# Fringe Search: Beating A* at Pathfinding on Game Maps

**Yngvi Björnsson**
School of Computer Science
Reykjavik University
Reykjavik, Iceland IS-103
yngvi@ru.is

**Markus Enzenberger, Robert C. Holte and Jonathan Schaeffer**
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{emarkus,holte,jonathan}@cs.ualberta.ca

**Abstract- The A* algorithm is the *de facto* standard used for pathfinding search. IDA* is a space-efficient version of A*, but it suffers from cycles in the search space (the price for using no storage), repeated visits to states (the overhead of iterative deepening), and a simplistic left-to-right traversal of the search tree. In this paper, the *Fringe Search* algorithm is introduced, a new algorithm inspired by the problem of eliminating the inefficiencies with IDA*. At one extreme, the Fringe Search algorithm expands frontier nodes in the exact same order as IDA*. At the other extreme, it can be made to expand them in the exact same order as A*. Experimental results show that Fringe Search runs roughly 10-40% faster than highly-optimized A* in our application domain of pathfinding on a grid.**

## 1 Introduction

Pathfinding is a core component of many intelligent agent applications, ranging in diversity from commercial computer games to robotics. The ability to have autonomous agents maneuver effectively across physical/virtual worlds is a key part of creating intelligent behavior. However, for many applications, especially those with tight real-time constraints, the computer cycles devoted to pathfinding can represent a large percentage of the CPU resources. Hence, more efficient ways for addressing this problem are needed.

Our application of interest is grid-based pathfinding. An agent has to traverse through a two-dimensional world. Moving from one location in the grid to another has a cost associated with it. The search objective is to travel from a $start$ location in the grid to a $goal$ location, and do so with the minimum cost. In many commercial computer games, for example, pathfinding is an essential requirement for computer agents (NPCs; non-player characters) [Stout, 1996]. For real-time strategy games, there may be hundreds of agents interacting at a time, each of which may have pathfinding needs. Grid-based pathfinding is also at the heart of many robotics applications, where the real-time nature of the applications requires expensive pathfinding planning and re-planning [Stentz, 1995].

A* [Hart et al., 1968] and IDA* [Korf, 1985] (and their variants) are the algorithms of choice for single-agent optimization search problems. A* does a best-first search; IDA* is depth-first. A* builds smaller search trees than IDA* because it benefits from using storage (the Open and Closed Lists), while IDA* uses storage which is only linear in the length of the shortest path length. A*'s best-first search does not come for free; it is expensive to maintain the Open List in sorted order. IDA*'s low storage solution also does not come for free; the algorithm ends up re-visiting nodes many times. On the other hand, IDA* is simple to implement, whereas fast versions of A* require a lot of effort to implement.

IDA* pays a huge price for the lack of storage; for a search space that contains cycles or repeated states (such as a grid), depth-first search ends up exploring all distinct paths to that node. Is there any hope for the simplicity of a depth-first search for these application domains?

A* has three advantages over IDA* that allows it to build smaller search trees:

1. IDA* cannot detect repeated states, whereas A* benefits from the information contained in the Open and Closed Lists to avoid repeating search effort.

2. IDA*'s iterative deepening results in the search repeatedly visiting states as it reconstructs the *frontier* (leaf nodes) of the search. A*'s Open List maintains the search frontier.

3. IDA* uses a left-to-right traversal of the search frontier. A* maintains the frontier in sorted order, expanding nodes in a best-first manner.

The first problem has been addressed by a transposition table [Reinefeld and Marsland, 1994]. This fixed-size data structure, typically implemented as a hash table, is used to store search results. Before searching a node, the table can be queried to see if further search at this node is needed.

The second and third problems are addressed by introducing the *Fringe Search* algorithm as an alternative to IDA* and A*. IDA* uses depth-first search to construct the set of leaf nodes (the frontier) to be considered in each iteration. By keeping track of these nodes, the overhead of iterative deepening can be eliminated. Further, the set of frontier nodes do not have to be kept in sorted order. At one extreme, the Fringe Search algorithm expands frontier nodes in the exact same order as IDA* (keeping the frontier in a left-to-right order). At the other extreme, it can be made to expand them in the exact same order as A* (through sorting).

This paper makes the following contributions to our understanding of pathfinding search:

1. Fringe Search is introduced, a new algorithm that spans the space/time trade-off between A* and IDA*.

2. Experimental results evaluating A*, memory-enhanced IDA*, and Fringe Search. In our test domain, pathfinding in computer games, Fringe Search is shown to run significantly faster than a

highly optimized version of A*, even though it examines considerably more nodes.

3. Insights into experimental methodologies for comparing search algorithm performance. There is a huge gap in performance between "textbook" A* and optimized A*, and a poor A* implementation can lead to misleading experimental conclusions.

4. Fringe Search is generalized into a search framework that encompasses many of the important single-agent search algorithms.

Section 2 motivates and discusses the Fringe Search algorithm. Section 3 presents experimental results comparing A*, memory-enhanced IDA*, and Fringe Search. Section 4 illustrates the generality of the Fringe Search idea by showing how it can be modified to produce other well-known single-agent search algorithms. Section 5 presents future work and the conclusions.

## 2 The Fringe Search Algorithm

We use the standard single-agent notation: $g$ is the cost of the search path from the $start$ node to the current node; $h$ is the heuristic estimate of the path cost from the current node to the $goal$ node; $f = g + h$; and $h^*$ is the real path cost to the $goal$.

Consider how IDA* works. There is a starting threshold ($h(root)$). The algorithm does a recursive left-to-right depth-first search, where the recursion stops when either a goal is found or a node is reached that has an $f$ value bigger than the threshold. If the search with the given threshold does not find a goal node, then the threshold is increased and another search is performed (the algorithm iterates on the threshold).

IDA* has three sources of search inefficiency when compared to A*. Each of these is discussed in turn.

### 2.1 Repeated states

When pathfinding on a grid, where there are multiple paths (possibly non-optimal) to a node, IDA* flounders [Korf and Zhang, 2000]. The repeated states problem can be solved using a transposition table [Reinefeld and Marsland, 1994] as a cache of visited states. The table is usually implemented as a (large) hash table to minimize the cost of doing state look-ups. Each visit to a state results in a table look-up that may result in further search for the current sub-tree being proven unnecessary.

A transposition table entry can be used to store the minimal $g$ value for this state and the backed-up $f$ value obtained from searching this state. The $g$ values can be used to eliminate provably non-optimal paths from the search. The $f$ values can be used to show that additional search at the node for the current iteration threshold is unnecessary. In this paper, IDA* supplemented with a transposition table is called Memory-Enhanced IDA* (ME-IDA*).

### 2.2 Iterating

Each IDA* iteration repeats *all* the work of the previous iteration. This is necessary because IDA* uses essentially no storage.

Consider Figure 1: each branch is labeled with a path cost (1 or 2) and the heuristic function $h$ is the number of moves required to reach the bottom of the tree (each move has an admissible cost of 1). The left column illustrates how IDA* works. IDA* starts out with a threshold of $h(start) = 4$. Two nodes are expanded (black circles) and two nodes are visited (gray circles) before the algorithm proves that no solution is possible with a cost of 4. An expanded node has its children generated. A visited node is one where no search is performed because the $f$ value exceeds the threshold. The $f$ threshold is increased to 5, and the search starts over. Each iteration builds a depth-first search, starting at $start$, recursing until the threshold is exceeded or $goal$ is found. As the figure illustrates, all the work of the previous iteration $i$ must be repeated to reconstruct the $frontier$ of the search tree where iteration $i + 1$ can begin to explore new nodes. For domains with a small branching factor, the cost of the iterating can dominate the overall execution time. In this example, a total of 17 nodes have to be expanded ($start$ gets expanded 3 times!) and 27 nodes are visited.

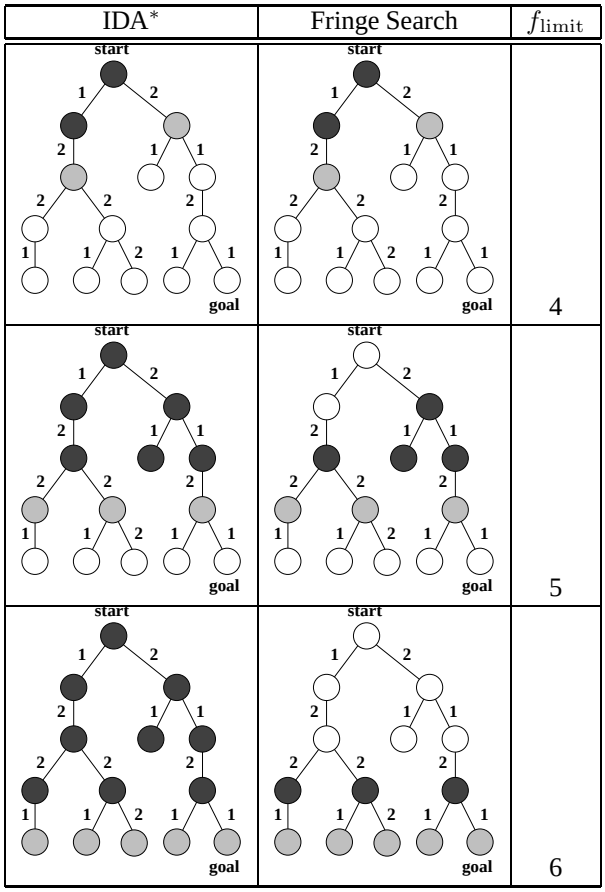


Figure 1: Comparison of IDA$^*$ and Fringe Search on an example graph. Visited nodes (gray) and expanded nodes (black) are given for each iteration

There exist IDA* variants, such as IDA*_CR [Sarkar et al., 1991], that can reduce the number of iterations. However, IDA* could be further improved if the repeated states overhead of iterating could be eliminated all together. This can be done by saving the leaf nodes (the $frontier$) of the iteration $i$ search tree and use it as the starting basis for iteration $i + 1$.

The middle column of Figure 1 illustrates how this algorithm works. It starts with $start$ and expands it exactly as in IDA*. The two leaf nodes of the first iteration are saved, and are then used as the starting point for the second iteration. The second iteration has 3 leaf nodes that are used for the third iteration. For the last iteration, IDA* has to visit every node in the tree; the new algorithm only visits the parts that have not yet been explored. In this example, a total of 9 nodes are expanded and 19 nodes are visited. Given that expanded nodes are considerably more expensive than visited nodes, this represents a substantial improvement over IDA*.

This new algorithm is called the Fringe Search, since the algorithm iterates over the fringe (frontier) of the search tree[1]. The data structure used by Fringe Search can be thought of as two lists: one for the current iteration ($now$) and one for the next iteration ($later$). Initially the $now$ list starts off with the root node and the $later$ list is empty. The algorithm repeatedly does the following. The node at the head of the $now$ list ($head$) is examined and one of the following actions is taken:

1. If $f(head)$ is greater than the threshold then $head$ is removed from $now$ and placed at the end of $later$. In other words, we do not need to consider $head$ on this iteration (we only visited $head$), so we save it for consideration in the next iteration.

2. If $f(head)$ is less or equal than the threshold then we need to consider $head$'s children (expand $head$). Add the children of $head$ to the front of $now$. Node $head$ is discarded.

When an iteration completes and a goal has not been found, then the search threshold is increased, the $later$ linked list becomes the $now$ list, and $later$ is set to empty.

When a node is expanded, the children can be added to the $now$ list in any order. However, if they are inserted at the front of the list and the left-to-right ordering of the children is preserved (the left-most child ends up at the front of the list), then the Fringe Search expands nodes *in the exact same order as IDA**. The children can be added in different ways, giving rise to different algorithms (see the following section).

### 2.3 Ordering

IDA* uses a left-to-right traversal of the search frontier. A* maintains the frontier in sorted order, expanding nodes in a best-first manner.

[1]Note that Frontier Search would be a better name, but that name has already been used [Korf and Zhang, 2000].

The algorithm given above does no sorting — a node is either in the current iteration ($now$) or the next ($later$). The Fringe Search can be modified to do sorting by having multiple $later$ buckets. In the extreme, with a bucket for every possible $f$ value, the Fringe will result in the same node expansion order as A*. The Fringe Search algorithm does not require that its $now$ list be ordered. At one extreme one gets the IDA* left-to-right ordering (no sorting) and at the other extreme one gets A*'s best-first ordering (sorting). In between, one could use a few buckets and get partial ordering that would get most of the best-first search benefits but without the expensive overhead of A* sorting.

### 2.4 Discussion

The Fringe Search algorithm essentially maintains an Open List (the concatenation of $now$ and $later$ in the previous discussion). This list does not have to be kept in sorted order, a big win when compared to A* where maintaining the Open List in sorted order can dominate the execution cost of the search. The price that the Fringe Search pays is that it has to re-visit nodes. Each iteration requires traversing the entire $now$ list. This list may contain nodes whose $f$ values are such that they do not have to be considered until much later (higher thresholds). A* solves this by placing nodes with bad $f$ values at/near the end of the Open List. Thus, the major performance differences in the two algorithms can be reduced to three factors:

1. Fringe Search may visit nodes that are irrelevant for the current iteration (the cost for each such node is a small constant),

2. A* must insert nodes into a sorted Open List (the cost of the insertion can be logarithmic in the length of the list), and

3. A*'s ordering means that it is more likely to find a goal state sooner than the Fringe Search.

The relative costs of these differences dictates which algorithm will have the better performance.

### 2.5 Implementation

The pseudo-code for the Fringe Search algorithm is shown in Figure 2. Several enhancements have been done to make the algorithm run as fast as possible (also done for A* and ME-IDA*). The $now$ and $later$ lists are implemented as a single double-linked list, where nodes in the list before the current node under consideration are the $later$ list, and the rest is the $now$ list. An array of pre-allocated list nodes for each node in the grid is used, allowing constant access time to nodes that are known to be in the list. An additional marker array is used for constant time look-up to determine whether some node is in the list. The $g$ value and iteration number (for ME-IDA*) cache is implemented as a perfect hash table. An additional marker array is used for constant time look-up to determine whether a node has already been visited and for checking whether an entry in the cache is

```
Initialize:
    Fringe F ← (s)
    Cache C[start] ← (0, null),
    C[n] ← null for n ≠ start
    f_limit ← h(start)
    found ← false
Repeat until found = true or F empty
    f_min ← ∞
    Iterate over nodes n ∈ F from left to right:
        (g, parent) ← C[n]
        f ← g + h(n)
        If f > f_limit
            f_min ← min(f, f_min)
            continue
        If n = goal
            found ← true
            break
        Iterate over s ∈ successors(n) from right to left:
            g_s ← g + cost(n, s)
            If C[s] ≠ null
                (g', parent) ← C[s]
                If g_s ≥ g'
                    continue
            If F contains s
                Remove s from F
            Insert s into F after n
            C[s] ← (g_s, n)
        Remove n from F
    f_limit ← f_min
If found = true
    Construct path from parent nodes in cache
```

Figure 2: Pseudo-code for Fringe Search

valid. As for ME-IDA* (see below), the marker arrays are implemented with a constant time clear operation.

If the successors of a node $n$ are considered from right-to-left, then they get added to the Fringe list $F$ such that the left-most one ends up immediately after $n$. This will give the same order of node expansions as IDA*.

## 3 Experiments

In this section we provide an empirical evaluation of the Fringe Search algorithm, as well as comparing its performance against that of both Memory-Enhanced IDA* (ME-IDA*) and A*. The test domain is pathfinding in computer-game worlds.

### 3.1 Algorithm Implementation Details

For a fair running-time comparison we need to use the "best" implementation of each algorithm. Consequently, we invested a considerable effort into optimizing the algorithms the best we could, in particular, by use of efficient data structures. For example, the state spaces for game and robotics grids are generally small enough to comfortably fit into the computer's main memory. Our implementations take advantage of this by using a lookup table that provides a constant-time access to all state information. Additional algorithm-dependent implementation/optimization details are listed below. It is worth mentioning that the most natural data structures for implementing both Fringe Search and ME-IDA* are inherently simple and efficient, whereas optimizing A* for maximum efficiency is a far more involved task.

#### 3.1.1 A$^*$ Implementation.

The Open List in $A^*$ is implemented as a balanced binary tree sorted on $f$ values, with tie-breaking in favor of higher $g$ values. This tie-breaking mechanism results in the $goal$ state being found on average earlier in the last $f$-value pass. In addition to the standard Open/Closed Lists, marker arrays are used for answering (in constant time) whether a state is in the Open or Closed List. We use a "lazy-clearing" scheme to avoid having to clear the marker arrays at the beginning of each search. Each pathfinding search is assigned a unique (increasing) *id* that is then used to label array entries relevant for the current search. The above optimizations provide an order of magnitude performance improvement over a standard "textbook" $A^*$ implementation.

#### 3.1.2 ME-IDA* Implementation.

Memory-Enhanced IDA* uses a transposition table that is large enough to store information about all states. The table keeps track of the length of the shortest path found so far to each state ($g$ value) and the backed-up $f$-value. There are three advantages to storing this information. First, a node is re-expanded only if entered via a shorter path ($g(s) < g_{cached}(s)$). Second, by storing the minimum backed-up $f$-value in the table (that is, the minimum $f$-value of all leaves in the sub-tree), the algorithm can detect earlier when following a non-optimal path (e.g., paths that lead to a dead-end). Combining the two caching strategies can drastically reduce the number of nodes expanded/visited in each iteration.

There is also a third benefit that, to the best of our knowledge, has not been reported in the literature before. When using non-uniform edge costs it is possible that ME-IDA* reduces the number of iterations by backing up a larger $f$-limit bound for the next iteration. We show an example of this in Figure 3. The current iteration is using a limit of 10. In the tree to the left transpositions are not detected (A and A' are the same node). Nodes with a $f$-value of 10 and less are expanded. The minimum $f$-value of the children is then backed up as the limit for the next iteration, in this case $min(12, 14)$ or 12. In the tree to the right, however, we detect that A' is a transposition into A via an inferior path, and we can thus safely ignore it (back up a $f$-value of $\inf$ to B). The $f$-limit that propagates up for use in the next iteration will now be 14.

### 3.2 Testing Environment

We extracted 120 game-world maps from the popular fantasy role-playing game *Baldur's Gate II* by *Bioware Inc.*

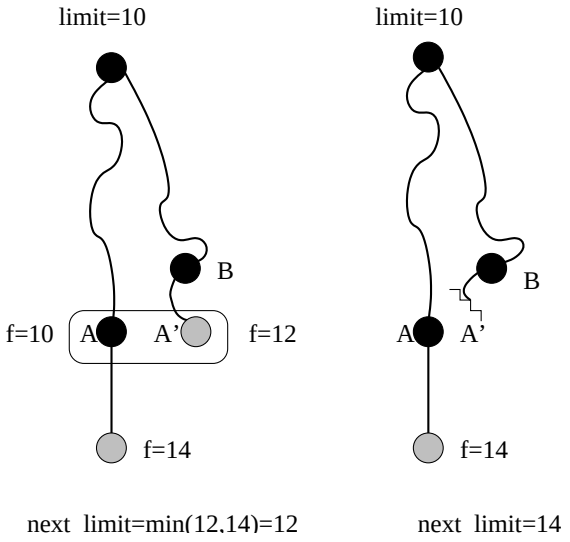


Figure 3: Caching reducing the number of iterations

Rectangular grids were superimposed over the maps to form discrete state spaces ranging in size from $50 \times 50$ to $320 \times 320$ cells, depending on the size of the game worlds (with an average of approximately $110 \times 110$ cells). A typical map is shown in Figure 4.

For the experiments we use two different grid-movement models: *tiles*, where the agent movement is restricted to the four orthogonal directions ($move\ cost = 100$), and *octiles*, where the agent can additionally move diagonally ($move\ cost = 150$). To better simulate game worlds that use variable terrain costs we also experiment with two different obstacle models: one where obstacles are impassable, and the other where they can be traversed, although at threefold the usual cost. As a heuristic function we used the minimum distance as if traveling on an obstacle-free map (e.g. Manhattan-distance for tiles). The heuristic is both admissible and consistent.

On each of the 120 maps we did 400 independent pathfinding searches between randomly generated start and goal locations, resulting in a total of 48,000 data points for each algorithm/model. We ran the experiments on a 1GHz Pentium III computer (a typical minimum required hardware-platform for newly released computer games), using a recently developed framework for testing pathfinding algorithms [Björnsson et al., 2003].

### 3.3 Fringe Search vs. ME-IDA*

The main motivation for the Fringe Search algorithm was to eliminate IDA*'s overhead of re-expanding the internal nodes in each and every iteration. Figure 5 shows the number of nodes expanded and visited by Fringe Search relative to that of ME-IDA* (note that the IDA* results are not shown; most runs did not complete). The graphs are plotted against the initial heuristic estimate error, that is, the difference between the actual and the estimated cost from *start* to *goal* ($h^*(start) - h(start)$). In general, the error increases as the game maps get more sophisticated (larger and/or more obstacles). We can see that as the heuristic

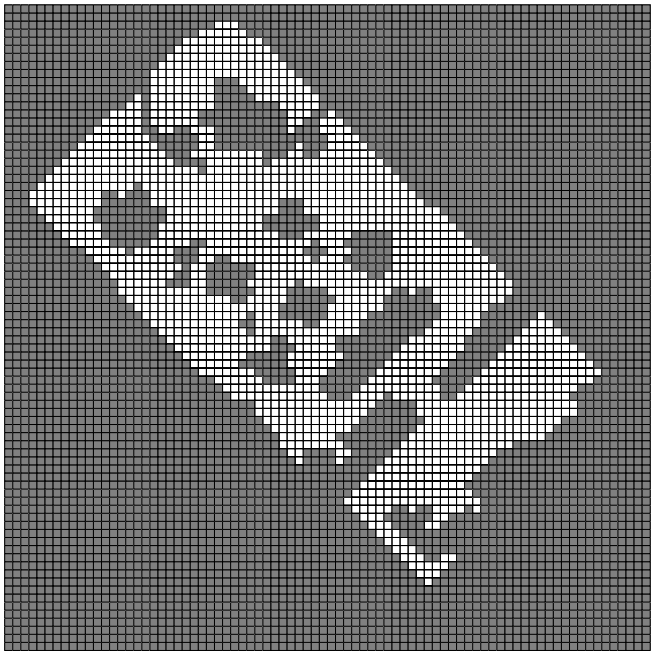

Figure 4: Example map

error increases, the better the Fringe Search algorithm performs relative to ME-IDA*. This can be explained by the observation that as the error increases, so will the number of iterations that *IDA** does. The data presented in Tables 1 and 2 allows us to compare the performance of the algorithms under the different pathfinding models we tested. They are based on pathfinding data on maps with impassable and passable obstacles, respectively. The tables give the CPU time (in milliseconds), iterations (number of times that the search threshold changed), visited (number of nodes visited), visited-last (number of nodes visited on the last iteration), expanded (number of nodes expanded), expanded-last (number of nodes expanded on the last iteration), path cost (the cost of the optimal path found), and path length (the number of nodes along the optimal path).

The data shows that Fringe Search is substantially faster than ME-IDA* under all models (by roughly an order of magnitude). The savings come from the huge reduction in visited and expanded nodes.

### 3.4 Fringe Search vs. A*

The A* algorithm is the *de facto* standard used for pathfinding search. We compared the performance of our new algorithm with that of a well-optimized version of $A^*$. As we can seen from Tables 1 and 2, both algorithms expand comparable number of nodes (the only difference is that because of its $g$-value ordering $A^*$ finds the target a little earlier in the last iteration). Fringe Search, on the other hand, visits many more nodes than A*. Visiting a node in Fringe Search is an inexpensive operation, because the algorithm iterates over the node-list in a linear fashion. In contrast A* requires far more overhead per node because of the extra work needed to maintain a sorted order. Time-wise the Fringe Search algorithm outperforms A* by a significant

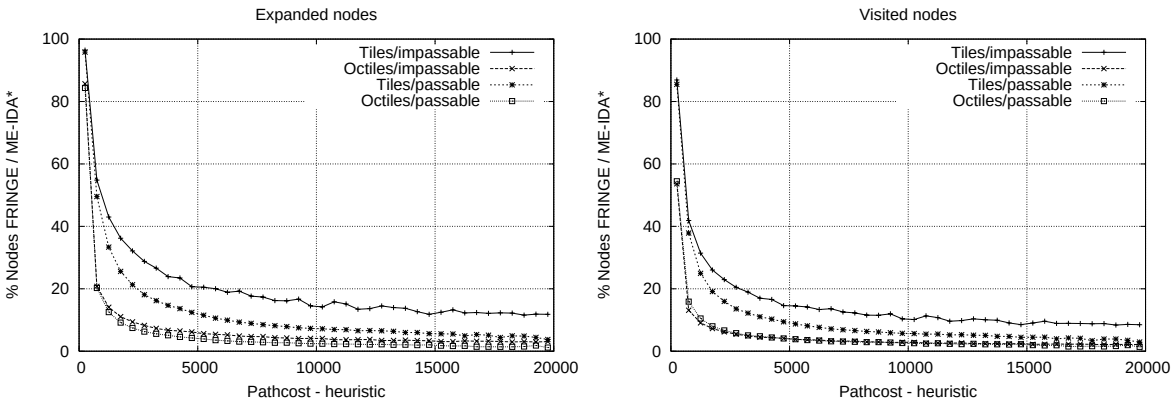


Figure 5: Comparison of nodes expanded/visited by Fringe Search vs. ME-IDA*

Table 1: Pathfinding statistics for impassable obstacles

| | Octiles | | | Tiles | | |
|---|---|---|---|---|---|---|
| | A* | Fringe | ME-IDA* | A* | Fringe | ME-IDA* |
| CPU/msec | 1.7 | 1.3 | 33.7 | 1.2 | 0.8 | 5.6 |
| Iterations | 25.8 | 25.8 | 26.9 | 9.2 | 9.2 | 9.2 |
| N-visited | 583.4 | 2490.7 | 91702.9 | 607.0 | 1155.3 | 11551.3 |
| N-visited-last | 27.7 | 79.5 | 620.1 | 54.5 | 103.8 | 276.9 |
| N-expanded | 582.4 | 586.5 | 14139.1 | 606.0 | 613.2 | 4327.6 |
| N-expanded-last | 26.7 | 30.7 | 115.9 | 53.5 | 60.7 | 127.8 |
| P-Cost | 5637.7 | 5637.7 | 5637.7 | 6758.6 | 6758.6 | 6758.6 |
| P-Length | 46.1 | 46.1 | 46.1 | 68.6 | 68.6 | 68.6 |

margin, running 25%-40% faster on average depending on the model.

Note that under the passable obstacle model, there is a small difference in the path lengths found by A* and Fringe/ME-IDA*. This is not a concern as long as the costs are the same (a length of a path is the number of grid cells on the path, but because under this model the cells can have different costs it is possible that two or more different length paths are both optimal cost-wise).

Our implementation of Fringe Search is using the IDA* order of expanding nodes. Using buckets, Fringe Search could do partial or even full sorting, reducing or eliminating A*'s best-first search advantage. The expanded-last row in Tables 1 and 2 shows that on the last iteration, Fringe Search expands more nodes (as expected). However, the difference is small, meaning that for this application domain, the advantages of best-first search are insignificant.

The ratio of nodes visited by Fringe Search versus A* is different for each model used. For example, in the impassable and passable obstacles model these ratios are approximately 4 and 6, respectively. It is of interest to note that a higher ratio does not necessarily translate into worse relative performance for Fringe Search; in both cases the relative performance gain is the same, or approximately 25%. The reason is that there is a "hidden" cost in A* not reflected in the above statistics, namely as the Open List gets larger so will the cost of maintaining it in a sorted order.

## 4 Related Algorithms

The Fringe Search algorithm can be seen either as a variant of A* or as a variant of IDA*.

Regarded as a variant of A*, the key idea in Fringe Search is that the Open List does not need to be fully sorted. The essential property that guarantees optimal solutions are found is that a state with an $f$-value exceeding the largest $f$-value expanded so far must not be expanded unless there is no state in the Open List with a smaller $f$-value. This observation was made in [Bagchi and Mahanti, 1983], which introduced a family of A*-like algorithms based on this principle, maintaining a record of the largest $f$-value of nodes expanded to date. This value, which we will call *Bound*, is exactly analogous to the cost bound used in iterative-deepening ($f$-limit); it plays the same role as and is updated in an identical manner.

Analogous to the FOCAL technique (pp. 88-89, [Pearl, 1984]) it is useful to distinguish the nodes in the Open List that have a value less than or equal to *Bound* from those that do not. The former are candidates for selection to be expanded in the present iteration, the latter are not.

This is a family of algorithms, not an individual algorithm, because it does not specify how to choose among the candidates for expansion, and different A*-like algorithms can be created by changing the selection criterion. For example, A* selects the candidate with the minimum $f$-value. By contrast algorithm C [Bagchi and Mahanti, 1983] selects the candidate with the minimum $g$-value.

Table 2: Pathfinding statistics for passable obstacles

| | Octiles | | | Tiles | | |
|---|---|---|---|---|---|---|
| | A* | Fringe | ME-IDA* | A* | Fringe | ME-IDA* |
| CPU/msec | 2.5 | 1.9 | 52.2 | 1.9 | 1.1 | 11.6 |
| Iterations | 14.2 | 14.2 | 14.2 | 5.0 | 5.0 | 5.0 |
| N-visited | 741.7 | 4728.3 | 132831.0 | 800.5 | 1828.3 | 23796.2 |
| N-visited-last | 27.0 | 119.7 | 1464.2 | 54.8 | 143.2 | 817.0 |
| N-expanded | 740.7 | 748.4 | 18990.1 | 799.5 | 816.3 | 7948.1 |
| N-expanded-last | 26.0 | 33.7 | 226.1 | 53.8 | 70.6 | 290.1 |
| P-Cost | 5000.6 | 5000.6 | 5000.6 | 5920.1 | 5920.1 | 5920.1 |
| P-Length | 39.5 | 39.5 | 39.5 | 56.3 | 56.5 | 56.5 |

If the heuristic being used is admissible, but not consistent, A* can do an exponential number of node expansions in the worst case, even if ties are broken as favorably as possible [Martelli, 1977]. By contrast, C can do at most a quadratic number of expansions, provided that ties in its selection criterion are broken in favor of the goal (but otherwise any tie-breaking rule will do)[2]. If the heuristic is not admissible, C maintains this worst-case speed advantage and in some circumstances finds superior solutions to A*.

Fringe Search is also a member of this family. It chooses the candidate that was most recently added. This gives it a depth-first behaviour mimicking IDA*'s order of node generation.

Among the variants of IDA*, Fringe Search is most closely related to ITS [Ghosh et al., 1994]. ITS is one of several "limited memory" search algorithms which aim to span the gap between A* and IDA* by using whatever memory is available to reduce the duplicated node generations that make IDA* inefficient. As [Ghosh et al., 1994] points out, ITS is the only limited memory algorithm which is not "best first". SMA* [Russell, 1992], which is typical of the others, chooses for expansion the "deepest, least-$f$-cost node". ITS, by contrast, is left-to-right depth first, just like IDA* and Fringe Search. New nodes are inserted into a data structure representing the search tree, and the node chosen for expansion is the deepest, left-most node whose $f$-value does not exceed the current cost bound. ITS requires the whole tree structure in order to retract nodes if it runs out of memory. Because Fringe Search assumes enough memory is available, it does not need the tree data structure to support its search, it needs only a linked list containing the leaf (frontier) nodes and a compact representation of the closed nodes.

## 5 Conclusions

Large memories are ubiquitous, and the amount of memory available will only increase. The class of single-agent search applications that need fast memory-resident solutions will only increase. As this paper shows, in this case, A* and IDA* are not the best choices for some applications. For example, Fringe Search out-performs optimized versions of A* and ME-IDA* by significant margins when pathfinding on grids typical of computer-game worlds. In comparison to ME-IDA*, the benefits come from avoiding repeatedly expanding interior nodes; compared to A*, Fringe Search avoids the overhead of maintaining a sorted open list. Although visiting more nodes than A* does, the low overhead per node visit in the Fringe Search algorithm results in an overall improved running time.

Since we ran the above experiments we have spent substantial time optimizing our already highly-optimized A* implementation even further. Despite of all that effort A* is still not competitive to our initial Fringe implementation, although the gap has closed somewhat (the speedup is ca. 10% for octiles and 20% for tiles). This is a testimony of one of Fringe search greatest strengths, its simplicity.

As for future work, one can possibly do better than Fringe Search. Although the algorithm is asymptotically optimal with respect to the size of the tree explored (since it can mimic A*), as this paper shows there is much to be gained by a well-crafted implementation. Although our implementation strove to be cache conscious, there still may be performance gains to be had with more cache-friendly data structures (e.g., [Niewiadomski et al., 2003]). Also, our current implementation of the Fringe Search algorithm traverses the fringe in exactly the same left-to-right order as IDA* does. Fringe Search could be modified to traverse the fringe in a different order, for example, by using buckets to partially sort the fringe. Although our experimental data suggests that this particular application domain is unlikely to benefit much from such an enhancement, other domains might.

## Acknowledgments

This research was supported by grants from the Natural Sciences and Engineering Council of Canada (NSERC) and Alberta's Informatics Center of Research Excellence (iCORE). This work benefited from the discussions of the University of Alberta Pathfinding Research Group, including Adi Botea and Peter Yap. The research was inspired by the needs of our industrial partners Electronic Arts (Canada) and Bioware, Inc.

[2] "Exponential" and "quadratic" are in terms of the parameter $N$ defined in [Bagchi and Mahanti, 1983].

## Bibliography


[Bagchi and Mahanti, 1983] Bagchi, A. and Mahanti, A. (1983). Search algorithms under different kinds of heuristics – a comparative study. *Journal of the Association of Computing Machinery*, 30(1):1–21.

[Björnsson et al., 2003] Björnsson, Y., Enzenberger, M., Holte, R., Schaeffer, J., and Yap., P. (2003). Comparison of different abstractions for pathfinding on maps. In *International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1511–1512.

[Ghosh et al., 1994] Ghosh, S., Mahanti, A., and Nau, D. S. (1994). ITS: An efficient limited-memory heuristic tree search algorithm. In *National Conference on Artificial Intelligence (AAAI'94)*, pages 1353–1358.

[Hart et al., 1968] Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107.

[Korf, 1985] Korf, R. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109.

[Korf and Zhang, 2000] Korf, R. and Zhang, W. (2000). Divide-and-conquer frontier search applied to optimal sequence alignment. In *National Conference on Artificial Intelligence (AAAI'02)*, pages 910–916.

[Martelli, 1977] Martelli, A. (1977). On the complexity of admissible search algorithms. *Artificial Intelligence*, 8:1–13.

[Niewiadomski et al., 2003] Niewiadomski, R., Amaral, N., and Holte, R. (2003). Crafting data structures: A study of reference locality in refinement-based path finding. In *International Conference on High Performance Computing (HiPC)*, number 2913 in Lecture Notes in Computer Science, pages 438–448. Springer.

[Pearl, 1984] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison & Wesley.

[Reinefeld and Marsland, 1994] Reinefeld, A. and Marsland, T. (1994). Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:701–710.

[Russell, 1992] Russell, S. J. (1992). Efficient memory-bounded search methods. In *European Conference on Artificial Intelligence (ECAI'92)*, pages 1–5, Vienna, Austria. Wiley.

[Sarkar et al., 1991] Sarkar, U., Chakrabarti, P., Ghose, S., and Sarkar, S. D. (1991). Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50:207–221.

[Stentz, 1995] Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1652–1659.

[Stout, 1996] Stout, B. (1996). Smart moves: Intelligent path-finding. *Game Developer Magazine*, (October):28–35.