The Fringe-Saving A* Search Algorithm - A Feasibility Study

Xiaoxun Sun
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
xiaoxuns@usc.edu

Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
skoenig@usc.edu

Abstract

In this paper, we develop Fringe-Saving A* (FSA*), an incremental version of A* that repeatedly finds shortest paths in a known gridworld from a given start cell to a given goal cell while the traversability costs of cells increase or decrease. The first search of FSA* is the same as that of A*. However, FSA* is able to find shortest paths during the subsequent searches faster than A* because it reuses the beginning of the immediately preceeding A* search tree that is identical to the current A* search tree. FSA* does this by restoring the content of the OPEN list of A* at the point in time when an A* search for the current search problem could deviate from the A* search for the immediately preceeding search problem. We present first experimental results that demonstrate that FSA* can have a runtime advantage over A* and Lifelong Planning A* (LPA*), an alternative incremental version of A*.

1 Introduction

Most research on heuristic search has studied how to solve one-shot search problems. However, search is often a repetitive process where one needs to find shortest paths to series of similar search problems because the world changes over time. For example, the traffic conditions can change when one drives to the airport and one might have to change the current route if the radio reports additional congestion. In this case, one has to replan for the new situation. Incremental search algorithms are often able to find shortest paths to series of similar search problems faster than is possible by solving each search problem independently, by re-using information from the preceeding searches [Koenig *et al.*, 2004b]. In this paper, we develop Fringe-Saving A* (FSA*), an incremental version of A* [Hart *et al.*, 1968] that repeatedly finds shortest paths in a known gridworld from a given start cell to a given goal cell while the traversability costs of cells increase or decrease. However, FSA* is often able to find shortest paths during the subsequent searches faster than A* because it reuses the beginning of the immediately preceeding A* search tree that is identical to the current A* search tree. FSA* does this by restoring the content of the OPEN list of A* at the point in time when an A* search for the current search problem could possibly deviate from the A* search for the immediately preceeding search problem because the A* search for the current search problem has encountered a changed edge cost. FSA* then restarts the A* search at that point. In the following sections, we first describe which search problems FSA* solves. Second, we explain in detail how FSA* reuses information from the immediately preceeding search to speed up the current search. Finally, we present first experimental results that demonstrate that FSA* can have a runtime advantage over A* and LPA* [Koenig *et al.*, 2004a], an alternative incremental version of A*.

2 Search Problems and Notation

Fringe-Saving A* (FSA*) solves path-planning problems in known finite gridworlds whose vertices correspond to the cells and whose edge costs increase or decrease over time. In this paper, we describe a version of FSA* that works on gridworlds of square cells that are either blocked or unblocked, where one can always move from an unblocked cell to one of the four adjacent cells with cost one, provided that the adjacent cell is unblocked. We use the following notation to be able to state the search problems formally: S denotes the finite set of unblocked cells. $Succ(s) \subseteq S$ denotes the set of unblocked cells that border unblocked cell $s \in S$. FSA* repeatedly determines a shortest (unblocked) path from a given unblocked cell $s_{start} \in S$ to a given unblocked goal cell $s_{goal} \in S$ with $s_{start} \neq s_{goal}$ as the traversability of cells changes, always knowing which cells are currently unblocked.

3 A*

A* [Hart *et al.*, 1968] is the most popular heuristic search algorithm. It maintains three values for every cell: First, the h-value $h(s)$ of a cell s is the user-given approximation of the distance from the cell to the goal cell (= heuristic), which we assume to be consistent, that is, it satisfies the triangle inequality [Pearl, 1985]. It is not changed during an A* search. Second, the g-value $g(s)$ of a cell s is an approximation of the distance from the start cell to the cell. It is changed during the A* search. Finally, the parent pointer $Parent(s)$ of a cell s points to its parent cell in the A* search tree. It is also changed during the A* search. A* maintains two data

structures: First, the CLOSED list contains all cells that have been expanded during the A* search. Initially, it is empty. Second, the OPEN list contains all cells that have been generated but not yet expanded during the A* search. Initially, it contains only the start cell with g-value zero. A* repeats the following procedure: A* removes a cell with the smallest sum of g-value and h-value from the OPEN list, inserts it into the CLOSED list and expands it by performing the following procedure for each unblocked adjacent cell. If the adjacent cell is neither in the OPEN nor CLOSED list, then A* generates the adjacent cell by setting the g-value of the adjacent cell to the g-value of the expanded cell plus one, setting the parent pointer of the adjacent cell to the expanded cell, and then inserting the adjacent cell into the OPEN list. If the adjacent cell is in the OPEN list and the g-value of the expanded cell plus one is smaller than the g-value of the adjacent cell, then A* sets the g-value of the adjacent cell to the g-value of the expanded cell plus one and sets the parent pointer of the adjacent cell to the expanded cell. A* terminates immediately when its OPEN list is empty or when it is about to expand the goal cell. (For simplicity, we count the goal cell as expanded.) Figure 1 shows the state of an A* search with start cell D2 (marked S) and goal cell F6 (marked G) after termination. We use the consistent Manhattan distance (the sum of the absolute x and y distances between a cell and the goal cell) as an approximation of the distance from a cell to the goal cell. Every cell has its h-value in the lower left corner. Every generated cell also has its g-value in the upper left corner, the sum of g-value and h-value in the upper right corner, and its parent pointer pointing to its parent cell in the A* search tree. Every expanded cell also has its sequence number in the lower right corner that indicates when it was expanded. FSA* makes use of the following four properties of A*: First, A* terminates. Second, the sum of g-value and h-value of the sequence of expanded cells is monotonically nondecreasing over time. Third, 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 (= the start distance of 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. This property implies that A* finds a shortest path from the start cell to the goal cell if it terminates because it is about to expand the goal cell. Fourth, no path exists from the start cell to the goal cell if A* terminates because its OPEN list is empty.

4 FSA*

Fringe-Saving A* (FSA*) is an incremental version of A*. Figure 6 gives its pseudo code.[1] The idea behind FSA* is

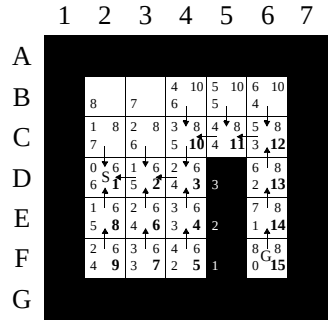


Figure 1: Search Problem 1

simple. The first search of FSA* to determine a shortest path from the start cell to the goal cell is the same as an A* search. Now assume that the traversability of some cells changes and consider another A* search to determine a shortest path from the start cell to the goal cell. The second A* search initially expands the same cells in the same order as the first A* search. FSA* restores the overall state of the first A* search at the point in time when the second A* search could possibly deviate from it, that is, when the second A* search encounters a cell whose traversability changed between the two A* searches. FSA* then restarts the first A* search at that point in time rather than performing the second A* search from scratch. The state of an A* search is given by the content of its OPEN and CLOSED lists and the g-values and parent pointers of the cells contained in them. FSA* executes the following steps.

Step 1 (Restoration of the CLOSED list): Assume that there are two complete A* searches and that *ExpandedId*(s) is

[1]The function *OPEN.Insert*(s) inserts a cell s into the OPEN list; and *OPEN*.$Pop(s)$ removes and returns a cell with the smallest sum of g-value and h-value from the OPEN list. The following comments are meant to help the reader understand the pseudo code better: 1) The initializations on lines $\{$02-05$\}$ will not actually be executed since memory cells are typically initialized with zero. 2) When a cell s is generated during the A* search then line $\{24\}$ sets *GeneratedIteration*(s) $=$ *Iteration* to indicate that the g-value of the cell is current. 3) Rather than identifying the reusable cells s repeatedly by checking whether *ExpandedId*(s) $<$ *BlockId*(*ExpandedIteration*(s)), lines $\{$55-56$\}$ set *GeneratedIteration*(s) $=$ *Iteration* for the reusable cells s that could otherwise get re-generated by the A* search. This way, the A* search knows that the g-values of these cells are current and, since it cannot decrease their g-values, will not re-generate the cells. Lines $\{$55-56$\}$ can be implemented very efficiently as part of identifying and traversing the relevant part of the fringe. 4) Lines $\{$43-45$\}$ can be efficiently implemented with a linked list to skip those elements *BlockId*(i) for which no cell s with *ExpandedId*(s) $<$ *BlockId*(*ExpandedIteration*(s)) exists. 5) To be more efficient, our implementation of FSA* traverses only the immediate outside of the continuous area of reusable cells between the borders of the gridworld in case the continuous area of reusable cells touches the border. To this end, FSA* traverses the immediate outside of the continuous area of reusable cells in the clockwise direction, starting with the anchor cell, until it is about to leave the anchor cell a second time in the same direction or reaches some cell of the border. In the second case, it then traverses the immediate outside of the continuous area in the counter-clockwise direction, starting again with the anchor cell, until it reaches some (potentially different) cell of the border. The fourth gridworld in Figure 2 illustrates this process for the search problem from Figure 1. 6) To be more efficient, our implementation of FSA* collapses the two variables *GeneratedIteration*(s) and *ExpandedIteration*(s) into one variable *Iteration*(s) although this requires code changes that make the pseudo code harder to understand.

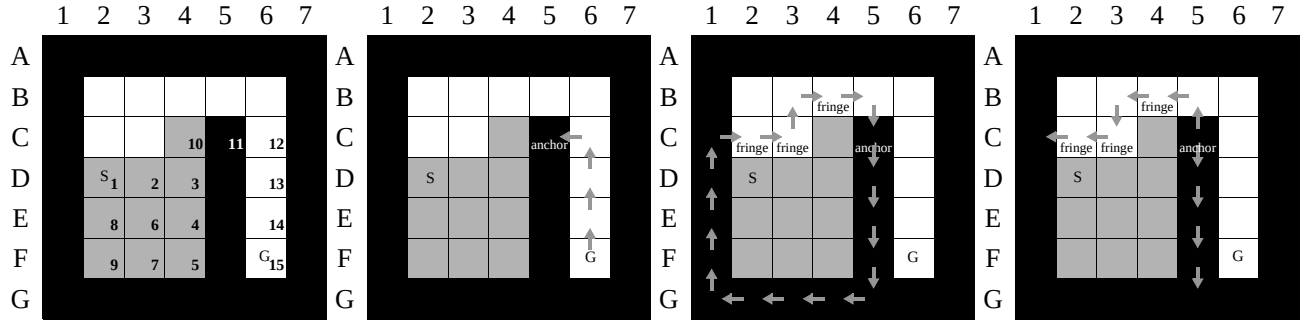


Figure 2: Search Problem 1 (continued)

the sequence number of cell s after the first A* search, that is, that cell s was expanded *ExpandedId*(s)th during the first A* search. (The start cell was expanded first. If cell s was not expanded during the first A* search, then *ExpandedId*$(s) = \infty$.) Assume further that only one cell s' changed its traversability between the two A* searches. We now determine a value for m so that the second A* search expands at least every cell s with *ExpandedId*$(s) < m$ in the same order as the first A* search. If cell s' became blocked, then the second A* search expands at least every cell s with *ExpandedId*$(s) <$ *ExpandedId*(s') in the same order as the first A* search, that is, every cell up to but not including cell s'. Thus, we set $m := m(s') :=$ *ExpandedId*(s'). If cell s' became unblocked, then the second 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 first A* search, that is, every cell up to and including the neighbor of cell s' that was expanded first in the first A* search. Thus, we set $m := m(s') := 1 + \min_{s'' \in Succ(s')}$ *ExpandedId*(s''). If several cells $s' \in S'$ changed their traversability between the two A* searches, then the second A* search expands at least every cell s with *ExpandedId*$(s) < m := \min_{s' \in S'} m(s')$ in the same order as the first A* search {32-42} (numbers in curly braces refer to line numbers in the pseudo code). Thus, these cells (which are guaranteed to be unblocked) are in the new CLOSED list. We refer to them as reusable cells. The first gridworld in Figure 2 illustrates this process for the search problem from Figure 1 after cell C5 with sequence number 11 became blocked. All cells with sequence numbers less than 11 are then reusable, as indicated in grey. It turns out that maintaining the sequence numbers is more complicated than discussed so far in case there are several searches in a row because FSA* might have to reuse cells from different searches. We would like a sequence number *ExpandedId*(s) to mean that cell s would have been expanded *ExpandedId*(s)th during the immediately preceeding A* search if the immediately preceeding search had been a complete A* search. Now consider the case where some cell s was expanded during the first search, but the cell was not reusable in the second search. Then, some other cell might have received the same sequence number as cell s during the second search, and FSA* might have to determine during the third search that the sequence number of cell s has become invalid. FSA* addresses this problem by maintaining the following global variables: *Iteration* is the number of the current search, that is, *Iteration* $= i$ during the ith search. FSA* also maintains a value *BlockId*(i) for the ith search and the values *ExpandedId*(s) and *ExpandedIteration*(s) for every cell s, initialized as follows: *BlockId*$(0) = 0$ and, for all cells s, *ExpandedId*$(s) = 0$ and *ExpandedIteration*$(s) = 0$ {01,04-05}. FSA* then maintains the following invariant directly before the ith search: If *ExpandedId*$(s) <$ *BlockId*(*ExpandedIteration*(s)) then cell s would have been expanded as *ExpandedId*(s)th cell during the immediately preceeding search if it had been a complete search {28-30}. If FSA* determines that every cell s with *ExpandedId*$(s) < m$ is reusable for the *Iteration*th search, then it proceeds as follows: First, it invalidates all cells with sequence numbers no smaller than m by setting *BlockId*$(i) :=$ $\min(m,$ *BlockId*$(i))$ for all $1 \leq i \leq$ *Iteration*, which is much faster than visiting all cells with invalid sequence numbers {43-45}. Second, whenever it expands a cell s during the *Iteration*th search it sets *ExpandedIteration*$(s) :=$ *Iteration*, *ExpandedId*$(s) := m$ and $m := m + 1$ {15-17}. Finally, it sets *BlockId*(*Iteration*) $:= \infty$ {12} and *Iteration* $:=$ *Iteration* $+ 1$ {48}.

Step 2 (Early Termination): If the goal cell is reusable, then FSA* does not need to replan because the shortest path from the start cell to the goal cell from the immediately preceeding search is also a shortest path for the current search {75}. If the start cell is not reusable and blocked, then FSA* terminates without replanning because there is no path from the start cell to the goal cell. If the start cell is not reusable and unblocked, then FSA* performs a complete A* search from scratch.

Step 3 (Restoration of the OPEN list): The unblocked and non-reusable cells that border one or more reusable cells are in the new OPEN list. First, FSA* identifies the anchor cell, a non-reusable cell that borders one or more reusable cells, by following the parent pointers from the goal cell to the start cell until it transitions from a non-reusable cell to a reusable cell {50-53}.[2] This non-reusable cell then is the anchor cell. The second gridworld in Figure 2 illustrates this

[2] FSA* cannot follow the parent pointers from the goal cell to the start cell if no path from the start cell to the goal cell existed in the immediately preceeding search. Whether a path exists or not, it can always simply follow a path from the goal cell to the start cell that first greedily decreases the x and then the y distance to the start cell.

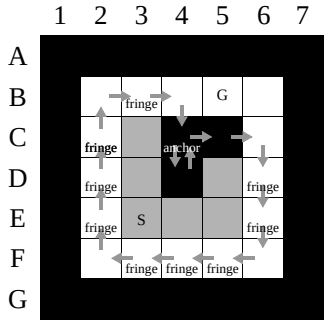


Figure 3: Search Problem 2

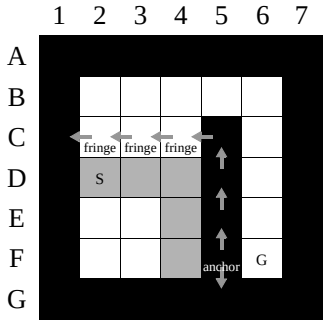
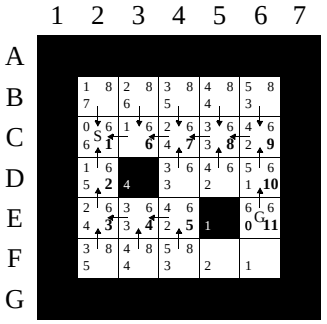


Figure 4: Search Problem 3



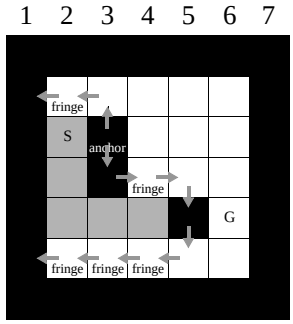


Figure 5: Search Problem 4

process for the search problem from Figure 1, resulting in anchor cell C5. Second, FSA* identifies the cells that belong to the relevant part of the new OPEN list (= fringe). The reusable cells form a continuous area since they are all reachable from the start cell. FSA* can traverse the immediate outside of this continuous area (say, in the clockwise direction), starting with the anchor cell, and insert all unblocked cells that border one or more reusable cells into the relevant part of the new OPEN list {54-66}. FSA* does not stop when it reaches the anchor cell a second time but only when it is about to leave the anchor cell a second time in the same direction. The third gridworld in Figure 2 illustrates this process for the search problem from Figure 1. The cells C2, C3 and B4 form the relevant part of the new OPEN list, as indicated in the figure. Figure 3 shows for a search problem different from our main one that the termination condition is indeed important. The start cell is E3 and the goal cell is B5. After cell C4 became blocked, FSA* visits cells C4, D4 and C4 again when it traverses the immediate outside of the continuous area of reusable cells, at which point it does not stop since the anchor cell C4 was first left towards the south but is now left towards the east. If it did stop, then it would not identify all cells that belong to the relevant part of the new OPEN list. Figure 4 shows for a search problem different from our main one that not all cells in the new OPEN list are relevant. The start cell is D2 and the goal cell is F6. After cell F5 became blocked, cells E2, E3 and F3 are irrelevant, and thus not included in the relevant part of the new OPEN list since the goal cell is not in the region of unblocked cells that they are part of and a shortest path from the start cell to a goal cell thus cannot pass through them. From now on, we just refer to the OPEN list when we mean the relevant part of the OPEN list.

Step 4 (Restoration of the G-Values and Parent Pointers): The g-values and parent pointers of all reusable cells are guaranteed to be correct. However, the g-values and parent pointers of cells in the new OPEN list are not necessarily correct in the following two cases: First, if a cell in the new OPEN list just became unblocked, then it was blocked before, and its g-value and parent pointer thus have not been updated in the preceeding searches and need to be corrected (Case 1). Second, if the parent pointer of a cell in the new OPEN list points to a cell that is not reusable, then the g-value and parent pointer need to be corrected (Case 2). To understand Case 2, consider a complete A* search and assume that the g-value and parent pointer of an unblocked cell changed during the A* search because the A* search first found a path from the start cell to the cell in question and then found a shorter path from the start cell to the cell in question. If the state of this A* search needs to be restored at a point in time after the first path was found but before the second path was found, then the g-value and parent pointer of the cell in question need to be corrected. Figure 5 (left and right) show for a search problem different from our main one that it is important to update the g-values and parent pointers of the cells in the new OPEN list. The start cell is C2 and the goal cell is E6. Figure 5 (left) shows the first A* search. Figure 5 (right) shows the situation after cell C3 became blocked. The g-value and parent pointer of cell D4 are incorrect and need to be corrected from 3 and C4, respectively, to 5 and E4. For Case 1 and 2, FSA* finds *any* reusable cell adjacent to the cell in question, sets the g-value of the cell in question to the g-value of the reusable cell plus one, and the parent pointer of the cell in question to the reusable cell {58-63}. It is correct for Case 1 since a cell in the new OPEN list that just became unblocked has exactly one adjacent reusable cell. We prove the correctness for Case 2 in the next section.

Step 5 (Sorting the new OPEN list): This step is important if the OPEN list is implemented as a binary heap because transforming a set in one step into a binary heap is more efficient than inserting the elements of the set one after the other into an empty binary heap [Corman *et al.*, 1990].

Step 6 (Restarting A*): Finally, FSA* restarts A* with the new OPEN and CLOSED lists {69}. A* is implemented by the function ComputeShortestPath.

5 Theoretical Results

The correctness of the restoration of the g-values and parent pointers of cells in the new OPEN list is not easy to see for Case 2. We thus prove it in the following, making use of the following theorem that is specific to the kinds of gridworlds used in this paper. The result also implies that the g-value and parent pointer of a cell in the new OPEN list are correct if the parent pointer already points to a cell that is reusable.

Theorem 1 *The parent pointer of any cell $s \in S$ can change at most once during an A* search from one cell to another. All expanded cells adjacent to a cell have the same g-value before the parent pointer of the cell changes.*

```
procedure Initialize()
{01} BlockId(0) := 0;
{02} Forall cells s
{03}     GeneratedIteration(s) := 0;
{04}     ExpandedIteration(s) := 0;
{05}     ExpandedId(s) := 0;
{06} m := 0;
{07} OPEN := ∅;
{08} g(s_start) := 0;
{09} OPEN.Insert(s_start);
{10} GeneratedIteration(s_start) := 1;
{11} Iteration := 1;

function ComputeShortestPath()
{12} BlockId(Iteration) := ∞;
{13} While (OPEN ≠ ∅)
{14}     s := OPEN.Pop();
{15}     ExpandedIteration(s) := Iteration;
{16}     ExpandedId(s) := m;
{17}     m := m + 1;
{18}     If (s = s_goal)
{19}         Return True;
{20}     Else
{21}         Forall s' ∈ Succ(s)
{22}             If (GeneratedIteration(s) ≠ Iteration Or g(s) + 1 < g(s'))
{23}                 g(s') := g(s) + 1;
{24}                 GeneratedIteration(s') := Iteration;
{25}                 Parent(s') := s;
{26}                 OPEN.Insert(s');
{27} Return False;

function CellReusable(s)
{28} If (ExpandedId(s) < BlockId(ExpandedIteration(s)))
{29}     Return True;
{30} Return False;

procedure UpdateMazeTraversability()
{31} TmpBlockId := ∞;
{32} Forall cells s whose traversability has changed
{33}     If (s is blocked)
{34}         If (CellReusable(s))
{35}             If (ExpandedId(s) < TmpBlockId)
{36}                 TmpBlockId := ExpandedId(s);
{37}     Else
{38}         Parent(s) := NULL;
{39}         Forall s' ∈ Succ(s)
{40}             If (CellReusable(s'))
{41}                 If (ExpandedId(s') + 1 < TmpBlockId)
{42}                     TmpBlockId := ExpandedId(s') + 1;
{43} Forall i = 1 . . . Iteration
{44}     If (TmpBlockId < BlockId(i))
{45}         BlockId(i) := TmpBlockId;
{46} m := BlockId(Iteration);

procedure RetrieveFringe()
{47} OPEN := ∅;
{48} Iteration := Iteration + 1;
{49} s := s_goal;
{50} While (Not CellReusable(Parent(s)))
{51}     s := Parent(s);
{52}     If (s = s_start)
{53}         Exit; /* there is no path */
{54} Move s around the area that contains exactly the cells s' with CellReusable(s')
{55}     Forall s' ∈ Succ(s) with CellReusable(s')
{56}         GeneratedIteration(s') := Iteration;
{57}     If (s is unblocked)
{58}         If (Parent(s) = NULL Or (Not CellReusable(Parent(s))))
{59}             Forall s' ∈ Succ(s)
{60}                 If (CellReusable(s'))
{61}                     Parent(s) := s';
{62}                     g(s) := g(s') + 1;
{63}                     break;
{64}         GeneratedIteration(s) := Iteration;
{65}         OPEN.Insert(s);
{66} Until the initial cell is about to be left in the same direction again;

procedure Main()
{67} Initialize();
{68} Repeat
{69}     If (Not ComputeShortestPath())
{70}         Exit; /* there is no path */
{71}     Repeat
{72}         Identify the path using the parent pointers and use it;
{73}         Wait for traversability changes;
{74}         UpdateMazeTraversability();
{75}     Until (BlockId(Iteration) ≤ ExpandedId(s_goal));
{76}     RetrieveFringe();
{77} Until False;
```

Figure 6: Fringe Saving A* (FSA*)

Proof: Consider any unblocked cell $s \in S$ and any two unblocked cells $s', s'' \in S$ adjacent to cell s. $h(s') - h(s'')$ is -2, 0 or 2 since the h-values are the Manhattan distances. This means that all cells adjacent to cell s together have at most two different h-values. Now assume that cell s' was expanded during an A* search and that cell s was generated during the expansion. Thus, the parent pointer of cell s was set to cell s'. Assume further that cell s'' was expanded later during the same A* search and that the parent pointer of cell s changed during the expansion. Thus, the parent pointer of cell s was set to cell s''. Then, $g(s')+h(s') \leq g(s'')+h(s'')$ since the sum of g-value and h-value of the sequence of expanded cells is monotonically nondecreasing over time. Furthermore, $g(s'') < g(s')$ since otherwise the parent pointer of cell s would not have changed. Thus, $g(s'') + c = g(s')$ for some $c > 0$. Put together, $g(s'') + c + h(s') = g(s') + h(s') \leq g(s'') + h(s'')$ or, alternatively, $h(s') - h(s'') \leq -c < 0$. Since $h(s') - h(s'')$ is -2, 0 or 2, it must be the case that $h(s') - h(s'') = -2$ and thus $h(s'') = h(s') + 2$. This implies that $-2 \leq -c < 0$. Since the start distances and thus the g-values of cells s' and s'' cannot differ by one, it must be the case that $g(s'') - g(s') = c = 2$. This means that the parent pointer of cell s can change at most once during an A* search because c increases with every change. Now consider any unblocked cell s''' adjacent to cell s that was expanded before the parent pointer of cell s changed. Then, $g(s')+h(s') \leq g(s''')+h(s''') \leq g(s'')+h(s'')$ since the sum of g-value and h-value of the sequence of expanded cells is monotonically nondecreasing over time. It holds that $g(s') + h(s') = g(s'') + 2 + h(s'') - 2 = g(s'') + h(s'')$. Thus, $g(s') + h(s') = g(s''') + h(s''') = g(s'') + h(s'')$. Since all cells adjacent to cell s together have only two different h-values, $h(s''')$ must equal either $h(s')$ or $h(s'')$. Thus, $g(s''')$ must equal either $g(s')$ or $g(s'')$. $g(s''')$ cannot equal $g(s'')$ since otherwise the parent pointer of cell s first had changed to cell s''' and later could no longer have changed to cell s''. Thus, $g(s''')$ must equal $g(s')$, which means that all expanded cells adjacent to cell s have the same g-value before the parent pointer of the cell changes. ■

We use this theorem as follows: During the course of an A* search, the g-value of a cell cannot increase. Thus, when restoring the (earlier) state of an A* search, the g-value of a cell cannot decrease. Assume that the parent pointer of a cell in the new OPEN list points to a cell that is reusable. The g-values of reusable cells are correct. Thus, the g-value of the cell in question does not need to increase. It remains the g-value of the reusable cell plus one. Thus, the g-value and parent pointer of the cell in question remain correct. Now assume that the parent pointer of a cell in the new OPEN list points to a cell that is not reusable. Clearly, the parent pointer needs to get corrected. Directly before it was set to the non-reusable cell during the immediately preceeding A* search, it pointed to some other cell with a larger g-value. At that point in time, all expanded cells adjacent to the cell in question had the same g-value according to Theorem 1, and the reusable cells adjacent to the cell in question are a subset of them since their sequence numbers are smaller than the sequence number of the non-reusable cell. FSA* can therefore find any reusable

cell adjacent to the cell in question and set the g-value of the cell in question to the g-value of the reusable cell plus one and the parent pointer of the cell in question to the reusable cell.

6 Related Work

Incremental A* (iA*) by Peter Yap (unpublished) reuses the beginning of the immediately preceeding A* search tree that is identical to the current A* search tree, like FSA*. It restores the content of the OPEN list of A* by repeating the A* search for the immediately preceeding search problem until it deviates from the A* search for the current search problem. Since the order of the cell expansions is already known, iA* does not need to use an OPEN list to determine the order of the cell expansions which makes the repeated A* search faster than the original one. FSA* restores the content of the OPEN list of A* without repeating the A* search for the immediately preceeding search problem. The main difference of both search algorithms is that iA* traverses the expanded cells while FSA* traverses the generated but not yet expanded cells. The expanded cells form a continuous area of cells, and the generated but not yet expanded cells basically form the outside perimeter of this area. Since an area can grow quadratically in the length of its perimeter, traversing the cells on the perimeter can potentially be much more efficient than traversing the cells in the area itself. There are three incremental versions of A* that operate according to principles different from FSA* and iA*. Adaptive A* [Koenig and Likhachev, 2006] runs A* unchanged but makes the h-values more informed. It cannot handle edge cost decreases and thus cannot always be used instead of FSA*. Lifelong Planning A* (LPA*) [Koenig *et al.*, 2004a] and Differential A* [Trovato and Dorst, 2002] leave the h-values unchanged but modify A*. They can handle edge cost increases and decreases and thus can be used instead of FSA*. They can be viewed as transforming the A* search tree of the immediately preceeding search into the A* search tree of the current search. We use LPA* instead of Differential A* because it is more advanced and available at idm-lab.org/project-a.html. LPA* can be slower than A* but tends to be more efficient than A* when the search problems change only slightly and the changes are close to the goal cell. It has been extended to moving agents and then used on robots as part of D* Lite [Koenig and Likhachev, 2005], which is a version of D* [Stentz, 1995] that is simpler to understand, implement and extend. We therefore compare FSA* against A*, iA* and LPA* experimentally.

7 Experimental Evaluation

We performed experiments in 100 gridworlds of size 1000×1000. Their start and goal cells were randomly chosen and *obstacle density* $\times 100$ percent of the cells were randomly chosen to be blocked, where *obstacle density* was between zero and one. For each gridworld, we changed the traversability of some cells and then found again a shortest path from the start cell to the goal cell. If no path from the start cell to the goal cell existed or after we had changed the traversability of cells and found a new path for 100 times, we continued with the next gridworld. (The runtime includes the gridworlds without a path from the start cell to the goal cell.) The pseudo code in Figure 6 exploits this assumption for simplicity, for example, does not handle the case where the start cell becomes unblocked. To maintain a constant obstacle density, we randomly changed the traversability of *change rate* $/2 \times 100$ percent of cells close to the goal cell from blocked to unblocked and an equal number of cells close to the goal cell from unblocked to blocked, where *change rate* was between zero and one. (This implies that *change rate* $\leq 2\times$ *obstacle density*.) A cell was close to the goal cell if its Manhattan distance was no larger than *closeness* times the Manhattan distance of the start cell, where *closeness* was between zero and one. This way of changing the terrain was inspired by real-time strategy games like Warcraft where one player often repeatedly attacks some other player. During these attacks, the attacking player tries to reach the location of the defending player. In the process, the attacking player destroys buildings of the defending player while the defending player builds new buildings. Thus, the traversability of the terrain changes around the destination of the attacking player, as simulated in our experiments. We compared the runtime of FSA* against the runtimes of A*, iA* and LPA* on a Pentium D 3.0 Ghz PC with 2 GByte of RAM. It is worth pointing out that runtime results always depend on the hardware, compiler and implementation, including the data structures, tie-breaking strategies and coding tricks used. However, there is currently no better testing methodology available since the three different search algorithms work according to very different principles and thus cannot be compared via proxies such as the number of expanded cells. All three search algorithms find shortest paths and, to be fair, were implemented in very similar ways. For example, they all used binary heaps to implement the OPEN list. When deciding which cell to expand next, they all broke ties among cells with the same smallest sum of g-value and h-value in favor of a cell with the largest g-value, since this tends to result in small runtimes. Table 1 reports their total runtimes over *all* searches they performed (including the calculation of the traversability changes and so on) for *obstacle density* = 0.25, showing the fastest runtime in bold. We say $X > Y$ iff the following relationship holds: "X runs consistently faster than Y for small values of *closeness*. On the other hand, Y runs faster than X for larger values of *closeness* if the value of *change rate* is sufficiently large. This threshold decreases as the value of *closeness* increases." The table then shows that FSA* $>$ A*, FSA* $>$ iA* and LPA* $>$ FSA*. We can explain these observations as follows: First, the value of *change rate* increases as we go from top to bottom in the table, which increases the number of traversability changes. The overhead of preprocessing each traversability change tends to be the smallest for A* (= none), followed by iA*, FSA* and LPA*, roughly in that order. This overhead gets amplified by the value of *change rate*. Second, the value of *closeness* increases as we go from left to right in the table, which decreases the part of the immediately preceeding A* search tree that can get reused. (Similarly, the value of *change rate* increases as we go from top to bottom in the table, which to some extent also decreases the part of the immediately preceeding A* search tree that can get reused.)

change	closeness = 0.1				closeness = 0.2				closeness = 0.3				closeness = 0.4				closeness = 0.5				closeness = 0.6			
rate	A*	iA*	LPA*	FSA*	A*	iA*	LPA*	FSA*	A*	iA*	LPA*	FSA*	A*	iA*	LPA*	FSA*	A*	iA*	LPA*	FSA*	A*	iA*	LPA*	FSA*
0.02	1.429	1.014	**0.260**	0.726	1.359	1.121	**0.352**	0.907	1.386	1.279	**0.492**	1.142	1.551	1.481	**0.723**	1.367	1.247	1.256	**0.930**	1.215	1.604	1.643	**1.348**	1.603
0.04	1.210	0.864	**0.293**	0.620	1.366	1.149	**0.443**	0.943	1.417	1.299	**0.628**	1.145	1.440	1.419	**1.058**	1.350	1.566	1.590	**1.382**	1.537	**1.683**	1.736	1.830	1.704
0.06	1.302	0.935	**0.342**	0.658	1.315	1.081	**0.525**	0.898	1.341	1.208	**0.814**	1.086	1.516	1.488	**1.272**	1.411	1.662	1.685	1.897	**1.640**	**1.948**	2.005	2.486	1.967
0.08	1.498	1.083	**0.400**	0.767	1.322	1.121	**0.588**	0.947	1.520	1.411	**1.016**	1.274	1.474	1.457	1.504	**1.390**	1.449	1.458	1.728	**1.409**	**1.889**	1.953	2.647	1.927
0.10	1.102	0.812	**0.406**	0.602	1.353	1.158	**0.664**	0.966	1.492	1.396	**1.068**	1.268	1.578	1.558	1.836	**1.484**	**1.882**	1.922	2.807	1.883	**2.013**	2.079	3.053	2.053
0.12	1.159	0.938	**0.448**	0.753	1.261	1.113	**0.707**	0.962	1.589	1.489	**1.269**	1.341	1.711	1.699	2.043	**1.637**	1.909	1.947	2.865	**1.906**	**2.056**	2.130	3.465	2.106
0.14	1.296	0.976	**0.487**	0.727	1.320	1.123	**0.752**	0.954	1.453	1.378	1.386	**1.272**	1.995	2.013	2.642	**1.942**	**2.075**	2.122	3.374	2.082	**2.064**	2.143	3.571	2.128
0.16	1.119	0.878	**0.449**	0.673	1.425	1.243	**0.906**	1.061	1.423	1.377	1.383	**1.291**	1.780	1.776	2.323	**1.707**	**2.045**	2.098	3.303	2.068	**2.609**	2.709	4.779	2.692
0.18	1.152	0.911	**0.482**	0.717	1.406	1.223	**0.903**	1.051	1.692	1.588	1.610	**1.444**	1.624	1.618	2.414	**1.570**	**2.148**	2.194	3.666	2.159	**2.535**	2.638	4.758	2.622
0.20	1.179	0.939	**0.478**	0.740	1.482	1.325	**1.017**	1.152	1.551	1.498	1.732	**1.403**	1.817	1.825	2.722	**1.762**	**2.187**	2.262	4.177	2.244	**3.295**	3.438	6.858	3.427
0.22	1.288	1.049	**0.567**	0.839	1.214	1.077	**0.854**	0.943	1.529	1.461	1.733	**1.355**	1.822	1.842	2.830	**1.799**	**2.846**	2.939	5.644	2.913	**2.601**	2.704	5.670	2.698
0.24	1.140	0.911	**0.523**	0.730	1.620	1.468	**1.295**	1.300	1.587	1.534	1.963	**1.447**	1.807	1.823	2.769	**1.775**	**2.737**	2.822	4.609	2.780	**2.753**	2.872	5.718	2.872
0.26	1.286	0.991	**0.550**	0.739	1.341	1.215	1.091	**1.083**	1.517	1.475	1.744	**1.393**	1.926	1.953	2.921	**1.900**	**2.754**	2.845	5.068	2.816	**3.054**	3.180	6.180	3.170
0.28	1.274	0.985	**0.568**	0.762	1.432	1.327	1.214	**1.201**	1.468	1.452	1.921	**1.381**	2.196	2.215	3.660	**2.157**	**2.443**	2.527	4.304	2.501	**3.546**	3.710	7.746	3.718
0.30	1.228	0.979	**0.588**	0.777	1.537	1.319	1.126	**1.125**	1.976	1.923	2.639	**1.808**	2.343	2.394	3.912	**2.341**	**3.079**	3.196	6.178	3.169	**3.528**	3.696	8.001	3.699
0.32	1.293	0.984	**0.554**	0.757	1.646	1.485	1.443	**1.313**	1.748	1.720	2.537	**1.638**	2.075	2.118	3.524	**2.074**	**3.042**	3.151	6.409	3.136	**3.863**	4.042	8.663	4.056
0.34	1.256	1.019	**0.571**	0.813	1.395	1.290	1.208	**1.152**	1.856	1.798	2.240	**1.685**	2.692	2.714	4.633	**2.639**	**2.698**	2.795	5.507	2.783	**3.958**	4.147	9.073	4.162
0.36	1.276	1.022	**0.631**	0.792	1.534	1.418	1.484	**1.285**	1.864	1.840	2.569	**1.751**	2.297	2.336	4.002	**2.283**	**3.140**	3.258	6.294	3.245	**3.801**	3.983	8.356	4.006
0.38	1.194	0.974	**0.602**	0.779	1.546	1.369	1.298	**1.189**	1.734	1.700	2.497	**1.613**	**2.284**	2.338	4.178	2.299	**3.178**	3.299	6.405	3.269	**3.670**	3.844	7.794	3.853
0.40	1.332	1.031	**0.627**	0.768	1.558	1.427	1.537	**1.279**	1.793	1.765	2.562	**1.673**	2.199	2.244	3.619	**2.198**	**2.651**	2.735	5.148	2.705	**3.847**	4.032	8.653	4.046
0.42	1.176	0.901	**0.586**	0.697	1.463	1.335	1.425	**1.203**	2.142	2.114	3.150	**2.011**	**2.590**	2.658	4.760	2.613	**3.129**	3.260	6.710	3.266	**4.436**	4.652	10.22	4.682
0.44	1.226	0.990	**0.627**	0.797	1.542	1.427	1.551	**1.287**	2.309	2.266	3.317	**2.163**	**1.978**	2.017	3.634	1.991	**4.006**	4.159	8.875	4.151	**5.080**	5.332	11.74	5.369
0.46	1.212	0.991	**0.637**	0.798	1.700	1.545	1.694	**1.367**	1.890	1.833	2.692	**1.728**	2.445	2.486	4.212	**2.430**	**3.938**	4.113	9.138	4.134	**4.291**	4.497	9.456	4.523
0.48	1.333	1.048	**0.692**	0.817	1.394	1.282	1.419	**1.146**	2.113	2.044	2.846	**1.927**	**3.104**	3.194	5.834	3.149	**3.841**	4.002	8.586	4.014	**4.098**	4.297	9.674	4.338
0.50	1.137	0.887	**0.578**	0.693	1.415	1.324	1.454	**1.210**	1.964	1.928	2.814	**1.846**	**2.852**	2.916	5.309	2.871	**3.715**	3.879	7.843	3.885	**4.909**	5.151	11.02	5.193

Table 1: Experimental Results (in seconds)

The part of the immediately preceeding A* search tree that actually gets reused tends to be the largest for LPA*, followed by iA*, FSA* (= same as for iA*) and A* (= none), roughly in that order. The overhead for identifying this part of the reusable A* search tree follows the same trend. Overall, iA* is never the fastest search algorithm. As the value of *closeness* increases, there is a decreasing threshold for the value of *change rate* below which LPA* is the fastest search algorithm and a larger decreasing threshold for the value of *change rate* above which A* is the fastest search algorithm. (LPA* becomes very slow with respect to all other search algorithms if the values of *change rate* and *closeness* are large.) Between the two thresholds, FSA* is the fastest search algorithm. Thus, our first feasibility study demonstrates that FSA* can find shortest paths faster than A*, iA* and LPA* in some situations and the principle behind it is thus worth being studied further.

8 Conclusions

In this paper, we developed Fringe-Saving A* (FSA*), an incremental version of A* that repeatedly finds shortest paths in a known gridworld from a given start cell to a given goal cell while the traversability costs of cells increase or decrease. Our first feasibility study demonstrates that FSA* can find shortest paths faster than A*, iA* and LPA* in some situations. It is future work to characterize these situations better and improve the efficiency of FSA* further.

Acknowledgments

We thank Peter Yap, with whom Sven Koenig collaborated in 2003 during a visit to the Alberta Ingenuity Centre for Machine Learning at the University of Alberta. This collaboration on comparing iA* and LPA* inspired our development of FSA* three years later, at a time where no one seems to know the current whereabouts of Peter. We also thank Ariel Felner for interesting discussions on FSA*. The Intelligent Decision-Making Group is partly supported by NSF awards to Sven Koenig under contracts IIS-0098807 and IIS-0350584. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

References

[Corman *et al.*, 1990] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, second edition, 1990.

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

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

[Koenig and Likhachev, 2006] S. Koenig and M. Likhachev. A new principle for incremental heuristic search: Theoretical results. In *Proceedings of the International Conference on Autonomous Planning and Scheduling*, pages 402–405, 2006.

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

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

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

[Stentz, 1995] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.

[Trovato and Dorst, 2002] K. Trovato and L. Dorst. Differential A*. *IEEE Transactions on Knowledge and Data Engineering*, 14:1218–1229, 2002.