) cells fast.

4. GENERALIZED ADAPTIVE A*

We use Generalized Adaptive A* (GAA*) [10] to demonstrate that our optimizations apply not only to A* but its extensions as well. The pseudocode of GAA* is shown in Figure 4. GAA* is an incremental search algorithm that solves a sequence of search problems. The current cell of the agent (start cell), goal cell and the blockage status of cells can change between searches. GAA* performs A* searches but updates the h-values to make future A* searches more focused. Eager GAA* updates the h-values immediately after each A* search, while Lazy GAA* updates them only when they are needed during a future A* search. We describe Eager GAA* in the following because it is easier to

```
01'  function Main()
02'  while s_start ≠ s_goal
03'      run A*;
04'      move the agent along the path returned by A* (using A*
         Property 2) until it reaches s_goal, s_goal changes, or action
         costs or blockages change;
05'      forall s ∈ CLOSED
06'          h(s) := g(s_goal) − g(s);
07'      s_start := current agent cell (if changed);
08'      s_goal := current goal cell (if changed);
09'      if the goal cell changed
10'          temp := h(s_goal);
11'          forall s ∈ S
12'              h(s) := max(H(s, s_goal), h(s) − temp);
13'      update the blockage status of cells and the action costs (if any);
14'      OPEN := ∅;
15'      forall s ∈ S \ {s_goal} and s' ∈ succ(s)
         whose c(s, s') decreased or for which s' ∈ succ(s) became true
16'          if h(s) > c(s, s') + h(s')
17'              h(s) := c(s, s') + h(s');
18'              if s ∈ OPEN
19'                  OPEN.Remove(s);
20'              OPEN.Insert(s, h(s));
21'      while OPEN ≠ ∅
22'          s' := OPEN.Pop();
23'          forall s ∈ S \ {s_goal} and s' ∈ succ(s)
24'              if h(s) > c(s, s') + h(s')
25'                  h(s) := c(s, s') + h(s');
26'                  if s ∈ OPEN
27'                      OPEN.Remove(s);
28'                  OPEN.Insert(s, h(s));
```

Figure 4: Generalized Adaptive A* (GAA*).

understand but use Lazy GAA* in the experiments because it runs faster.

4.1 Algorithm: Improving the h-Values

GAA* updates (= overwrites) the consistent h-values with respect to the goal cell of all expanded cells $s \in S$ after an A* search by assigning

$$h(s) := g(s_{goal}) - g(s) \qquad (1)$$

(Lines 05'-06'). This principle was first used in [2] and later resulted in the independent development of GAA*. The updated h-values are again consistent with respect to the goal cell and dominate the immediately preceeding h-values at least weakly [3]. Thus, future A* searches of GAA* are more focused since an A* search with the updated h-values expands no more cells than an otherwise identical A* search with the immediately preceeding h-values (A* Property 3).

4.2 Algorithm: Maintaining Consistency

It is important that the h-values remain consistent with respect to the goal cell from A* search to A* search. The following changes can occur between A* searches:

- **Change 1:** The start cell changes. In this case, GAA* does not need to do anything since the h-values remain consistent with respect to the goal cell.

- **Change 2:** The goal cell changes. In this case, GAA* needs to update the h-values since they might not be consistent with respect to the new goal cell. Assume that the goal cell changes from $s_{goal} \in S$ to $s'_{goal} \in S$. GAA* then updates the h-values of all cells $s \in S$ by assigning

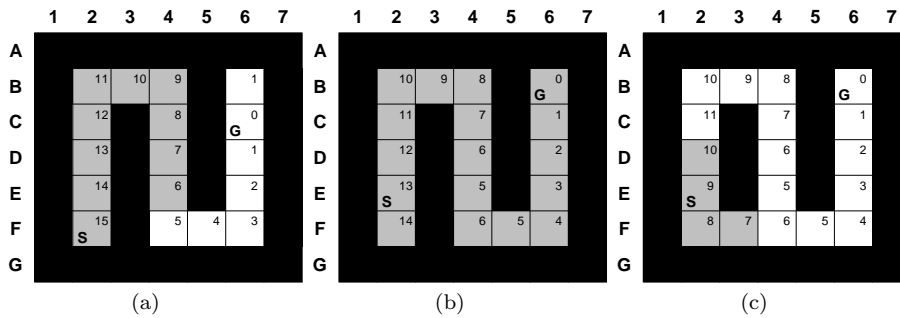$$h(s) := \max(H(s, s'_{goal}), h(s) - h(s'_{goal})) \qquad (2)$$
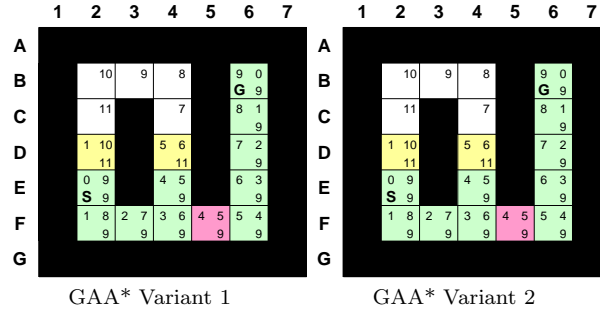
Figure 5: h-Value Updates.

Figure 6: Execution Traces of GAA* Variants.

(Lines 10'-12'). The updated h-values are consistent with respect to the new goal cell [4]. However, they are potentially smaller than the immediately preceeding h-values. Taking the maximum of $h(s) - h(s'_{goal})$ and the user-supplied H-value $H(s, s'_{goal})$ with respect to the new goal cell ensures that the updated h-values dominate the user-supplied H-values at least weakly.

- **Change 3:** Costs or blockages changed but no cost decreased and no successor set contains additional cells. In this case, GAA* does not need to do anything since the h-values remain consistent with respect to the goal cell [10].

- **Change 4:** At least one cost decreased or at least one successor set contains additional cells. In this case, GAA* needs to update the h-values since they might no longer be consistent with respect to the goal cell. GAA* does so by executing the Consistency Procedure, which reuses the $OPEN$ list as its priority queue: If a cell s and its successor $s' \in succ(s)$ no longer satisfy the Consistency Property $h(s) \leq c(s, s') + h(s')$, then it assigns $h(s) := c(s, s') + h(s')$ and inserts cell s into the $OPEN$ list with h-value $h(s)$ or, if it is already in the $OPEN$ list, updates its h-value to $h(s)$ (Lines 14'-20'). Then, it repeats the following procedure until the $OPEN$ list is empty: It removes a cell s' with the smallest h-value from the $OPEN$ list. If a cell s and its successor $s' \in succ(s)$ no longer satisfy the Consistency Property $h(s) \leq c(s, s') + h(s')$, then it assigns $h(s) := c(s, s') + h(s')$ and inserts cell s into the $OPEN$ list with h-value $h(s)$ or, if it is already in the $OPEN$ list, updates its h-value to $h(s)$ (Lines 21'-28'). The updated h-values are consistent with respect to the goal cell [10].

4.3   Optimizations

GAA* performs A* searches and can thus use A* Variants 1 and 2, resulting in GAA* Variants 1 and 2, respectively. GAA* Variants 1 and 2 are often able to expand cells fast due to the way GAA* updates its h-values. For example, after Assignment (1) assigns $h(s) := g(s_{goal}) - g(s)$ and $h(s') := g(s_{goal}) - g(s')$ after an A* search to two expanded cells $s, s' \in S$ with $s' \in succ(s)$, it holds that $f(s) = g(s) + h(s) = g(s_{goal}) = g(s') + h(s') = f(s')$. Similarly, after the Consistency Procedure assigns $h(s) := c(s, s') + h(s')$ to two cells $s, s' \in S$ with $s' \in succ(s)$, it holds that $f(s) = g(s) + h(s) = g(s) + c(s, s') + h(s') = g(s') + h(s') = f(s')$ when A* expands cell s and updates the g-value of its successor s' to $g(s') := g(s) + c(s, s')$. Assignment (2) will often leave these relationships unchanged.

4.4   Example

We again use the grid from Figure 2 to illustrate GAA* Variants 1 and 2. The first A* search of GAA* Variants 1 and 2 is identical to that of A* Variants 1 and 2, respectively, shown in Figure 3. Assume that the start cell then changes to $E2$, the goal cell changes to $B6$, and cell $F3$ becomes unblocked. The h-values are shown in the top right corners of the cells. GAA* first improves the h-values of all expanded cells with Assignment (1), shown in Figure 5(a). Cells whose h-values changed are shown in grey. GAA* then updates the start and goal cells and corrects the h-values of all cells with Assignment (2) to make them consistent with respect to the new goal cell, shown in Figure 5(b). Cells whose h-values changed are shown in grey. GAA* then makes cell $F3$ unblocked and corrects the h-values of cells $D2$, $E2$, $F2$ and $F3$ with the Consistency Procedure to make them again consistent, shown in Figure 5(c). Cells whose h-values changed are shown in grey. Finally, GAA* performs its second A* search. The execution trace of GAA* Variants 1 and 2 is

shown in Figure 6. The g-values and f-values are shown in the top left and bottom right corners, respectively, of the cells. The second A* search of GAA* Variant 1 generates but does not expand the yellow cells. It expands the red cells slowly and the green cells fast. When it expands cell $F4$ with f-value 9, it could expand either cell $E4$ or $F5$ fast since both of them have f-value 9. We assume that it generates cell $E4$ first and thus expands it fast. The second A* search of GAA* Variant 2 is identical to the one of GAA* Variant 1.

5. EXPERIMENTAL EVALUATION

We perform several experiments to compare the runtimes of A* Variants 1 and 2 against the runtime of A* and the runtimes of GAA* Variants 1 and 2 against the runtime of GAA*.

5.1 Experimental Setup

We perform experiments in randomly generated four-connected torus-shaped grids of size 200×200. We generate their random corridor structure with a depth-first search, which guarantees that there exists a path between any two unblocked cells. We choose the start and goal cells randomly. We base all search algorithms on the same implementation but use two different data structures for implementing their priority queues, namely binary heaps and buckets, because they differ in the runtime of their priority queue operations. We perform two different kinds of experiments, both of which require repeated searches. In both cases, all search algorithms search from the agent (start cell) to the target (goal cell).

- **Stationary-Target Search:** We study a version of stationary-target search in which an agent needs to reach a stationary target on an initially unknown grid. The agent always knows which (unblocked) cell it is in and which (unblocked) cell the target is in. Initially, the agent does not know which cells are blocked but it can always sense the blockage status of its four neighboring cells. 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 to the cell of the target (freespace assumption) [5]. It recomputes the shortest presumed unblocked path whenever it observes its current path to be blocked.

- **Moving-Target Search:** We also study a version of moving-target search in which an agent needs to reach a moving target on an initially known grid. We unblock k blocked cells and block k unblocked cells every tenth time step, while ensuring that there always exists a path from the current cell of the agent to the current cell of the target. The target moves randomly with the restriction that it does not move every tenth time step, which allows the agent to reach the current cell of the target eventually. Additionally, we do not allow the target to return to its previous cell, unless that is its only possible move. The agent always knows which cells are currently blocked and moves on a shortest path from its current cell to the current cell of the target. It recomputes the shortest path whenever the target moves off its current path or the blockage status of cells changes.

5.2 Experimental Results

Tables 1 and 2 compare A* Variants 1 and 2 against A* and GAA* Variants 1 and 2 against GAA*, averaged over 100 grids on a Pentium D 3.0 GHz computer with 2 GByte of RAM. We report one measure for the difficulty of the search problems, namely the average number of searches performed until the agent reaches the target. We also report three measures for the efficiency of the search algorithms, namely the number of slowly expanded cells per search, the number of fast expanded cells per search (for A* Variants 1 and 2 and GAA* Variants 1 and 2) and the total runtime per search in microseconds. We also 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 reports on stationary-target search. In general, A* Variant 1 or GAA* Variant 1 is always faster than A* Variant 2 or GAA* Variant 2, respectively. If the priority queues are implemented with buckets, then A* or GAA* is faster than A* Variant 1 or GAA* Variant 1, respectively. However, it is not always feasible to implement the priority queues with buckets, for example, when the f-values are not integers. If the priority queues are implemented with binary heaps, then A* Variant 1 or GAA* Variant 1 is faster than A* or GAA*, respectively. In fact, A* Variant 1 is 14.5 percent faster than A*, and GAA* Variant 1 is 33.7 percent faster than GAA*. We draw the following conclusions.

- **Conclusion 1:** If the priority queues are implemented with buckets, then the priority queue operations are fast and the runtime overhead of the optimizations outweighs the runtime savings gained from reducing the number of priority queue operations. If the priority queues are implemented with binary heaps, then the priority queue operations are slow and the runtime savings gained from reducing the number of priority queue operations outweighs the runtime overhead of the optimizations.

- **Conclusion 2:** The additional runtime overhead of A* Variant 2 or GAA* Variant 2 over A* Variant 1 or GAA* Variant 1, respectively, outweighs the additional runtime savings gained from increasing the number of fast expansions.

- **Conclusion 3:** If the priority queues are implemented with binary heaps, then GAA* Variant 1 speeds up GAA* more than A* Variant 1 speeds up A* because GAA* Variant 1 can expand more cells fast than A* Variant 1 due to the way GAA* updates its h-values, as discussed earlier.

Table 2 reports on moving-target search. In general, A* Variant 1 or GAA* Variant 1 is always faster than A* or GAA* Variant 2, respectively. If the priority queues are implemented with buckets, then A* or GAA* is (almost) always faster than A* Variant 1 or GAA* Variant 1, respectively. If the priority queues are implemented with binary heaps, then A* Variant 1 is sometimes faster than A*, while GAA* Variant 1 is always faster than GAA*. GAA* Variant 1 is 30.7 percent faster than GAA* for $k = 1$, 24.9 percent faster than GAA* for $k = 10$, 14.7 percent faster than GAA* for $k = 20$, 12.9 percent faster than GAA* for $k = 50$ and 7.2 percent faster than GAA* for $k = 100$. Conclusions 1-3 from above continue to hold.

Table 1: Stationary Target Search in Initially Unknown Grids.

	Binary Heap					Buckets				
	searches until target reached	slow expansions per search	fast expansions per search	fast expansions per slow expansion	runtime per search	searches until target reached	slow expansions per search	fast expansions per search	fast expansions per slow expansion	runtime per search
A*	3417	1707 (7.5)	N/A	N/A	421	3381	1057 (3.7)	N/A	N/A	**134**
A* Variant 1	3418	980 (1.6)	726	0.7413	**360**	3485	803 (1.3)	635	0.7909	191
A* Variant 2	3417	977 (1.6)	730	0.7473	387	3485	778 (1.3)	659	0.8464	208
GAA*	3481	346 (1.6)	N/A	N/A	92	3308	277 (1.0)	N/A	N/A	**33**
GAA* Variant 1	3428	28 (0.1)	333	12.087	**61**	3509	24 (0.1)	315	13.199	39
GAA* Variant 2	3428	26 (0.1)	334	12.987	63	3552	22 (0.1)	321	14.599	42

Table 2: Moving Target Search in Known Grids.

	Binary Heap					Buckets				
	searches until target reached	slow expansions per search	fast expansions per search	fast expansions per slow expansion	runtime per search	searches until target reached	slow expansions per search	fast expansions per search	fast expansions per slow expansion	runtime per search
$k = 1$										
A*	716	4583 (15.6)	N/A	N/A	953	711	4602 (15.7)	N/A	N/A	**807**
A* Variant 1	790	2474 (8.8)	2463	0.9954	**909**	791	2474 (8.8)	2463	0.9956	878
A* Variant 2	790	2334 (8.4)	2602	1.1147	962	791	1935 (7.0)	3001	1.5509	886
GAA*	731	2056 (8.4)	N/A	N/A	414	731	2013 (8.2)	N/A	N/A	297
GAA* Variant 1	814	101 (1.5)	2132	21.059	**287**	814	101 (1.5)	2139	21.100	**281**
GAA* Variant 2	814	99 (1.5)	2133	21.444	299	814	97 (1.5)	2141	21.816	297
$k = 10$										
A*	371	3344 (18.5)	N/A	N/A	**652**	372	3355 (18.6)	N/A	N/A	**557**
A* Variant 1	447	1970 (9.6)	1934	0.9817	663	447	1970 (9.6)	1932	0.9809	652
A* Variant 2	447	1900 (9.2)	2003	1.0541	698	447	1668 (8.2)	2232	1.3376	671
GAA*	381	1310 (9.6)	N/A	N/A	301	371	1313 (9.8)	N/A	N/A	**223**
GAA* Variant 1	410	158 (3.0)	1303	8.9181	**226**	410	159 (3.0)	1306	8.2122	224
GAA* Variant 2	410	155 (2.9)	1306	8.4263	232	410	150 (2.9)	1313	8.7446	225
$k = 20$										
A*	278	2657 (19.4)	N/A	N/A	**504**	278	2712 (19.5)	N/A	N/A	**437**
A* Variant 1	361	1613 (8.9)	1563	0.9689	516	360	1633 (9.8)	1579	0.9672	515
A* Variant 2	361	1563 (8.7)	1609	1.0277	553	360	1406 (7.9)	1806	1.2840	529
GAA*	304	1088 (10.3)	N/A	N/A	266	305	1094 (10.2)	N/A	N/A	**195**
GAA* Variant 1	381	188 (3.2)	1201	6.3929	**227**	383	186 (3.2)	1191	6.3975	216
GAA* Variant 2	383	183 (3.2)	1203	6.5686	235	381	176 (3.1)	1203	6.8291	224
$k = 50$										
A*	212	2511 (21.8)	N/A	N/A	491	209	2501 (22.0)	N/A	N/A	**397**
A* Variant 1	255	1462 (10.1)	1395	0.9542	**477**	255	1466 (10.1)	1398	0.9537	463
A* Variant 2	254	1430 (9.9)	1428	0.9987	507	255	1293 (9.1)	1568	1.2129	479
GAA*	230	969 (12.0)	N/A	N/A	271	236	989 (11.8)	N/A	N/A	**209**
GAA* Variant 1	288	214 (4.3)	975	4.5397	**236**	287	213 (4.3)	968	4.5420	211
GAA* Variant 2	288	209 (4.2)	980	4.6729	247	287	199 (4.2)	980	4.9045	220
$k = 100$										
A*	183	2272 (22.6)	N/A	N/A	467	178	2311 (23.1)	N/A	N/A	**387**
A* Variant 1	225	1316 (9.9)	1221	0.9277	**441**	222	1339 (10.2)	1238	0.9247	412
A* Variant 2	225	1292 (9.8)	1242	0.9606	468	223	1206 (9.3)	1360	1.1274	433
GAA*	200	875 (12.8)	N/A	N/A	265	197	912 (13.0)	N/A	N/A	**213**
GAA* Variant 1	263	228 (4.7)	895	3.9189	**249**	259	229 (4.7)	890	3.8870	222
GAA* Variant 2	261	220 (4.7)	897	4.0672	256	257	214 (4.7)	901	4.2104	232

6. CONCLUSIONS

In this paper, we presented two simple optimizations that can reduce the number of priority queue operations for A* and its extensions. Basically, when the optimized search algorithms expand a cell, they check whether they will expand a successor of the cell next. If so, they do not first insert it into the priority queue and then immediately remove it again. Our experimental results show that our optimizations speed up Generalized Adaptive A* by up to 30 percent if its priority queue is implemented as a binary heap.

7. REFERENCES

[1] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 2:100–107, 1968.

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

[3] 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.

[4] S. Koenig, M. Likhachev, and X. Sun. Speeding up moving-target search. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 1136–1143, 2007.

[5] S. Koenig, Y. Smirnov, and C. Tovey. Performance bounds for planning in unknown terrain. *Artificial Intelligence Journal*, 147(1–2):253–279, 2003.

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

[7] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of the Neural Information Processing Systems Conference*, 2003.

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

[9] X. Sun, M. Druzdzel, and C. Yuan. Dynamic Weighting A* search-based MAP algorithm for Bayesian networks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2385–2390, 2007.

[10] X. Sun, S. Koenig, and W. Yeoh. Generalized Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 469–476, 2008.