Iterative-Expansion A*

Colin M. Potts and Kurt D. Krebsbach
Department of Mathematics and Computer Science
Lawrence University, Appleton, Wisconsin 54911
{colin.m.potts, kurt.krebsbach}@lawrence.edu

March 14, 2012

Abstract

In this paper we describe an improvement to the popular IDA* search algorithm that emphasizes a different space-for-time trade-off than previously suggested. In particular, our algorithm, called Iterative-Expansion A* (IEA*), focuses on reducing redundant node expansions within individual depth-first search (DFS) iterations of IDA* by employing a relatively small amount of available memory—bounded by the error in the heuristic—to store selected nodes. The additional memory required is exponential not in the solution depth, but only in the difference between the solution depth and the estimated solution depth. A constant-time hash set lookup can then be used to prune entire subtrees as DFS proceeds. Overall, we show 2- to 26-fold time speedups vs. an optimized version of IDA* across several domains, and compare IEA* with several other competing approaches. We also sketch proofs of optimality and completeness for IEA*, and note that IEA* is particularly efficient for solving implicitly-defined general graph search problems.

1 Introduction

Heuristic search techniques suffer from two types of memory-related problems: too much, or too little. Optimal algorithms, such as A*, require an amount of memory exponential in the length of the solution path. This causes the algorithm to run out of memory before producing a solution, or to spend an impractical amount of time generating, storing, and revisiting the stored search information. A common approach to this problem is embodied in the IDA* algorithm, which combines the linear memory requirements of depth-first search (DFS) with the heuristic estimate from A* search. While IDA* was in fact the first algorithm to successfully solve many early search problems, it notoriously suffers from over-reliance on trading time for space spending time regenerating nodes of the search tree. There are two main types of redundancy. First, IDA* periodically restarts the entire DFS from the root of the tree after each successive f-limit is reached. This "iterational" redundancy is traditionally accepted as an effective "time for memory" trade-off to exploit DFS's linear memory requirement, although other researchers have made progress in addressing this type of redundancy as well [9, 10].
The second type involves redundancy *within* a single iteration of DFS. Because DFS only keeps the current path in memory at any given time, it regenerates entire subtrees of the graph, expanding all successors of such a state once for each path to the state. This can be a major—and often underestimated—time inefficiency of IDA*, but one that can be effectively addressed within the context of the multiple iterations of IDA*, and is therefore the focus of our technique. We present the IEA* algorithm, and will show that IEA* drastically outperforms IDA* under a variety of assumptions.

2 A* and Iterative-Deepening A*

Since its introduction, the A* search algorithm [4, 5] has become the standard by which best-first search algorithms are judged. A* expands nodes based on the sum (denoted f) of the cost accumulated along a path (g), and a heuristic estimate of the distance from that node to the nearest goal state (h). This node-expansion strategy guarantees that A* is complete and optimal, provided that h never overestimates the actual distance to a goal state [2]. Unfortunately, A* requires an amount of memory exponential in the length of the solution path, causing the algorithm to run out of memory before producing a solution on difficult problems. The IDA* algorithm [6] was developed to address this memory limitation, and does so by combining the linear memory requirements of (uninformed) DFS with the f function of A*. IDA* performs a sequence of DFS iterations with increasing f-limits until an optimal solution is found.

2.1 IDA* Optimizations

A first and extremely effective step in curtailing the node count is to remove cycles from the graph. This can be done without sacrificing speed and without increasing the memory requirement. A 2-cycle occurs when we expand a node P, pick one of its children C to expand, and then one of those expanded nodes is the parent P. Similarly, an n-cycle is when we attempt to expand one of the $n - 1$ states on the current path. Both types of cycles are easily eliminated.

3 Iterative-Expansion A*

Low memory techniques like IDA* are extremely fast at expanding nodes as compared to memory-intensive searches (like A*). For example, depending on the node's *successor* function, a redundant node can often be regenerated much more quickly than checking to see whether it has already been generated; however, always regenerating a node implies regenerating the entire subtree rooted at that node. In IDA*, regenerating subtrees can be a major inefficiency both within a single iteration and multiplied across f-limited iterations. We now compare IEA* to the optimized version of IDA*. For purposes of this paper, we assume identical action costs. IEA* strikes a balance between A* and IDA* to speed up the search by using a modest and bounded increase to IDA*'s memory requirement. Since we eliminate cycles consistently in both algorithms, we seek to eliminate other unnecessary expansions within single iterations.
A useful way to think about tree search is that we explore all paths that appear—based on a heuristic—to be of a specified length until we discover that they belong to a longer path. All of these paths are unique but may contain the same subsections which occur whenever there are transpositions in the tree (a single state with multiple paths to it). The subtree of a transposition node is explored once for each path to it. IEA* seeks to eliminate the transpositions which will result in the greatest redundancy elimination. It is clear that those higher up in the tree result in larger subtrees being eliminated, both in the current iteration of DFS, and in all future iterations. In fact, we demonstrate that this strategy eliminates an exponential number of node regenerations using memory bounded by the error in the heuristic function.

3.1 IDA* vs. IEA* Node Expansion

Figure 1 demonstrates how IDA* and IEA* node expansion differs. Each tree represents a successive iteration of both IDA* and IEA*. The f-values of all nodes shown in an iteration are equal, except for the nodes from the previous iterations. The highlighted nodes are on the closed list that IEA* keeps. IEA* begins its search by adding the root to the fringe. It then performs an f-Limited-Search from that node to the f-limit of $h(root)$. When that search finishes, we have completed the first iteration. IDA* begins in exactly the same way by performing the same f-Limited-Search starting at the root using the same f-limit. However, IEA* keeps track of the children of the root with f-value $\leq h(root)$—the nodes connected to the fringe in this figure. These children are then added to the fringe for the next iteration and thus added to the closed list as well (once a node is added to the closed list, it remains there for the duration of the search.) When IDA* begins its second iteration, it restarts from the root using a new f-limit based on the last iteration. IEA* uses the same f-limit, but instead of a complete restart it begins with the nodes on the fringe (the periphery of the closed list—similar to a search like A*.) IDA* will eventually re-expand all nodes that IEA* has stored, and thus do the equivalent of starting f-Limited-Searches from each node on the fringe, which is how IEA* proceeds. It follows that IEA* will expand the same set of nodes as IDA*, but whenever a transposition is detected using the closed list, that node will not be re-expanded, eliminating the entire subtree rooted there.

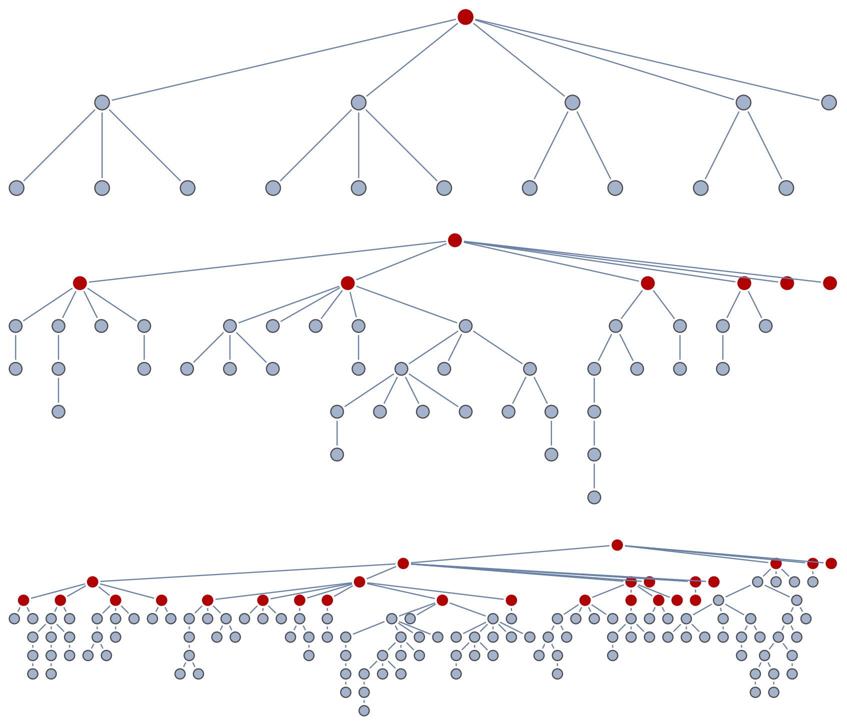

Figure 1: IDA* vs. IEA* node expansion and closed-list maintenance.

3.2 The IEA* Algorithm

As shown in the top-level function, Iterative-Expansion-A*, the first while loop of the algorithm (line 4) iterates until a solution is found, or we can prove no solution exists. It also sets a new higher f-limit on each iteration using the minimum f-value of the generated nodes (line 15), and creates a new variable called new-$fringe$ (line 5), which is where we store information for the next iteration. The next while loop simply selects the nodes in the fringe based on priority (lines 6-7), and begins an f-Limited-Search from each (line 8). After performing the FLS, Expand-Fringe is invoked, and each successor s of the current best node is then added to new-fringe and the closed list if and only if $f(s) \leq f$-limit, and s is not on the closed list (lines 2-4). If all children were not added (i.e., a cutoff) then the parent node is put back onto new-$fringe$, otherwise the parent can be left off new-$fringe$ for good. The FLS is the standard IDA* one, except for line 8 where we trim the successors based on the closed list.

Function Iterative-Expansion-A*(initial)

```
result $\leftarrow$ **cutoff**;
$f$-limit $\leftarrow$ $f$(initial);
fringe, closed $\leftarrow$ {initial};
**while** result = **cutoff** **do**
    new-fringe $\leftarrow \emptyset$;
    **while** fringe $\neq \emptyset$ **do**
        best $\leftarrow$ **pop** min $f$-value node **in** fringe;
        result $\leftarrow$ f-Limited-Search(best, $f$-limit);
        **if** result $\neq$ **cutoff** **then**
            **return** result
        expansion $\leftarrow$ Expand-Fringe(best, $f$-limit);
        new-fringe $\leftarrow$ new-fringe $\cup$ expansion;
        closed $\leftarrow$ closed $\cup$ expansion;
    fringe $\leftarrow$ new-fringe;
    $f$-limit $\leftarrow \min\{f(s) | s \in Gen, f(s) > f\text{-limit}\}$;
**return** result
```

Function f-Limited-Search(node, f-limit)

Result: a solution, failure, or cutoff

```
cutoff-occurred $\leftarrow$ **false**;
**if** $f$(node) $>$ $f$-limit **then**
    **return** **cutoff**
**else if** Goal-Test(node) **then**
    **return** result $\cup$ {node}
**else**
    **foreach** *state* $s$ **in** successors(*node*) **do**
        **if** $s \notin$ closed **then**
            result $\leftarrow$ f-Limited-Search($s$, $f$-limit);
            **if** result = **cutoff** **then**
                cutoff-occurred $\leftarrow$ **true**;
            **else if** result $\neq$ **failure** **then**
                **return** result
**if** cutoff-occurred **then**
    **return** **cutoff**
**else**
    **return** **failure**
```

Function Expand-Fringe(node, f-limit)

Result: nodes for new fringe

```
nodes $\leftarrow \emptyset$;
**foreach** *state* $s$ in (successors(node) $-$ closed) **do**
    **if** $f$($s$) $\leq$ $f$-limit **then**
        nodes $\leftarrow$ nodes $\cup$ {$s$};
**if** nodes $\neq$ (successors(node) $-$ closed) **then**
    nodes $\leftarrow$ nodes $\cup$ {node};
**return** nodes
```

4 Empirical Results

We now present empirical results obtained across two demonstration domains. All experiments were run on a Intel Core i7-2820QM 2.30 Ghz processor with 8Gb of available RAM. Both the IDA* and IEA* implementations use to the same f-Limited-Search function. We implement full cycle checking in both to keep the comparisons consistent. We use a hash set to do cycle and closed-list checking. Given this, the hash set passed to the f-Limited-Search function is the only variation between the two depth-first searches. The algorithmic difference is that between iterations we update IEA*'s closed list by inserting new states into it, as previously described. Finally, we use the Manhattan distance heuristic function on both algorithms and domains.

4.1 Fifteen-Puzzle Domain

Figure 2 illustrates how IEA* runtimes compare with IDA* for each of the 100 Korf Fifteen-Puzzle problems (sorted by the amount of time IEA* took.) The important feature of this graph is how it uses a log scale to show the differences between IEA* and IDA*. We note that the difference between the two shows up as relatively constant on the graph, demonstrating an exponential difference in runtimes. As we will see, these times correspond linearly with differences in nodes expanded.

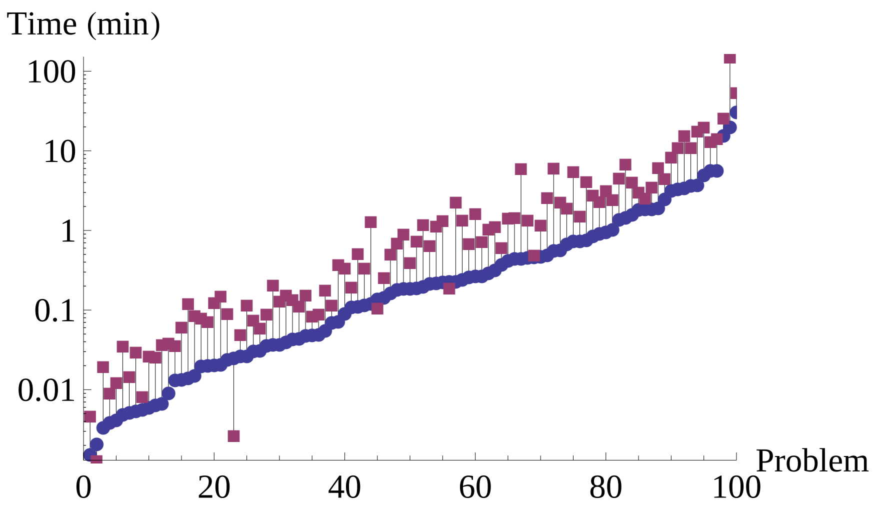

Figure 2: Runtime comparison for IEA* vs. IDA* for each of Korf's 100 Fifteen-Puzzle Problems, plotted against a log scale. Note that each difference is approximately the same, reflecting an exponential difference.

Figure 3 shows the same data plotted as the ratio of IDA* to IEA* runtime, effectively representing IEA*'s speedup factor for each of the problems. As we see, IEA* performs almost 4 times better than IDA* on average. Computing the IDA*/IEA* runtime ratios, we get a range of 0.8 to 14.0, a large disparity that hugely favors IEA*. While this result is related to an exponential reduction in the number of node expansions, IEA* incurs a slightly greater time cost for expansions. Therefore, the benefit in terms of nodes expanded comes from two sources: the number of transpositions that occur in the given f-limit, and the number of iterations. For easier problems this benefit has not produced a large disparity between IDA* and IEA* node expansions, and thus we see IEA* losing out. However, in increasingly difficult cases, IEA* starts to win at around 4 times better. With more iterations, and more costly effects by missed transpositions high in the tree for IDA*, this ratio begins sky-rocketing and we enjoy 14 times better performance.

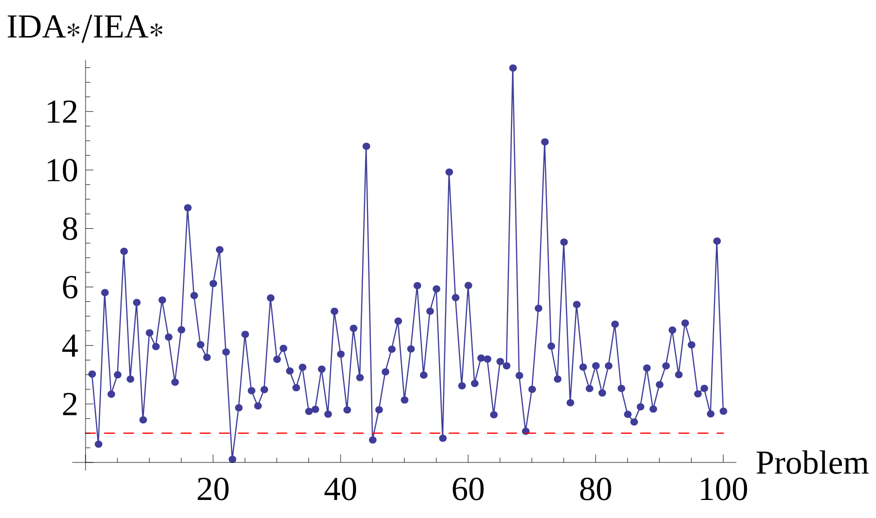

Figure 3: Speed-up factor for IEA* vs. IDA* for each of Korf's 100 Fifteen-Puzzle Problems.

Figure 4 shows us the closed list for IEA* on the same problem set, sorted by the number of iterations. This is plotted against lines for the complexity classes of A* and IEA*. We see that in every case, IEA* uses less memory than the predicted amount, which is exponentially better than A*.
For example, we look at a Fifteen-Puzzle instance with optimal solution depth $d = 55$, and $h(root) = 43$. The Manhattan distance heuristic has parity, that is $M = 2$. So $k = \lceil \frac{55-43}{2} \rceil = 6$ and $b = 4$, so we get $|closed\text{-}list| \leq 4^6 = 4096$. However, if we assume an average branching factor of $b = 2.5$, then that IEA* stores an average of $2.5^6 \approx 245$ nodes in the closed list. For IEA*, $245 \leq nodes \leq 4096$, but for A*, $2.5^{12} \approx$ 59,605 $\leq nodes \leq 4^{12} =$ 16,777,216. Where our experimental results are 184 nodes for IEA*, and 6,488,168 for A*. Further, IDA* require bd nodes, so a maximum of $4 * 55 = 220$ and an average of $2.5 * 55 = 137.5$.
A second way to illuminate the exponential difference noted above is to look at the average exponential curve. We take the number of nodes expanded at each depth for both IDA* and IEA*, take the difference, and curve fit a function of the form $a \times b^x$. Figure 5 plots the data points (the actual differences) along with the fitted curve for each problem against a logarithmic scale. It is easy to see how closely the curves approximate the points, and that all appear as a line in the graph showing the exponential difference. For our fitted function, a is a measure of how quickly we will see benefit, b is a gauge of long-term node count, and x specifies the iteration. Then, we take the average of all values to get roughly $a = 15$, and $b = 6.4$. The standard deviation for b is approximately 1.2, but for a it is 36. This

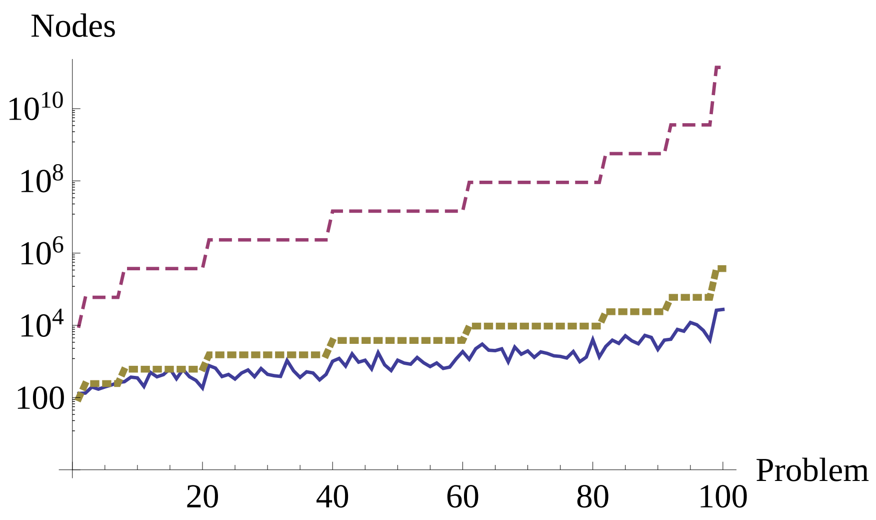

Figure 4: Number of nodes stored on the closed list for each of Korf's 100 Fifteen-Puzzle Problems (bottom line), plotted against the average complexity class of A*: $2.5^{d-h(root)}$ (top line), and the average complexity class of IEA*: $2.5^{\frac{d-h(root)}{2}}$ (middle line) .

is to be expected, since a varies greatly with how soon the fringe becomes effective, but b remains relatively constant as time wears on. Thus we expect in general—and we see in practice—that IEA* really starts to perform better as we perform more iterations; i.e., as more node expansions are required. We forgo a discussion of A* here because it is well known that for the Fifteen-Puzzle, IDA* with cycle checking is a faster method, and thus IEA* surpasses it even further. However, in our second domain, Grid Navigation, A* has superior results. This is largely because we use a 256 by 256 grid which is an explicit graph that easily fits into memory. We test on the Grid Navigation domain because it is rampant with transpositions, which DFS historically struggles with. We selected this domain to show how extremely effective IEA*'s closed list is at detecting these transpositions. As expected, we observe IEA* outperforming IDA* exponentially.

4.2 Grid Navigation Domain

As a second test domain, we present results of searching for an optimal path through a grid-based map with obstacles. We seek an optimal route for an agent allowed to move between adjacent cells in one of (at most) four directions. Figure 6 shows the sampling of test problems, presented as the ratio of IDA* to IEA* runtime, effectively telling us how much better or worse IEA* did compared to IDA*. We observe that IEA* runs from 2 to 26 times faster than IDA* on all but 3 problems, with an average speedup factor of about 5. These tests were run on 1000 different problems, and we present a representative selection of 41 of them here (which, if anything, are biased to IDA*'s advantage). For example, one exceptionally difficult problem not included in Figure 6 showed IEA* performing 400 times faster. In addition, most of the low ratios (including losses) came from problems where the runtimes were on the order of only 10ms.

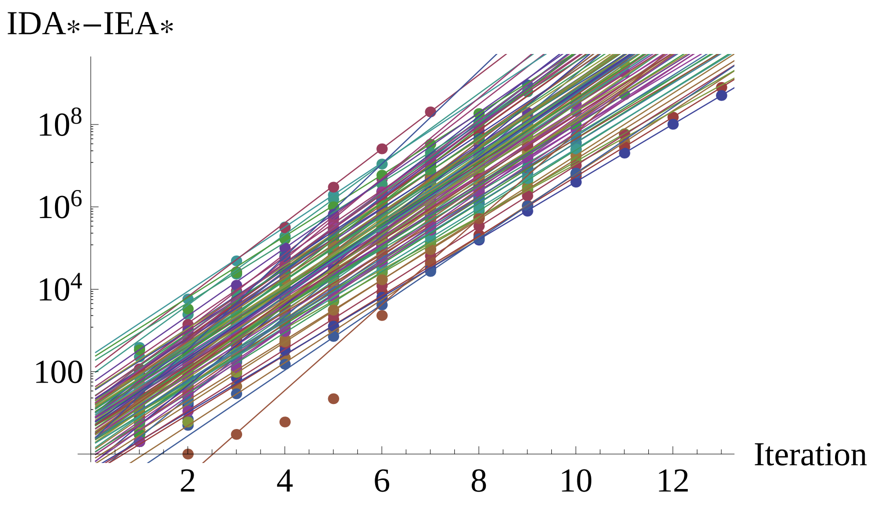

Figure 5: Difference between number of nodes expanded by IDA* vs. IEA*, plotted by iteration. The 100 curves are fitted to the actual data of the 100 Korf Fifteen-Puzzle problems, with the individual points representing the actual differences by iteration.

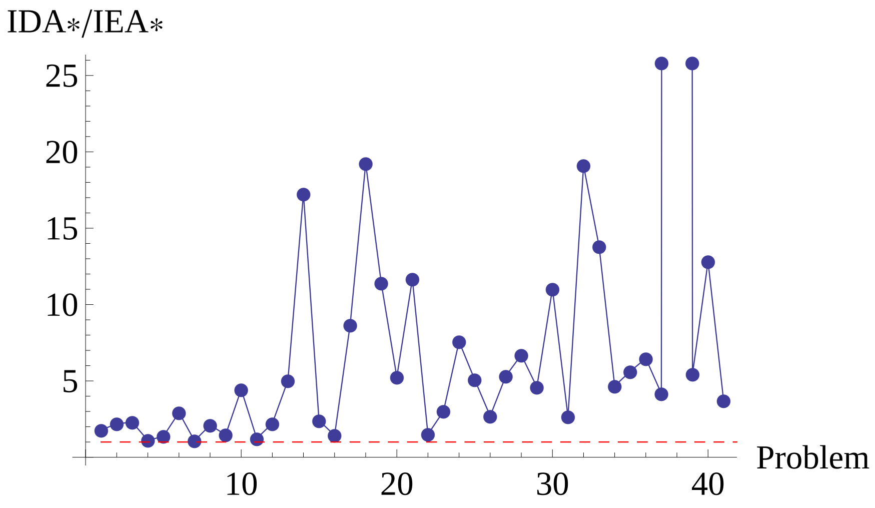

Figure 6: Speed-up factor for IEA* vs. IDA* for a random sampling of Grid Navigation problems.

5 Theoretical Results

5.1 Space complexity

In general, the memory requirement for depth-limited search is $O(bd)$, assuming that b is the maximum branching factor and d is the depth of the shallowest solution. Assuming h is the heuristic function, let M be the minimum increase in f-limit, and r be the root node. Finally, we let $k = \lceil \frac{d-h(r)}{M} \rceil$ (which is the number of iterations.) Then IEA* stores a maximum of $O(b^k)$ nodes in memory—an increase bounded by the error in the heuristic function h.

Each iteration causes the f-limit to increase by at least M, so at most, we will perform k iterations, thus increasing the closed lists k times. In the worst case, suppose during each iteration that every possible node is added to the closed list. Thus, at a given iteration i, $O(b^i)$ nodes will be on the closed list.
To see how this works, we compare the number of nodes added by IEA* to that of A*. To do this, we view A* as an iterative search where each "iteration" represents all the nodes of the same f-value, which must all be expanded before expanding any higher-valued nodes. Now, we consider the number of nodes that can be added per iteration to the closed list. For A*, any node with $f \leq f$-limit will be added to the closed list, which includes all nodes of all subtrees of the fringe satisfying this property—often an exponential number of nodes. Now suppose these subtrees contained only *some* of the children of the fringe nodes, and no nodes at greater depths. This number represents an extreme lower bound for A*; however, IEA* will always expand exactly that many nodes.

5.2 Completeness

To show completeness, we rely upon the completeness of IDA*. The only way for IEA* to be incomplete is if—by keeping a closed list—IEA* somehow fails to consider a node that IDA* does consider. So we only need to show that the first n nodes of a path P to the goal must be on the closed list. The initial state is trivially on the closed list after the very first expansion. Now the next node on P is either on the closed list or it is not. If it is, we proceed; if it is not, then the initial state was on the fringe and the f-Limited-Search would proceed normally from there. Through iteration, we see that the IEA* algorithm is complete.

5.3 Optimality

Proof of IEA*'s optimality follows from the proof of optimality for IDA*. Since IEA* iterates at increasing f-depths, which are set each time using the minimum f-value that was found greater than the current f-depth, we know that no solution exists at any f-depth checked previously due to the proof of IEA*'s completeness. Therefore, the first time we find a solution s at depth d, we know there can exist no solution at any depth less than d. Thus, s is an optimal solution.

6 Related Work

While IDA* was in fact the first algorithm to successfully solve many early search problems, it notoriously suffers from over-reliance on trading time for space, using too little memory. Because DFS remembers only the current path, many nodes are re-expanded redundantly both within and across f-limit iterations.
When h is consistent, A* handles a graph like a tree and never expands a node more than once; however, IDA* cannot prevent the re-expansion of nodes in a graph given only the current path. The only way to completely eliminate duplicate nodes from a graph search is to store all generated nodes (i.e., the union of the closed and open lists); however, it has

long been known that some duplicates can be cheaply eliminated by comparing new nodes to nodes on the current path from the root [7]. This optimization requires no additional memory, but analysis shows that it can be expensive, as the set of nodes on the current path is constantly changing, incurring overhead costs with every DFS node expansion.
A variety of proposals have also been published that use additional available memory to improve the runtime performance of IDA*. Sen and Bagchi's MREC algorithm [9] accepts a runtime parameter M denoting the amount of additional memory MREC is allowed to use beyond IDA* requirements. MREC then uses this to store as much of the explicit graph as possible to prevent duplicate generation.
As discussed earlier, an obvious improvement for problems involving general graph search is to eliminate cycles. Dillenburg and Nelson describe two types – full cycle-checking and parent cycle-checking, and provide guidelines for which type should be used with IDA* on a given problem [3] . Taylor and Korf manage to eliminate some duplicates without explicitly storing them by performing a limited breadth-first search of the space, and creating a finite-state machine (FSM) describing a set of operator strings known to produce duplicate nodes [10]. The FSA represents a more efficient abstraction of the set of duplicates in the shallow portion of the search tree. Then, whenever a string is encountered during DFS that matches an FSM string, the rest of that path can be safely pruned. We should note that this technique is only useful for problems described implicitly (e.g., the Fifteen-Puzzle). Reinefeld, et.al., have suggested adapting strategies from two-player games to dynamically reorder node successors as well as combining these strategies with standard transposition table storage techniques [8]. The reordering techniques described differ from ours in that they only potentially speed up the final iteration of IDA*, whereas IEA* produces savings from each transposition node forward through all future iterations. Finally, so-called *fringe search* appears to add both the closed list and the open list from A* to implement an algorithm with the same memory requirements that IDA* was initially developed to address [1].

7 Future Work

We are currently exploring two threads of research related to the work reported on here. The most immediate work concerns developing a parallelized version of IEA* to exploit multicore machine architectures. One of the biggest issues in parallelization is partitioning the work effectively. Search algorithms in particular tend to be executed sequentially, and each step must synchronize across the board; however, the way we have structured the IEA* fringe lends itself to a useful and natural partitioning of the work. We are currently developing a method to cache the closed list across multiple processors while maintaining cache consistency. We also intend to apply results of this work to heuristic search planners, especially in domains which tend to be dense with solutions, and for which iterative-deepening techniques are particularly efficient at finding the first optimal solution.

8 Conclusion

We introduce an improvement on the classical IDA* algorithm that uses additional available memory to find solutions faster. Our algorithm, IEA*, reduces redundant node expansions within individual DFS iterations by keeping a relatively small amount of extra memory which we can show is bounded by the error in the heuristic. The additional memory required is exponential not in the solution depth, but only in the difference between the solution depth and the estimated solution depth. We show 2- to 14-fold speedups in one domain, and 2- to 26-fold speedups in a majority of the problems in the other. We also sketch proofs of optimality and completeness for IEA*, and note that this algorithm is particularly efficient for solving implicitly-defined general graph search problems.

References

[1] Bjornsson, Y.; Enzenberger, M.; Holte, R. C.; and Schaeffer, J. 2005. Fringe search: Beating A* at pathfinding on game maps. In *IEEE Symposium on Computational Intelligence and Games*.

[2] Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 32(3):505–536.

[3] Dillenburg, J. F., and Nelson, P. C. 1993. Improving the efficiency of depth-first search by cycle elimination. *Information Processing Letters* 45:5–10.

[4] Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

[5] Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1972. Correction to A formal basis for the heuristic determination of minimum cost paths. *SIGART Newsletter* 37:28–29.

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

[7] Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, Massachusetts: Addison-Wesley.

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

[9] Sen, A. K., and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, 297–302.

[10] Taylor, L. A., and Korf, R. E. 1993. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence*, 756–761.