Incremental ARA*: An Incremental Anytime Search Algorithm for Moving-Target Search*

Xiaoxun Sun Tansel Uras Sven Koenig
Computer Science Department
University of Southern California
941 W 37th Street, Los Angeles, CA 90089, USA
{xiaoxuns,turas,skoenig}@usc.edu

William Yeoh
Living Analytics Research Center
Singapore Management University
80 Stamford Road, Singapore 178902
williamyeoh@smu.edu.sg

Abstract

Moving-target search, where a hunter has to catch a moving target, is an important problem for video game developers. In our case, the hunter repeatedly moves towards the target and thus has to solve similar search problems repeatedly. We develop Incremental ARA* (I-ARA*) for this purpose, the first incremental anytime search algorithm for moving-target search in known terrain. We provide an error bound on the lengths of the paths found by I-ARA* and show experimentally in known four-neighbor gridworlds that I-ARA* can be used with smaller time limits between moves of the hunter than competing state-of-the-art moving-target search algorithms, namely repeated A*, G-FRA*, FRA*, and sometimes repeated ARA*. The hunter tends to make more moves with I-ARA* than repeated A*, G-FRA* or FRA*, which find shortest paths for the hunter, but fewer moves with I-ARA* than repeated ARA*, which finds suboptimal paths for the hunter like I-ARA*. Also, the error bounds on the lengths of the paths of the hunter tend to be smaller with I-ARA* than repeated ARA*.

Introduction

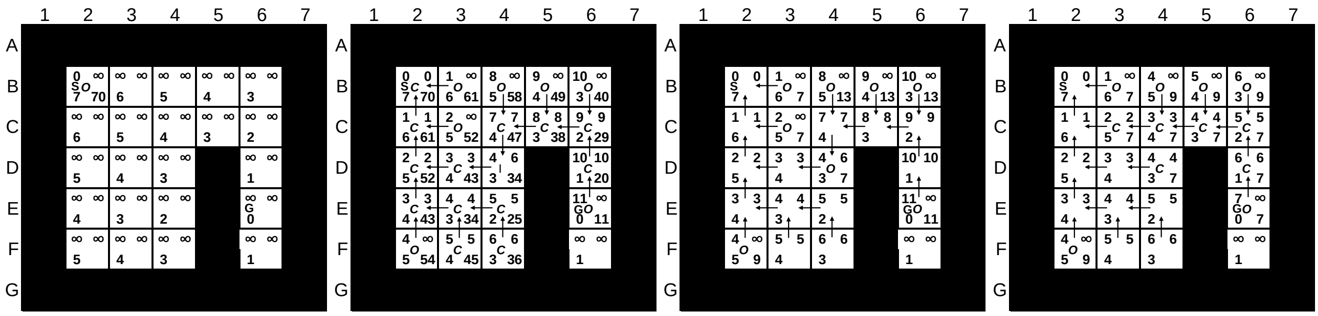
Moving-target search (Ishida and Korf 1991), where a hunter has to catch a moving target in known terrain, is an important problem for video game developers because video game characters often need to chase hostile characters or catch up with friendly characters. In our case, the hunter finds a path from its current location to the current location of the target and moves along it. If the target moves off the path, then the hunter repeats the process (Koenig, Likhachev, and Sun 2007). Thus, the hunter has to solve similar search problems repeatedly and has to do so fast to move smoothly. The video game company Bioware, for example, imposes a time limit of 1-3 ms per search (Bulitko et al. 2007a), which is not surprising for video games since their searches can use up to 70 percent of the available CPU time (Loh and Prakash 2009). Repeated A* with consistent h-values tends to run too slowly to stay within such stringent time limits between moves of the hunter but can be sped up in two different ways.[1]

- First, incremental search algorithms (Koenig et al. 2004) modify A* to reduce the runtime per search over repeated A* by re-using information from previous searches to speed up the current search. G-FRA* (Sun, Yeoh, and Koenig 2010), for example, is the currently fastest incremental moving-target search algorithm on known graphs, and FRA* optimizes it for known gridworlds (Sun, Yeoh, and Koenig 2009). G-FRA*, FRA* and repeated A* with consistent h-values find shortest paths but the runtime of FRA* tends to be smaller than the one of G-FRA* and the runtime of G-FRA* tends to be smaller than the one of repeated A*.
- Second, weighted A* (Pohl 1970) performs an A* search with h-values obtained by multiplying consistent h-values with a constant user-provided weight $\epsilon > 1$. It trades off solution quality and efficiency: Weighted A* with weight ϵ finds an ϵ-suboptimal path (a path that is at most a factor of ϵ longer than the shortest path) with a runtime that tends to be the smaller the larger the weight is. For weights larger than one, the runtime of weighted A* tends to be smaller than the one of A* with consistent h-values. However, it is difficult to identify the smallest weight that makes weighted A* run within a user-provided time limit. Anytime Repairing A* (ARA*) (Likhachev, Gordon, and Thrun 2003) avoids this problem by using incremental search to perform a series of weighted A* searches (each one called a repair iteration) with decreasing weights to find paths from the current location of the hunter to the current location of the target with smaller and smaller error bounds on their lengths until it finds a shortest path or the user-provided time limit has been reached.

In this paper, we develop Incremental ARA* (I-ARA*), the first incremental anytime search algorithm for moving-

*This material is based upon research supported by NSF (while Sven Koenig was serving at NSF), ARL/ARO under contract/grant number W911NF-08-1-0468, and ONR under contract/grant number N00014-09-1-1031. William Yeoh is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

[1] If a single-shot search algorithm X is used repeatedly for every search of the hunter, we refer to it as repeated X.



(a) Before Repair Iteration 1 (b) After Repair Iteration 1 (c) Before Repair Iteration 2 (d) After Repair Iteration 2

Figure 1: First ARA* Search

target search in known terrain. I-ARA* operates in the same way as repeated ARA*, except that it also uses incremental search (as used in G-FRA*) to speed up the first (and often the slowest) repair iteration of each search by reusing information from the previous search. I-ARA* operates in the same way as G-FRA*, except that it performs a series of (modified) weighted A* searches (that can result in a suboptimal path) instead of a single (modified) A* search (that results in a shortest path). We provide an error bound on the lengths of the paths found by I-ARA* (by showing that I-ARA* repair iterations with weight ϵ find ϵ-suboptimal paths) and show experimentally in known four-neighbor gridworlds that I-ARA* can be used with smaller time limits than repeated A*, G-FRA*, FRA*, and sometimes repeated ARA*. The hunter tends to make more moves with I-ARA* than repeated A*, G-FRA* or FRA*, which find shortest paths for the hunter, but fewer moves with I-ARA* than repeated ARA* with the same time limits and the same schedule for decreasing the weights, which finds suboptimal paths for the hunter like I-ARA*. Also, the error bounds on the lengths of the paths of the hunter tend to be smaller with I-ARA* than repeated ARA* because I-ARA* is often able to decrease the weights more than repeated ARA* within the time limits.

Notation

Although I-ARA* can operate on arbitrary graphs, we describe its operation on four-neighbor gridworlds with blocked and unblocked cells for ease of description. S is the finite set of unblocked cells, $s_{start} \in S$ is the current cell of the hunter and the start cell of each search, and $s_{goal} \in S$ is the current cell of the target and the goal cell of each search. *Neighbor*$(s) \subseteq S$ is the set of unblocked neighbors of cell $s \in S$. *parent*$(s) \in$ *Neighbor*(s) is the parent of cell $s \in S$ in the search tree, where a search tree contains exactly s_{start} and all cells with parents, that is, whose parents are different from *NULL*. $c(s, s')$ is the length of a shortest path from cell $s \in S$ to cell $s' \in S$ (measured in the number of moves needed to move along the path).

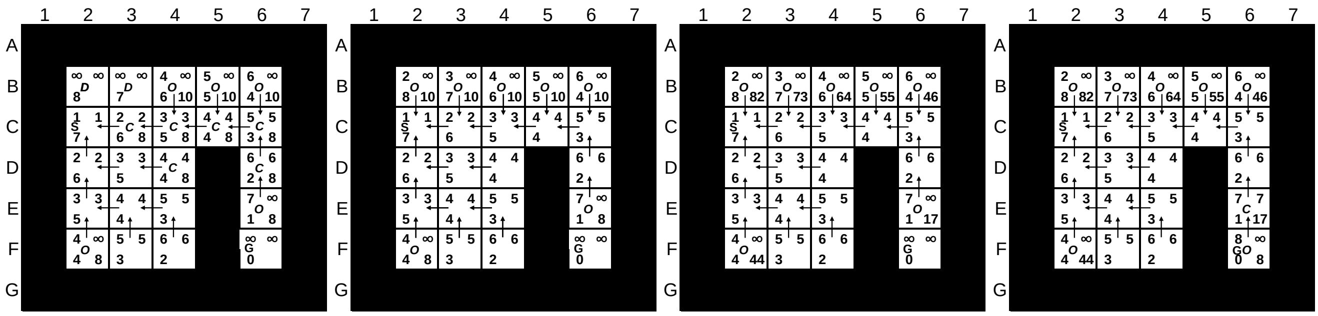
Existing Research: Repeated ARA*

The hunter always knows the gridworld and the current cells of both itself and the target. The hunter and target alternate moves (but do not need to move). The hunter can always move from s_{start} to any of its neighboring unblocked cells. I-ARA* builds on Anytime Repairing A* (ARA*) (Likhachev, Gordon, and Thrun 2003). I-ARA* and repeated ARA* operate as follows until the hunter occupies the same cell as the target: They perform a series of weighted A* searches from s_{start} to s_{goal} with decreasing weights to find paths with smaller and smaller error bounds on their lengths until they find a shortest path or the user-provided time limit has been reached. (If they find that no path from s_{start} to s_{goal} exists, then they exit unsuccessfully since the hunter will never be able to occupy the same cell as the target.) Then, they move the hunter along the path. If the target moves off the path, then they repeat the process.

We explain repeated ARA* first. Repeated ARA* maintains several values for each cell $s \in S$: (1) The g-value $g(s)$ is the length of the shortest path from s_{start} to s found so far. Initially, it is infinity. (2) The v-value $v(s)$ is the g-value at the time of the last expansion of cell s. Initially, it is infinity. We call a cell s *locally consistent* if $v(s) = g(s)$ and *locally inconsistent* otherwise. (3) The h-value $h(s, s')$ is a user-provided approximation of the length of a shortest path from s to s'. The h-values have to be consistent, that is, obey the triangle inequality (Pearl 1985). We use the Manhattan distances as consistent h-values throughout the paper. (4) The f-value $f(s) = g(s) + \epsilon \times h(s, s_{goal})$ is an approximation of the length of an ϵ-suboptimal path from s_{start} via s to s_{goal}, where ϵ is the weight of the current repair iteration.

We call each weighted A* search a *repair iteration*, and the series of weighted A* searches between two moves of the hunter an *ARA* search*. Repeated ARA* proceeds as follows, using the sets *OPEN*, *CLOSED* and *INCONS*:[2] Repeated ARA* first sets $g(s_{start})$ to zero, *OPEN* to contain only s_{start}, and *CLOSED* and *INCONS* to the empty set. Repeated ARA* then starts the first repair iteration by setting the weight of the current repair iteration ϵ to a user-provided initial weight ϵ_{max} (to find a path from s_{start} to s_{goal} with a large error bound on its length quickly) and then repeats the

[2] *OPEN* is the set of all locally inconsistent cells that have not yet been expanded in the current repair iteration, *CLOSED* is the set of all locally consistent cells that have been expanded in the current repair iteration, and *INCONS* is the set of all locally inconsistent cells that have been expanded in the current repair iteration.



(a) Before Repair Iteration 1 (After Step 2) (b) Before Repair Iteration 1 (After Step 3) (c) Before Repair Iteration 1 (After Step 4) (d) After Repair Iteration 1

Figure 2: Second I-ARA* Search

following procedure: Repeated ARA* deletes a cell s with the smallest f-value from *OPEN*, inserts it into *CLOSED* and *expands* it by setting $v(s)$ to $g(s)$ and performing the following operations for each neighbor $s' \in Neighbor(s)$ with $g(s') > v(s) + c(s, s')$: Repeated ARA* sets $g(s')$ to $v(s) + c(s, s')$ and *parent*(s') to s. If s' is neither in *OPEN*, *CLOSED* nor *INCONS*, then repeated ARA* *generates* it by inserting it into *OPEN*. If s' is in *CLOSED*, then repeated ARA* re-generates it by moving it to *INCONS*. Repeated ARA* terminates the current repair iteration when *OPEN* is empty (which indicates that no path exists from s_{start} to s_{goal}) or when the f-value of s_{goal} is no larger than the smallest f-value of any cell in *OPEN* (which indicates that repeated ARA* found an ϵ-suboptimal path from s_{start} to s_{goal}, which can be traced in reverse by following the parents from s_{goal} to s_{start}). If $\epsilon > 1$ and the user-provided time limit has not yet been reached, then repeated ARA* performs the following operations before the next repair iteration: Repeated ARA* decreases ϵ by a user-provided constant δ_ϵ (to find a path with a smaller error bound on its length), sets *OPEN* to the union of *OPEN* and *INCONS*, and sets *CLOSED* and *INCONS* to the empty sets. Repeated ARA* then starts the next repair iteration. It thus improves upon repeated weighted A*, which starts each repair iteration with *OPEN* containing only s_{start} and thus does not reuse information from the previous repair iteration.

Figures 1(a-d) show a gridworld of size 7×7 that we use as running example throughout the paper. $B2$ is s_{start} (marked "S"), and $E6$ is s_{goal} (marked "G"). Cells in *OPEN* are marked "*O*" in the center, cells in *CLOSED* are marked "*C*" in the center, and cells in *INCONS* are marked "*I*" in the center. Arrows point from cells to their parents. We show the g-, v-, h- and f-values of each cell in its top-left, top-right, bottom-left and bottom-right corners, respectively. The f-values are shown only for cells in *OPEN*, *CLOSED* or *INCONS*. Ties among cells with the same f-values are broken in favor of cells with larger g-values because this typically results in smaller runtimes per repair iteration. Figures 1(a) and 1(b) show the cells before and after the first repair iteration, respectively. The first repair iteration uses $\epsilon = 10$, expands cells in the order $B2$, $C2$, $D2$, $E2$, $E3$, $E4$, $F4$, $D4$, $D3$, $F3$, $C4$, $C5$, $C6$ and $D6$, and returns a suboptimal path from $B2$ to $E6$ of length 9. Note that it expands $D4$ with $g(D4) = 6$ and later $D3$ with $g(D3) = 3$. At this point, it detects that $g(D3) + c(D3, D4) = 3 + 1 < 6 = g(D4)$, re-generates $D4$, and inserts it into *INCONS*. Figures 1(c) and 1(d) show the cells before and after the second repair iteration, respectively. The second repair iteration uses $\epsilon = 1$ and returns a shortest path of length 7.

New Research: I-ARA*

I-ARA* operates in the same way as repeated ARA* but reuses part of the search tree at the end of the last repair iteration of the previous I-ARA* search as the search tree in the beginning of the first repair iteration of the current I-ARA* search, namely the subtree that is rooted in s_{start}. Figure 3 shows the pseudocode of I-ARA*. The first I-ARA* search is an ARA* search. Thus, Figure 1 shows the first I-ARA* search. Figure 2 shows the second I-ARA* search after the hunter moved from $B2$ to $C2$ and the target moved from $E6$ to $F6$. The second I-ARA* search proceeds as follows, using the additional set *DELETED*:[3]

Step 1 (Making s_{start} Locally Consistent) s_{start} never needs to be re-expanded since the search tree trivially contains an ϵ-suboptimal path from s_{start} to itself (namely, the empty path). To avoid unnecessary re-expansions of s_{start} and potentially many of its descendants in the search tree, I-ARA* makes s_{start} locally consistent if it is locally inconsistent, by setting its g-value to its v-value. I-ARA* then deletes s_{start} from *OPEN* and *INCONS*. In our example, I-ARA* skips Step 1 since $C2$ is locally consistent already.

Step 2 (Deleting Cells from the Previous Search Tree) If s_{start} is different from the previous cell of the hunter *previous_*s_{start}, I-ARA* sets the parent of s_{start} to *NULL* and deletes all cells from the previous search tree that are not in the subtree rooted in s_{start} by setting their g-values and v-values to infinity, setting their parents to *NULL* and deleting them from *OPEN* and *INCONS*. All cells deleted from the previous search tree are inserted into *DELETED*. In our example, I-ARA* deletes $B2$ and $B3$ and inserts them into

[3] *DELETED* is the set of all cells that have been deleted from the search tree in the current repair iteration.

DELETED. All cells in *DELETED* are marked "D" in the center, see Figure 2(a).

Step 3 (Completing OPEN) *OPEN* is still incomplete since it is missing all cells in *DELETED* that are neighbors of at least one cell with a finite v-value and missing all cells in *INCONS*. Therefore, I-ARA* iterates through all cells in *DELETED* and performs the following check: If a cell s' in *DELETED* is a neighbor of at least one cell with a finite v-value, I-ARA* sets $g(s')$ to $v(s) + c(s, s')$ and $\textit{parent}(s')$ to s, where s is $\arg\min_{s'' \in \textit{Neighbor}(s')}(v(s'') + c(s'', s'))$, and inserts s' into *OPEN*. In our example, I-ARA* inserts $B2$ and $B3$ into *OPEN*. It then inserts all cells in *INCONS* into *OPEN* and sets *CLOSED*, *INCONS* and *DELETED* to the empty sets, see Figure 2(b).

Step 4 (Updating Weight ϵ) If the f-value of s_{goal} is no larger than the smallest f-value of all cells in *OPEN*, then the search tree already contains an ϵ-suboptimal path from s_{start} to s_{goal}. Thus, I-ARA* decreases the weight ϵ by the user-provided constant δ_ϵ by setting ϵ to $\max(1, \epsilon - \delta_\epsilon)$ to find a path with a smaller error bound on its length. Otherwise, I-ARA* has yet to find an ϵ-suboptimal path. Thus, it sets weight ϵ to the user-provided initial weight ϵ_{max} to find a path with a large error bound on its length quickly. In our example, the second case holds and I-ARA* sets the weight to $\epsilon_{max} = 10$, see Figure 2(c).

Step 5 (Starting Repair Iterations) I-ARA* starts the next repair iteration of the current I-ARA* search. In our example, this repair iteration with weight 10 expands only $E6$, see Figure 2(d). In contrast, it would expand $C2$, $C3$, $C4$, $C5$, $C6$, $D6$ and $E6$ without re-using part of the search tree at the end of the last repair iteration of the previous I-ARA* search.

```
function ImprovePath()
while g(s_goal) + ε × h(s_goal, s_goal) > min_{s∈OPEN}(g(s) + ε × h(s, s_goal))
    move s ∈ OPEN with the smallest g(s) + ε × h(s, s_goal) from OPEN to CLOSED;
    v(s) := g(s);
    forall s' ∈ Neighbor(s)
        if g(s') > v(s) + c(s, s')
            g(s') := v(s) + c(s, s');
            parent(s') := s;
            if s' ∉ CLOSED
                if s' ∉ OPEN AND s' ∉ INCONS
                    insert s' into OPEN;
            else
                move s' from CLOSED to INCONS;
if g(s_goal) = ∞
    return false;
else
    return true;

function ComputePath()
repeat
    return false if ImprovePath() = false;
    return true if ε = 1 OR the time limit has been reached;
    OPEN := OPEN ∪ INCONS;
    CLOSED := INCONS := ∅;
    ε := max(1, ε − δ_ε);

procedure Step1()
if g(s_start) ≠ v(s_start)
    g(s_start) := v(s_start);
    delete s_start from INCONS if s_start ∈ INCONS;
    delete s_start from OPEN if s_start ∈ OPEN;

procedure Step2()
if s_start ≠ previous_s_start
    parent(s_start) := NULL;
    forall s in the search tree rooted at previous_s_start but not in the subtree rooted at s_start
        v(s) := g(s) := ∞;
        parent(s) := NULL;
        delete s from INCONS if s ∈ INCONS;
        delete s from OPEN if s ∈ OPEN;
        insert s into DELETED;

procedure Step3()
forall s ∈ DELETED
    forall s' ∈ Neighbor(s)
        if g(s) > v(s') + c(s', s)
            g(s) := v(s') + c(s', s);
            parent(s) := s';
    if g(s) ≠ ∞
        insert s into OPEN;
OPEN := OPEN ∪ INCONS;
CLOSED := INCONS := DELETED := ∅;

procedure Step4()
if g(s_goal) + ε × h(s_goal, s_goal) > min_{s∈OPEN}(g(s) + ε × h(s, s_goal))
    ε := ε_max;
else
    ε := max(1, ε − δ_ε);

function Main()
forall s ∈ S
    v(s) := g(s) := ∞;
    parent(s) := NULL;
ε := ε_max;
s_start := the current cell of the hunter;
s_goal := the current cell of the target;
OPEN := CLOSED := INCONS := DELETED := ∅;
g(s_start) := 0;
insert s_start into OPEN;
while s_start ≠ s_goal
    return false if ComputePath() = false; /* Step 5 */
    previous_s_start := s_start;
    identify a path from s_start to s_goal using the parents;
    while the target has not been caught yet AND is still on the path from s_start to s_goal
        the hunter follows the path from s_start to s_goal;
    return true if the target has been caught;
    s_start := the current cell of the hunter;
    s_goal := the current cell of the target;
    Step1(); /* Step 1 */
    Step2(); /* Step 2 */
    Step3(); /* Step 3 */
    Step4(); /* Step 4 */
return true;
```

Figure 3: I-ARA*

Correctness of I-ARA*

Due to space constraints, we are only able to sketch the correctness proof of I-ARA*, that is, that ImprovePath() with weight ϵ finds an ϵ-suboptimal path from s_{start} to s_{goal} if one exists and reports that no path exists otherwise. We start with a useful lemma.

Lemma 1. *For all cells $s \in S$, $g(s) \leq v(s)$.*

Proof. Initially, $g(s) = v(s) = \infty$. Whenever $g(s)$ or $v(s)$ changes, they are both set to infinity, $v(s)$ is set to $g(s)$, or $g(s)$ is reduced. The lemma follows from these observations. □

Corollary 15 in (Likhachev 2005) can be adapted to prove that $g(s_{goal}) - g(s_{start}) \leq \epsilon \times c(s_{start}, s_{goal})$ when ImprovePath() with weight ϵ terminates if (A1) *CLOSED* is the empty set, (A2) *OPEN* is the set of all locally inconsistent cells, and (A3) for all cells $s \in S \setminus \{s_{start}\}$, $g(s) = v(\textit{parent}(s)) + c(\textit{parent}(s), s) = \min_{s' \in \textit{Neighbor}(s)}(v(s') + c(s', s))$ when ImprovePath() is called. [4] In the following,

[4]The version of ImprovePath() used in (Likhachev 2005) does not use *INCONS* and requires that $g(s_{start}) = 0$. The version of ImprovePath() used by I-ARA* uses *INCONS* only for bookkeeping and the proof in (Likhachev 2005) applies to it if one uses $g(s) - g(s_{start})$ instead of $g(s)$ for all cells $s \in S$.

we first prove the premise by proving A1, A2 and A3 separately. We then use the conclusion for the correctness proof of I-ARA*.

We now prove that A1 holds whenever ImprovePath() is called. This is easy to see since both *CLOSED* and *INCONS* are always set to the empty set before and between calls to ImprovePath() (Lines 23, 48 and 61).

We now prove that A2 holds whenever ImprovePath() is called. We start with a useful lemma.

Lemma 2. *OPEN* $\cup$ *INCONS is the set of all locally inconsistent cells when ImprovePath() terminates if A2 holds when ImprovePath() is called.*

Proof. We prove the lemma by induction on the number of times ImprovePath() executes Line 2: The conclusion of the lemma holds for the first execution of Line 2 since A2 holds due to the premise of the lemma and *CLOSED* and *INCONS* are the empty set. The conclusion of the lemma also holds for each subsequent execution of Line 2: ImprovePath() sets the v-value of a cell to its g-value on Line 4 and thus makes it locally consistent, and it might reduce the g-value of a cell on Line 7, thus making it locally inconsistent according to Lemma 1. Whenever a cell is made locally consistent, it is deleted from *OPEN* $\cup$ *INCONS* (Line 3). Whenever a cell is made locally inconsistent, it is inserted into *OPEN* $\cup$ *INCONS* (Lines 11 and 13). These are the only ways how cells can be inserted into or deleted from *OPEN* $\cup$ *INCONS*. □

Next, we prove that A2 holds whenever ImprovePath() is called if A2 holds whenever ComputePath() is called. We prove the statement by induction on the number of times ComputePath() calls ImprovePath(). A2 holds when ComputePath() calls ImprovePath() for the first time since A2 holds whenever ComputePath() is called. A2 also holds when ComputePath() calls ImprovePath() each subsequent time: *OPEN* $\cup$ *INCONS* is the set of all locally inconsistent cells when ImprovePath() terminates according to Lemma 2. ComputePath() then restores A2 since it inserts all cells in *INCONS* into *OPEN* (Line 22).

Finally, we prove that A2 holds whenever ComputePath() is called. We prove the statement by induction on the number of times Main() calls ComputePath(). A2 holds when Main() calls ComputePath() for the first time since *OPEN* contains the only locally inconsistent cell, namely s_{start}. A2 also holds when Main() calls ComputePath() each subsequent time: *OPEN* $\cup$ *INCONS* is the set of all locally inconsistent cells when ComputePath() terminates since it is the set of all locally inconsistent cells when ImprovePath() terminates according to Lemma 2. Main() sets the g-value of a cell to its v-value on Lines 27 and 34 and thus makes it locally consistent, and it reduces the g-value of a cell on Line 43 and thus makes it locally inconsistent according to Lemma 1. Whenever a cell is made locally consistent, it is deleted from *OPEN* $\cup$ *INCONS* (Lines 28-29 and 36-37). Whenever a cell is made locally inconsistent, it is inserted into *OPEN* $\cup$ *INCONS* (Line 46). These are the only ways how cells can be inserted into or deleted from *OPEN* $\cup$ *INCONS*. Main() then restores A2 since it inserts all cells in *INCONS* into *OPEN* (Line 47).

We have proven that A2 holds whenever ImprovePath() is called if A2 holds whenever ComputePath() is called. We have also proven that A2 holds whenever ComputePath() is called. Consequently, we have proven that A2 holds whenever ImprovePath() is called.

We now prove that A3 holds whenever ImprovePath() is called. We first prove that A3 holds whenever ImprovePath() terminates if A3 holds whenever ImprovePath() is called. We prove the statement by induction on the number of times ImprovePath() executes Line 2, taking into account Lines 4-8. We then use this statement to prove that A3 holds whenever ImprovePath() is called if A3 holds whenever ComputePath() is called. We prove this statement, in turn, by induction on the number of times ComputePath() calls ImprovePath(), taking into account that ImprovePath() but not the rest of ComputePath() can change the g- and v-values.

Finally, we prove that A3 holds whenever ComputePath() is called. We prove the statement by induction on the number of times Main() calls ComputePath(). A3 holds when Main() calls ComputePath() for the first time due to the initialization of the g-values, parents, and v-values of all cells (Lines 56-57 and 62). A3 also holds when Main() calls ComputePath() each subsequent time: A3 holds whenever ComputePath() terminates since it holds whenever ImprovePath() terminates. We prove that A3 also holds after Main() executes Line 76 if it holds before Main() executes Line 66: If the hunter does not move and its current cell s_{start} is thus equal to its previous cell *previous_s*$_{start}$, then the g-values, parents and v-values of no cell changes when Main() executes Lines 66-76 and A3 continues to hold. If the hunter moves, we partition all cells into three sets: S_{new} is the set of all cells in the subtree of the search tree rooted in s_{start}, S_{old} is the set of all cells in the search tree (rooted in *previous_s*$_{start}$) but not in S_{new}, and S_{rest} is the set of all remaining cells. Consider any cell $s \in S \setminus \{s_{start}\}$.

- If $s \in S_{new}$, then *parent*$(s) \in S_{new}$ before Main() executes Line 66 since S_{new} is the set of all cells in a subtree of the search tree. Neither the g-value of s, the parent of s nor the v-value of *parent*(s) changes when Main() executes Lines 66-76. $\min_{s' \in Neighbor(s)}(v(s') + c(s', s))$ does not decrease since the v-value of no cell decreases. Consequently, s does not violate A3 after Main() executes Line 76.
- If $s \in S_{rest}$, then the g-value of s is infinity before Main() executes Line 66 since S_{rest} is the set of cells that are not in the search tree and whose g-values are thus infinity. For all $s' \in Neighbor(s)$, the v-value of s' is also infinity according to A3. Neither the g-value nor parent of s changes when Main() executes Lines 66-76. $\min_{s' \in Neighbor(s)}(v(s') + c(s', s))$ does not decrease since the v-value of no cell decreases. Consequently, s does not violate A3 after Main() executes Line 76.
- If $s \in S_{old}$, then the g- and v-value of s are set to infinity on Line 34. *DELETED* is the empty set (Lines 48 and 61) before the cells in S_{old} are inserted into it (Lines 33 and 38) and thus S_{old} is the set of all cells in *DELETED*. The g-value of s is then set to $\min_{s' \in Neighbor(s)}(v(s') + c(s', s))$,

and the parent of s is set to $\arg\min_{s' \in \mathit{Neighbor}(s)}(v(s') + c(s', s))$ (Lines 41-44). After the g-value and parent of s have been set the last time on Lines 43-44, then the g-value of s, the parent of s and the v-value of no cell changes. Consequently, s does not violate A3 after Main() executes Line 76.

We have proven that A3 holds whenever ComputePath() is called. Thus, A3 also holds whenever ImprovePath() is called.

Overall, we have proven that $g(s_{goal}) - g(s_{start}) \leq \epsilon \times c(s_{start}, s_{goal})$ when ImprovePath() with weight ϵ terminates if A1, A2, and A3 hold when ImprovePath() is called. We have also proven that A1, A2, and A3 hold when ImprovePath() is called. Consequently, we have proven that $g(s_{goal}) - g(s_{start}) \leq \epsilon \times c(s_{start}, s_{goal})$ when ImprovePath() with weight ϵ terminates. We now use this result for the correctness proof of I-ARA*.

Theorem 1. *ImprovePath() returns true if a path from s_{start} to s_{goal} exists and false otherwise.*

Proof. ImprovePath() always expands a cell in OPEN and then does not reinsert it into OPEN. Since S is finite, ImprovePath() eventually terminates. We have shown that $g(s_{goal}) - g(s_{start}) \leq \epsilon \times c(s_{start}, s_{goal})$ when ImprovePath() with weight ϵ terminates. If a path from s_{start} to s_{goal} exists, then $c(s_{start}, s_{goal})$ is finite, which means that $g(s_{goal})$ is also finite (since $g(s_{start})$ is finite) and ImprovePath() returns true (Lines 14 and 17). If no such path exists, then $g(s_{goal})$ is infinity (which can be proven by contradiction using Lemma 1 and A3) and ImprovePath() returns false (Lines 14-15). □

Theorem 2. *When ImprovePath() with weight ϵ returns true, an ϵ-suboptimal path from s_{start} to s_{goal} of length at most $g(s_{goal}) - g(s_{start})$ can be traced in reverse by following the parents from s_{goal} to s_{start}.*

Proof. We have shown that $g(s_{goal}) - g(s_{start}) \leq \epsilon \times c(s_{start}, s_{goal})$ and A3 holds whenever ImprovePath() with weight ϵ terminates. A3 and Lemma 1 ensure that the g-values of the cells are strictly monotonically decreasing when following the parents from s_{goal}. One eventually reaches s_{start} since all cells with finite g-values, except for s_{start}, have parents. Now consider the resulting path $(s_0 = s_{start}, \ldots, s_k = s_{goal})$. We prove that the path $(s_0 = s_{start}, \ldots, s_i)$ is of length at most $g(s_i) - g(s_{start})$ by induction on i: The path $(s_0 = s_{start}, \ldots, s_0)$ is trivially of length at most $g(s_i) - g(s_{start}) = 0$. Now assume that the path $(s_0 = s_{start}, \ldots, s_i)$ for $i > 0$ is of length at most $g(s_i) - g(s_{start})$. $g(s_{i+1}) = v(s_i) + c(s_i, s_{i+1}) \geq g(s_i) + c(s_i, s_{i+1})$ and thus $g(s_{i+1}) - g(s_i) \geq c(s_i, s_{i+1})$ according to A3 and Lemma 1 since $\mathit{parent}(s_{i+1}) = s_i$. The length of path $(s_0 = s_{start}, \ldots, s_{i+1})$ is equal to the sum of the length of path $(s_0 = s_{start}, \ldots, s_i)$, which is at most $g(s_i) - g(s_{start})$, and $c(s_i, s_{i+1})$, which is at most $g(s_{i+1}) - g(s_i)$. □

Experimental Results

Moving-target search algorithms in known terrain can be classified into off-line and on-line algorithms. Off-line moving-target search algorithms, such as Reverse Minimax A* (Moldenhauer and Sturtevant 2009), determine the optimal strategy for the hunter once by taking into account all strategies for the target, which tends to make their searches too slow on larger maps. On-line search algorithms react to the actual moves of the target and thus need to search repeatedly. Some on-line search algorithms use the strategy for the hunter to find a path from s_{start} to s_{goal} and move along it. If the target moves off the path, then they repeat the process. Real-time moving-target search algorithms, such as MTS (Ishida and Korf 1991), find a prefix of a path in constant time, which tends to result in a large number of moves for the hunter until it catches the target or needs a large amount of memory (Bulitko et al. 2007b). We therefore study moving-target search algorithms that find a complete path, for example, via repeated A* or weighted A* searches. Incremental search has been used to speed up both kinds of moving-target search algorithms. For example, USP (Edelkamp 1998) is an incremental version of a version of MTS (Sasaki, Chimura, and Tokoro 1995), and G-FRA* (Sun, Yeoh, and Koenig 2010) is an incremental version of repeated A*. G-FRA* finds shortest paths for the hunter and is the currently fastest such incremental moving-target search algorithm on known graphs. FRA* optimizes it for gridworlds (Sun, Yeoh, and Koenig 2009).

We therefore compare I-ARA* experimentally to repeated A*, G-FRA*, FRA* and repeated ARA* (Likhachev, Gordon, and Thrun 2003). It is important to realize that experimental results, such as the runtimes of the search algorithms, depend on a variety of factors, including implementation details (such as the data structures, tie-breaking strategies, and coding tricks used) and experimental setups (such as whether the gridworlds are four-neighbor or eight-neighbor gridworlds). We do not know of any better method for evaluating search algorithms than to implement them as best as possible, publish their runtimes, and let other researchers experiment with their own and thus potentially different implementations and experimental setups. For fairness, we use comparable implementations. For example, all search algorithms use binary heaps as priority queues and replan when the target moves off the path. We perform our experiments in (1) four-neighbor gridworlds of size $1,000 \times 1,000$ with 25 percent randomly blocked cells and (2) a four-neighbor video game map of size 676×676 adapted from Warcraft III (courtesy of Nathan Sturtevant). We average over 100 test cases (corresponding to 100 different gridworlds but the same video game map) with randomly selected unblocked cells for the hunter and target for both kinds of maps, with the restriction that a path exists between both cells. The target always follows a shortest path from its current cell to a randomly selected unblocked cell and repeats the process once it reaches that cell. It skips every tenth move, which allows the hunter to catch it in all cases.

Experiment 1

We first let I-ARA* and repeated ARA* perform only one repair iteration per search with a user-provided weight, which we vary from 1.0 to 5.0. Thus, repeated ARA* reduces to repeated weighted A*. Table 1 reports one measure for the solution quality of the search algorithms, namely the

	Gridworlds				Game Map			
	moves per test case	expansions per search	average runtime per search	largest runtime of any search	moves per test case	expansions per search	average runtime per search	largest runtime of any search
A*	746 [1.00]	13801.1 (95.0)	6,768	28,114	546 [1.00]	9,079.4 (69.1)	3,789	12,169
FRA*	747 [1.00]	283.5 (14.4)	427	21,417	545 [1.00]	321.1 (26.4)	225	10,196
G-FRA*	746 [1.00]	646.5 (18.1)	592	21,630	545 [1.00]	673.9 (22.3)	340	10,302
ARA* ($\epsilon = 1.0$)	746 [1.00]	13,800.5 (95.0)	7,221	29,500	546 [1.00]	9,082.2 (66.4)	4,071	12,591
ARA* ($\epsilon = 1.1$)	772 [1.03]	6,510.7 (42.2)	3,058	7,416	557 [1.02]	6,288.3 (47.2)	2,675	7,512
ARA* ($\epsilon = 1.2$)	800 [1.07]	3,458.4 (20.6)	1,602	4,669	574 [1.05]	5,021.0 (36.7)	2,065	5,767
ARA* ($\epsilon = 1.3$)	824 [1.10]	1,641.5 (7.7)	732	2,133	590 [1.08]	4,515.0 (31.8)	1,825	4,664
ARA* ($\epsilon = 1.4$)	840 [1.13]	1,087.2 (3.8)	492	1,156	602 [1.10]	4,167.0 (28.9)	1,665	4,323
ARA* ($\epsilon = 1.5$)	851 [1.14]	919.2 (3.0)	430	968	616 [1.13]	3,785.6 (24.0)	1,472	3,699
ARA* ($\epsilon = 2.0$)	882 [1.18]	733.4 (2.3)	370	832	659 [1.21]	2,855.9 (16.2)	1,037	2,583
ARA* ($\epsilon = 3.0$)	919 [1.23]	664.7 (2.0)	351	802	701 [1.28]	2,121.8 (10.6)	697	1,826
ARA* ($\epsilon = 4.0$)	919 [1.23]	656.0 (2.0)	350	805	805 [1.47]	1,731.1 (8.2)	559	1,607
ARA* ($\epsilon = 5.0$)	929 [1.25]	645.3 (2.0)	349	799	915 [1.68]	1,864.0 (7.8)	599	1,486
I-ARA* ($\epsilon = 1.0$)	746 [1.00]	646.5 (18.1)	619	23,485	545 [1.00]	673.9 (22.3)	368	11,681
I-ARA* ($\epsilon = 1.1$)	760 [1.02]	163.2 (6.2)	174	5,939	549 [1.01]	310.4 (20.3)	190	6,298
I-ARA* ($\epsilon = 1.2$)	778 [1.04]	62.3 (3.1)	128	3,303	555 [1.02]	198.2 (19.8)	154	4,944
I-ARA* ($\epsilon = 1.3$)	799 [1.07]	14.7 (0.9)	101	1,375	568 [1.04]	140.4 (18.8)	127	4,065
I-ARA* ($\epsilon = 1.4$)	814 [1.09]	5.9 (0.4)	92	975	578 [1.06]	115.4 (18.1)	116	3,795
I-ARA* ($\epsilon = 1.5$)	828 [1.11]	4.1 (0.3)	93	877	588 [1.08]	110.5 (18.4)	107	3,249
I-ARA* ($\epsilon = 2.0$)	858 [1.15]	3.0 (0.3)	101	796	630 [1.15]	59.3 (17.1)	79	2,204
I-ARA* ($\epsilon = 3.0$)	885 [1.19]	2.6 (0.2)	106	761	672 [1.23]	44.8 (15.7)	71	1,633
I-ARA* ($\epsilon = 4.0$)	890 [1.19]	2.5 (0.2)	105	772	683 [1.25]	42.1 (15.3)	68	1,463
I-ARA* ($\epsilon = 5.0$)	898 [1.20]	2.4 (0.2)	106	759	698 [1.28]	40.0 (14.8)	65	1,352

Table 1: User-Provided Error Bound ϵ on the Length of the Path of each Search

	Gridworlds						Game Map					
	moves per test case	expansions per search	average runtime per search	largest runtime of any repair iteration	searches exceeding time limit t	repair iterations per search	moves per test case	expansions per search	average runtime per search	largest runtime of any repair iteration	searches exceeding time limit t	repair iterations per search
FRA* ($t = 100\mu s$)	747 [1.00]	283.5 (14.4)	427	-	17.6% {17.4%}	-	545 [1.00]	321.1 (26.4)	225	-	19.4% {19.2%}	-
FRA* ($t = 200\mu s$)	747 [1.00]	283.5 (14.4)	426	-	16.1% {15.9%}	-	545 [1.00]	321.1 (26.4)	225	-	15.5% {15.3%}	-
FRA* ($t = 300\mu s$)	747 [1.00]	283.5 (14.4)	426	-	14.9% {14.7%}	-	545 [1.00]	321.1 (26.4)	225	-	13.4% {13.1%}	-
FRA* ($t = 400\mu s$)	747 [1.00]	283.5 (14.4)	427	-	13.9% {13.7%}	-	545 [1.00]	321.1 (26.4)	224	-	11.5% {11.2%}	-
FRA* ($t = 500\mu s$)	747 [1.00]	283.5 (14.4)	426	-	12.8% {12.6%}	-	545 [1.00]	321.1 (26.4)	226	-	10.0% {9.8%}	-
FRA* ($t = 1,000\mu s$)	747 [1.00]	283.5 (14.4)	427	-	9.0% {8.9%}	-	545 [1.00]	321.1 (26.4)	224	-	5.7% {5.5%}	-
FRA* ($t = 2,000\mu s$)	747 [1.00]	283.5 (14.4)	427	-	5.4% {5.2%}	-	545 [1.00]	321.1 (26.4)	226	-	2.2% {2.0%}	-
FRA* ($t = 3,000\mu s$)	747 [1.00]	283.5 (14.4)	427	-	3.6% {3.5%}	-	545 [1.00]	321.1 (26.4)	224	-	1.3% {1.1%}	-
ARA* ($t = 100\mu s$)	877 [1.18]	741.4 (1.4)	382	827	78.8% {78.8%}	2.40	656 [1.20]	2,874.1 (7.8)	1,062	2,631	79.0% {79.0%}	2.50
ARA* ($t = 200\mu s$)	868 [1.16]	760.2 (1.1)	408	838	65.2% {65.1%}	3.29	652 [1.19]	2,897.2 (6.1)	1,087	2,636	69.1% {69.1%}	3.29
ARA* ($t = 300\mu s$)	862 [1.16]	787.2 (0.9)	444	835	51.9% {51.8%}	4.06	649 [1.19]	2,919.4 (5.3)	1,103	2,651	60.5% {60.4%}	3.87
ARA* ($t = 400\mu s$)	857 [1.15]	818.9 (0.8)	497	874	39.6% {39.5%}	4.80	647 [1.18]	2,952.2 (4.8)	1,146	2,644	54.2% {54.2%}	4.30
ARA* ($t = 500\mu s$)	852 [1.14]	858.5 (0.7)	557	836	28.4% {28.3%}	5.53	645 [1.18]	2,983.5 (4.4)	1,182	2,634	49.0% {48.9%}	4.69
ARA* ($t = 1,000\mu s$)	836 [1.12]	1,140.2 (0.5)	944	844	4.1% {4.1%}	7.82	639 [1.17]	3,183.5 (3.5)	1,388	2,637	34.3% {34.3%}	5.97
ARA* ($t = 2,000\mu s$)	818 [1.10]	1,921.3 (0.6)	1,746	837	0.0% {0.0%}	9.17	627 [1.15]	3,736.4 (3.0)	1,883	2,680	17.9% {17.9%}	7.28
ARA* ($t = 3,000\mu s$)	809 [1.08]	2,698.1 (0.8)	2,473	822	0.0% {0.0%}	9.59	620 [1.14]	4,336.3 (2.8)	2,418	2,636	11.2% {11.1%}	7.91
I-ARA* ($t = 100\mu s$)	847 [1.14]	5.3 (0.1)	158	742	26.8% {26.6%}	4.07	623 [1.14]	62.8 (6.1)	139	2,190	24.8% {24.6%}	4.08
I-ARA* ($t = 200\mu s$)	827 [1.11]	12.2 (0.1)	246	746	6.8% {6.6%}	5.47	613 [1.12]	76.1 (4.2)	221	2,187	6.1% {5.8%}	5.47
I-ARA* ($t = 300\mu s$)	818 [1.10]	22.4 (0.1)	325	751	2.5% {2.3%}	6.33	605 [1.11]	94.6 (3.4)	297	2,189	2.3% {2.1%}	6.33
I-ARA* ($t = 400\mu s$)	809 [1.08]	34.8 (0.1)	402	748	0.7% {0.5%}	7.02	601 [1.10]	119.5 (3.0)	366	2,175	1.2% {1.0%}	6.99
I-ARA* ($t = 500\mu s$)	804 [1.08]	49.7 (0.1)	473	740	0.5% {0.4%}	7.55	596 [1.09]	148.4 (2.8)	429	2,174	0.7% {0.4%}	7.51
I-ARA* ($t = 1,000\mu s$)	785 [1.05]	136.4 (0.1)	762	750	0.2% {0.1%}	9.07	577 [1.06]	309.0 (2.4)	687	2,192	0.3% {0.1%}	8.87
I-ARA* ($t = 2,000\mu s$)	766 [1.03]	299.8 (0.3)	1,076	749	0.2% {0.2%}	9.73	555 [1.02]	534.1 (2.4)	912	2,194	0.3% {0.2%}	9.60
I-ARA* ($t = 3,000\mu s$)	757 [1.01]	414.5 (0.5)	1,251	745	0.3% {0.3%}	9.88	550 [1.01]	634.3 (2.5)	980	2,196	0.2% {0.1%}	9.79

Table 2: User-Provided Time Limit t for each Search

number of moves of the hunter until it catches the target. The ratio of this number and the number of moves of the hunter of repeated A* is shown in square brackets. The table reports three measures for the efficiency of the search algorithms, namely the number of expanded cells per search and two runtimes in microseconds on an Intel Xeon 3.20 Ghz PC with 2 GByte of RAM, namely the average runtime per search over all searches until the hunter catches the target and the largest runtime of any search until the hunter catches the target. The trends for the number of expanded cells per search, the average runtime per search and the largest runtime of any search are similar, and we thus refer only to the runtime per search. The standard deviation of the mean for the number of expanded cells per search is shown in parentheses. We notice the following relationships:

- The numbers of moves of all search algorithms tend to be similar when I-ARA* and repeated ARA* use weight one since they all find the same paths (modulo tie breaking).
- The runtime per search of repeated A* tends to be smaller than the one of repeated ARA* with weight one since they operate in the same way and thus tend to have the same number of expanded cells per search but repeated ARA* needs extra bookkeeping operations. The runtime per search of I-ARA* with weight one tends to be smaller than the one of repeated A* since it uses incremental search. The runtime per search of G-FRA* tends to be smaller than the one of I-ARA* with weight one since they operate in the same way and thus tend to have the same number of expanded cells per search but I-ARA* needs extra bookkeeping operations. The runtime per search of FRA* tends to be smaller than the one of G-FRA* since it is optimized for gridworlds.
- The numbers of moves of I-ARA* tends to be smaller than the one of repeated ARA* with the same weight and both increase slowly as the weight increases since they trade-off solution quality and efficiency.

- The runtime per search of I-ARA* tends to be smaller than the one of repeated ARA* with the same weight since it uses incremental search to re-use information from the last repair iteration of the previous search to speed up the first repair iteration of the current search. The runtimes per search of I-ARA* and repeated ARA* with the same weight tend to decrease as the weight increases. The runtime per search of I-ARA* tends to be smaller than the ones of all other tested search algorithms for sufficiently large weights.

The advantage of I-ARA* over G-FRA* and repeated ARA* searches is due to the following reasons:

- The average runtime per search of G-FRA* tends to be smaller than the one of repeated A* but its largest runtime of any search tends to be only slightly smaller. Both the average runtime per search and the largest runtime of any search of I-ARA* with weights larger than one tend to be smaller than the ones of G-FRA* but the largest runtime of any search tends to be much smaller. The reason for this is that the first search is often the one with the largest runtime. The first G-FRA* search is an A* search (and the largest runtime of any search of G-FRA* thus tends to be similar to the one of repeated A*), while the first I-ARA* search is a faster weighted A* search.
- Both the average runtime per search and the largest runtime of any search of I-ARA* tend to be smaller than the ones of repeated ARA* with the same weight but the average runtime per search tends to be much smaller. The reason for this is that the first search is often the one with the largest runtime. The first searches of I-ARA* and repeated ARA* operate in the same way (and the largest runtime of any search of I-ARA* thus tends to be similar to the one of repeated ARA*), while I-ARA* uses incremental search to speed up the subsequent searches.

Experiment 2

We now impose a time limit per search, which we vary from 100 to 3,000 μs. Since all search algorithms have to find a path from s_{start} to s_{goal}, we let FRA* complete its searches independent of the time limit and repeated ARA* and I-ARA* complete their first repair iterations of all searches independent of the time limit. We use $\epsilon_{max} = 2.0$ and $\delta_\epsilon = 0.1$ for repeated ARA* and I-ARA*. Table 2 reports, in addition to some of the measures from Table 1, the largest runtime of any repair iteration per search, the number of repair iterations per search, and the percentage of searches that exceed the time limit. The percentage of searches different from the first search that exceed the time limit is shown in curly braces since the first search is performed before the hunter starts to move and thus might not be subject to the time limit. We notice the following relationships:

- The number of repair iterations per search of I-ARA* tends to be larger than the one of repeated ARA* with the same time limit, as already suggested by the average runtime per search in Experiment 1. Consequently, I-ARA* tends to be able to provide smaller error bounds on the lengths of the paths found than repeated ARA* and its number of moves tends to be smaller since it tends to be able to decrease the weight more. The number of moves of I-ARA* tends to be similar to the one of FRA* for sufficiently large time limits.

	Gridworlds		Game Map	
	Off the Path	Every Move	Off the Path	Every Move
I-ARA* ($t = 100\ \mu s$)	847 [1.14]	829 [1.11]	623 [1.14]	613 [1.12]
I-ARA* ($t = 200\ \mu s$)	827 [1.11]	806 [1.08]	613 [1.12]	599 [1.10]
I-ARA* ($t = 300\ \mu s$)	818 [1.10]	795 [1.07]	605 [1.11]	591 [1.08]
I-ARA* ($t = 400\ \mu s$)	809 [1.08]	786 [1.05]	601 [1.10]	585 [1.07]
I-ARA* ($t = 500\ \mu s$)	804 [1.08]	777 [1.04]	596 [1.09]	579 [1.06]
I-ARA* ($t = 1,000\ \mu s$)	785 [1.05]	764 [1.02]	577 [1.06]	559 [1.02]
I-ARA* ($t = 2,000\ \mu s$)	766 [1.03]	754 [1.01]	555 [1.02]	551 [1.01]
I-ARA* ($t = 3,000\ \mu s$)	757 [1.01]	750 [1.01]	550 [1.01]	548 [1.00]

Table 3: Moves per Test Case

- The percentage of searches of I-ARA* that exceed the time limit tends to be smaller than the one of repeated ARA* since the largest runtime of any repair iteration per search of I-ARA* tends to be smaller than the one of repeated ARA*, as already suggested by the largest runtime of any search in Experiment 1. (The largest runtimes of any repair iteration tend not to depend on the time limit since the first repair iteration per search is often one with the largest runtime.) Large time limits in gridworlds are an exception: Large time limits allow I-ARA* to perform searches with smaller weights, which result in larger search trees, whose preprocessing times can sometimes exceed the time limit.
- The percentage of searches of I-ARA* that exceed the time limit tends to be smaller than the one of FRA*, as already suggested by the largest runtime of any search in Experiment 1. Small time limits are an exception: The time needed to preprocess the search trees is often longer for I-ARA* than FRA* (and can be longer than the time limit for small time limits) since it needs extra bookkeeping operations and it is not optimized for gridworlds.

So far, all search algorithms replanned when the target moved off the path (marked "Off the Path"). However, the hunter might have the time to perform a search before every move (marked "Every Move"). The number of moves of repeated ARA* remains about the same, but the number of moves of I-ARA* decreases slightly since it uses incremental search, see Table 3.

Conclusions

We developed Incremental ARA* (I-ARA*), the first incremental anytime search algorithm for moving-target search in known terrain. We provided an error bound on the lengths of the paths found by I-ARA* and showed experimentally in known four-neighbor gridworlds that I-ARA* can be used with smaller time limits between moves of the hunter than competing state-of-the-art moving-target search algorithms. The hunter tends to make more moves with I-ARA* than repeated A*, G-FRA* or FRA*, which find shortest paths for the hunter, but fewer moves with I-ARA* than repeated ARA*, which finds suboptimal paths for the hunter like I-ARA*. Also, the error bounds on the lengths of the paths of the hunter tend to be smaller with I-ARA* than repeated ARA*.

References

Bulitko, V.; Björnsson, Y.; Luvstrek, M.; Schaeffer, J.; and Sigmundarson, S. 2007a. Dynamic control in path-planning with real-time heuristic search. In *Proc. of ICAPS*, 49–56.

Bulitko, V.; Sturtevant, N.; Lu, J.; and Yau, T. 2007b. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research* 30(1):51–100.

Edelkamp, S. 1998. Updating shortest paths. In *Proc. of ECAI*, 655–659.

Ishida, T., and Korf, R. 1991. Moving target search. In *Proc. of IJCAI*, 204–211.

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

Koenig, S.; Likhachev, M.; and Sun, X. 2007. Speeding up moving-target search. In *Proc. of AAMAS*, 1136–1143.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proc. of NIPS*.

Likhachev, M. 2005. *Search-based Planning for Large Dynamic Environments*. Ph.D. Dissertation, Carnegie Mellon University.

Loh, P., and Prakash, E. 2009. Performance simulations of moving target search algorithms. *International Journal of Computer Games Technology* 2009:1–6.

Moldenhauer, C., and Sturtevant, N. 2009. Optimal solutions for moving target search. In *Proc. of AAMAS*, 1249–1250.

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

Pohl, I. 1970. First results on the effect of error in heuristic search. *Machine Intelligence* 5:219–236.

Sasaki, T.; Chimura, F.; and Tokoro, M. 1995. The trailblazer search with a hierarchical abstract map. In *Proc. of IJCAI*, 259–265.

Sun, X.; Yeoh, W.; and Koenig, S. 2009. Efficient incremental search for moving target search. In *Proc. of IJCAI*, 615–620.

Sun, X.; Yeoh, W.; and Koenig, S. 2010. Generalized Fringe-Retrieving A*: Faster moving target search on state lattices. In *Proc. of AAMAS*, 1081–1087.