Dynamic Fringe-Saving A*

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

ABSTRACT

Fringe-Saving A* is an incremental version of A* that repeatedly finds shortest paths from a fixed start cell to a fixed goal cell in a known gridworld in case the traversability of cells changes over time. It restores the content of the OPEN and CLOSED lists of A* at the point in time when an A* search for the current search problem could deviate from the A* search for the previous search problem. Thus, Fringe-Saving A* reuses the beginning of the previous A* search that is identical to the current A* search. In this paper, we generalize the correctness proof of Fringe-Saving A* to cover the case where the goal cell changes over time in addition to the traversability of cells. We then apply Fringe-Saving A* to the problem of moving an agent along a shortest path from its current cell to a fixed destination cell in a known gridworld, where the shortest path is replanned whenever the traversability of cells changes. Unfortunately, our experimental results show that the resulting Dynamic Fringe-Saving A* algorithm tends to be dominated by either repeated A* searches or D* Lite (a state-of-the-art incremental version of A*).[1]

[1]This paper is an updated version of our original paper. We updated the experimental results after we noticed a problem in the implementation of our experiments: We performed the experiments in gridworlds of size 1000 x 1000. After each move of the agent from its current cell to a neighbor, we intended to block a certain number of randomly chosen unblocked cells and unblock the same number of randomly chosen blocked cells. We had ported code to a different operating system and, in the process, changed the random number generator. The new random number generator used to determine these cells generated random numbers in the range from 0 to 32,767 only, which is too small to choose among all cells. We apologize for this very embarassing programming mistake.

Categories and Subject Descriptors

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

General Terms

Algorithm; Experimentation; Theory

Keywords

A*; D* Lite; Fringe-Saving A*; Incremental Heuristic Search; Path Planning

1. INTRODUCTION

Search is about finding shortest paths, say from a fixed start cell to a fixed goal cell in a known gridworld. One has to search repeatedly if the traversability of cells changes over time. In this case, incremental search algorithms reuse information from previous searches to speed up the current search and hence are often able to find shortest paths much faster than is possible by searching from scratch [4]. There are three different classes of incremental versions of A*. Search algorithms from Class 1 start an A* search at the point where the current A* search deviates from the previous one. An example is Fringe Saving A* (FSA*) [7], which is related to iA* by Peter Yap (unpublished). Search algorithms from Class 2 update the h-values from the previous A* search during the current A* search to make them more informed [1]. An example is Generalized Adaptive A* [8]. Search algorithms from Class 3 update the g-values from the previous A* search during the current A* search to correct them when necessary, which can be interpreted as transforming the A* search tree of the previous A* search to the A* search tree of the current A* search. An example is Lifelong Planning A* (LPA*) [3]. In this paper, we are interested in finding shortest paths for series of similar search problems from a fixed start cell to a goal cell in a known gridworld in case the goal cell changes over time in addition to the traversability of cells. Generalized Adaptive A* from Class 2 can handle such series of similar search problems, and LPA* from Class 3 has been extended to D* Lite [2] to handle them. However, no search algorithm from Class 1 is known to handle them. We therefore show in this paper that FSA* can handle them as well because its restored CLOSED list contains those cells expanded in all previous A* searches whose g-values and parent pointers are still guaranteed to be correct. We then apply it to the problem of moving an agent along a shortest path from its current cell to a fixed destination cell in a known gridworld, where the shortest path is

replanned whenever the traversability of cells changes. Unfortunately, our experimental results show that the resulting Dynamic Fringe-Saving A* algorithm tends to be dominated by either repeated A* searches or D* Lite (a state-of-the-art incremental version of A*).

2. SEARCH PROBLEMS AND NOTATION

Consider a finite gridworld with blocked and unblocked cells. S denotes the set of unblocked cells. $Succ(s) \subseteq S$ denotes the set of unblocked neighbors of cell $s \in S$. The cost of moving from an unblocked cell to an unblocked neighbor is one. The cost of moving between any other pair of cells is ∞. FSA* finds shortest paths from a fixed start cell $s_{start} \in S$ to a fixed goal cell $s_{goal} \in S$ whenever the traversability of cells changes over time, always knowing which cells are currently blocked. We describe A* before we describe FSA* since FSA* uses A*.

3. A*

A* is arguably the most popular search algorithm in artificial intelligence. We describe the version of A* shown in Figure 1 that is similar to the version of A* used by FSA*.

3.1 Values

A* maintains five values for every cell s: (1) The h-value $h(s)$ is an approximation of the distance from cell s to the goal cell. We require the h-values to be consistent [6]. (2) The g-value $g(s)$ is the length of the shortest path 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 length of a shortest path 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* search tree. The parent pointers are used to extract the path after the A* search terminates. (5) The expansion order $ExpandedId(s)$ specifies that A* expanded the cell $ExpandedId(s)$th, where $ExpandedId(s_{start}) = 0$ and $ExpandedId(s) = \infty$ for unexpanded cells s (not shown in the pseudocode)

3.2 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. $OPEN.Insert(s)$ inserts cell s into the $OPEN$ list if it is not already contained in it; and $OPEN.Pop()$ removes a cell with the smallest f-value from the $OPEN$ list and returns it. (2) The $CLOSED$ list is a set that contains all cells that have already been expanded.

3.3 Algorithm

A* repeats the following procedure until the $OPEN$ list is empty (Line 20) or it has expanded the goal cell (Line 13): It removes a cell s with the smallest f-value from the $OPEN$ list (Line 08), inserts the cell into the $CLOSED$ list (Line 09), sets its expansion order (Line 10) and expands it by performing the following operations for each unblocked neighbor $s' \in Succ(s)$ of cell s: If cell s' is not in the $OPEN$ or $CLOSED$ lists or $g(s) + 1 < g(s')$, then A* generates the cell by assigning $g(s') := g(s) + 1$, setting the parent pointer of cell s' to cell s, and then inserting cell s' into the $OPEN$ list (Lines 17-19).

3.4 Properties

```
function ComputeShortestPath()
{01}  m := 0;
{02}  g(s_start) := 0;
{03}  Parent(s_start) := NULL;
{04}  OPEN := ∅;
{05}  OPEN.Insert(s_start);
{06}  CLOSED := ∅;
{07}  While OPEN ≠ ∅
{08}    s := OPEN.Pop();
{09}    CLOSED := CLOSED ∪ {s};
{10}    ExpandedId(s) := m;
{11}    m := m + 1;
{12}    If s = s_goal
{13}      Return "path found";
{14}    Else
{15}      Forall s' ∈ Succ(s)
{16}        If ((s' ∉ OPEN And s' ∉ CLOSED) Or g(s) + 1 < g(s'))
{17}          g(s') := g(s) + 1;
{18}          Parent(s') := s;
{19}          OPEN.Insert(s');
{20}  Return "no path found";
```

Figure 1: Pseudocode of A*

A* has the following properties [6].

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

- **A* Property 2:** A* terminates.

- **A* Property 3:** The g-value and parent pointer of any cell are correct when it is expanded and then do not change any longer, that is, the g-value of an expanded cell is equal to the distance from the start cell to the cell and a shortest path from the start cell to the cell can be identified in reverse by following the parent pointers from the cell to the start cell. We consider the goal cell expanded after the A* search terminates on Line 13 since the goal cell has this property then, which implies that A* finds a shortest path from the start cell to the goal cell if it terminates because it expands the goal cell.

- **A* Property 4:** No path exists from the start cell to the goal cell if A* terminates because the OPEN list is empty.

4. FSA*

Assume that the traversability of some cells changes after an A* search. We refer to the A* search before the traversability change as previous A* search and to the A* search after the traversability change as current A* search. The current A* search initially expands the same cells in the same order as the previous A* search. FSA* restores the state of the previous A* search (given by the content of its OPEN and CLOSED lists and the g-values and parent pointers of the cells contained in them) at the point in time when the current A* search could deviate from the previous A* search, that is, when the current A* search encounters a cell whose traversability changed. FSA* then starts an A* search at that point in time rather than performing it from scratch. Thus, it reuses the beginning of the previous A* search that is identical to the current A* search. The complete pseudocode of FSA* is given in [7]. We illustrate its operation with an example. Consider an A* search in the four-neighbor gridworld with start cell $E6$ and goal cell $E2$ shown in Figure 2(a). We break ties among cells with the same smallest f-value in favor of a cell with the largest g-
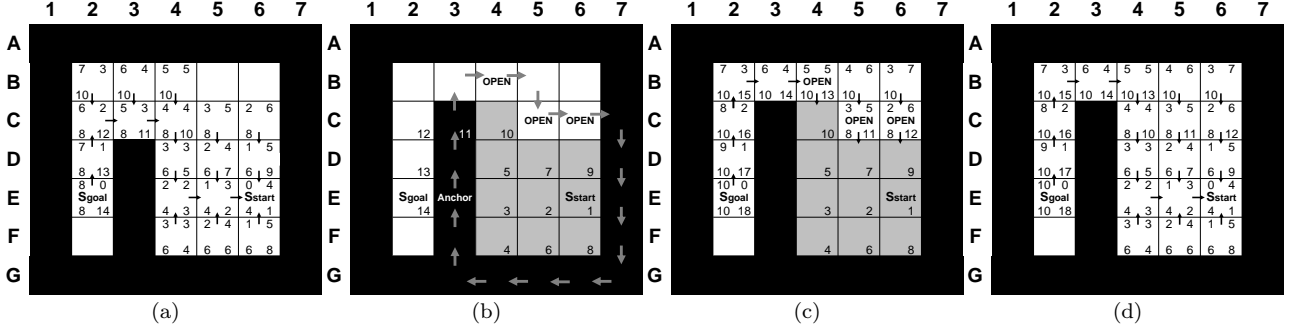
Figure 2: Example Search Problem with Fixed Goal Cell

value. Every generated cell contains its g-value in the upper left corner, its h-value (here: Manhattan distance to the goal cell) in the upper right corner and its f-value in the lower left corner. The outgoing arrow shows its parent pointer. Every expanded cell also contains its expansion order in the lower right corner. Assume that cell $C3$ becomes blocked after the A* search. FSA* then executes the following steps.

4.1 Step 1: Restoration of the CLOSED List

FSA* first restores the CLOSED list. Assume for now that, as for our example, only one cell s' changed its traversability. FSA* determines a value for m so that the current A* search expands at least every cell s with $ExpandedId(s) < m$ in the same order as the previous A* search.

• If cell s' became blocked, then the current A* search expands at least every cell s with $ExpandedId(s) < ExpandedId(s')$ in the same order as the previous A* search. Thus, FSA* sets

$$m := m(s') := ExpandedId(s'). \qquad (1)$$

For our example, cell $C3$ became blocked and $m := ExpandedId(C3) = 11$.

• If the start cell became unblocked, then the current A* search needs to be run from scratch. If cell $s' \neq s_{start}$ became unblocked, then the current A* search expands at least every cell s with $ExpandedId(s) < 1 + \min_{s'' \in Succ(s')} ExpandedId(s'')$ in the same order as the previous A* search. Thus, FSA* sets

$$m := m(s') := 1 + \min_{s'' \in Succ(s')} ExpandedId(s''). \qquad (2)$$

If several cells $s' \in S' \subseteq S$ changed their traversability, then the current A* search expands at least every cell s with $ExpandedId(s) < \min_{s' \in S'} m(s')$ in the same order as the previous A* search. Thus, FSA* sets $m := \min_{s' \in S'} m(s')$. The restored CLOSED list then contains every cell s with $ExpandedId(s) < m$. FSA* thus determines the restored CLOSED list by setting the value of m, which tends to be much faster than having to expand all cells in the restored CLOSED list again. The g-values and parent pointers of the cells in the restored CLOSED list are still correct. For our example, the restored CLOSED list contains the first ten cells expanded by the previous A* search, namely cells $E6$,

$E5$, $E4$, $F4$, $D4$, $F5$, $D5$, $F6$, $D6$ and $C4$, which are shown in grey in Figure 2(b).

4.2 Step 2: Early Termination of Iteration

FSA* checks whether it can skip replanning. If the goal cell is in the restored CLOSED list, then FSA* skips replanning because the shortest path from the start cell to the goal cell found by the previous A* search remains a shortest path. If the start cell is blocked, then FSA* skips replanning because there is no path from the start cell to the goal cell. If the start cell is unblocked and not in the restored CLOSED list, then FSA* performs an A* search from scratch. Otherwise, FSA* proceeds to prepare for replanning. For our example, the start cell is in the restored CLOSED list but the goal cell is not.

4.3 Step 3: Restoration of the OPEN List

FSA* restores the OPEN list at the point in time when the previous A* search had expanded the cells in the restored CLOSED list. The OPEN list (= fringe, which gives FSA* its name) at this point in time contains the unblocked cells that are not in the restored CLOSED list but are neighbors of one of more cells in the restored CLOSED list. FSA* does not necessarily restore all of the OPEN list but only the relevant part of the OPEN list. The relevant part of the OPEN list contains those cells in the OPEN list that are not separated from the goal cell by cells in the restored CLOSED list. First, FSA* identifies an anchor cell among the cells that are not in the restored CLOSED list but are neighbors of one or more cells in the restored CLOSED list. It follows a shortest path from the goal cell to the start cell under the assumption that all cells are unblocked until it is about to move from a cell not in the restored CLOSED list to a cell in the restored CLOSED list. The current cell then is the anchor cell. For our example, cell $E3$ is the anchor cell, as shown in Figure 2(b). Second, FSA* identifies the cells that belong to the restored OPEN list. The cells in the restored CLOSED list form a contiguous area since they are all reachable from the start cell. FSA* follows the outside perimeter of this contiguous area, starting with the anchor cell, and inserts all unblocked cells that are neighbors of one or more cells in the restored CLOSED list into the restored OPEN list. FSA* thus restores the OPEN list by following the outside perimeter of the restored CLOSED list, which tends to be faster than having to expand all cells in the restored CLOSED list again. For our example, FSA* follows the grey arrows shown in Figure 2(b). The restored OPEN
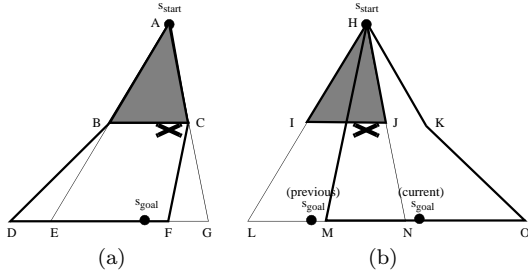
Figure 4: A* Search Trees

list contains cells $B4$, $C5$ and $C6$. The g-values and parent pointers of cells in the restored OPEN list are not necessarily correct. For each cell in the restored OPEN list that just became unblocked or whose parent pointer points to a cell that is not in the restored CLOSED list, FSA* therefore finds a neighbor of the cell in the restored CLOSED list with a smallest g-value. It then sets the g-value of the cell in the restored OPEN list to the g-value of that neighbor plus one and the parent pointer of the cell in the restored OPEN list to that neighbor. For our example, the g-values and parent pointers of all cells in the restored OPEN list are correct right away.

4.4 Step 4: Starting A*

FSA* finally replans by starting an A* search with the restored OPEN and CLOSED lists. The expansion order of the first cell s expanded by the A* search is $ExpandedId(s) = m$. For our example, the A* search reuses the ten cells in the restored CLOSED list and expands eight cells, as shown in Figure 2(c). A current A* search, on the other hand, would expand eighteen cells, including the ten cells reused by FSA*, as shown in Figure 2(d).

5. RE-INTERPRETING FSA*

FSA* finds shortest paths for series of search problems from a fixed start cell to a fixed goal cell in a known gridworld in case the traversability of cells changes over time. It restores the OPEN and CLOSED lists of the previous A* search at the point in time when the current A* search could deviate from it. For our example, we assumed that cell $C3$ becomes blocked and discussed the steps that FSA* executes. Figure 4(a) illustrates the relationships graphically. The previous A* search tree is A-E-G-A. Then, the traversability of a cell changes, as denoted by the cross. The current A* search tree is A-B-D-F-C-A. It is guaranteed to contain all of the previous A* search tree up to the change in traversability, which is A-B-C-A. This part therefore forms the restored CLOSED list, shown in grey. However, we are interested in finding shortest paths from a fixed start cell to a goal cell in a known gridworld in case the goal cell changes over time in addition to the traversability of cells. For our example, we assume now that cell $C2$ becomes the goal cell in addition to cell $C3$ becoming blocked, as shown in Figure 3(b). If FSA* executes the same steps as before, it no longer restores the OPEN and CLOSED lists of the previous A* search at the point in time when the current A* search could deviate from it. For example, the restored CLOSED list contains the cells $E6$, $E5$, $E4$, $F4$, $D4$, $F5$, $D5$, $F6$, $D6$ and $C4$ as before (listed in the order in which the previous A* search expanded them), shown in Figure

3(b), but the current A* search does not expand cells $F4$, $F5$ and $F6$ and expands the remaining cells in the different order $E6$, $E5$, $E4$, $D4$, $C4$, $D5$ and $D6$, as shown in Figure 3(d). Fortunately, it is unnecessary to change FSA* because its restored CLOSED list contains those cells expanded in all previous A* searches whose g-values and parent pointers are still guaranteed to be correct. They can thus be reused by the current A* search. This property continues to hold when the goal cell changes since the correctness of the g-values and parent pointers does not depend on the goal cell. The current A* search might not reuse all of the cells in the restored CLOSED list but future A* searches might. Figure 4(b) illustrates these relationships graphically. The previous A* search tree is H-L-N-H. Then, the goal cell changes in addition to the traversability of a cell, as denoted by the cross. The current A* search tree is H-M-O-K-H. It does not contain all of the previous A* search tree up to the change in traversability, which is H-I-J-H. However, it contains part of it. The other part can become again important in the future, for example, if the goal cell changes back to the previous one. Both parts therefore form the restored CLOSED list, shown in grey. FSA* finally replans by starting an A* search with the restored OPEN and CLOSED lists. For our example, this A* search reuses the ten cells in the restored CLOSED list and expands six cells, as shown in Figure 3(c). The current A* search, on the other hand, would expand thirteen cells, including seven of the ten cells reused by FSA*, as shown in Figure 3(d).

5.1 Correctness Proof

We now generalize the correctness proof of FSA* to cover the case where the goal cell changes over time in addition to the traversability of cells. The correctness proof of FSA* in case the goal cell remains fixed is simple since one can argue that FSA* restores the state of the previous A* search at the point in time when the current A* search could deviate from it and then apply the correctness proof of A*. This argument does not apply in case the goal cell changes over time since the restored CLOSED list might contain cells that are never in the CLOSED list of the current A* search and those cells that are in both CLOSED lists and are thus expanded by both the previous and current A* search can be expanded in different orders.

THEOREM 1. *Assume that the goal cell is not in the restored CLOSED list, that the start cell is in the restored CLOSED list, $g(s_{start}) = sd(s_{start}) = 0$ and, for every cell $s \neq s_{start}$ in the restored CLOSED list, $Parent(s)$ is in the restored CLOSED list, $ExpandedId(Parent(s)) < ExpandedId(s)$ and $g(s) = g(Parent(s)) + 1 = sd(s)$, where $sd(s)$ is the distance from the start cell to cell s. Then, the current A* search with the restored CLOSED and OPEN lists finds a shortest path from the start cell to the goal cell if a path exists and otherwise reports that no path exists. Furthermore, $g(s_{start}) = sd(s_{start}) = 0$ and, for every cell $s \neq s_{start}$ in the CLOSED list after the current A* search, $Parent(s)$ is in the CLOSED list, $ExpandedId(Parent(s)) < ExpandedId(s)$ and $g(s) = g(Parent(s)) + 1 = sd(s)$.*

Proof Sketch: The standard A* correctness proof applies almost unchanged to the A* search performed by FSA*. We give an abbreviated version of such a proof here.

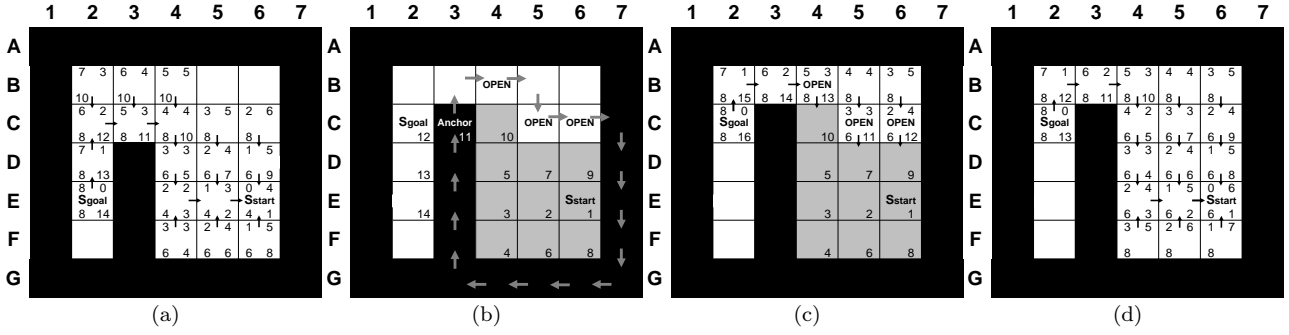- A* always expands a cell s with the smallest f-value

Figure 3: Example Search Problem with Changing Goal Cell

in the OPEN list and then replaces it in the OPEN list with zero or more of its unblocked neighbors $s' \in Succ(s)$. These neighbors become the children of cell s in the A* search tree. A* sets $Parent(s') = s$ (where cell s is now in the CLOSED list) and $g(s') = g(s) + 1 = g(Parent(s')) + 1$. The f-values of these neighbors are no smaller than the f-value of cell s since the h-values are consistent. Thus, the f-values along any branch of the A* search tree are monotonically nondecreasing (Monotonicity Property 1). Furthermore, the f-value of all cells in the resulting OPEN list (including the cell that will be expanded next) are no smaller than the f-value of cell s. Thus, the sequence of f-values of the expanded cells is monotonically non-decreasing (Monotonicity Property 2).

• Assume that A* expands a cell s with f-value $f(s) = g(s) + h(s)$ and later expands it again with f-value $f'(s) = g'(s) + h(s)$. Then, $f(s) \leq f'(s)$ according to Monotonicity Property 2 and thus $g(s) \leq g'(s)$. Thus, A* expands a cell only once since it cannot decrease its g-value further.

• Assume that A* expands a cell s for the first time with f-value $f(s) = g(s) + h(s)$ and that $sd(s) < g(s)$. Then, there exists a path of length $sd(s)$ from the start cell to cell s. The f-value of cell s on this path is $f'(s) = sd(s) + h(s)$. The f-values of all cells on this path are no larger than $f'(s)$ according to Monotonicity Property 1, where $f'(s) < f(s)$. At least one of them is always in the OPEN list (starting with the restored OPEN list) since every path from the start cell to the goal cell passes through the restored OPEN list. A* always expands a cell with the smallest f-value in the OPEN list and thus expands all cells on the path before it expands cell s with f-value $f(s)$. Thus, it expands cell s for the first time with f-value $f'(s)$, which is a contradiction. Thus, A* expands a cell s for the first time with f-value $f(s) = sd(s) + h(s)$. When A* expands cell s, the cell satisfies that $g(s) = g(Parent(s)) + 1 = sd(s)$ and $Parent(s)$ is already in the CLOSED list, which implies that $ExpandedId(Parent(s)) < ExpandedId(s)$.

• Assume that A* never expands the goal cell although there exists a path from the start cell to the goal cell. Since A* never expands the goal cell, the sequence of g-values of the expanded cells and thus also the sequence of f-values of the expanded cells goes towards infinity. The f-value of the goal cell on a shortest path from the start cell to

the goal cell is $f(s_{goal}) = sd(s_{goal})$ and thus finite since there exists a path from the start cell to the goal cell. The f-values of all cells on the path are no larger than $f(s_{goal})$ according to Monotonicity Property 1 and thus finite. At least one of them is always in the OPEN list, for the reason given above. A* always expands a cell with the smallest f-value in the OPEN list and thus expands all cells on the path eventually. Thus, it expands the goal cell, which is a contraction. Thus, A* eventually expands the goal cell if there exists a path from the start cell to the goal cell. Then, A* terminates.

• Assume that there does not exist a path from the start cell to the goal cell. The number of cells is finite. A* inserts every cell at most once into the OPEN list. It expands every cell in the OPEN list and removes it from the OPEN list in the process. Thus, if A* does not terminate earlier because it expands the goal cell, it eventually terminates because its OPEN list is empty and correctly reports that no path from the start cell to the goal cell exists. ∎

THEOREM 2. *Assume that* $g(s_{start}) = sd(s_{start}) = 0$ *and, for every cell* $s \neq s_{start}$ *in the CLOSED list of the previous A* search,* $Parent(s)$ *is in the CLOSED list,* $ExpandedId(Parent(s)) < ExpandedId(s)$ *and* $g(s) = g(Parent(s)) + 1 = sd(s)$. *Then,* $g(s_{start}) = sd(s_{start}) = 0$ *and, for every cell* $s \neq s_{start}$ *in the restored CLOSED list,* $Parent(s)$ *is in the restored CLOSED list,* $ExpandedId(Parent(s)) < ExpandedId(s)$ *and* $g(s) = g(Parent(s)) + 1 = sd(s)$.

Proof Sketch: The restored CLOSED list contains exactly the cells s in the CLOSED list with $ExpandedId(s) < m$, where FSA* sets m as described in Section 4.1. Initially, the g-values and parent pointers of all cells in the CLOSED list are correct according to Theorem 1. A change of the goal cell does not affect their correctness but a change in traversability might.

• Assume that the cost of moving from a cell s'' to its neighbor s' increases. If cell s'' is in the CLOSED list with $Parent(s') = s''$ then the g-values and parent pointers of cell s' and its descendants in the A* search tree, that is formed by the cells in the CLOSED list and their parent pointers, can become incorrect because there might now exist a shorter path from the start cell to cell s' or its descendants that does not pass through s'' and s'. Thus, the g-values and parent pointers of some cells s

in the CLOSED list with $ExpandedId(s) \geq ExpandedId(s')$ can become incorrect. The g-values and parent pointers of all other cells in the CLOSED list remain correct since the current shortest paths from the start cell to them do not pass through s'' and s'.

If the start cell becomes blocked, then FSA* skips replanning. Assume that a different cell s' in the CLOSED list becomes blocked. Then, the cost of moving into s' can increase from one to infinity. The g-values and parent pointers of s' and its descendants in the A* search tree can become incorrect according to the argument above since the parent of cell s' is in the CLOSED list. Thus, the g-values and parent pointers of some cells s in the CLOSED list with $ExpandedId(s) \geq ExpandedId(s')$ can become incorrect. Similarly, the cost of moving out of s' can increase from one to infinity. Consider any unblocked neighbor s'' of cell s' with $Parent(s'') = s'$. The g-values and parent pointers of cell s'' and its descendants in the A* search tree can become incorrect according to the argument above since cell s' is in the CLOSED list. Thus, the g-values and parent pointers of some cells s in the CLOSED list with $ExpandedId(s) \geq ExpandedId(s'') > ExpandedId(s')$ can become incorrect. Thus, in both cases, the g-values and parent pointers of some cells s in the CLOSED list with $ExpandedId(s) \geq ExpandedId(s')$ can become incorrect.

• Assume that the cost of moving from a cell s'' to its neighbor s' decreases. If cell s'' is in the CLOSED list then the g-values and parent pointers of cells s in the CLOSED list with $ExpandedId(s) \geq 1 + ExpandedId(s'')$ can become incorrect because there might now exist a shorter path from the start cell to cell s that passes through s'' and s'.

If the start cell becomes unblocked, then FSA* performs an A* search from scratch. Assume that a different cell s' in the CLOSED list becomes unblocked. Then, the cost of moving into s' can decrease from infinity to one. Consider any unblocked neighbor s'' of cell s' that is in the CLOSED list. The g-values and parent pointers of cells s in the CLOSED list with $ExpandedId(s) \geq 1 + ExpandedId(s'')$ can become incorrect according to the argument above since cell s'' is in the CLOSED list. Thus, the g-values and parent pointers of some cells s in the CLOSED list with $ExpandedId(s) \geq 1 + \min_{s'' \in Succ(s')} ExpandedId(s'')$ can become incorrect. Similarly, the cost of moving out of s' can decrease from infinity to one. The g-values and parent pointers of cells s in the CLOSED list with $ExpandedId(s) \geq 1 + ExpandedId(s') > 1 + \min_{s'' \in Succ(s')} ExpandedId(s'')$ can become incorrect according to the argument above since cell s' is in the CLOSED list. (The last inequality holds since the parent of cell s' is one of its unblocked neighbors.) Thus, in both cases, the g-values and parent pointers of some cells s in the CLOSED list with $ExpandedId(s) \geq 1 + \min_{s'' \in Succ(s')} ExpandedId(s'')$ can become incorrect.

FSA* sets m so that all cells s in the CLOSED list whose g-values and parent pointers could have become incorrect are excluded from the restored CLOSED list. ∎

Thus, by induction, $g(s_{start}) = sd(s_{start})$ and, for every cell $s \neq s_{start}$ in the CLOSED list (no matter whether it was in the restored CLOSED list or expanded by the A* search),

```
procedure UpdateMazeTraversability()
{01'}  TmpBlockId := ExpandedId(s_goal) + 1;
{02'}  Forall cells s whose traversability has changed
{03'}     If (s is blocked)
{04'}        If (CellReusable(s))
{05'}           If (ExpandedId(s) < TmpBlockId)
{06'}              TmpBlockId := ExpandedId(s);
{07'}     Else
{08'}        Parent(s) := NULL;
{09'}        Forall s' ∈ Succ(s)
{10'}           If (CellReusable(s'))
{11'}              If (ExpandedId(s') + 1 < TmpBlockId)
{12'}                 TmpBlockId := ExpandedId(s') + 1;
{13'}  If (TmpBlockId ≤ ExpandedId(s_goal))
{14'}     Forall i = 1...Iteration
{15'}        If (TmpBlockId < BlockId(i))
{16'}           BlockId(i) := TmpBlockId;
{17'}  m := BlockId(Iteration);
```

Figure 5: Pseudocode Fragment of FSA*

$Parent(s)$ is in the CLOSED list, $ExpandedId(Parent(s)) < ExpandedId(s)$ and $g(s) = g(Parent(s)) + 1 = sd(s)$. Thus, the g-value of every cell in the CLOSED list is equal to the distance from the start cell to the cell and a shortest path from the start cell to the cell can be identified in reverse by following the parent pointers from the cell to the start cell.

6. APPLYING FSA*

Our insights into FSA* allow us to apply FSA* to agent navigation problems, namely the problem of moving an agent on a shortest path from its current cell to a fixed destination cell in a known gridworld, where the shortest path is replanned whenever the traversability of cells changes. This problem is similar to the problem of moving a game character in a real-time strategy game to a fixed destination while the world changes, for example, bridges and fences are built by other game characters. It is also similar to the problem of moving a robot in unknown terrain via a presumed unblocked path to a fixed destination cell, which is called planning with the freespace assumption and has been extensively studied in robotics [5]. In these cases, the current cell of the agent changes over time while the destination cell remains fixed. FSA* therefore determines a shortest path from the current cell of the agent to the fixed destination cell by searching from the destination cell to the current cell of the agent. The agent then follows the parent pointers from its current cell to the fixed destination cell until it either reaches the destination cell or the traversability of cells changes. In the latter case, FSA* determines another shortest path from the the current cell of the agent to the fixed destination cell, and the process repeats. Our example from Figures 2 and 3 is consistent with an agent moving on a shortest path from its current cell $E2$ to the fixed destination cell $E6$ and then replanning the shortest path when cell $C3$ becomes blocked after it reaches cell $C2$. FSA* can be optimized slightly for agent navigation problems, resulting in Dynamic FSA*, since the goal cell does not change arbitrarily but rather follows the parent pointers from the current cell of the agent to cells with smaller and smaller expansion orders. Assume that the current cell of the agent is s_{goal} when the traversability of cells changes and FSA* would ordinarily perform its bookkeeping operations to exclude the cells s with $ExpandedId(s) \geq m$ from the restored CLOSED list. Assume further that, in case $ExpandedId(s_{goal}) < m$, the agent continues to follow the shortest path from its current cell to the destination cell without the bookkeeping op-

	Four-Neighbor Gridworld						Eight-Neighbor Gridworld					
	(a)	(b)	(c)	(d)	(e)	(f)	(a)	(b)	(c)	(d)	(e)	(f)
p = 0.001												
A* (optimized)	668	1994 (28.7)	N/A	0	797	1.76	451	329 (13.8)	N/A	0	158	1.42
A* (unoptimized)	668	10273 (29.6)	N/A	0	3822	8.42	451	3132 (32.1)	N/A	0	1391	12.53
D* Lite	668	56 (3.2)	N/A	1	*46*	*0.10*	451	27 (4.6)	N/A	1	*34*	*0.31*
Dynamic FSA*	667	1042 (27.1)	97	0	454	1.00	450	189 (0.9)	32	0	111	1.00
p = 0.01												
A* (optimized)	669	9815 (40.0)	N/A	0	3698	1.17	451	2720 (33.6)	N/A	0	1253	1.22
A* (unoptimized)	669	10882 (40.6)	N/A	0	4054	1.28	451	3442 (33.1)	N/A	0	1535	1.49
D* Lite	669	194 (3.9)	N/A	16	*161*	*0.05*	451	74 (6.5)	N/A	32	*111*	*0.11*
Dynamic FSA*	668	7522 (28.6)	481	1	3161	1.00	450	1904 (8.9)	263	6	1031	1.00
p = 0.1												
A* (optimized)	665	12268 (46.7)	N/A	0	4594	0.96	453	4167 (36.5)	N/A	0	1936	0.93
A* (unoptimized)	665	12311 (49.2)	N/A	0	4628	0.97	453	4209 (36.4)	N/A	0	1940	0.93
D* Lite	665	1468 (7.5)	N/A	179	*1295*	*0.27*	452	501 (15.3)	N/A	307	*866*	*0.41*
Dynamic FSA*	664	11796 (36.5)	378	12	4795	1.00	452	3869 (18.1)	358	83	2088	1.00
p = 0.5												
A* (optimized)	629	13941 (66.2)	N/A	0	*5262*	*0.95*	461	4925 (38.4)	N/A	0	2299	0.77
A* (unoptimized)	629	13945 (68.1)	N/A	0	5273	0.95	457	4962 (37.9)	N/A	0	*2295*	*0.77*
D* Lite	626	5782 (15.3)	N/A	808	5276	0.95	457	1843 (28.5)	N/A	1488	3491	1.17
Dynamic FSA*	629	13840 (55.7)	150	63	5532	1.00	456	4914 (22.9)	172	385	2973	1.00
p = 1												
A* (optimized)	574	15012 (103.5)	N/A	0	*5666*	*0.95*	461	4925 (38.4)	N/A	0	*2287*	*0.70*
A* (unoptimized)	574	15013 (105.1)	N/A	0	5678	0.96	461	4925 (38.4)	N/A	0	2299	0.71
D* Lite	580	9334 (21.4)	N/A	1506	8813	1.48	461	3011 (36.7)	N/A	2819	6063	1.87
Dynamic FSA*	577	15063 (96.7)	89	128	5945	1.00	461	4910 (22.8)	101	763	3247	1.00
p = 2												
A* (optimized)	483	16696 (147.9)	N/A	0	*6312*	*0.96*	459	5146 (39.5)	N/A	0	*2401*	*0.60*
A* (unoptimized)	483	16696 (149.6)	N/A	0	6332	0.96	459	5146 (39.5)	N/A	0	2405	0.60
D* Lite	474	14468 (32.3)	N/A	2869	14409	2.18	459	4121 (41.8)	N/A	5618	10052	2.50
Dynamic FSA*	475	16985 (151.4)	52	280	6595	1.00	458	5142 (24.0)	57	1414	4025	1.00
p = 3												
A* (optimized)	393	18556 (227.0)	N/A	0	*7050*	*0.99*	468	5309 (41.3)	N/A	0	2489	0.51
A* (unoptimized)	393	18556 (228.6)	N/A	0	7340	1.03	468	5306 (41.2)	N/A	0	*2474*	*0.50*
D* Lite	412	18134 (40.3)	N/A	4240	18940	2.66	467	4884 (45.1)	N/A	8209	13449	2.73
Dynamic FSA*	402	18610 (232.8)	38	421	7114	1.00	467	5324 (24.6)	41	2229	4923	1.00
p = 4												
A* (optimized)	342	19749 (309.2)	N/A	0	*7493*	*0.97*	474	5276 (40.4)	N/A	0	*2467*	*0.45*
A* (unoptimized)	342	19749 (306.6)	N/A	0	7519	0.98	474	5276 (40.4)	N/A	0	2480	0.46
D* Lite	345	21531 (50.3)	N/A	5359	22352	2.90	474	5282 (45.8)	N/A	10903	16570	3.05
Dynamic FSA*	335	19844 (288.8)	31	553	7697	1.00	473	5301 (24.3)	32	2734	5433	1.00
p = 5												
A* (optimized)	298	20988 (389.4)	N/A	0	7977	0.99	470	5335 (40.4)	N/A	0	2497	0.41
A* (unoptimized)	298	20988 (386.5)	N/A	0	*7559*	*0.94*	470	5335 (40.4)	N/A	0	*2496*	*0.41*
D* Lite	299	23777 (60.2)	N/A	6881	26792	3.33	471	5808 (51.8)	N/A	13416	19662	3.23
Dynamic FSA*	306	20737 (364.3)	26	856	8046	1.00	468	5370 (24.8)	27	3377	6086	1.00
p = 10												
A* (optimized)	188	26350 (713.0)	N/A	0	10075	1.00	481	5508 (44.5)	N/A	0	2590	0.30
A* (unoptimized)	188	26350 (706.8)	N/A	0	*9660*	*0.95*	481	5508 (44.5)	N/A	0	*2581*	*0.29*
D* Lite	182	34531 (106.1)	N/A	14151	44827	4.43	483	6875 (55.6)	N/A	25764	33246	3.79
Dynamic FSA*	182	27062 (684.2)	17	1423	10125	1.00	482	5502 (25.1)	16	5961	8763	1.00

Table 1: Experimental Results

erations. Then, the agent either reaches its destination without any replanning or, at some point in time, the current cell of the agent is s'_{goal} when the traversability of cells changes so that the current cell of the agent is not reusable. In the latter case, FSA* now performs its bookkeeping operations to exclude the cells s with $ExpandedId(s) \geq m'$ from the restored CLOSED list, which automatically excludes the cells s with $ExpandedId(s) \geq m$ since $m' \leq ExpandedId(s'_{goal}) \leq ExpandedId(s_{goal}) < m$. For completeness, Figure 5 shows a fragment of the pseudocode of FSA* from [7], which is described in detail there. Line 13' is new and implements the optimization. This optimization then allows the further optimization to change Line 01' from "*TmpBlockId* := ∞;" to "*TmpBlockId* := $ExpandedId(s_{goal}) + 1$;" which avoids some of the assignments on Lines 06' and 12' from being executed.

7. EXPERIMENTAL EVALUATION

We evaluated Dynamic FSA* on agent navigation prob-

lems. We solved 500 agent navigation problems in known gridworlds of size 1000×1000 with 250,000 randomly chosen blocked cells on a Pentium D 3.0 GHz PC with 2 GByte of RAM. The current cell of the agent and the destination cell were randomly chosen. After each move of the agent from its current cell to a neighbor, a number of randomly chosen unblocked cells that corresponds to $p/2$ percent of all cells were blocked and a number of randomly chosen blocked cells that corresponds to $p/2$ percent of all cells were unblocked, so that the total number of blocked cells remained the same. We then determined a new shortest path from the current cell of the agent to the destination cell (knowing about all blockage changes) and repeated the process. We considered an agent navigation problem to be solved once the agent reached the destination cell or there were no paths from the current cell of the agent to the destination cell.

7.1 Search Algorithms

We compared Dynamic FSA* experimentally to D* Lite

[2] and two versions of repeated A* searches. One version of A*, called A* (unoptimized), always performed an A* search. The other version of A*, called A* (optimized), only performed an A* search if at least one blockage change was in the previous search tree. (We counted it as an A* search with no cell expansions if it did not perform an A* search.) We excluded Generalized Adaptive A* from the comparison because it has already been shown to be slower than D* Lite on a variety of agent navigation problems with fixed destinations [8]. D* Lite is the application of LPA* to agent navigation problems, similar to how Dynamic FSA* is the application of FSA* to agent navigation problems. D* Lite and Dynamic FSA* have in common that the root of the A* search tree and thus the start cell must remain fixed while the goal cell can change over time. Thus, they both search from the destination cell to the current cell of the agent, which is why we compare them. D* Lite runs fast if the A* search tree of the previous A* search is similar to the one of the current A* search, which is typically the case if the number of traversability changes is small and they occur close to the goal cell. The three search algorithms differ in their runtime per cell expansion. D* Lite uses modified A* searches and expands cells more slowly than A*. Dynamic FSA* uses unmodified A* searches and thus expands cells about as fast as A*. The three search algorithms also differ in their processing time of each traversability change. A* does not performs any processing. D* Lite updates the g-values of all neighbors of the cell whose traversability changed to the minimum of the g-values of their neighbors plus one, which can require b^2 operations on b-neighbor gridworlds. Dynamic FSA* evaluates Equations 1 and 2 to determine the value of m, which can require b operations on b-neighbor gridworlds. We thus varied b (by using four-neighbor and eight-neighbor gridworlds) and p in the experiments. We used the Manhattan distances (= the sum of the absolute differences of the x and y coordinates of a cell and the goal cell) as h-values in four-neighbor gridworlds and the maximum of the absolute differences of the x and y coordinates of a cell and the goal cell as h-values in eight-neighbor gridworlds. In both cases, the cost of moving from an unblocked cell to an unblocked neighbor was one. In general, the runtime of search algorithms depends on the hardware, compiler and implementation, including the data structures, tie-breaking strategies and coding tricks used. However, there is currently no better methodology available for comparing the runtimes of search algorithms that work according to very different principles other than to implement them and measure their runtime. We implemented the three search algorithms in a very similar way for fairness. For example, all of them didn't use code optimizations, used binary heaps to implement the OPEN list and broke ties among cells with the same smallest f-value in favor of a cell with the largest g-value, which is considered to be a good tie-breaking strategy.

7.2 Results and Interpretation

Table 1 reports one measure for the difficulty of agent navigation problems, namely the number of A* searches until an agent navigation problem was solved (a). This number is similar for all three search algorithms since they all find shortest paths. The table reports several measures for the efficiency of the search algorithms. First, it reports the number of cell expansions per A* search (b) with the standard deviation of the mean in parentheses to demonstrate the statistical significance of our results. Second, it reports the number of cells in the restored OPEN list of Dynamic FSA* before it starts an A* search (c). Third, it reports the runtime needed to process all traversability changes before an A* search (d). Fourth, it reports the runtime per A* search (e), which includes the runtime needed to process all traversability changes before the A* search but not the runtime needed to choose the cells that change their traversability. Finally, it reports the normalized runtime per A* search (f) as the ratio of the runtime per A* search and the runtime per A* search of Dynamic FSA*. The smallest (unnormalized and normalized) runtimes per A* search are shown in italics. The table shows that D* Lite is best for small values of p. A* (unoptimized or optimized) becomes best as the value of p grows. Dynamic FSA* is always dominated by one of the other search algorithms. We were able to construct isolated cases where Dynamic FSA* is best, namely when blockage changes occur only around the agent, but even then the runtime of Dynamic FSA* is no more than four percent smaller than the runtime of the next best search algorithm.

8. CONCLUSIONS

Fringe-Saving A* (FSA*) is an incremental version of A* that repeatedly finds shortest paths from a fixed start cell to a fixed goal cell in a known gridworld in case the traversability of cells changes over time. In this paper, we generalized the correctness proof of FSA* to cover the case where the goal cell changes over time in addition to the traversability of cells. We then applied it to the problem of moving an agent along a shortest path from its current cell to a fixed destination cell in a known gridworld, where the shortest path is replanned whenever necessary due to changes in the traversability of cells. Unfortunately, our experimental results show that the resulting Dynamic Fringe-Saving A* algorithm tends to be dominated by either repeated A* searches or D* Lite (a state-of-the-art incremental version of A*).

9. REFERENCES

[1] 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.

[2] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *Transaction on Robotics*, 21(3):354–363, 2005.

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

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

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

[7] X. Sun and S. Koenig. The Fringe-Saving A* search algorithm - a feasibility study. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2391–2397, 2007.

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