Improving the Efficiency of Limited-Memory Heuristic Search*

Subrata Ghosh
Department of Computer Science
University of Maryland
College Park, MD 20742
USA
Telephone: (301) 405-2717
Fax: (301) 405-6707
subrata@cs.umd.edu

Ambuj Mahanti
Indian Institute of Management Calcutta
Diamond Harbour Road
P.O. Box No. 16757
Calcutta 700 027
INDIA

Dana S. Nau
Department of Computer Science and
Institute for Systems Research
University of Maryland
College Park, MD 20742
USA
Telephone: (301) 405-2684
Fax: (301) 405-6707
nau@cs.umd.edu

Address all correspondence to Dana S. Nau

*Supported in part by NSF Grants NSFD CDR-88003012, IRI-9306580, and CMDS project (work order no. 019/7-148/CMDS-1039/90-91).

Abstract

This paper describes a new admissible tree search algorithm called Iterative Threshold Search (ITS). ITS can be viewed as a much-simplified version of MA* [2], and a generalized version of MREC [15]. ITS's node selection and retraction (pruning) overhead is much less expensive than MA*'s. We also present the following results:

1. Every node generated by ITS is also generated by IDA*, even if ITS is given no more memory than IDA*. In addition, there are trees on which ITS generates $O(N)$ nodes in comparison to $O(N \log N)$ nodes generated by IDA*, where N is the number of nodes eligible for generation by A*.

2. Experimental tests show that if the heuristic branching factor is low and the node-generation time is high (as in most practical problems), then ITS can provide significant savings in both number of node generations and running time.

3. Our experimental results also suggest that on the Traveling Salesman Problem, both IDA* and ITS are asymptotically optimal on the average if the costs between the cities are drawn from a fixed range. However, if the range of costs grows in proportion to the problem size, then IDA* is not asymptotically optimal. ITS's asymptotic complexity in the latter case depends on the amount of memory available to it.

Key words: heuristic search, node generation, pruning, memory bound

1 Introduction

Although A* is usually very efficient in terms of number of node expansions [3], it requires an exponential amount of memory, and thus runs out of memory even on problem instances of moderate size. This problem led to the development of algorithm IDA* [8]. IDA*'s memory requirement is only linear in the depth of the search, enabling it to solve larger problems than A* can solve in practice. However, when additional memory is available, IDA* does not make use of this memory to reduce the number of node expansions. This led to the development of several other limited-memory heuristic search algorithms, including MREC and MA*. In this paper, we present the following results:[1]

1. We present a new admissible tree search algorithm called Iterative Threshold Search (ITS). Like IDA*, ITS maintains a threshold z, expands each path until its cost exceeds z, and then revises z. But if given additional memory, it keeps track of additional nodes, and backs up path information at parents when nodes get pruned. ITS can be viewed as a much simplified version of MA*, and a generalized version of MREC. ITS's node selection and retraction (pruning) overhead is much less expensive than MA*'s.

2. We have proved that ITS dominates IDA*; i.e., even if ITS is given no more memory than IDA*, every node generated by ITS is also generated by IDA*. In addition, we demonstrate that there are cases in which ITS generates $O(N)$ nodes in comparison to $O(N \log N)$ nodes generated by IDA*, where N is the number of nodes eligible for generation by A*.

3. We present extensive experimental tests on ITS on three problem domains: the flow-shop scheduling problem, the 15-puzzle, and the traveling salesman problem. Our results show that if the heuristic branching factor is low and the node-generation time is high (which is

[1] Some of these results have also been summarized briefly in [5].

the case for most practical problems), ITS can provide significant savings in both number of node generations and running time.

4. Our experimental results also suggest that on the Traveling Salesman Problem, both IDA* and ITS are asymptotically optimal on the average if the costs between the cities are taken from a fixed range. However, if the range of costs grows in proportion to the problem size then IDA* is not asymptotically optimal. ITS's asymptotic complexity in that case depends on the amount of memory available to it.

2 Background

The objective of many heuristic search algorithms is to find a minimum cost solution path in a directed graph G. To find such a path, these algorithms use a node evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of a minimum cost path currently known from the start node s to n, and $h(n) \geq 0$, the heuristic value of node n, is an estimate of $h^*(n)$. $h^*(n)$ is the cost of a minimum cost path from n to a goal node. In this paper, we assume that the heuristic function h is admissible, i.e $\forall n \in G$, $h(n) \leq h^*(n)$. The cost of an arc (m, n) in G is denoted by $c(m, n)$.

Let P be any path from the start node s. Then the function pathmax(P) [1] is defined as follows:

$$\text{pathmax}(P) = \max_{n \in P}(c(P, s, n) + h(n)),$$

where $c(P, s, n)$ is the cost of the subpath of P that goes from s to n.

We use L to denote the maximum number of nodes on any path P for which pathmax(P) $\leq$

$h^*(s)$. Since the number of nodes on any path is one more than the length of the path,[2]

$$L = 1 + \max\{length(P) : \text{pathmax}(P) \leq h^*(s)\}.$$

2.1 Algorithm A*

The best known admissible algorithm is A* [6, 11]. A* works in a best-first manner. It maintains two lists: OPEN, which contains nodes that are to be expanded, and CLOSED, which contains nodes that have already been expanded. At each iteration, A* selects a node n from OPEN with minimum f-value, generates all of its children, and puts these children into OPEN after setting their g, h, and f-values. If a child p of n is already present in OPEN and $g(p) > g(n) + c(n,p)$, then $g(p)$ is reset to $g(n) + c(n,p)$. If a child p of n is already present in CLOSED and a better path to it is now found, then $g(p)$ is reset to the newly found smaller path cost and p is brought back to OPEN from CLOSED. The process of node selection and expansion continues until a goal node is selected for expansion.

One major problem with A* is the amount of memory required to store nodes in OPEN and CLOSED. As shown in [3], every admissible search algorithm must expand all *surely expandable nodes* before finding a solution. The number of such nodes often grows exponentially with some measure of the problem size—and since A* keeps track of all of these nodes, it needs an exponential amount of memory in which to store these nodes.

2.2 Algorithm IDA*

To overcome the storage problem, a variant of A* called IDA* (Iterative Deepening A*) was introduced by Korf [8, 9]. Basically, IDA* performs a depth-first search, backtracking whenever

[2]The length of a path P is the number of arcs in P.

```
procedure IDA*:
    Let z, z' be global variables.
    Set z := h(s), where s is the start node. Set z' := ∞.
    Do the following steps repeatedly:
        set P := the path containing only s;
        call Depth-First(P);
        set z := z'.

procedure Depth-First(P):
    Set f := cost(P) + h(last(P)).
    If f > z, then set z' := min(z', f).
    Otherwise, if last(P) is a goal node, then exit from IDA*, returning P.
    Otherwise, do the following steps for every child n of last(P):
        set P' := the path formed by appending n to P;
        call Depth-First(P').
```

Figure 1: Pseudocode for IDA*.

it finds a path whose cost exceeds a *threshold* value z. It repeats this search for larger and larger values of z, until it finds a solution. Figure 1 shows a pseudocode version of IDA*.

Unlike A*, IDA* is a tree search algorithm—in other words, it does not keep track of alternate paths to each node. Because IDA* only keeps track of the nodes on the path it is currently exploring, it requires $O(L)$ memory. In other words, IDA*'s memory requirement grows only linearly with the depth of the search. This enables IDA* to solve much larger problems than A* can solve, such as the 15-puzzle [9]. Thus IDA* has drawn significant attention from the AI research community.

2.3 Other Limited-Memory Algorithms

Following IDA*, several other limited-memory algorithms have been designed to reduce the number of node generations compared to IDA*. These algorithms can be categorized into two classes: (1) the first class uses additional memory to store more nodes than IDA*, and thereby reduce regeneration of some nodes. The algorithms which belong to this class are MREC, MA*, RA* [4], SMA* [13], and ITS, and (2) the second class of algorithms attempts to reduce node regenerations by reducing the number of iterations, by increasing the threshold more liberally than IDA*. IDA*_CR [14], DFS* [12], and MIDA* [16] belong to this class.

Like IDA*, MREC is a recursive search algorithm. The difference between MREC and other algorithms in its class is that MREC allocates its memory statically, in the order in which nodes are generated. Algorithm MA* makes use of the available memory in a more intelligent fashion, by storing the best nodes generated so far. MA* does top-down and bottom-up propagation of heuristics and generates one successor at a time. RA* and SMA* are simplified versions of MA*, with some differences.

Although algorithms MA*, RA*, and SMA* are limited-memory algorithms, their formulation is more similar to A*'s than IDA*'s. They all maintain OPEN and CLOSED, select the best/worst node from OPEN for expansion and pruning. Therefore, their node generation/pruning overhead is much higher than IDA*'s. As a result, although they generate fewer nodes than IDA*, they do not always run faster than IDA*. ITS's formulation is similar to IDA*'s and therefore has a low node-generation overhead than any of them.

Algorithms IDA*_CR, MIDA*, and DFS* work similar to IDA* except that they set successive thresholds to values larger than the minimum value that exceeded the previous threshold. This reduces the number of iterations and therefore the total number of node generations. However, unlike IDA*, the first solution found by these algorithms is not necessarily optimal and therefore to guarantee optimal solution, these algorithms revert to depth-first branch-and-bound in the last iteration.

It should be noted that the techniques used in the two classes of algorithms can be combined.

3 Algorithm ITS

Most heuristic search algorithms maintain a search tree T containing data about each node n that has been installed in the tree. Nodes of G are generated one at a time and installed into T, until a solution path is found in T that duplicates the least-cost solution path of G. Usually the branches of T are represented only as links among the data structures representing the nodes. However, ITS (see Figure 2) maintains heuristic information not only for each node of its search tree, but also for each branch of the tree. Thus, rather than considering a branch (p, q) merely to be a link between the node p and its child q, we consider it as a separate entity in T.

Conceptually, ITS installs (p, q) into T at the same time that it installs p into T, even though ITS has not yet generated q. It is possible to implement such a scheme without incurring the overhead of generating all of p's children, by creating one branch $(p, R(p))$ for each operator R applicable to p without actually invoking the operator R. A *tip branch* of T is a branch (p, q) in T such that q is not in T. A *tip node* of T is a node p of T such that every branch (p, q) in T is a tip branch. Such nodes are eligible for *retraction* by ITS. Retracting p consists of removing from T the node p and every branch (p, q).

For each branch (p, q) in T a variable B is maintained, which stores an estimate of the cost of the minimum cost solution path containing the branch (p, q). $B(p, q)$ is initialized to $f(p) = g(p) + h(p)$, when the node p is installed in T. However, unlike the f value of a node, $B(p, q)$ is updated every time the node q is retracted.

S is the amount of storage (number of nodes) available to ITS.

Remark. For any given $S \geq 0$, ITS never stores more than $\max(L, S)$ nodes in memory where L is as defined in Section 2. Thus, if the given memory S is more than L nodes, then it uses S amount of memory, otherwise it uses L amount of memory as used by IDA*.

Algorithm ITS:

1. Call Install$(s, 0)$.
2. Do the following steps repeatedly:
   (a) Set $z \leftarrow \min\{B(p,q) : (p,q)$ is a tip branch$\}$.
   (b) Do the following steps repeatedly, until $B(p,q) > z$ for every tip branch (p,q):
      i. Select the leftmost tip branch (m,n) such that $B(m,n) \leq z$.
      ii. If m is a goal node then EXIT, returning $g(m)$.
      iii. If $n = $ DUMMY, then set $B(m,n) \leftarrow \infty$. Otherwise, do the following:
         A. If T contains $\geq S$ nodes and has at least two tip nodes, then retract a node, as follows:
         If there is a tip node x such that $B(x,y) > z$ for every branch (x,y), then let q be the leftmost such node. Otherwise, let q be the rightmost tip node of T. Set $B(p,q) \leftarrow \min_r B(q,r)$, where p is q's parent in T. Remove q and every branch (q,r) from T.
         B. Call Install$(n, g(m) + c(m,n))$.

procedure Install(n, g)**:**

1. Put n into T.
2. If there are no operators applicable to n, then put a dummy branch $(n,$ DUMMY$)$ into T. Otherwise, for each operator R applicable to n, put a branch $(n, R(n))$ into T.
3. Set $g(n) \leftarrow g$.
4. For each branch (n,r), set $B(n,r) \leftarrow g(n) + h(n)$.

Figure 2: Pseudocode for ITS.

Example 1 Consider the search tree G shown in Figure 3. Arc costs are shown beside each arc. Heuristic value at each node is zero. The doubly circled nodes are the goal nodes. When ITS is run on this tree with $S = 3$ nodes, the explicit trees after each node generation are shown in Figures 4(a) through (m).

First, ITS generates the root node 1, creates three branches for its three children and initializes the B-value of each branch to $g(1) + h(1)$=0, as shown in Figure 4(a). The threshold z is set to 0 (the minimum of the B-values of the three tip branches). It then generates nodes 2 and 3 as shown in Figures 4(b) and (c). At this stage, ITS already has 3 nodes in memory and wants to

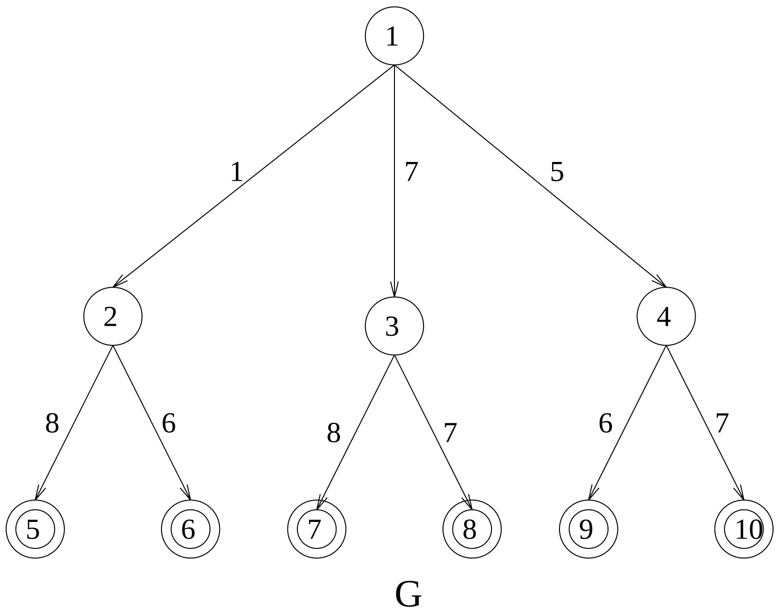

Figure 3: An example search tree.

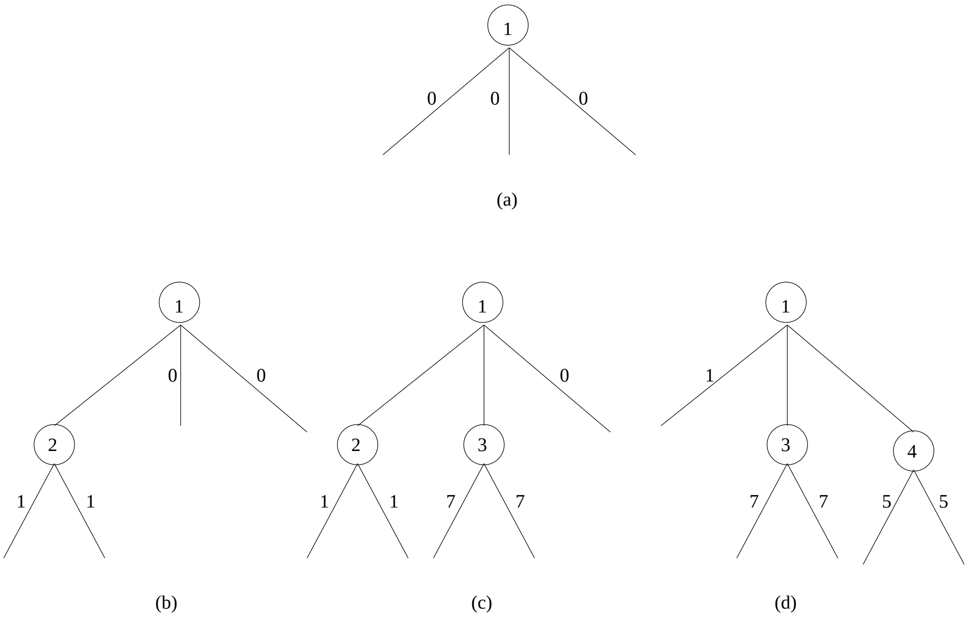

Figure 4: ITS on the example search tree. (*continued on next page*)

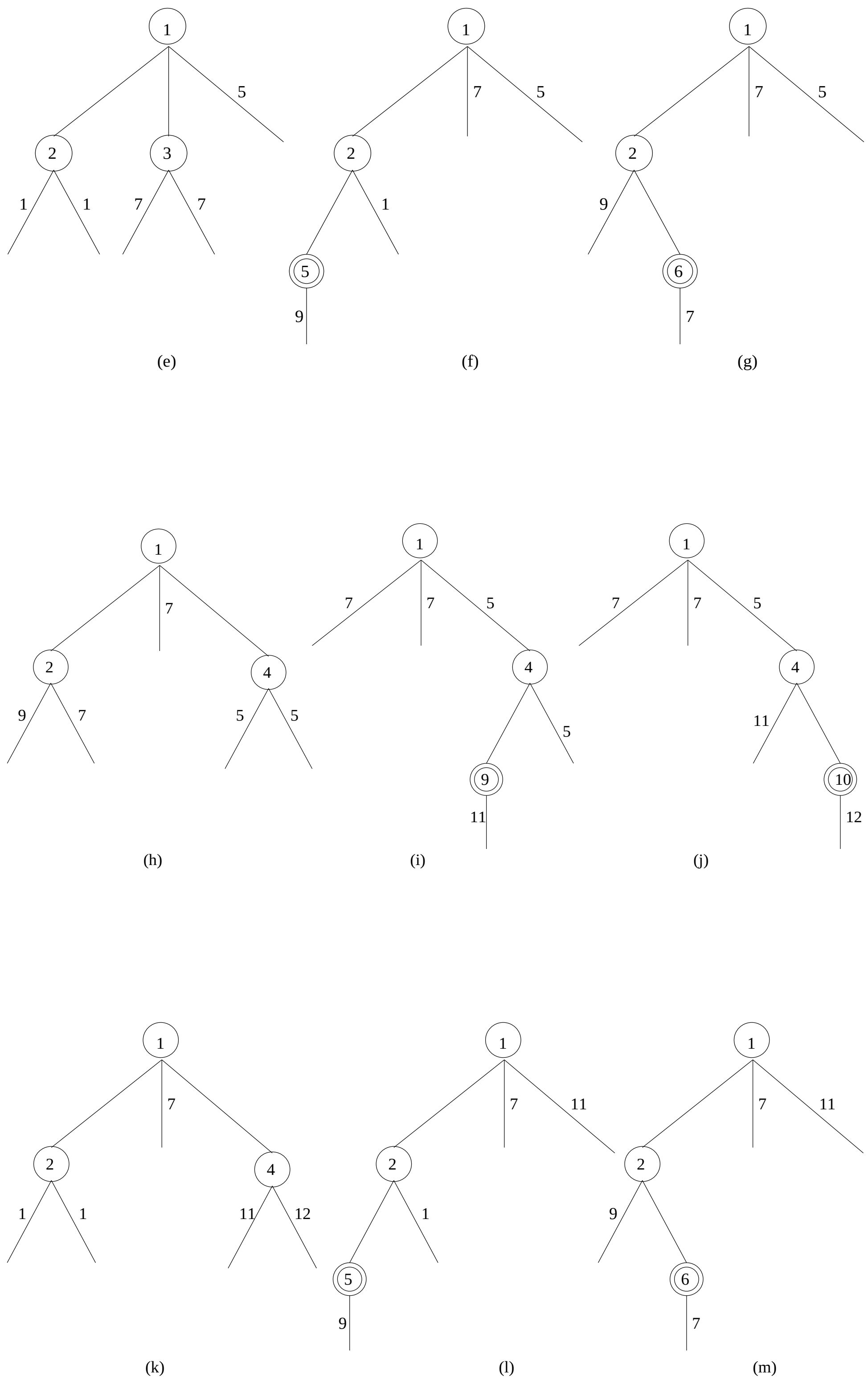

Figure 4 (continued): ITS on the example search tree.

generate node 4. Therefore, it needs to retract a node at this point. It selects the leftmost node (which is node 2), backs up the minimum B-value (which is 1) of its tip branches and removes node 2 from memory. It then generates node 4. The result is shown in Figure 4(d). At this point all tip branches have B-values greater than 0 and therefore the threshold z increases to 1. The node generation and retraction continues as shown in Figures 4(e) through (m). In Figure 4(e), ITS has regenerated node 2 and retracted the rightmost node (which is node 4). Finally, ITS terminates with the optimal solution (of cost 7) when it selects the goal node 6 for expansion after it has reached the stage shown in Figure 4(m).

The node generation sequence for ITS on this tree is as follows:

z	Nodes generated
0	1, 2, 3, 4
1	2, 5, 6
5	4, 9, 10
7	2, 5, 6

On the same tree, the node generation sequence for IDA* is as given below:

z	Nodes generated
0	1, 2, 3, 4
1	2, 5, 6, 3, 4
5	2, 5, 6, 3, 4, 9, 10
7	2, 5, 6

As can be seen, on this example ITS does fewer node generations than IDA*, although both algorithms store at most 3 nodes in memory at any one time. In the next section we will prove that in fact ITS always does fewer node generations than IDA*.

4 Properties of ITS

4.1 Definitions

As we will see later, the quantities defined below are useful to prove some properties of ITS and also for comparison of ITS with IDA*.

If $P = (n_0, \ldots, n_k)$ is any path, then all-but-last(P) is the path $(n_0, \ldots, n_{k-1})$ and $last(P)$ is the node n_k. For each $z \geq 0$, we define

$$
\begin{aligned}
W^G(z) &= \left\{ \begin{array}{l} \text{all paths } P \text{ from } s \text{ such that } \mathrm{cost}(P) + h(\mathrm{last}(P)) > z, \\ \text{pathmax(all-but-last}(P)) \leq z, \text{ and all-but-last}(P) \text{ con-} \\ \text{tains no goal nodes} \end{array} \right\}; \\
V^G(z) &= \{\text{all nodes of all paths in } W^G(z)\}; \\
X^G(z) &= \{\text{all non-tip nodes of all paths in } W^G(z)\}; \\
x^G(z) &= |X^G(z)|; \\
N(G) &= x^G(h^*(s)).
\end{aligned}
$$

In the terms defined above, we will usually omit G if its identity is clear.

From these definitions, it follows immediately that if we run A* on a tree, then every path generated by A* will be a subpath of a path in $W(h^*(s))$, every node generated by A* will be in $V(h^*(s))$, and every node expanded by A* will be in $X(h^*(s))$. Furthermore, in the worst case (which occurs if A* expands all nodes on OPEN that have f-values $\leq h^*(s)$ before it selects a goal node), A* will generate all paths in $W(h^*(s))$ and all nodes in $V(h^*(s))$, and will expand all nodes in $X(h^*(s))$, for a total of $x(h^*(s))$ node expansions. Thus $X(h^*(s))$ is the set of all nodes eligible for expansion by A*, and $N = x(h^*(s))$ is the number of nodes eligible for expansion by A*. The quantity N is known as the number of *possibly expandable* nodes.

For $j = 1, \ldots,$ we inductively define

$$
\begin{aligned}
z^G(j) &= \begin{cases} h(s), \text{ where } s \text{ is } G\text{'s start node}, & \text{if } j = 1, \\ \min\{\text{cost}(P) + h(\text{last}(P)) : P \text{ is a path in } W^G(z^G(j-1))\}, & \text{otherwise;} \end{cases} \\
X^G_{\text{new}}(j) &= \begin{cases} X^G(z^G(1)), & \text{if } j = 1, \\ X^G(z^G(j)) - X^G(z^G(j-1)), & \text{otherwise;} \end{cases} \\
x^G_{\text{new}}(j) &= |X^G_{\text{new}}(j)|; \\
I^G &= \min\{j : \text{there is a solution path } P \text{ such that pathmax}(P) \le z^G(j)\}; \\
x^G_{\text{tot}} &= \sum_{j=1}^{I^G} x^G(z^G(j)) \quad = \quad \sum_{j=1}^{I} |X^G(z^G(j))|.
\end{aligned}
$$

As before, we will usually omit the superscript G if the identity of G is clear.

Suppose we run IDA* on a tree. The definitions of $V(z(j))$ and $z(j)$ presented above correspond precisely to the way that IDA* generates nodes and sets thresholds. Thus it follows that IDA* does I iterations, and that for $j = 1, 2, \ldots, I-1$,

$$\text{the threshold used during IDA*'s } j\text{'th iteration} = z(j); \tag{1}$$

$$\{\text{nodes generated during IDA*'s } j\text{'th iteration}\} = V(z(j)); \tag{2}$$

$$\{\text{nodes expanded during IDA*'s } j\text{'th iteration}\} = X(z(j)); \tag{3}$$

$$\{\text{new nodes expanded during IDA*'s } j\text{'th iteration}\} = X_{\text{new}}(j). \tag{4}$$

Since IDA* generates each node at most once during each iteration, it follows that

$$\text{the number of nodes expanded during IDA*'s } j\text{'th iteration} = x(z(j)); \tag{5}$$

$$\text{the number of new nodes expanded during IDA*'s } j\text{'th iteration} = x_{\text{new}}(j). \tag{6}$$

During the final iteration (i.e., $j = I$), IDA* can find a goal node and terminate before every node of $V(z(I))$ has been generated. Thus

$$\begin{aligned}
\text{the threshold used during IDA*'s } I\text{'th iteration} &= z(I); & (7)\\
\{\text{nodes generated during IDA*'s } I\text{'th iteration}\} &\subseteq V((z(I)); & (8)\\
\{\text{nodes expanded during IDA*'s } I\text{'th iteration}\} &\subseteq X(z(I)); & (9)\\
\{\text{new nodes expanded during IDA*'s } I\text{'th iteration}\} &\subseteq X_{\text{new}}(I); & (10)\\
\text{the number of nodes expanded during IDA*'s } I\text{'th iteration} &\leq x(z(I)); & (11)\\
\text{the number of new nodes expanded during IDA*'s } I\text{'th iteration} &\leq x_{\text{new}}(I); & (12)
\end{aligned}$$

with equality in the worst case. Furthermore, from the correctness of IDA*, it follows that

$$z(I) = h^*(s). \tag{13}$$

The *heuristic branching factor* is defined as the average, over $j = 2, \ldots, I$, of the quantity

$$\frac{x_{\text{new}}(j)}{x_{\text{new}}(j-1)}.$$

Intuitively, this is the average ratio of the number of nodes of each f-value (assuming that the heuristic is monotone) to the the number of nodes at the next smaller f-value [9].

4.2 Basic Properties

For $i = 1, 2, \ldots$, the i'th *instant* in the operation of ITS is the i'th time that Step 2(b)i is executed, i.e., the i'th time that ITS selects a tip branch for expansion. ITS's j'th *iteration* is the j'th iteration of the outer loop in Step 2. ITS's j'th *threshold value* is the value of z during this iteration. We will later prove this is identical to IDA*'s j'th threshold value.

In Theorem 1 below, we prove that no node is generated more than once by ITS during iteration j, and from this it follows that the number of instants in iteration j equals the number of nodes generated in iteration j. At each instant i, ITS either exits at Step 2(b)ii or generates a node n_i at Step 2(b)iiiB. In the latter case, either n_i is a new node (i.e., a node that has never before been generated), or else it is a node that was previously generated and retracted.

Theorem 1 ITS satisfies the following properties:

1. A tip branch (m, n) of T will be selected during an iteration if and only if $B(m, n) \leq z$ during that iteration.

2. The value of ITS's threshold z increases monotonically after each iteration.

3. For each instant i, for each branch (m, n) of T, $g(m) + h(m) \leq B(m, n) \leq \text{cost}(P)$, where P is the least costly solution path containing (m, n).

4. Let i be any instant in iteration j, and suppose that at instant i, ITS selects some branch (m, n) and generates n. Let (n, p) be the leftmost branch from n. Then unless $B(n, p) > z$, (n, p) will be selected at instant $i + 1$.

5. No node is generated more than once during each iteration.

Proof. The proof is by induction. The proof that each property holds at the current inductive step depends on the induction assumption that all of the first four of the properties hold at the previous inductive step. Since the proof is fairly straightforward, below we present an outline of the proof, leaving the details to the reader.

Property 1. This part of the proof involves what happens to a node c generated in Step 2(b)iiiB. If $g(c)+h(c) \le z$, then clearly a branch (c,d) from c will be selected at the next instant. Selection and generation will continue below d until every tip node e below d has $g(e)+h(e) > z$. At this point, another branch (c,d') will be selected. This process will continue until all branches from c have been selected.

Property 2. This follows from the above argument that selection and generation continues to occur until every tip node has an B-value greater than z.

Property 3. When (m,n) is initially installed in T, $B(m,n) = g(m)+h(m)$. The only time $B(m,n)$ is reset is in the case of node retraction—and as explained in the proof of Property 5, this can never decrease $B(m,n)$. Thus $g(m)+h(m) \le B(m,n)$ at every instant i.

The initial value $B(m,n) = g(m)+h(m)$ is clearly $\le \mathrm{cost}(P)$. If ITS subsequently generates and later retracts n, then it sets $B(m,n) \leftarrow \min_q B(n,q)$, which by induction is also $\le \mathrm{cost}(P)$. Thus $B(m,n) \le \mathrm{cost}(P)$ at every instant i.

Property 4. This follows directly from the selection strategy given in Step 2(b)i.

Property 5. From Step 2(b)i of ITS, it follows that once a branch (m,n) is selected, ITS will keep selecting branches below n until every tip branch (q,r) below n has $B(q,r) > z$. Whenever there is a node q below n such that every branch (q,r) from q is a tip branch with $B(q,r) > z$, then ITS may retract q in Step 2(b)iiiA, but if it does this, it will set $B(p,q) > z$ where p is q's parent, and this guarantees that (p,q) will never again be selected during this iteration.

■

Theorem 2 Every path generated by ITS is a subpath of some path in $W(h^*(s))$.

Proof. Suppose not. Let $P = (n_1, n_2, \ldots, n_k)$ be the first path generated by ITS which is not in

$W(h^*(s))$. Let $P' = (n_1, n_2, \ldots, n_{k-1})$. The only way this can happen is if $\text{cost}(P') + h(n_{k-1}) > h^*(s)$. Let i be the instant at which ITS selected the branch (n_{k-1}, n_k).

Let P'' be a minimum-cost solution path in G. Let (p, q) be the tip branch of P'' at the instant i. From Property 3 of Theorem 1, $B(p,q) \leq h^*(s)$. Thus, since ITS selected (n_{k-1}, n_k) rather than (p, q), it must be the case that $B(n_{k-1}, n_k) \leq z \leq h^*(s)$ at the instant i, which contradicts $\text{cost}(P') + h(n_{k-1}) > h^*(s)$. ■

Corollary 1 ITS terminates and returns an optimal solution.

Proof. By Theorem 2, every path generated by ITS is in $W(h^*(s))$, which is finite. Since ITS does not generate any node more than once in an iteration (by Theorem 1 property 5), and there can be only finitely many iterations (since path cost is additive, minimum arc cost $\delta > 0$ and $h^*(s)$ is finite); ITS is bound to terminate after a finite number of node generations.

When ITS terminates, it cannot terminate by expanding any branch (m, n) having $B(m, n) > h^*(s)$, because this would contradict Property 1 of Theorem 1. ■

5 Comparison of ITS with MA*

Although ITS is in some respects similar to MA*, there are also several significant differences. We briefly describe some of them below.

1. MA* does top-down propagation of h-values. In particular, if MA* generates a node m as a child of some other node n, then for each child q of m MA* computes heuristic value with respect to the heuristic value at node n (see [2], pp. 199). Thus, the f-values of generated nodes are strictly nondecreasing, regardless of whether the heuristic function h is monotone. In contrast, ITS does not do downward propagation of h-values.

2. When MA* expands a node n, it installs a most promising child m, and computes not only the h-value of m, but also the h-value of each child q of m (using the downward propagation technique stated above). If we assume that the branching factor is uniform, then in a b-ary tree MA* will make b^2 heuristic computations for each full expansion of a node. In comparison, ITS makes only b heuristic computations for full expansion of a node.

   When ITS expands a node n, it generates "placeholders" for all of n's children $n_1, \ldots, n_k$, but only evaluates one of these children, say, n'. Furthermore, ITS's evaluation of n' is much simpler than MA*'s evaluation of n', because ITS does not look at the children of n' in order to evaluate n'.

3. Every time MA* generates a node p such that $f(p)$ exceeds the current threshold z, MA* recursively propagates the f-values upwards in a manner similar to the upward propagation done in the AO* algorithm [11].

   In contrast, ITS does not do upward propagation of this kind. The closest that it comes to anything similar to this is a one-level back-up of B-values when it retracts a node.

6 Comparison of ITS with IDA*

6.1 Theoretical Results

In this section we compare ITS with IDA*. In particular, we show two things:

1. ITS never generates a node more times than IDA*. As a consequence, ITS generates every node generated by IDA*, and that for every node n, ITS generates n no more times than IDA* does.

2. There are classes of trees on which ITS will have better asymptotic time complexity than IDA*, even when given no more memory than IDA* (i.e., $S = 0$). In particular, we show that

there are trees on which ITS does only $O(N)$ node expansions. The main reason for this is that when ITS retracts nodes, it backs up values, which allows it to avoid re-generating many nodes.

Lemma 1 During each iteration of ITS, every path generated is a subpath of some path in $W(z)$, every node generated is in $V(z)$, and the number of node generations is $\leq |V(z)|$, where z is the current threshold value.

Proof. Suppose the claim is false. Then there is an iteration in which ITS generates a path $P = (n_1, n_2, \ldots, n_k)$ that is not a subpath of some path in $W(z)$. The only way this can happen is if $\text{cost}(P') + h(n_{k-1}) > z(j)$, where $P' = (n_1, n_2, \ldots, n_{k-1})$. Thus, from Property 3 of Theorem 1, $B(n_{k-1}, n_k) \geq \text{cost}(P') + h(n_{k-1}) > z(j)$. But from Step 2(b)i of ITS, we know that every branch (m, n) selected by ITS during this iteration has the property that $B(m, n) \leq z(j)$, which is a contradiction. Thus, every path generated during this iteration is a subpath of a path in $W(z)$, so every node generated by ITS during this iteration is in $V(z)$. From Property 5 of Theorem 1, each node is generated at most once during this iteration, so the number of nodes generated during this iteration is $\leq |V(z)|$. ∎

Lemma 2 Let z be ITS's threshold during its j'th iteration. Then

$$\{\text{all nodes generated during iterations } 1, 2, \ldots, j\} \subseteq V(z),$$

with equality if j is not ITS's final iteration.

Proof. For $i = 1, \ldots, j$, Let z_i be the threshold value during ITS's i'th iteration. From Theorem 1, $z_1 < z_2 < \ldots < z_j$, and thus

$$V(z_1) \subseteq V(z_2) \subseteq \ldots \subseteq V(z_j).$$

Thus from Lemma 1,

$$\{\text{all nodes generated during iterations } 1, 2, \ldots, j\} = \bigcup_{i=1}^{j} V(z_i) \subseteq V(z_j) = V(z).$$

To prove equality if j is not the final iteration, the proof is by induction on j. For the base case (i.e., $j = 1$), the proof is immediate. For the induction step, suppose the theorem holds for the iterations $1, 2, \ldots, j-1$, and consider iteration j. If ITS has not yet begun to retract nodes, the proof is immediate. Whenever ITS retracts a node q, there are two cases:

1. $\min_r B(q, r) \leq z_j$. In this case, during iteration j, ITS regenerates q and continues to generate nodes below q until it has generated all successors of q in any of the paths in $W(z_j)$.

2. $\min_r B(q, r) > z_j$. In this case, all successors of q in any of the paths in $W(z_j)$ are also successors of q in $W(z_{j-1})$. Thus by the induction hypothesis, they were generated by ITS during some iteration prior to j.

■

Theorem 3 For every j, $z(j)$ is the threshold used by ITS during its j'th iteration.

Proof. The proof is by induction.

Base case: $j = 1$. Then the threshold is $h(s)$, which is the same as $z(1)$.

Induction step: Let $j > 1$, and suppose the theorem holds for every iteration $\leq j$. We need to prove that it holds for iteration $j + 1$.

First, we prove that at the end of iteration j, for every tip branch (m, n), $B(m, n) = \min_P\{f(\text{last}(P)) : P \in W(z(j)) \text{ and } P \text{ contains } m\}$. To prove this, there are two cases:

1. n has never been generated. Then there is only one path $P \in W(z(j))$ that contains m, namely the path from s to m; and in Step 4 of Install, $B(m, n)$ was set to $g(m) + h(m)$.

2. n has been generated and later retracted. From Step 2(b)i of ITS, it follows that once a branch (m, n) is selected, ITS will keep selecting branches below n until every tip branch (q, r) below n has $B(q, r) > z(j)$. At this point, ITS may retract q, but if it does this, it will set $B(p, q) = \min_r B(q, r) = g(q) + h(q)$, so the theorem holds for p. If ITS later retracts p or any of its ancestors, it follows from an inductive argument that the theorem holds for them as well.

Thus at the end of iteration j, ITS sets the threshold for its next iteration to $\min\{f(\text{last}(P)) : P \in W(z(j))\}$. But this is precisely $z(j + 1)$. Thus during iteration $j + 1$, the threshold is $z(j + 1)$.

■

Corollary 2 IDA* and ITS do the same number of iterations, and for every j,

$$\begin{aligned} &\{\text{all nodes generated during IDA*'s } j\text{'th iteration}\} \\ &\quad = \{\text{all nodes generated during ITS's iterations } 1, 2, \ldots, j\}. \end{aligned}$$

Proof. For every iteration of IDA* except the final iteration I, this is an immediate consequence of Lemma 2. Let $P_1, P_2, \ldots, P_k$ be those paths in $W(z(I))$ generated by IDA* during its final iteration, with P_k being the solution path found by IDA*. IDA* generates these paths in left-to-right order. Within each iteration, ITS uses this same node selection strategy as IDA*, and therefore must generate these same paths in left-to-right order, except for those paths generated during previous iterations. Thus during this iteration, ITS will return the same goal path returned by IDA*. ■

Theorem 4 Let G be any state space, and n be any node of G. If IDA* and ITS expands nodes from G in left-to-right order following the same sequence of operators, then

1. ITS and IDA* generate exactly the same set of nodes;

2. for every node n, ITS generates n no more times than IDA* does.

Proof.

1. Suppose IDA* generates some node n. Then there is some iteration j of IDA* such that $n \in V(z(j))$. Thus from Corollary 2 , ITS generates n too.

   Suppose ITS generates some node n. Let j be the first iteration in which ITS generates n. Then $n \in V(z(j))$, so from Corollary 2, IDA* also generates n in its j'th iteration.

2. IDA* has $I-j$ subsequent iterations, and it generates n exactly once in each of these iterations. From Corollary 2, ITS als o has $I-j$ subsequent iterations, and from Property 5 of Theorem 1, ITS generates n at most once in each of these iterations. Thus ITS generates n no more times than IDA* does.

■

The above theorem shows that ITS's time complexity is never any worse than IDA*'s. Below, we show that there are classes of trees on which ITS does only $O(N)$ node expansions compared to IDA*'s $O(N \log N)$ node expansions on the same trees. The same result also holds for node generations. In the tree in Example 2, it is simpler to count the number of node expansions, and therefore we present the result in terms of node expansions.

Example 2 In the search tree G shown in Figure 5(a), each non-leaf node has a node-branching factor $b = 2$, and each arc has unit cost. G consists of two subtrees G_1 and G_2 where each one is a full binary tree of height k. G_2 is rooted at the right most node of G_1. Every leaf node, except the one labeled as goal, is a non-terminal. For each node n in G, $h(n) = 0$.

Clearly G_1 and G_2 each contain $N' = \lceil N/2 \rceil$ nodes, where N is the number of nodes eligible for expansion by A*. The cost of the solution path is $2k = 2[\log_2(N' + 1) - 1]$. Let $N_0 =$

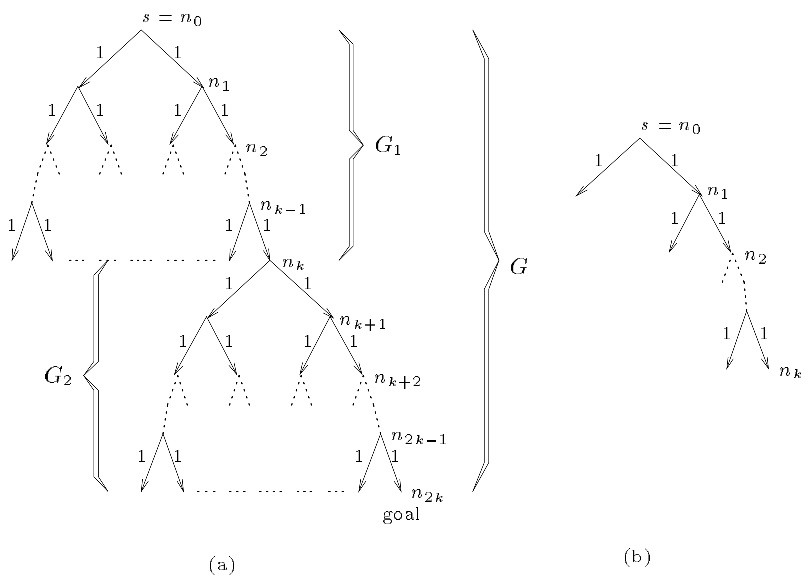

Figure 5: A tree G on which IDA* is $O(N \log N)$ and ITS is $O(N)$

$b^k + 2b^{k-1} + 3b^{k-2} + \ldots + kb$. Then the total number of node expansions by IDA* in the worst-case is

$$N_0 + kN' + N_0 \geq kN' + N' = k(N' + 1) = O(N \log N).$$

Now we count the total number of node expansions by ITS on G. As in the case of IDA* no node of G_2 will be expanded prior to the expansion of all the nodes of G_1 at least once. Using the theorem 4, we can infer that the total number of node expansions by ITS on G_1 is $O(N)$. Once ITS begins expanding nodes of G_2, the portion of G_1 that will be retained in memory is shown in Figure 5(b). The branches of G_1 which do not lead a goal node (all left branches) will have B value of ∞. Therefore no node of G_1 will be reexpanded while expanding nodes of G_2. Since G_1 and G_2 are symmetric, by the same argument as in case of G_1, ITS will not make more than $O(N)$ node expansions on G_2. Thus the worst-case time complexity of ITS on trees like G will always be $O(N)$.

6.2 Experimental Results

In the examples above, we have shown that there are classes of trees on which ITS's asymptotic complexity is better than IDA*'s. In this section we report results of our experiments on three problem domains namely 3-machine flow-shop scheduling, traveling salesman and 15-puzzle. These problems were selected mainly to encompass a wide range of node generation times. While the node generation time for the 15-puzzle is very small, it is significant for the traveling salesman problem. The node generation time for 3-machine flow-shop scheduling problem is also small but higher than that of 15-puzzle. All the programs were run on a SUN sparcstation. We describe our results in the following subsections.

One purpose of our experiments was to compare ITS with IDA*, and another purpose was to see how giving ITS additional memory would improve its performance in terms of both node generation and running time. For the latter purpose, we ran ITS with varying amounts of memory. The definition of ITS includes a parameter S which gives the total amount of memory available to ITS for storing nodes. If $S = 0$, then ITS retracts all nodes except those on the current path. For each problem instance p, let ITS(v) be ITS with $S = vM$, where M is the number of distinct nodes generated by ITS on p. Thus, $v = S/M$ is what fraction ITS gets of the amount of memory it would need in order to avoid doing any retractions.[3] For example, ITS(1) is ITS with enough memory that it doesn't need to retract any nodes, and ITS(1/4) is ITS running with 1/4 of the amount of memory as ITS(1).

[3]If we had expressed S as an absolute number rather than a fraction of M, this would not have given useful results, because the number of distinct nodes generated by ITS on each problem instance varies widely. For example, with 100,000 nodes, on some problem instances ITS would have exhausted the available memory very quickly, and on others, it would not even have used all of the available memory.

Table 1: IDA* and ITS′(0) on the 10-job 3-machine Flow-Shop Scheduling Problem.

algorithm	node generations	time (sec)
IDA*	211308.76	3.93
ITS′(0)	210842.96	4.43

Table 2: ITS(v) on the 10-job 3-machine Flow-Shop Scheduling Problem.

v	node generations	time (sec)
0	210842.96	23.22
1/4	123764.71	13.64
1/2	61690.79	6.92
3/4	28174.31	3.32
1	17663.28	1.80

6.2.1 Flow-Shop Scheduling Problem

The flow-shop scheduling problem is to schedule a given set of jobs on a set of machines such that the time to finish all of the jobs is minimized. In our experiments, we selected the number of machines to be 3. We used a search-space representation and admissible node evaluation function of Ignall and Schrage [7].

For ITS(0), there is a special case to consider. In the flow-shop scheduling problem, it is very easy to generate the successor n' of a node n. Thus, since IDA* and ITS(0) will need to keep track of only one successor of n at a time, both IDA* and ITS(0) can generate n' by modifying the record for n (and undoing this modification later when retracting n'), rather than generating an entirely new record (this same technique was used by Korf in his implementation of IDA* on the 15-puzzle). For the 3-machine flow-shop scheduling problem, we used this technique to improve the efficiency of both IDA* and ITS(0). To distinguish between the normal version of ITS(0) and the improved version, we call the latter ITS′(0).

We ran IDA* and ITS′(0) on one hundred problem instances with ten jobs in the jobset. The processing times of the jobs on the three machines were generated randomly from the range [0,100] using a uniform distribution. Table 1 presents the average node generation and running time figures for IDA* and ITS$'$(0) on these problem instances. As can be seen, ITS$'$(0) generated fewer nodes than IDA*. However, ITS$'$(0) took slightly more time than IDA*. This is primarily because the node generation time for this problem is small, and therefore the smaller number of nodes generated by ITS$'$(0) did not compensate for its slightly higher overhead than IDA* in node selection and retraction.

We also ran ITS(v) on the same problem instances, with various values of v. The average node generation and running-time figures for ITS(v) are given in Table 2. The table shows that as the amount of available memory increases, ITS improves its performance in terms of both node generations and running time.

6.2.2 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is as follows: given a set of K cities with nonnegative cost between each pair of cities, find the cheapest tour. A tour is a path that starting at some initial city visits every city once and only once, and returns to the initial city. We chose the well known method of Little et al. [10] to represent the search space and the lower bound heuristic for the Traveling Salesman Problem. The search space in this formulation is a binary tree.

The technique that we used to improve the efficiency of IDA* and ITS(0) in the 3-machine flow-shop scheduling problem cannot be used in the Traveling Salesman Problem, because in this problem it is much more difficult to generate the successors of a node. Thus, on those problems we simply compared IDA* against ITS(v) for various values of v, without making any special-case modifications.

We generated two sets of data TSP Set 1 and TSP Set 2 and ran both IDA* and ITS on each set. For both sets we selected the number of cities equal to 5, 10, 15, 20, 25, and 30. For each value of the number of cities, one hundred cost matrices were generated. For TSP Set 1 the cost values

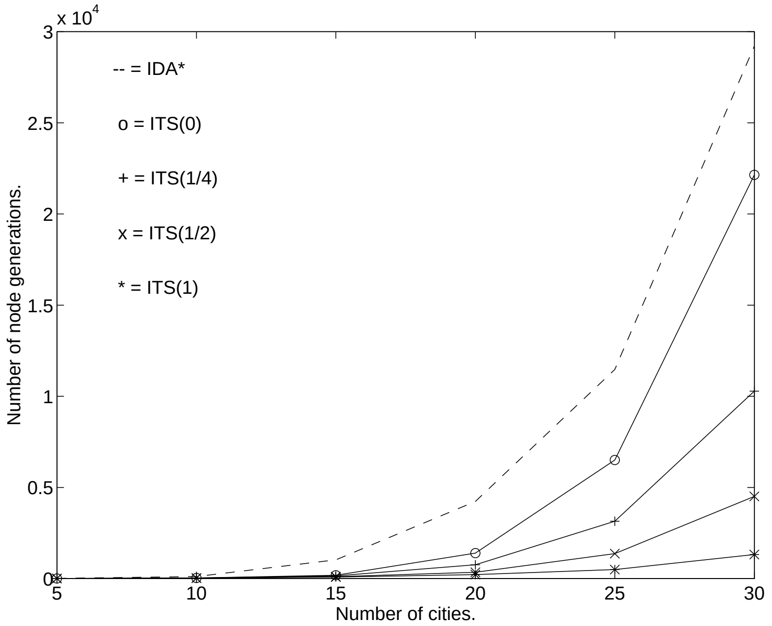

Figure 6: Number of node generations versus number of cities for IDA* and ITS on TSP Set 1.

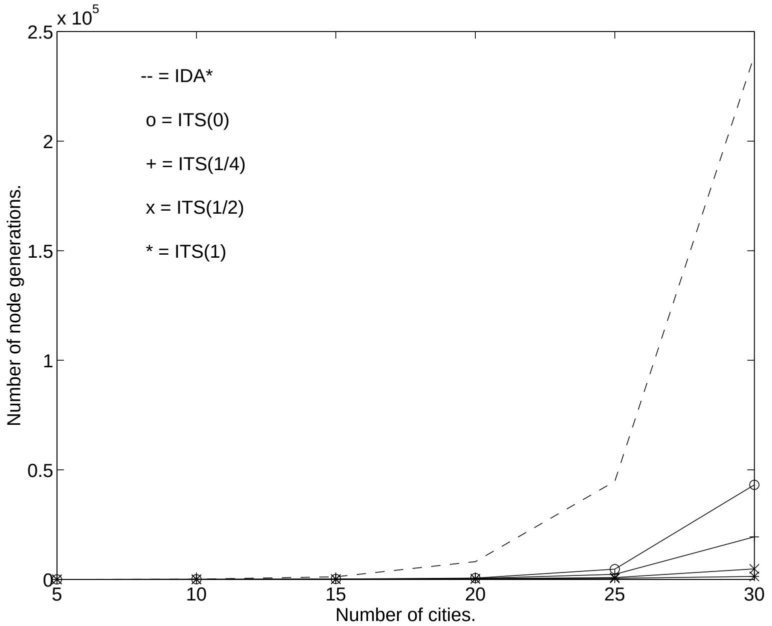

Figure 7: Number of node generations versus number of cities for IDA* and ITS on TSP Set 2.

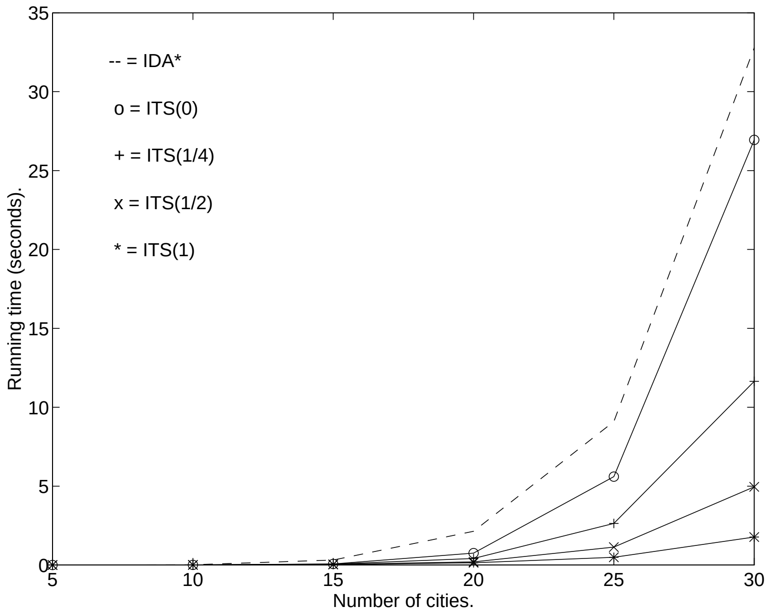

Figure 8: Running time versus number of cities for IDA* and ITS on TSP Set 1.

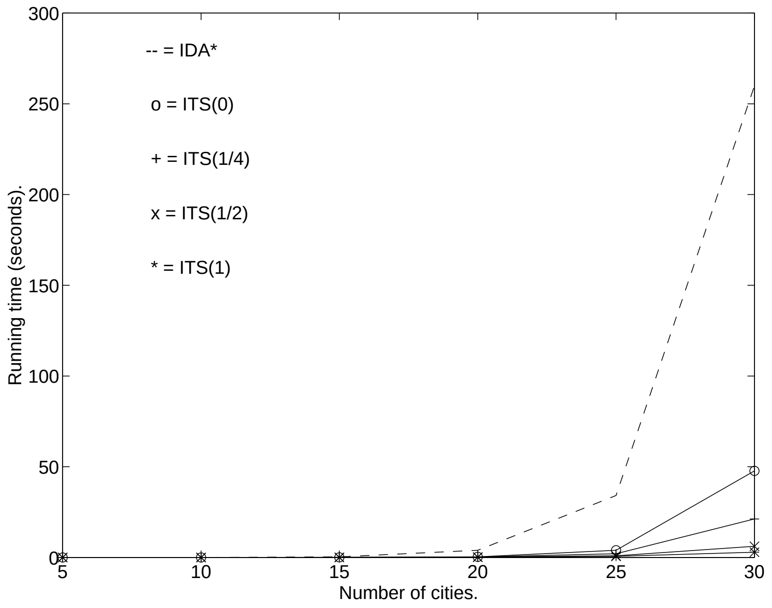

Figure 9: Running time versus number of cities for IDA* and ITS on TSP Set 2.

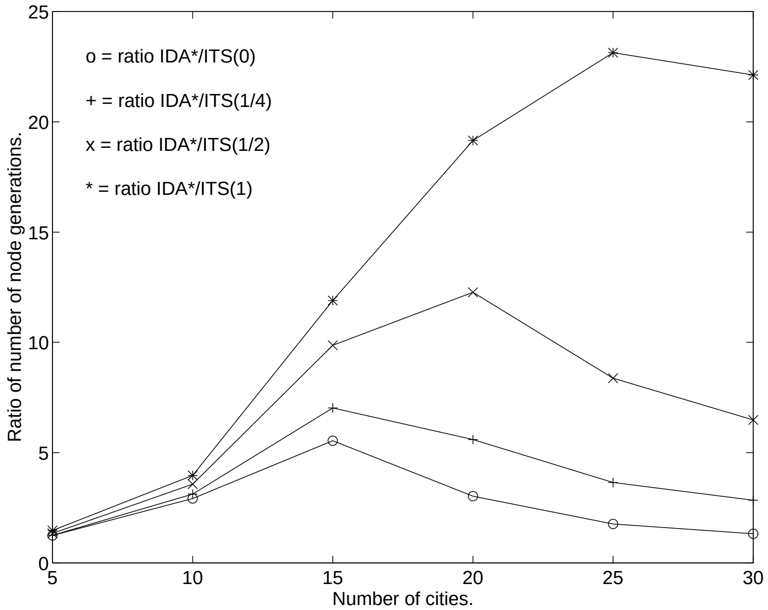

Figure 10: Ratio of IDA* node generations to ITS node generations, versus number of cities on TSP Set 1.

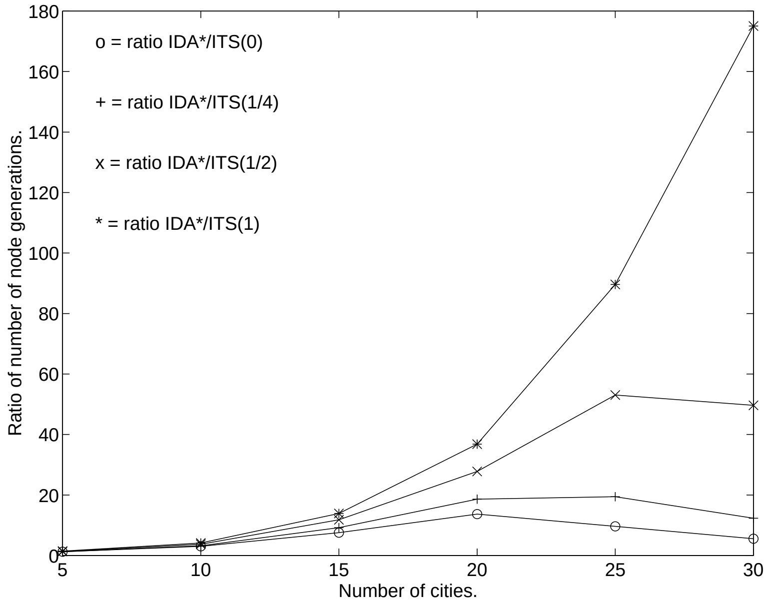

Figure 11: Ratio of IDA* node generations to ITS node generations, versus number of cities on TSP Set 2.

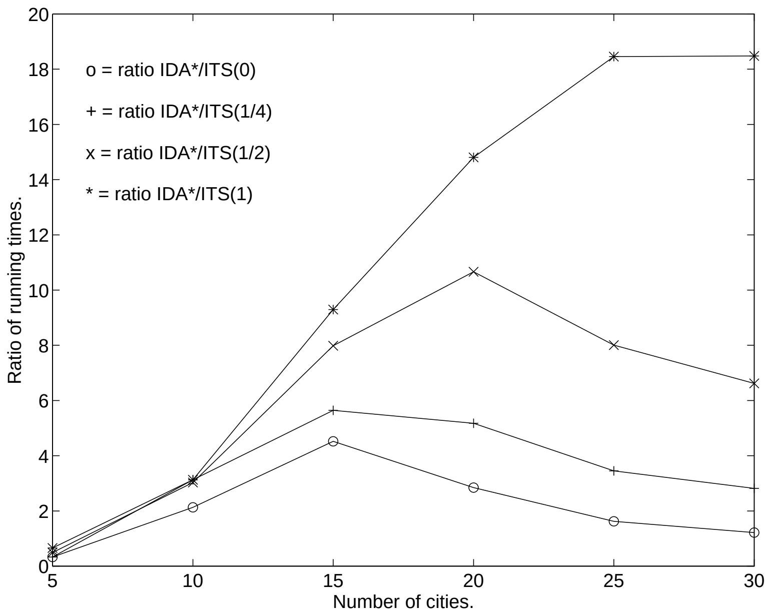

Figure 12: Ratio of IDA* running time to ITS running time, versus number of cities on TSP Set 1.

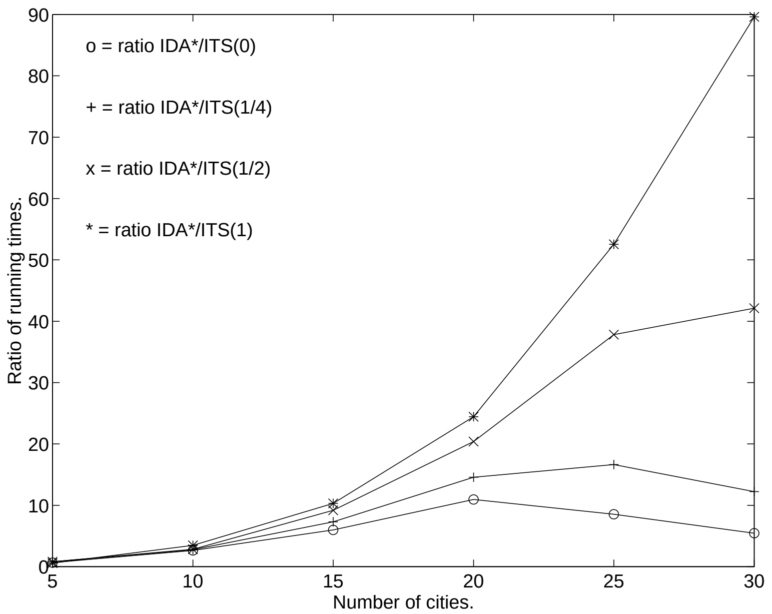

Figure 13: Ratio of IDA* running time to ITS running time, versus number of cities on TSP Set 2.

$c(i,j)$ were taken at random from the interval [0,100] using a uniform distribution except for $i = j$, in which case $c(i,j)$ was set to ∞. For TSP Set 2 the cost values $c(i,j)$ were taken at random from the interval $[0,(10 \times \text{number of cities}^2)]$ using a uniform distribution except for $i = j$, in which case $c(i,j)$ was set to ∞. In general the cost matrices of TSP Set 1 and TSP Set 2 were not symmetric and did not satisfy the triangle inequality.

The results of our experiments are summarized in Figures 6 through 13, which graph the performance of IDA*, ITS(0), ITS(1/4), ITS(1/2), and ITS(1) for TSP Set 1 and TSP Set 2. Each data point in Figures 6 through 13 is the average over the one hundred problem instances.

From Figures 6 through 9, it can be seen that on this problem, ITS(0) makes fewer node generations and runs faster than IDA*. This is because the node generation time is large enough that the extra overhead of ITS over IDA* becomes relatively insignificant, and therefore the reduction in number of node generations does reduce the running time. Furthermore, the additional memory used by ITS significantly reduces the number of node generations as well as the running time. By comparing Figure 6 versus 7 or Figure 8 versus 9, it can be seen that the savings provided by ITS for any given amount of memory is more for TSP Set 2 than TSP Set 1. This is because the heuristic branching factor is lower in TSP Set 2 than in TSP Set 1.

In order to study how IDA*'s average-case asymptotic behavior compares to ITS's, in Figures 10 through 13 we have plotted ratios of node generations and running time of IDA* and ITS for TSP Set 1 and TSP Set 2. The interesting point to be noted about these graphs is the difference between TSP Set 1 and TSP Set 2. For TSP Set 1, in each case, the ratio first goes up and then goes down. If ITS's asymptotic performance were strictly better than IDA*'s, we would have expected the ratios to keep going up. Since Theorem 4 shows that ITS's asymptotic performance is at least as good as IDA*'s, this suggests that both algorithms have the same asymptotic performance on this problem on TSP Set 1. Since this behavior also occurs for ITS(1), which is essentially a version of A*, our results suggest that both ITS and IDA* are asymptotically optimal on the Traveling Salesman Problem when the costs between cities are generated uniformly from a fixed range.

For TSP Set 2, the ratios eventually go down for each case except for IDA*/ITS(1). This suggests that IDA* is not asymptotically optimal in this case. For the reasons explained above, IDA*, ITS(0), ITS(1/4), and ITS(1/2) have the same asymptotic performance on TSP Set 2. This suggests an interesting result that ITS with a fraction (less than 1) of memory is not asymptotically optimal when the range of costs varies in proportion to the number of cities.

6.2.3 15-Puzzle

For the 15-puzzle, we used the manhattan distance heuristic in our experiments.

For this problem, we made the same efficiency-improving modification to IDA* that we made in the 3-machine flow-shop scheduling problem. We considered making the same modification to ITS(0), but decided not to run ITS(0) at all on this problem, for the following reason. In the 15-puzzle, with the manhattan distance heuristic, the threshold in every iteration of IDA* and ITS increases by exactly two. Also, if z is the threshold during the current iteration, every tip branch (p, q) whose B value exceeds z has $B(p, q) = z + 2$. This makes it useless to back-up B values during retraction, because every node that is retracted in iteration i must be regenerated in iteration $i + 1$. Thus, in order to improve the efficiency of ITS(0) on this this problem, we should not only simplify the node-generation scheme as described in Section 6.2.1, but should also remove the back-up step. But these modifications make ITS(0) essentially identical to IDA*.

The same reasoning suggests that on the 15-puzzle, even if $S \neq 0$, ITS will not reduce the number of node generations very much in comparison with IDA*. If IDA* makes I iterations on a problem, then ITS with S amount of memory will save at most $S * I$ number of node generations. Since I is usually small for 15-puzzle (between 5 and 10), the actual savings is expected to be relatively small. Thus, since ITS has higher overhead than IDA*, we would expect ITS to take more time than IDA* on this problem.

To confirm these hypotheses, we ran ITS and IDA* with three values of $S = 100,000$, 300,000, and 600,000 on the twenty problem instances on which Chakrabarti *et al.* ran MA*(0). We could not run ITS(v) on these problem instances because the number of distinct nodes is so large on some of the problem instances that they exceed the available memory. Therefore, we had to run ITS with fixed amounts of S. The results are summarized in Figures 14 and 15. As expected, ITS did not achieve a significant reduction in the number of node generations, and took significantly more time than IDA*.[4] Thus, for the 15-puzzle, IDA* is the preferable algorithm.

6.2.4 Discussion of Experimental Results

The experimental results indicate that to what extent this savings of node generations is worthwhile depends on two problem characteristics: the amount of time required to generate each node, and the heuristic branching factor (i.e., the average ratio of new nodes generated during consecutive thresholds). The tradeoffs are as follows:

- When the heuristic branching factor is low, ITS will generate many fewer nodes than IDA*, in some cases achieving better asymptotic performance than IDA*. If in addition the node generation time is high, then this reduction in the number of node generations can enable ITS to take much less time than IDA*. This is why ITS outperformed IDA* in the Traveling Salesman Problem.

- If the heuristic branching factor is high, then ITS's reduction in node generations won't be as great. If in addition the node generation time is low, then ITS will be hurt by its higher

[4] Oddly, Figure 15 shows a relative improvement for ITS at the two largest problem sizes. However, we suspect that these data are spurious, because at these two problem sizes, we exceeded the maximum integer size of some of our counters and also encountered thrashing.

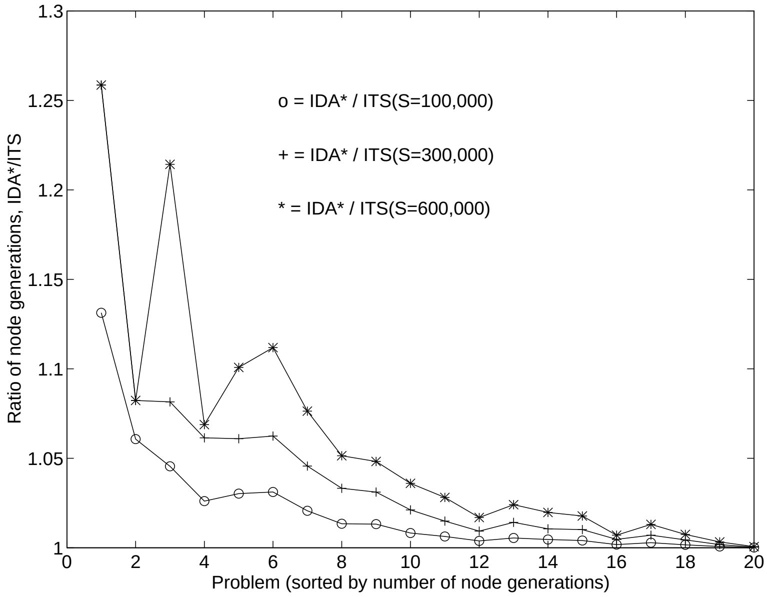

Figure 14: Ratio of IDA* node generations to ITS node generations, on twenty 15-puzzle problem instances. On the x axis, the problem instances are sorted by the number of node generations they required.

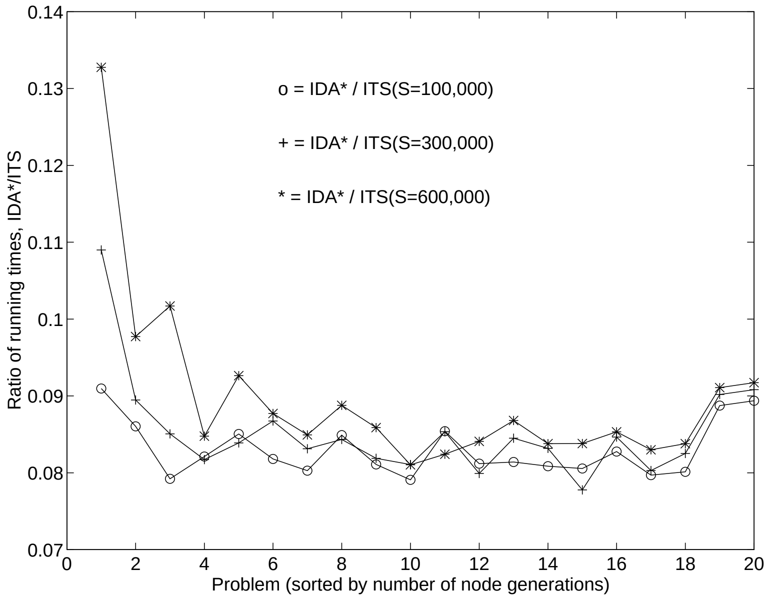

Figure 15: Ratio of IDA* running time to ITS running time, on twenty 15-puzzle problem instances. On the x axis, the problem instances are sorted by the number of node generations they required.

node-generation overhead. This is why IDA* outperformed ITS in the 15-puzzle.

Our experimental results also suggest that on the Traveling Salesman Problem, both IDA* and ITS are asymptotically optimal on the average if the costs between the cities are taken from a fixed range. However, if the range of costs grows in proportion to the problem size then IDA* is not asymptotically optimal. ITS's asymptotic complexity in the latter case depends on the amount of memory available to it.

Conclusion

We have presented a new algorithm called ITS for tree search in limited memory. Like MA*, ITS makes better use of the available memory than IDA* does. However, MA* did this at the expense of having a very high node-generation overhead. ITS's node-generation overhead is somewhat higher than IDA*'s, but is much lower than MA*'s.

Our theoretical analysis shows that ITS never does more node generations than IDA*, and that in some cases it generates asymptotically fewer nodes than IDA*.

In our experimental studies, ITS always generated fewer nodes than IDA*. However, these studies also show that to what extent this savings of node generations is worthwhile depends on the amount of time required to generate each node, and the heuristic branching factor. In problems for which the node generation time is high and the heuristic branching factor is low, ITS can provide significant savings in both number of node generations and running time.

References

[1] A. Bagchi and A. Mahanti. Search algorithms under different kinds of heuristics-a comparative study. *JACM*, 30(1):1–21, 1983.

[2] P. P. Chakrabarti, S. Ghosh, A. Acharya, and S. C. De Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 47:197–221, 1989.

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

[4] M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: A memory-limited heuristic search procedure for the connection machine. In *Frontiers'90: Frontiers of Massively Parallel Computation*, 1990.

[5] S. Ghosh, A. Mahanti, and D. S. Nau. ITS: An efficient limited-memory heuristic tree search algorithm. In *AAAI 1994*, pages 1353–1358, 1994.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Sciences and Cybernetics*, pages 1556–1562, 1968.

[7] E. Ignall and L. Schrage. Applications of the branch and bound technique to some flow-shop scheduling problems. *Operations Research*, 13(3):400–412, 1965.

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

[9] R. E. Korf. Optimal path finding algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 200–222. Springer Verlag, 1988.

[10] J. D. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[11] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, 1980.

[12] V. N. Rao, V. Kumar, and R. E. Korf. Depth-first vs. best-first search. In *AAAI-1991*, pages 434–440, Anaheim, California, 1991.

[13] S. Russell. Efficient memory-bounded search methods. In *ECAI-1992*, Vienna, Austria, 1992.

[14] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. De Sarkar. Reducing reexpansions in iterative deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50(2):207–221, 1991.

[15] A. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI-89*, pages 274–277, 1989.

[16] B. W. Wah. MIDA*, an IDA* search with dynamic control. Technical Report UILU-ENG-91-2216, University of Illinois at Urbana, Champaign-Urbana, IL, 1991.