Efficient memory-bounded search methods

Stuart Russell*
Computer Science Division
University of California, Berkeley, CA 94720, USA

Abstract. Memory-bounded algorithms such as Korf's IDA* and Chakrabarti *et al*'s MA* are designed to overcome the impractical memory requirements of heuristic search algorithms such as A* . It is shown that IDA* is inefficient when the heuristic function can take on a large number of values; this is a consequence of using too little memory. Two new algorithms are developed. The first, SMA*, simplifies and improves upon MA*, making the best use of all available memory. The second, Iterative Expansion (IE), is a simple recursive algorithm that uses linear space and incurs little overhead. Experiments indicate that both algorithms perform well.

1 Introduction

This paper adopts the standard framework of heuristic search, in which the object is to find a sequence of operators leading from a given initial state to any goal state. The search is guided by a heuristic function $h(n)$, which estimates the lowest cost of any path from state n to a goal state. Although heuristic search is a well-established area of research, new developments have great practical significance, both because of the large number of applications and because search often underlies the operation of more complex AI systems.

We will be principally concerned with finding the optimal solution sequence, using heuristics that are *admissible*; that is, $h(n) \leq h^*(n)$ where h^* represents the exact distance to the nearest goal state.[1] With admissible h, the A* algorithm [5] is known to return optimal solutions. Furthermore, A* examines the minimum number of nodes necessary to do this, up to tie-breaks [3] (see below for a simple proof).

Given these results, and the existence of more efficient, ϵ-admissible algorithms that relax the optimality requirement, one might imagine that this area of research is more or less sewn up. Unfortunately, A* retains in memory all the nodes it has generated, and will run out of space long before it exhausts the patience of the user. (A good 15-puzzle implementation can exhaust main memory on a 64MB workstation in under ten minutes.) Recognizing this, several researchers have developed memory-bounded variants of A*. The principal difficulties are 1) ensuring optimal solutions, and 2) avoiding the continual re-expansion of nodes that have been visited before but necessarily forgotten because of memory restrictions.

The Graph Traverser [4], one of the earliest search programs, commits to an operator after searching best-first up to the memory limit. As with other "staged search" algorithms, optimality cannot be ensured because until the best path has been found the optimality of the first step remains in doubt. IDA* [6], one of the earliest admissible, memory-bounded algorithms, uses space linear in the length of the solution. The MA* algorithm [2] can utilize whatever memory is available, and thereby avoids some re-expansions. The latter paper also describes several other memory-bounded algorithms.

The next section discusses IDA* and MA* in detail. Section 3 describes the SMA* algorithm, which improves on MA*. Section 4 describes IE, a simple, low-overhead recursive algorithm that uses linear space and is more efficient than IDA*. Section 5 provides performance data to support these claims.

2 IDA* and MA*

IDA* was derived from the old idea of iterative deepening. In ordinary depth-first iterative deepening, search proceeds by gradually increasing the depth limit until a goal is found. IDA* takes advantage of the admissibility of the heuristic function by limiting the *f-cost* of the nodes examined by the depth-first search, rather than the depth. The f-cost of a node is given by $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of the path from the start node. The IDA* algorithm is defined as follows ($S(n)$ denotes n's successors):

Algorithm IDA*(n):
$limit \leftarrow f(n)$;
do until *success* or *limit* unchanged
  $limit \leftarrow DFS(n, limit)$;

Function DFS(n,limit):
if $f(n) > limit$ return $f(n)$;
if $goal(n)$ then return with success
else return lowest value of $DFS(s, limit)$ for $s \in S(n)$.

IDA* has much lower overhead than A*; furthermore, in Korf's experiments on the 15-puzzle, the

*Work supported by an NSF PYI Award, an SERC Visiting Fellowship, and the Turing Institute, Glasgow. Figures by Ashwin.

[1] We will also enforce *monotonicity* of h, which amounts to satisfaction of the triangle inequality. Monotonicity can be ensured for any admissible h by the application of *pathmax*, which compares the value of a state with those of its ancestors.

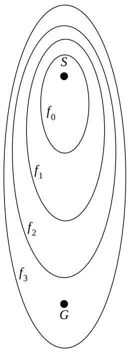

Figure 1: f-cost diagram for A* and IDA*

number of nodes re-expanded was small enough to make IDA* competitive with A* in terms of time. The easiest way to understand the behaviour and complexity of IDA* is to consider the state space statically, with the nodes ordered by f-cost. Figure 1 shows some idealized f-cost boundaries around the start node S. In such f-cost diagrams, all nodes between boundaries f_i and f_{i+1} have $f(n) = f_{i+1}$. Nodes inside the f_0 boundary have the same f-cost as the start node.[2]

A* operates by expanding all the nodes in each layer before continuing to the next.[3] On the other hand, IDA* starts again from scratch with f-cost limits in the increasing sequence $f_0, f_1, \ldots$. Each iteration goes over the layers examined by the previous one, plus one new layer.

In the worst case, every node in the state space has a different f-cost, so that each layer contains exactly one node. If A* examines k nodes to solve a problem, then IDA* will examine $1 + 2 + \cdots + k$, i.e., $O(k^2)$ nodes. When k is large (for instance, large enough that k nodes cannot be stored in the memory available), the slowdown compared to A* may be unacceptable. The worst case does not arise in domains such as the 8-puzzle because the manhattan-distance heuristic takes on only a small number of integer values, and many nodes have the same f-cost. Realistic optimization problems such as Travelling Salesman or VLSI layout would certainly cause worst-case behaviour because of the continuous variables involved. In fact, applying an infinitesimal random perturbation to the heuristic values in the 8-puzzle is sufficient to degrade IDA*'s performance drastically, as we show below, whereas A*'s performance is unaffected.

State spaces in which nodes have a wide variety of f-costs are difficult to search because as the algorithm extends the search frontier, it will frequently change its mind about which path is most promising. A* can handle this because all paths are available to be extended if they become the most promising candidates; bounded-memory algorithms will have difficulty because some previously-explored paths are necessarily purged from memory and will have to be re-expanded. IDA* is particularly vulnerable because it retains *no* path information between iterations.

A simple meta-level argument suggests that a good algorithm should retain as many nodes as possible, and should prefer to retain the most promising ones. This is because the purpose of retention is to increase speed by avoiding re-expansion (in general, the purpose of memory is to avoid recomputation as well as to retain state). If nodes are selected for expansion by lowest f-cost, then if $f(n_1) < f(n_2)$, n_1 must become the most promising node before n_2 because until then only nodes with $f(n) \leq f(n_1)$ are expanded. Hence nodes with highest f-cost should be pruned first. Furthermore, when a node is pruned, as much cost information should be retained in its ancestors as possible, consistent with the constraint that each node takes constant space.

The MA* algorithm [2] embodies these principles reasonably well. (Unfortunately, the algorithm is too complicated to reproduce here, so a sketch will have to suffice.) Like A*, it maintains two sets of nodes: CLOSED contains nodes all of whose successors are present in memory, and OPEN contains all other nodes in memory. Successors are generated from the node in OPEN with lowest f-cost, and added to OPEN. When the number of nodes in OPEN and CLOSED reaches some preset limit, MA* begins to prune the OPEN list by removing the leaf-node with highest f-cost.[4] When a new successor is generated, its f-cost is propagated back up the tree so that, roughly speaking, the f-cost of each internal node is always the most informed bound derived from all its examined descendants. (Actually, this is how SMA* works. MA* uses a different set of quantities for each node, to the same effect.) Thus, for example, if a node n whose original f-cost is 4 is found to have successors all of whose f-costs are 6, then the f-cost of n is revised to 6, representing the new lower bound on solution paths through n. Although pruning the descendants of n then loses information about *which way* to go from n, the retention of backed-up values means that the algorithm still knows *how worthwhile* it is to go anywhere from n.

MA* makes good use of available memory, and is able to solve selected 15-puzzles expanding fewer nodes than IDA*. As soon as the memory limit is reached, MA* abandons the least promising part of the space and reallocates memory to push forward

[2] That such a diagram can be consistently drawn follows from the monotonicity property of admissible heuristics with pathmax, which means that f-costs are non-decreasing along any path.

[3] Hence we can see that A* is optimally efficient among all admissible algorithms with the same information; if an algorithm skips any node in an interior layer, it will fail to return the optimal solution if that node happens to be on the best path.

[4] This means that MA*, and SMA*, need to deal with partially-expanded nodes and to generate successors one at a time.

towards the goal. As time progresses, the nodes in memory will occupy a narrow band around the best solution path.

3 SMA*

The MA* algorithm can be improved in several ways. The resulting algorithm, called SMA*, incorporates the following improvements:

1. Because the backing up of f-costs means that many nodes have the same f-cost, and because the algorithms need to select deepest and shallowest leaves of lowest and highest f-cost, SMA* uses a binary tree of binary trees to store OPEN, sorted by f and depth respectively. MA*'s data structures are less efficient.
2. SMA* is easier to implement and understand than MA*, since it maintains just two f-cost quantities for each node rather than four. Also, SMA* backs up once per fully-expanded node, rather than once per node generated.
3. When MA* begins pruning, it continues until only the current 'principal variations' — nodes with the best f-cost — remain. As argued above, only the worst node should be pruned, and only when space is needed for a better one. SMA* adds and prunes only one node at a time.
4. MA* loses information by not using pathmax with the backed-up f-costs. This is the most crucial improvement in SMA*.

3.1 Algorithm description

Algorithm SMA*(start):
put *start* on OPEN; USED $\leftarrow$ 1;
loop
  if *empty*(OPEN) return with failure;
  best $\leftarrow$ deepest least-f-cost leaf in OPEN;
  if *goal*(*best*) then return with success;
  $succ \leftarrow next\text{-}successor(best)$;
  $f(succ) \leftarrow max(f(best), g(succ) + h(succ))$;
  if *completed*(*best*), $BACKUP(best)$;
  if $S(best)$ all in memory, remove *best* from OPEN.
  USED $\leftarrow$ USED+1;
  if USED $>$ MAX then
    delete shallowest, highest-f-cost node in OPEN;
    remove it from its parent's successor list;
    insert its parent on OPEN if necessary;
    USED $\leftarrow$ USED-1;
  insert *succ* on OPEN.

Procedure BACKUP(n):
if n is completed and has a parent then
  $f(n) \leftarrow$ least f-cost of all successors;
  if $f(n)$ changed, $BACKUP(parent(n))$.

SMA* is called with the start node. A global variable MAX is set to the maximum number of nodes that can be accommodated, and the variable USED keeps track of how many nodes are currently in memory. Each node contains its g, h and f-costs, and the minimum f-cost of its examined successors; it also retains some information used by the successor function to indicate the next successor to be generated. A successor that has not been generated since its parent was last generated is called *unexamined*. A node with no unexamined successors is called *completed*.

3.2 Properties

SMA* has the following properties:

Lemma 1 *f-costs are maintained to give a correct lower bound on the cost of solution paths through any unexamined descendant of a node. The bounds are stricter than those maintained by MA*.*

Lemma 2 *SMA* always expands the node that has the best lower bound on its unexamined descendants.*

Theorem 3 *SMA* is guaranteed to return an optimal solution, provided MAX is at least as large as the number of nodes on the optimal solution path.*

Theorem 4 *Except for its ability to generate single successors, SMA* behaves identically to A* when MAX is larger than the number of nodes generated by A*.*

The complexity of SMA* is discussed briefly in section 6.

3.3 Example

Figure 2 shows a typical tree; the left subtree has been largely pruned away to make room for the more promising right subtree, but its f-cost information has been backed up to its frontier ancestor **C**. If **C** is later re-expanded, pathmax will give **F** a value of 5, as we would hope. Unfortunately, node **I** will get a value of 5 although it was once known to have a value of 6. This form of information loss seems unavoidable given the requirement of constant space per node.

4 IE

IE is a simple recursive algorithm developed from an idea in DTA* [8]. There it was pointed out that with an admissible heuristic, the backed-up f-cost of a child of the root can only increase, and therefore search should only be carried out under the current best child, until its cost exceeds the current second-best child. Thus IE is called on a node with a bound equal to the backed-up f-cost of the second-best path from any ancestor of that node. (The second-best-value idea is also used in RBFS, a similar algorithm developed independently by Korf [7], and in Bratko's implementation of A* [1].)

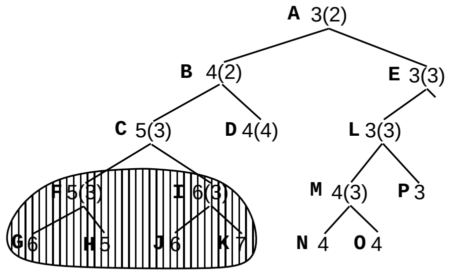

Figure 2: Example search tree generated by SMA* Backed-up f-costs shown with original f-costs in parentheses. Shaded region indicates pruned nodes. Nodes are *generated* in alphabetical order, and *pruned* in the order **GKJIHF**. Memory holds 10 nodes, and becomes full when **J** is added.

4.1 Algorithm description

IE is called with the start node and a bound of ∞. f-costs are maintained in exactly the same way as in SMA*, except that backing up occurs when the bound is violated and the recursive path unwinds back to the point at which the previous second-best path begins.

Algorithm IE(n,bound):
if $f(n) > bound$ then return;
if $goal(n)$ then return with success;
generate $S(n)$, assign f-costs using pathmax;
if $S(n) = \{\}$ return ∞;
do until success or while $f(n) \leq bound$
  $best \leftarrow$ node in $S(n)$ with lowest f-cost;
  $newbound \leftarrow min(bound,$ other f-costs in $S(n))$;
  call $IE(best, newbound)$;
  $f(n) \leftarrow$ lowest f-cost in $S(n)$.

IE has the same formal properties as SMA*, but since it prunes away all but the current best path and its sibling nodes, one would expect a higher rate of re-expansion.

4.2 Example

Figure 3 shows three snapshots of IE operating in the same search space shown in the SMA* example. In the first snapshot, IE has just been called on **E**; since $f(\mathbf{E})$ exceeds the bound of 3 (which derives from **A**'s right child), IE will return and $f(\mathbf{D})$ is set to 5. This exceeds **D**'s bound, so IE returns from **D** and **C** then calls IE on its right child, **F**. This basically suffers the same fate as **D** (see second snapshot), and the recursion unwinds back to **A**. Then **A** calls IE on its right child **H**, with a bound of 4 from the left child. The search proceeds, producing the third snapshot.

5 Performance

The experiments reported here have all been performed on the *perturbed 8-puzzle*. Small perturbations are made to the manhattan-distance heuristic function, giving each node a different f-cost without changing the structure of the puzzle.[5] The experiments were run using Allegro Common Lisp on a Macintosh Powerbook 140 laptop with 4MB memory.

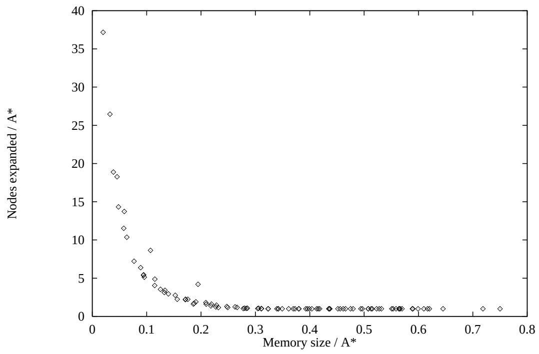

Figure 4: SMA*: Nodes expanded vs. memory size for solution-length = 20

The first set of experiments shows the effect of memory size on the number of nodes expanded by SMA* (Figure 4). The scatter plot was generated by solving 10 puzzles with various memory allocations covering the full range from the minimum up to the memory used by A*. The x-axis shows memory size as a fraction of the memory needed by A*. The y-axis shows the ratio of nodes expanded by SMA* to nodes expanded by A*. The data show that for these puzzles, good performance can be obtained using only a small fraction of the memory required by A*; also, the algorithm always expanded the same number of nodes as A* when the allocation was more than 32% of the memory required by A*. The data exhibit an intriguingly good fit to the relationship $nodes \propto memory^{-1.33}$.

The second set of experiments compares the average number of nodes expanded by IDA*, IE, SMA* and A*, as a function of solution length (Figure 5). 20 puzzles at each solution length were used, where possible. SMA* was run with memory size equal to twice the solution length. Even this small additional amount of memory allows SMA* to dominate IE, which in turn dominates IDA*. In terms of execution time, the rank order remains the same, except on very small problems, despite the additional overhead incurred by SMA* (ranging from 1.5 to 3.5 times more expensive per node). This should be less important on problems where node expansion time is significant. No exhaustive comparisons with MA* have yet been run, but the overhead for SMA* is about five times less. In some

[5] For reproducibility, the details of the perturbation method are as follows. The perturbed heuristic value $h' = h + (1 - \epsilon - h^2)/h(h+1)$, where $\epsilon < 1$ is unique to the state in question. For example, the state ((1 0 3) (4 5 6) (7 8 2)) is represented as 0.103456782, base 9. The perturbed heuristic is admissible and monotonic.

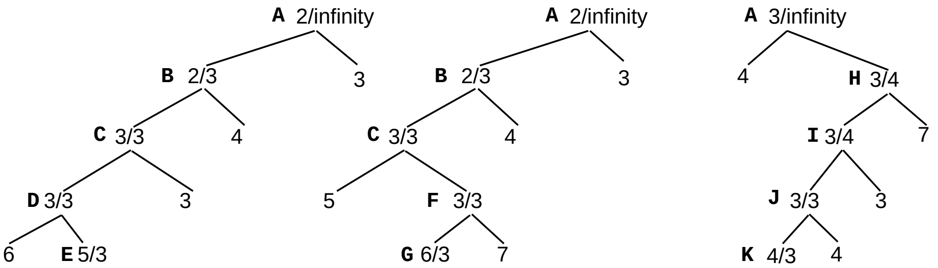

Figure 3: Three stages in a search by IE.
Each tree is a snapshot of the recursion stack. Labels are f-cost/bound. Calls to IE occur in alphabetical order.

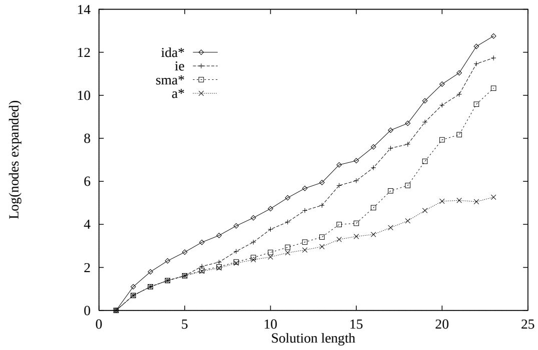

Figure 5: Log(nodes expanded) vs. solution length for IDA*, IE, SMA*, A*.

harder problems, MA* also expands up to 20 times more nodes because of its less strict use of pathmax.

6 Conclusions

Problem domains are difficult for memory-bounded optimization algorithms if they contain a large number of distinct values for the heuristic function.[6] This is because the algorithm will change its mind many times about which solutions are most promising. Two algorithms have been demonstrated that seem to have reasonable performance, but it would be useful to have a deeper analysis of the limits on achievable efficiency. Clearly, in the limit of large problems, the best path will change with every expansion, since the current best node's successor will appear some finite amount worse because of reduced error in h, whereas the current second-best node will be only infinitesimally worse than the current best node.

In practical terms, the impact of memory-bounded search algorithms can be quite large; for example, Soderland's SNLP general-purpose planning system has been modified to use IE instead of A*, enabling a previously infeasible tyre-changing problem to be solved quite easily. Conversely, IE and SMA* (and IDA*, for that matter) enable a heuristic to be used in systems, such as Stickel's Prolog Technology Theorem Prover, that currently use depth-first iterative deepening. The choice of which of the three memory-bounded algorithms to use depends on several factors, including the size of individual nodes; node expansion time; available memory; the number of different heuristic values; and the expected size of OPEN in SMA*.

While we have focussed on a restricted class of problems, the same issues arise in any form of search guided by heuristics. Many kinds of systems—neural nets and simulated annealing, for example—resort to hill-climbing, but could benefit from a less blinkered search.

[6] Memory-bounded algorithms also suffer from transpositions in graph-structured spaces, but this issue is beyond the scope of the paper.

References

[1] Bratko, I. (1986) The art of PROLOG programming. London: Academic Press.

[2] Chakrabarti, P. P., Ghose, S., Acharya, A., and de Sarkar, S. C. (1989) Heuristic search in restricted memory. *AIJ*, **41**, 197-221.

[3] Dechter, R., and Pearl, J. (1985) Generalized best-first strategies and the optimality of A*. *JACM* **32**, 505-536.

[4] Doran, J., and Michie, D. (1966) Experiments with the graph traverser. *Proc. R. Soc. (A)* **294**, 235-259.

[5] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968) A formal basis for the heuristic determination of minimum-cost paths. *IEEE Trans. Sys. Sci. and Cybernetics* **SSC-4**(2) 100-107.

[6] Korf, R. E. (1985) Depth-first iterative deepening: An optimal admissible tree search. *AIJ*, **27**(1), 97-109.

[7] Korf, R. E. (1991) Best-first search with limited memory. *UCLA Comp. Sci.Ann.*

[8] Wefald, E. H., and Russell, S. J. (1989) Estimating the value of computation: The case of real-time search. In *Proc. AAAI Spring Symp. AI and Limited Rationality*, Stanford, CA.