SPECULATIVE SEARCH: AN EFFICIENT SEARCH ALGORITHM FOR LIMITED MEMORY

Markian M. Gooley and Benjamin W. Wah
Center for Reliable and High-performance Computing
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, Illinois 61801

Abstract — Solving a problem optimally using an A^* search minimizes the number of nodes expanded. In the worst case, however, it uses an amount of memory exponential in the size of the problem. More general forms of best-first search have the same problem. Depth-first search uses a linear amount of storage, but expands more nodes on the average than best-first. Furthermore, it leaves most of a computer's primary memory idle. rather than making it help in the search. We propose a search method, *speculative branch-and-bound*, or *SB&B*, which combines elements of best-first and depth-first searches in an attempt to put this extra memory to work. We describe the search, and a variant, *SIDA**, that is analogous to A^*. We show how they are related to such searches as *IDA** and *MA**, show how to add pruning to *SIDA**, and give performance results. *SIDA** is superior to *IDA** and competitive with *MA** in the number of nodes it expands, and its overhead lies between theirs.

1. INTRODUCTION

Suppose that we want to find an optimal solution to a problem by branch-and-bound search, or by a specialization such as A^* [4]. In the worst case, we must expand a number of nodes exponential in the size of the problem [5]. Best-first expansion reduces the number of nodes we expand; with an admissible function (in this paper we assume one), it minimizes the number, and guarantees that the first solution found is optimal. But best-first searches must store exponentially many nodes in memory [5]. Unless our problems are of small size, or have very tight lower-bound and upper-bound functions, this soon exhausts even the secondary memory of any computer.

Instead we can expand nodes depth-first, using a stack and therefore a linear amount of memory. Typically it expands more nodes than a best-first search. Also, if the bounding function is loose, it might seek far deeper than where the optimal solution(s) will lie, doing futile work. Moreover, it uses only a fraction of the primary memory of a computer. *Can we use the extra memory to reduce the number of nodes expanded in a search?*

Various researchers have proposed search methods that combine aspects of depth-first and best-first searches. Pearl [5] describes such methods in his book. Our method is related more closely to those we describe next.

Iterative-deepening A*, or *IDA** [2] simulates an A^* search using a series of depth-first searches. Like A^*, it requires an admissible lower-bound function. It is a variant of *depth-first iterative deepening* (*DFID*): a series of distinct depth-first searches to progressively greater depths to mimic a breadth-first search. As originally described, *IDA** initially sets an *incumbent* to the (lower-bound) value of the root node *s*, and searches depth-first from *s*, backtracking when it reaches nodes whose values exceed the incumbent. Such a depth-first search is a *stage* or *iteration*. If a stage (not counting its leaves) finds a solution, that solution is optimal; if not, *IDA** sets the incumbent to the smallest value borne by a leaf of the stage (its *periphery*). Then it does the next stage: a new depth-first search from the root, discarding all results of the previous stage apart from the new incumbent.

The first solution found by A^* or *IDA** is optimal. Nodes first seen in a given stage of *IDA** all have the same value, apart from the leaves; thus the search examines new nodes in nondecreasing order of cost — just as does A^*. Asymptotically *IDA** expands the same number of nodes as A^* [2]; in practice it is clearly discarding information and wasting effort. Nor can it make use of more than the linear memory used by its stack.

MA* [1] uses any additional memory available, and prunes nodes when memory is exhausted. It resembles *IDA** performed using an informed depth-first search.

Consider the tree that *IDA** examines during a stage i given an incumbent inc_i. Call it $tree_i$: it has all nodes of value inc_i or less, with some children of larger value. Given inc_i, *MA** also examines the nodes of $tree_i$ before it proceeds to others, but in another way. At stage i, *MA** has saved on a priority-queue (its *open* list) the nodes of a subtree rooted at s; this is a subset of $tree_{i-1}$ (just s itself, initially). Each node of this subtree bears the value of the smallest leaf of $tree_{i-1}$ that is its descendant: this is inc_i or greater. It also contains this "best-descendant" information for each of its children.

Suppose that *MA** is at an early stage of a search, so that it is not yet pruning. Remove from the front of *open* a node n of value inc_i; generate n's best untried child m. (If there is no m, n goes onto another priority-queue, the *closed* list.) Expand m's children, back up their values to m, and expand next m's best child. Expand thereby the nodes of a path from m to the parent of a leaf of $tree_i$, and all of their children. However, save only the nodes on the path itself. (Note: before the back-up of values, the saved nodes of the path all had value inc_i.) Expand all such paths until $tree_i$ has been examined completely.

MA* usually reaches its fixed limit of *MAX* nodes (the number that fit in main memory, say). When more than *MAX* nodes are in memory, *MA** starts to prune, though some nodes being pruned may yet lead to optimal solutions. It prunes the

Research supported by a National Aeronautics and Space Administration Graduate Fellowship Program and Contract NCC 2-481, and National Science Foundation grant MIP 88-10584.

nodes of largest values, and these must have no saved descendants; their ancestors may have to have their backed-up values altered. Pruning continues until all nodes with backed-up values greater than that of the root are deleted.

Note a detail not stated explicitly in the principal paper on MA^* [1]: we must know the heuristic values of all of a node's children before we enqueue it, since an informed depth-first search is carried out. The children might not have to be expanded completely, or the effort may be small, as with the Fifteen Puzzle. For other problems, expanding a child "far enough" to evaluate it may be almost as hard as expanding it entirely.

MREC [7] is essentially an IDA^* that saves, in the "extra" memory, nodes near the root. So long as memory remains, it extends the stored search tree (or graph) with a series of depth-first stages from the root to nodes whose values just exceed an incumbent. The only differences from IDA^* are that it doesn't re-expand (just scans) nodes already in storage, and that it stores new nodes. When memory is exhausted, it searches as before, but stops storing new nodes. With IDA^*, once a node is expanded in any stage but the last, it is re-expanded in every stage thereafter. *MREC* prevents this re-expansion for some nodes near the root. It has the pleasing property of not re-expanding any nodes until it runs out of memory [7].

Section 2 describes *SB&B*, first informally, then formally. Section 3 examines its behavior, given a simple model of search, and compares it with that of IDA^* and MA^*; also it presents in detail the problems that must be shared by IDA^*, MA^*, *SB&B*, their variants, and any future methods of similar type. Section 4 shows how to add a pruning mechanism to *SB&B*, describes the extended method formally, and compares it with MA^* (which is a specialization). Section 5 gives preliminary results, and Section 6 concludes the paper.

2. SPECULATIVE SEARCH ALGORITHM

In doing a best-first search, we begin by putting an initial node on a priority queue. Then, until we find a solution, we repeat a series of steps: remove the "best" node (that of lowest lower-bound) from the front of the priority queue, expand all of its children, discard those whose values exceed the best known upper bound, insert the rest in the queue, and, if we have just found a tighter upper bound on the solution, prune the queue of any nodes with values exceeding this upper bound.

With our technique, speculative branch-and-bound (*SB&B*), we still use a priority queue, and still remove and expand the node at its front. This node, or one of its children (depending on circumstances we describe presently), is the *rootlet*. Instead of expanding the selected node, we use the rootlet as the root of a depth-first search to some given depth or value of node: a *speculation*.

We speculate from the rootlet using the usual last-in-first-out stack of depth-first search. We keep the *minimum leaf* (or *minimum* for short) of the search, and a corresponding *minimum path* from the rootlet to this leaf. At the start of a speculation there is no minimum. The first leaf of the speculation becomes the initial one; the state of the stack at that point, the minimum path. Thereafter the minimum becomes the leaf of smallest value yet seen in the speculation (if there are duplicates we keep the first to avoid effort), and the minimum path the corresponding state of the stack. When the speculation is done, the minimum path leads to the minimum leaf of the entire speculation.

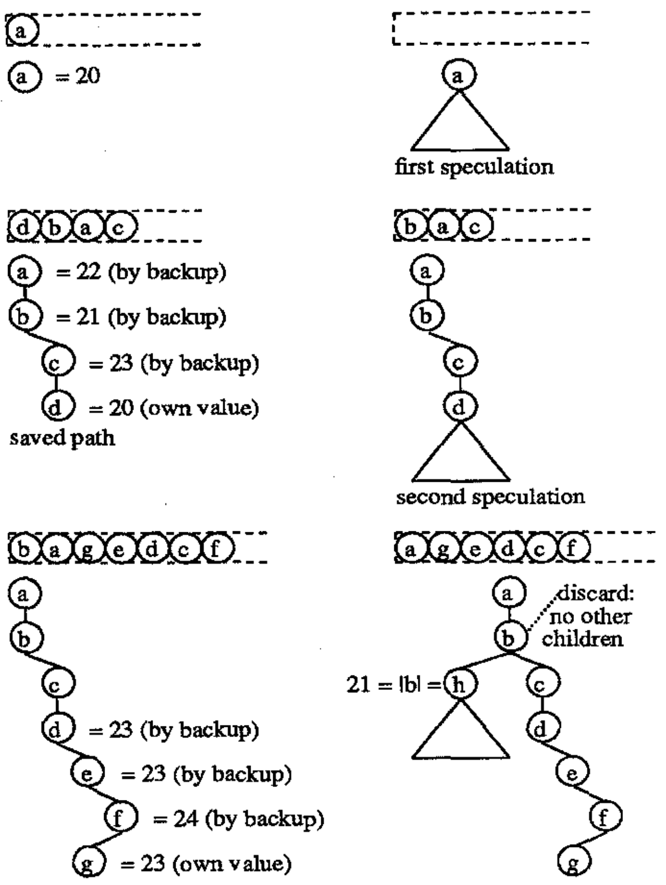

Figure 2.1: Speculative branch-and-bound ($d = 3$)

During the speculation, we propagate back to the minimum path the values of the leaves. By the end of a speculation, each node of the path has been labeled with the value of the best leaf (apart from the minimum itself) descending from it; thus the nodes of the path bear the best values that we can expect of their descendants should we re-expand them later. (As with MA^*, we back up to a node the best leaf-values reachable via each child.) We enqueue the nodes of the incumbent path, omitting those of cost greater than the upper bound on the solution. We prune the queue as necessary, remove the best node, and perform another speculation. Figure 2.1 shows a few operations of *SB&B*.

2.1. Details of Algorithm

First note that for *SB&B* we store some extra information in each node that we enqueue. Namely, a node has a flag that is set if it is a *virgin*; *i.e.*, none of its children has been expanded. The starting node and all minima are initially virgins. A node also has a flag for each child, telling whether or not the child may be expanded as a rootlet when the node is selected from the queue. Finally, for each child eligible for expansion, the node has the backed-up value of its best descendant leaf.

If the node removed from the queue is a virgin, we use it as the actual rootlet of a speculation. Otherwise, we generate its most-promising eligible child, and speculate from that, returning the parent to the queue, its flag for the chosen child set to ineligibility and the value of the parent suitably altered.

We can back up leaf-values during the depth-first search without much difficulty. We alter the stack used for speculation, adding extra slots to its frames; these hold values backed up from each child of the node. We must also keep track of how large a section of the current minimum path and the stack is identical, using a counter called *common*.

When a speculation reaches its first minimum, we copy the entire state of the stack at that point to the minimum path. Stack and path are therefore identical at this point: we set *common* to the depth of the leaf. Now, when we backtrack a level, we both pop the stack and decrement *common*; *common* remains unaltered when we generate new nodes on the same level or deeper.

When do we propagate the values of leaves upwards to their ancestors? The rule of thumb is simple: *propagate the value of a node upwards only when you have searched all the node's descendants in the speculation.* Suppose that we speculate to a fixed depth d. We immediately propagate the value of leaves l_{i0}, l_{i1}, and so on to their parent p_i at depth $d-1$; however, we needn't propagate the value of p_i to the grandparent g at depth $d-2$ until we've generated all the children $l_{i,j}$ of p_i. Only when we know for certain the best value of the children of p_i, and hence the final value p_i takes in this speculation, should we propagate it up to g. The same principle holds for the upper levels of the speculation. Values trickle up from the leaves, each node of the path acquiring the value of its least-costly child (not including children on the path).

Often we will not generate an entire speculation-tree: some nodes may have values exceeding the upper bound, or we may find a solution (not necessarily optimal) shallower than the depth of speculation. In such cases we can simply propagate an "infinite" value back to the parent of the node, showing that this child is not available for further expansion and therefore has no value to propagate upwards. If all leaves or their ancestors have values exceeding the upper bound, the speculation fails.

Note also that propagation can potentially alter all backed-up values in the stack, but it alters only those of the incumbent path that are in common with the stack. In this way the incumbent path always gets the correct backed-up values for its nodes.

```
Procedure s_b&b:
(1) [Initialize]
  Queue contains the root node s.
  Set global upper bound $u = \infty$; $best\_sol = nil$.
(2) [Termination test]
  If Queue is empty, stop; best_sol is the solution.
(3) [Selection]
  Dequeue the node n of lowest cost.
(4) [Speculation]
  $u =$ minimum of u and upper-bound value of n.
  Set solution = speculate(n, Queue, u).
  If solution is nil, go to (2).
(5) [Comparison]
  If $cost(solution) < cost(best\_sol)$, $best\_sol = solution$.
(6) [Pruning]
  prune(Queue, u).
  Go to (2).
```

Figure 2.2: Main procedure of *SB&B* algorithm.

```
node Function speculate(n, Queue, u)
(1) [Determine the rootlet]
  If n is a virgin,
    Set rootlet = n.
  else
    Set rootlet = best eligible child of n.
    Mark rootlet's record in n as ineligible.
    If n has eligible children,
      Give n the value of its best eligible child;
      Return n to Queue.
  If rootlet is a solution, return rootlet.
(2) [Further initialization]
  Set $depth = common = 0$; $minimum = lb = \infty$; $solution = nil$.
  Translate rootlet into node and an initial frame frame.
  Push frame onto the stack.
(3) [Make a new frame]
  Examine the frame atop the stack.
  Make frame frame for its 1st untried child; alter node.
  If no such frame can be made, go to (7).
  Push frame onto the stack.
  Increment depth; set lb = lower-bound value of node.
(4) [Test for too-large node]
  If $lb > u$ go to (6).
(5) [Test for solution, leaves]
  If node is a solution,
    Set $u = lb - 1$; set solution = node.
    Propagate values up stack & min_path (using common).
  else if $depth = d$
    Propagate values up stack & min_path (using common).
    If $lb < minimum$
      Set $minimum = lb$.
      Set u = min. of u and upper_bound(node).
      Copy stack to min_path.
      Set $common = d$.
    Undo last frame & changes to node.
  Pop the stack.
  Decrement common and depth.
  Go to (3).
(6) [New node too large]
  Propagate values up stack & min_path (using common).
  Undo last frame and changes to node.
  Go to (3).
(7) [Cannot make a child]
  Pop the stack.
  If the stack is not empty, go to (3).
(8) [Cleaning up]
  Create nodes from node and min_path.
  Put those of value $\le u$ on Queue.
  Return solution [usually nil].
```

Figure 2.3: Speculation procedure for *SB&B* algorithm

2.2. A More Formal Description

Figures 2.2 and 2.3 present the algorithm for *SB&B* in a more-precise form. We have a priority-queue *Queue*, a global upper-bound *u*, and assorted procedures for pruning the queue, enqueuing and dequeuing nodes, and so forth.

We do speculations by calling a function *speculate* (Figure 2.3) that speculates from a node n or its child, putting the minimum path onto *Queue* and altering u as needed. It translates the rootlet into a single record *node* and an initial frame that it pushes onto a stack. It then uses the stack for a conventional depth-first search (the speculation), altering *node* with each change of frame. Whenever a new minimum appears, it copies the stack to the minimum-path; it calls a procedure *propagate* to pass leaf-values up the stack and the portion of the saved minimum-path that coincides with it. At the end of the speculation, it constructs nodes by combining information from *node* and the saved minimum-path, and puts them on *Queue*.

2.3. Variants

SB&B speculating to a fixed depth d is analogous to depth-first iterative deepening. Instead, we can make speculations mimic the iterations of IDA^*. To speculate, we do a depth-first search from the rootlet in the style of IDA^*, using the initial value of the rootlet as the incumbent. We backtrack on reaching nodes of value higher than the incumbent — just as with IDA^*. In this way we get an analog to IDA^* which we call *Speculative* IDA^*, or $SIDA^*$. We show in Section 3 that $SIDA^*$ is optimal given an admissible heuristic, and typically expands fewer nodes than IDA^* — as with MA^*, the savings is linear in the size of the extra memory used. As with the other methods, $SIDA^*$ can use an incumbent larger than the smallest value borne by the previous stage, at the cost of not finding an optimal solution initially.

We can also use a local A^* speculation, expanding just enough new nodes each time to ensure that the number of new nodes is sufficiently large to limit repetition. This requires that we keep track of how many descendants of each node we have already expanded. One problem: successive speculations may grow larger and larger, with more nodes each time, in order to ensure good progress; eventually they may grow too large to fit in a computer's memory.

3. BEHAVIOR OF ALGORITHM

How does SB&B behave? Namely, how many nodes can we expect it to expand, compared to a best-first search, or to IDA^* or MA^*, and how much memory can we expect it to use? What problems might *SB&B* have in common with the other searches?

3.1. Node-Expansions

Consider a simplified view of search, such as Korf used in analyzing the performance of IDA^* [2]. Suppose that our search tree has a uniform branching-factor b (each interior node has b children), and all speculations are to a fixed depth d. Every node seen in the initial speculation is new. But what of the speculations starting from the nodes of the saved path? The worst case is for the $b-1$ speculations spawned when s is re-selected from the priority queue. With these, only the deepest level contains nodes not seen before. Figure 3.1 shows the situation: only the solid lines of the picture lead to new nodes; the dashed lines show repetitions.

Consider the rootlet of a speculation as level 0. A speculation to depth d, given the uniform b, expands $\frac{b}{b-1}(b^d-1)$ nodes [9]. For the worst-case speculations mentioned above, the first $d-1$ levels, or $\frac{b}{b-1}(b^{d-1}-1)$ of these, are old, or almost exactly $\frac{1}{b}$ of the total. Hence, even for the worst-case speculations, at

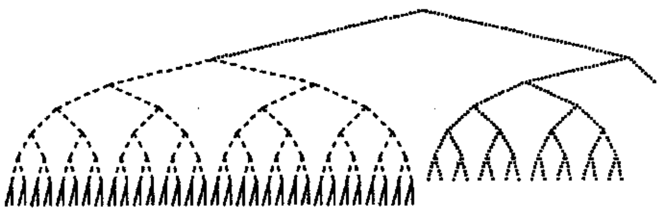
Figure 3.1: Worst-case repetition for $b=2$, $d=6$

least $\frac{b-1}{b}$ of the nodes are new.

A similar result holds for IDA^* under such a model [2]. This is unrealistic for some real-life searches, and the result is therefore questionable, but we will see that IDA^*, MA^*, and both *SB&B* and $SIDA^*$ encounter similar problems when the search tree is not so "well-behaved."

It should be obvious that in examining an entire stage, $SIDA^*$ never examines more nodes than does IDA^*. Both methods look at the same number of nodes, but $SIDA^*$ already has the saved nodes stored in memory, and so it does not re-expand them or their ancestors. Neither method typically examines the entire final stage, but $SIDA^*$ looks at it using many small speculations, performed best-rootlet-first, whereas IDA^* uses a single search. Typically $SIDA^*$ will find a solution sooner, just as an informed depth-first search typically does better than a blind one.

3.2. Space

In the worst case, A^* stores a number of nodes exponential in the size of the problem. *SB&B* generates at least $\frac{b}{b-1}(b^d-1)-\frac{b}{b-1}(b^{d-1}-1)=b^d$ new nodes with every speculation, but stores only d of them. Therefore, under this simplified model, *SB&B* stores $\frac{d}{b^d}$ of the nodes stored by A^*. Hence the worst-case use of memory for *SB&B* is still exponential, but with a very small constant. A judicious choice of d (and b, if the nature of the problem makes that practical) can make an *SB&B* search fit in a primary memory that could not have held the nodes of an A^* search. Table 3.1 shows the ratio for various values of b and d. If this does not control the use of memory tightly enough, we can add to *SB&B* a pruning mechanism similar to that of MA^*. Such pruning keeps use of memory under a fixed amount, though at an increase in overhead. We describe this in Section 4.

3.3. A Family of Search Methods

We can consider IDA^*, MA^* and *SB&B* ($SIDA^*$ specifically, for the purposes of the following discussion) as members of a family of search methods. MA^* is essentially a generalization of IDA^* [1], and $SIDA^*$ with pruning is a generalization of MA^*. The problem is that all of these methods depend for their effectiveness, to some degree, on this assumption: *The number of nodes in the search tree increases exponentially with the depth of the search tree or the lower-bound value of the nodes.*

In other words, we can look at the tree generated in a search, even though the upper-bound function causes pruning and the representation of the problem lowers the branching-factor as we go deeper, and there is still some "*de facto*" branching-factor $\bar{b}$, $\bar{b}>1$, such that level $i+1$ of the tree has at least $\bar{b}$ times as

Table 3.1: Effects of b and d on saving of nodes.

b	d	$\frac{\textit{nodes expanded}}{\textit{nodes stored}}$
2	2	2
	4	4
	8	32
	16	2048
3	2	4.5
	4	20
	8	820
	16	2690420
4	2	8
	4	64
	8	8192
	16	268435456

many nodes as has level i, or such that about $\bar{b}$ times as many nodes have values in the range $(h, h+\delta]$ as in the range $(h-\delta, h]$, for some $\delta>0$ (presumably $\delta \ll h$). If there is this dependence on depth, we can speculate to a fixed depth with *SB&B*, or do *DFID*, and know that most nodes of a speculation or a stage will be new. Similarly, the dependence on value lets us do *IDA**, *MA**, or *SIDA**, and know that most nodes of a stage, path, or speculation are new.

For what problems does the number of nodes increase exponentially with depth or value? Note that the papers on *IDA** [2, 3] and *MA** [1] concentrate on the Fifteen Puzzle. The usual lower-bound heuristic based on Manhattan distance behaves very well: values of a parent and its child differ by either zero or two. Hence the search tree has "slices," at least one level thick at their thinnest, containing nodes of equal value. Even a slice one level thick typically has as many nodes as all higher levels of the tree combined, because the average branching-factor for the Puzzle is 2. An upper-bound function would effectively reduce the branching factor, especially deeper in the tree, but apparently no such function exists. Thus the three methods all do well with the Fifteen Puzzle because the amount of work deep in search tree overwhelms even the oft-repeated work near its top.

Korf [3], notes that *IDA** as presented originally [2] performs poorly on the traveling-salesman problem (TSP), but claims that by using thresholds sufficiently exceeding the value of the minimum leaf of the previous stage, we can ensure that the next stage has overwhelmingly many new nodes. Analogous claims can be made for the other methods. For *SB&B* or *DFID* we can increase d, for *SIDA** or *MA** we can increase the threshold. We are trying to avoid the following: If there are many discrete values of node, each borne by relatively few nodes, setting the threshold to successive values means that only a small fraction of the nodes explored are new. (Note that all nodes of lower value are re-expanded at each stage.) Therefore the search covers the same ground repeatedly. New speculations of *SB&B* or *DFID*, or new paths of *MA**, will be similarly repetitive.

Smith [8] has studied *relaxation-guided* branch-and-bound searches, particularly for the asymmetric TSP. (A relaxation of a problem is a version of it with some constraint removed; a solution to the original also solves the relaxation.) Frequently such methods give excellent lower bounds on the cost of an optimal solution. Smith, using random trees, found that for such searches, the size of a depth-first search tree is *linear* in the depth of the first leaf (the length of the path to the leftmost solution). This implies that when heuristics are strong, the number of nodes might not increase exponentially with depth or cost. The exponential increase that *IDA** and related searches required would demand an exponential increase of the threshold.

Korf [2] quotes results summarized by Pearl [5] to justify *IDA** (and hence of related methods including *SIDA**): if a heuristic used by A^* exhibits constant relative error, the number of nodes generated by A^* (and hence in the space that one of the methods must search) grows exponentially with depth. A heuristic must have logarithmic relative error for the growth to be polynomial, and assumes that the branching factor b is constant. The model applies well in the Fifteen Puzzle problem. No alternative methods have been developed when the above assumption is invalid.

The *IDA** family of search algorithms expand more nodes in cases in which either the bounding functions are tight or solutions to a problem lie at a known level of the search tree. For example, if a heuristics for some problem in *NP* is not based on relaxation, solutions may be assembled one element at a time: a child has one more element of a partial solution than its parent, and solutions all have the same size. In this case, an informed depth-first search to the right depth is obviously superior. In problems in which the bounding functions are tight, an informed depth-first search generally searches approximately the same number of nodes as a best-first search [10].

What happens if the incumbent for *IDA**, the d for *SB&B*, or the change in node-values for *SIDA** is too large? Each stage or speculation expands overwhelmingly many new nodes, but most of these might be superfluous: A^* would not have expanded them. Say that an optimal solution has value 34: A^* dequeues only nodes of value 34 or less, and expands (at worst) all of their children. *IDA** (say), with an outsize incumbent of 38 on its final stage, might well expand almost all nodes of value 38 or less, and most of their children.

A good upper-bound function for a problem also renders members of the *IDA** family less useful. If speculations or stages go too deep, they are cut off not by their own limits (d or incumbent), but by the upper-bound function. In that case, an informed depth-first search would search about the same space, but without repetitions, and therefore fare better.

3.4. Optimality of SIDA*

The argument of Lemma 6.3 of Korf's paper [2], that *IDA** finds a solution of least cost if one exists, holds for *SIDA** as well. *SIDA** also expands all nodes of a given cost before proceeding to those of higher cost, given an admissible heuristic function. The optimality argument follows the one for *IDA** (Korf's Theorem 6.4).

4. ADDING PRUNING

Although *SB&B* uses substantially less memory than does A^*, it can still use an exponentially large amount. Sometimes we cannot adjust b and d to ensure that the search fits in a machine's primary memory; *e.g.*, some problems may be hard to recast to make b large; b tends to shrink with depth. Too large a d, and the search degenerates into depth-first. Also, it may be hard to estimate precisely the number of distinct nodes we will examine in a search; we may have to bound it loosely and choose a large d to ensure that the search fits in memory.

SIDA* is the most important case: increasing the depth of speculation destroys optimality. Therefore we will present a version of *SIDA** with pruning, and then discuss its behavior. Pruning is possible with other forms of *SB&B*, but it requires more bookkeeping (*e.g.*, keeping count of descendants).

4.1. Informal description

The original *SB&B* simply enqueues the nodes generated by speculations. Now it must also keep track of *saved paths*, not just their nodes in another priority queue. This *Path* queue holds a *header* for each path; a header has information about its path, and the head of a doubly-linked list through the path's nodes. Nodes now point to their header and the neighbors in their path.

A speculation generates a path: enqueue the nodes, as usual, but also put their header on the *Path* queue. Initially a header bears the value of its speculation's minimum leaf, but that changes when child paths are pruned. The header keeps a count *Children* of child paths, a pointer *Parent* to the parent of the rootlet, a pointer *Rootlet* to the rootlet, and a pointer *Minimum* to the minimum leaf. Note that after a successful speculation from a node (*i.e.*, do a speculation that reaches at least one leaf, and therefore returns a path), we increment *Children* in the header of its path.

Disjoint paths keep the algorithm tidy: we no longer treat the minimum leaf as part of the path leading to it. We enqueue it as usual; it generally becomes the rootlet of its own speculation, and hence the first node of another path. (The pointer from the associated header lets us find it if we must delete it before then.)

When use of memory exceeds the threshold *MAX*, we delete paths. The headers on the *Path* queue are in order of suitability for pruning: worst-first and newest-first; the first path the *Path* queue has no child paths. (Because the lower-bound function is admissible, children of a path have values no less than that of the path itself, and they are newer — hence they appear earlier in the *Path* queue than their parent.) Dequeue the first header, and, following the links down its path, delete its elements from the main queue. If that minimum leaf associated with a path has not become the rootlet of a later speculation, delete it as well.

Before we delete a path, we alter the corresponding child-value stored in its parent node, and, if necessary, the parent's own value. We decrement the count of active child paths, using the pointer from the parent node to the header of *its* path. If the path that we are deleting is the last child path, we update the value of the parent path, scanning through its nodes and copying the best value to the header. If this changes the value of the path, we remove it from the *Path* queue and re-insert it before all nodes of equal value: because it no longer has child paths, it is at least as eligible for deletion as any other path of its value. Any paths we later generate will be inserted near to the front of the *Path* queue, because they will be newer and have values no lower, and therefore be prone to deletion sooner.

Figure 4.1 describes of *SIDA** with pruning more formally. The *speculate* function is much like that in Figure 2.3, except that it creates a path header and sets the added pointers in the nodes to make the doubly-linked list representing the path.

4.2. Behavior

It should now be obvious that *SIDA** with pruning is a generalization of *MA**. Instead of searching only one level deep, *SIDA** goes all the way to the bottom of an *IDA**-stage. In earlier stages, *MA** may expand fewer nodes than would *SIDA**, because *MA** quickly fills the allotted memory with nodes, whereas *SIDA** takes longer. Recall that the savings in searching a stage

```
Procedure pruning_s_b&b(Min, Max):
(1) [Initialize]
  Queue contains the root node s.
  PathQueue is empty.
  Set global upper bound $u = \infty$; best_sol = nil.
(2) [Termination test]
  If Queue is empty, stop; best_sol is the solution.
(3) [Selection]
  Remove from Queue the node n of lowest cost.
(4) [Speculation]
  Set u = minimum of u and upper-bound value of n.
  Set solution = speculate(n, Queue, u, header).
  If header is not nil,
    Increment count of Children in header of n's path.
    Set header.Children = 0.
    Enqueue header on PathQueue.
  If less than Max memory is in use, go to (8).
(5) [Pruning to conserve memory]
  Get header h at the head of PathQueue.
  Delete from Queue all nodes of h's path.
  Propagate value of h to its parent p; adjust p as needed.
  Decrement Children in header of p's path P.
  If P.Children > 0,go to (7).
(6) [Updating parent path]
  Start from P and scan its path for the smallest cost c.
  If c > P.cost
    P.cost = c.
    Remove P from PathQueue; restore with its new value.
(7) [Test to stop pruning]
  If more than Min memory is in use, go to (5).
(8) [Loop]
  If solution is nil, go to (2).
(9) [Comparison]
  If cost(solution) < cost(best_sol), set best_sol = solution.
(10) [Pruning to eliminate costly nodes]
  Remove headers h, h.value $\geq$ u, and path [as in (5), (6)]
  For every node n of value $\geq$ u:
    Remove n from Queue.
    Adjust pointers of neighbors of n on path, to skip n.
  Go to (2).
```

Figure 4.1: *SIDA** with pruning.

with *SIDA** or *MA** instead of *IDA** is proportional to the number of nodes in storage. On the other hand, *SIDA** saves the best path of a speculation, which gives it an element of an informed search. The comparison of the two methods will be shown in a future paper.

5. RESULTS

Our results are preliminary, but they show that if we use the same heuristic and the same amount of "overshoot," *SIDA** expands fewer nodes that *IDA**. *IDA** sometimes solves a problem more quickly, however, because its overhead is lower. Table 5.1 shows some results for sets of 100 randomly-generated problems, with given overshoot in the value of the incumbent (written "o = 20," for example) or fixed depth of speculation ("d = 12"). We give average number of nodes and standard deviation for each set. Instances of the jobshop scheduling problem are for 4 jobs, each having 4 tasks, scheduled on 4 machines (hence "4/4/4"); they were generated by adding tasks of random length (in this case, nonzero but less than 64) to a job, checking to ensure that no contiguous tasks demanded the same processor. They were solved using a naive heuristic, showing strengths and weaknesses peculiar to various methods. Using a weak heuristic makes the number of nodes increase exponentially with value; however, all solutions are at the same depth. Conventional *IDA** fares poorly because of the many different values that nodes can take. Adding an "overshoot" to the incumbent value reduces the number of nodes, but *SIDA** still fares better under the same advantage. Note that *SB&B* with fixed *d* does poorly, because the branching-factor drops sharply (from 4 near the root of the tree to 1 just before the leaves). For this sort of problem, even a blind depth-first search fares better than any method of the *IDA** family, because it repeats no nodes and cannot search past the level where the solutions lie.

Table 5.2 shows some results for instances of the Fifteen Puzzle presented by Korf [2] and also used as examples in the paper on *MA** [1]. Usually *SIDA** performs better than *IDA** and not as well as *MA** with no additional memory; however, the *MA** paper does not appear to count its examinations of child nodes as node-expansion. Also, because we do not at this writing have the pruning version of *SIDA** working, to ensure that the search would fit in machine memory, we had to collect these data using *SIDA** with an overshoot of 10 added to the threshold: speculations are expanding needless nodes deep in the tree. Furthermore, for the results to be comparable to those for *MA**, *SIDA** should use informed depth-first search in its speculations.

6. CONCLUSIONS

We have presented a new algorithm, *speculative branch and bound* (*SB&B*), for performing heuristic search in restricted memory. It is analogous to depth-first iterative deepening (*DFID*). We have also presented a variant, *speculative IDA** (*SIDA**), analogous to *IDA**. Both of these use substantially less memory than conventional branch-and-bound or A^* methods. We have also devised a pruning mechanism, analogous to that of *MA**, to restrict use of memory.

All of these methods (*DFID*, *IDA**, *MA**, *SB&B*, *SIDA**) perform best on search spaces for which the number of nodes increases exponentially with depth (*DFID* and *SB&B*) or heuristic value (the others). This requires a heuristic with constant relative error (which is usual), and a representation of the problem that ensures that the branching factor of the search tree is roughly constant with depth.

In the future we will compare *SIDA** more thoroughly to competing methods, and try for some tighter theoretical bounds on its performance. Parallelizing our search methods is straightforward, and we have already run some tests that show near-linear speedups to be possible, and perhaps competitive with other parallel searches such as the parallel *IDA** of Rao *et al.* [6].

Table 5.1: Comparison of methods for 100 random problems.

Problem	Search	$nodes_{avg}$	$nodes_{std.dev.}$
4/4/4 jobshop	*IDA**	148859139	296580397
	IDA*, o = 20	23536998	34971060
	IDA*, o = 30	20381512	29519058
	SIDA*, o = 20	20685240	30757773
	SIDA*, o = 30	20034856	30950953
	blind DFS	15932349	19437159
	SB&B, d = 12	36242055	44740843
	SB&B, d = 13	40707373	44376572
8-puzzle	*IDA**	5864	9218
	SIDA*	4465	8812

Table 5.2: Instances of the 15 Puzzle from Korf's paper.

No.	*IDA**	*MA**(0)	*SIDA**, overshoot = 10
15	543598067	270016222	298196963
30	2196593	898573	1682282
40	63276188	34010298	54167881
45	6158733	2396522	8309799
50	63036422	17989651	62453094
55	927212	454994	391449
85	2725456	1426580	2154883
100	67880056	19221994	70243385

REFERENCES

[1] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar, "Heuristic Search in Restricted Memory," *Artificial Intelligence*, no. 41, pp. 197-221, 1989.

[2] R. E. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.

[3] R. E. Korf, "Optimal Path Finding Algorithms," pp. 223-267 in *Search in Artificial Intelligence*, ed. V. Kumar, Springer-Verlag, New York, 1988.

[4] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, 1980.

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

[6] V. N. Rao, V. Kumar, and K. Ramesh, "A Parallel Implementation of Iterative-Deepening-A*," *Proc. 6th. National Conf. on Artificial Intelligence*, pp. 178-182, Morgan Kaufman, Los Altos, CA, July 1989.

[7] A. K. Sen and A. Bagchi, "Fast Recursive Formulations for Best-Fist Search That Allow Controlled Use of Memory," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 297-302, IJCAI, Inc., Detroit, MI, Aug. 1989.

[8] D. R. Smith, "Random Trees and the Analysis of Branch-and-Bound Procedures," *J. of the ACM*, vol. 31, no. 1, pp. 163-188, 1984.

[9] M. E. Stickel and W. M. Tyson, "An Analysis of Consecutively Bounded Depth-First Search with Applications in Automated Deduction," *9th IJCAI*, pp. 1073-5, 1985.

[10] B. W. Wah and C. F. Yu, "Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search," *Trans. on Software Engineering*, vol. SE-11, no. 9, pp. 922-934, IEEE, Sept. 1985.