ITS: An Efficient Limited-Memory Heuristic Tree Search Algorithm[*]

Subrata Ghosh
Department of Computer Science
University of Maryland
College Park,MD 20742
subrata@cs.umd.edu

Ambuj Mahanti
IIM, Calcutta
Calcutta 700 027
India
iimcal!am@veccal.ernet.in

Dana S. Nau
Dept. of Computer Science, and
Institute for Systems Research
University of Maryland
College Park, MD 20742
nau@cs.umd.edu

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* [1], and a generalized version of MREC [12]. 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 node-generation time is high (as in most practical problems), ITS can provide significant savings in both number of node generations and running time. Our experimental results also suggest that in the average case both IDA* and ITS are asymptotically optimal on the traveling salesman problem.

Introduction

Although A* is usually very efficient in terms of number of node expansions [2], it requires an exponential amount of memory, and thus runs out of memory even on problem instances of moderate size. This problem led to Korf's development of IDA* [6]. 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. 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 (for proofs, see [4]) 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 present example trees in which ITS expands $O(N)$ nodes in comparison to $O(N \log N)$ nodes expanded by IDA* where N is the number of nodes eligible for expansion 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 node-generation time is high (which is the case for most practical problems), ITS can provide significant savings in both number of node generations and running time.

4. Our experiments suggest that in the average case both IDA* and ITS are asymptotically optimal on the traveling salesman problem. Although Patrick *et al.* [8] showed that there exists a class of traveling salesman problems in which IDA* is not asymptotically optimal, our results suggest that such problems are not common enough to affect IDA*'s average performance over a large number of problem instances.

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 edge (m, n) in G is denoted by $c(m, n)$.

Algorithm ITS

Most heuristic search algorithms maintain a search tree T containing data about each node n that has been

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

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, in the search tree T maintained by ITS, ITS maintains heuristic information not only for each node of the 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 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.

Procedure ITS:

```
1. Call Install($s, 0$).
2. Do the following steps repeatedly:
  (a) Set $z := \min\{B(p, q) : (p, q)$ 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 $B(m, n) := \infty$. 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 $B(p, q) := \min_r B(q, r)$, 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 no operators are applicable to $n$, then put a dummy branch $(n,$ DUMMY$)$ in $T$. Else for each operator $R$ applicable to $n$, put a branch $(n, R(n))$ in $T$.
3. Set $g(n) := g$.
4. For each branch $(n, r)$, set $B(n, r) := g(n) + h(n)$.
```

Basic Properties of ITS

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.

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 iff $B(m, n) \leq z$ 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.

Theorem 2 ITS terminates and returns an optimal solution.

Comparison of ITS with IDA*

Theoretical Results

In this section we show the following:

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., $S = 0$). The main reason for this is that when ITS retracts nodes, it backs up path information, which allows it to avoid re-generating many subtrees.

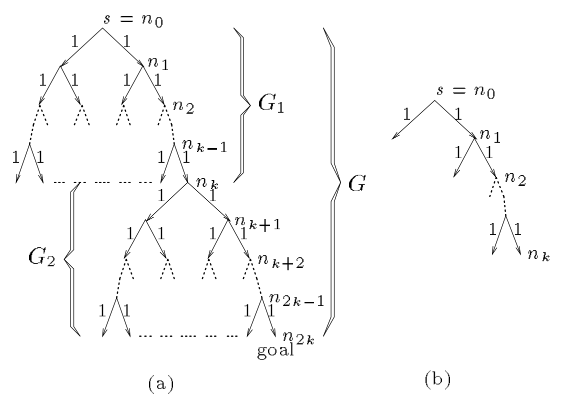

Figure 1: A tree G on which IDA* is $O(N \log N)$ and ITS is $O(N)$

Theorem 3 IDA* and ITS do the same number of iterations, and for every j,

$$\{\text{nodes generated in IDA*'s } j\text{'th iteration}\} = \{\text{nodes generated in ITS's iterations } 1, 2, \ldots, j\}.$$

Theorem 4 Let G be any state space, and n be any node of G. If IDA* and ITS expand 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.

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 1, it is simpler to count the number of node expansions, and therefore we present the result in terms of node expansions.

Example 1. In the search tree G shown in Figure 1(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 $N_0 = b^k + 2b^{k-1} + 3b^{k-2} + \ldots + kb$. 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 1(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 $O(N)$.

Experimental Results

In the example 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 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 flow-shop scheduling problem is also small but higher than that of 15-puzzle. All the programs were written in C and run on a SUN sparcstation. We describe the problems and our results in the following sections.

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.[1] 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).

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

[1]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 the whole 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

number of machines to be 3. We used a search-space representation and admissible node evaluation function of Ignall and Schrage [5].

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 For the 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 100 problem instances with 10 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.

Traveling Salesman Problem The traveling salesman problem 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. [7] to represent the search space and the lower bound heuristic for the traveling salesman problem.

The technique that we used to improve the efficiency of IDA* and ITS(0) in the 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.

We ran our experiments with the number of cities K equal to 5, 10, 15, 20, 25, 30, 35 and 40. For each value of K, one hundred cost matrices were generated, taking the cost values $c(i, j)$ at random from the interval [0,100] using a uniform distribution (except when $i = j$, in which case $c(i, j) = \infty$). Thus, in general the cost matrices were not symmetric and did not satisfy the triangle inequality.

The results of our experiments are summarized in Figures 2 through 5, which graph the performance of IDA*, ITS(0), ITS(1/4), ITS(1/2), and ITS(1). From figures 2 and 3, it can be seen that on this problem, ITS(0) makes fewer node generations and runs slightly 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.

In order to study how IDA*'s average-case asymptotic behavior compares to ITS's, in figures 4 and 5 we have plotted ratios of node generations and running time of IDA* and ITS. The interesting point to be noted about these graphs is that 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, that both algorithms have the same asymptotic performance on this problem. Since this behavior also occurs for ITS(1), which is essentially a version of A*, this suggests that both ITS and IDA* are asymptotically optimal on the traveling salesman problem (at least in the case when the costs between cities are generated uniformly from a fixed range).

15-Puzzle The 15-puzzle problem consists of a 4×4 frame containing fifteen numbered tiles and an empty position usually known as the "blank". The valid moves slide any tile adjacent to the blank horizontally or vertically to the adjacent blank position. The task is to find a sequence of valid moves which transform some random initial configuration to a desired goal configuration. The manhattan distance function was used as the heuristic in our experiments.

In the 15-puzzle, we made the same efficiency-improving modification to IDA* that we made in the 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

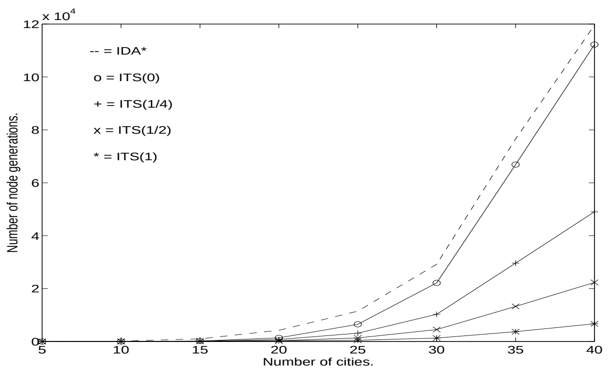

Figure 2: Nodes versus no. of cities.

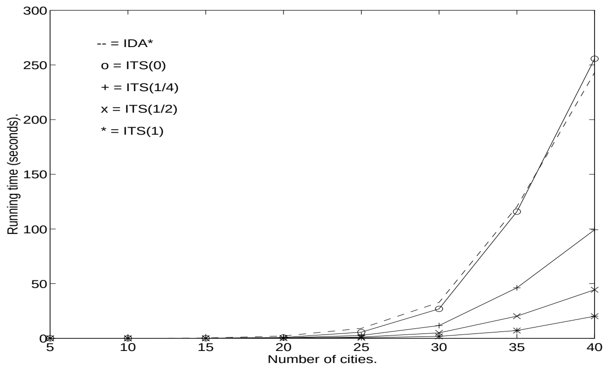

Figure 3: Time versus no. of cities.

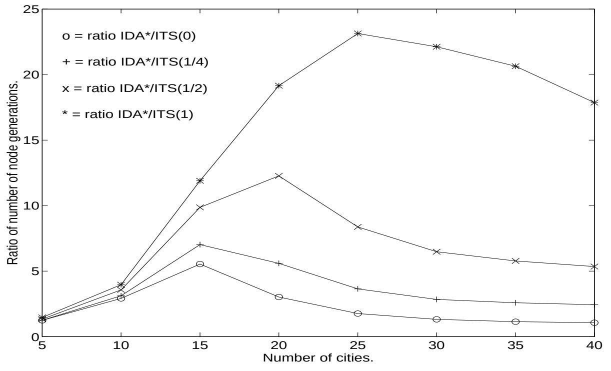

Figure 4: IDA* to ITS nodes, versus no. of cities.

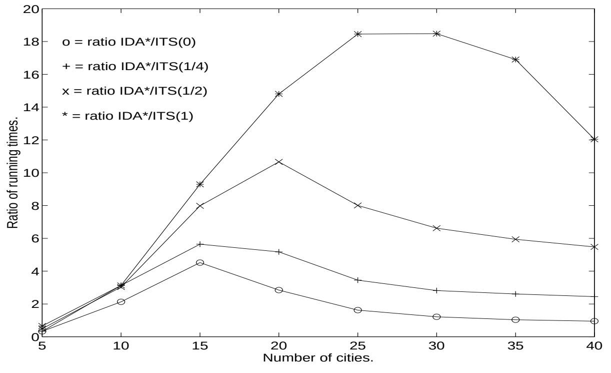

Figure 5: IDA* to ITS time, versus no. of cities.

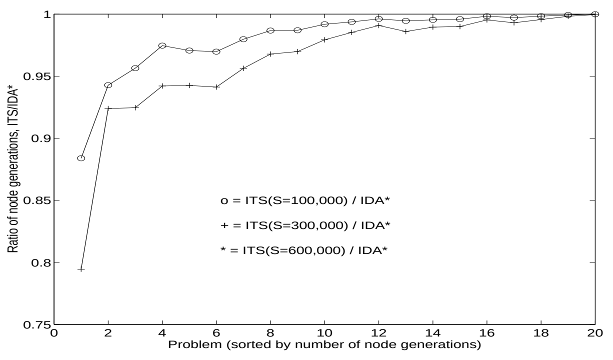

Figure 6: ITS to IDA* nodes on 20 problem instances.

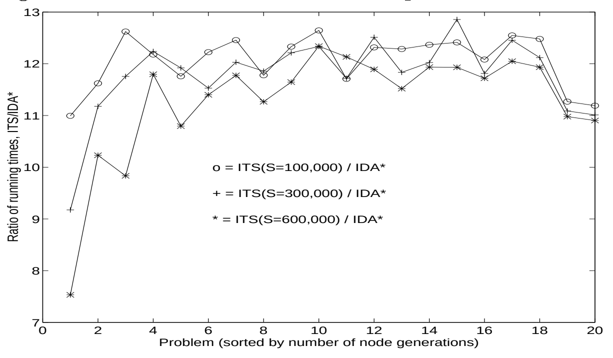

Figure 7: ITS to IDA* time on 20 problem instances.

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 the flow-shop scheduling problem, but should also remove the back-up step. But that makes 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 $S = 100,000$, 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 values for S. The results are summarized in Figures 6 and 7. As expected, ITS did not achieve a significant reduction in the number of node generations, and took significantly more time than IDA*.[2] Thus, for the 15-puzzle, IDA* is the preferable algorithm.

Related Work

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* [3], SMA* [10], 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 [11], DFS* [9], and MIDA* [13] 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, even if 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.

Finally, it should be noted that the techniques used in the two classes of algorithms can be combined.

Conclusion

We have presented a new algorithm called ITS for tree search in limited memory. Like IDA*, ITS has low node-generation overhead—and like MA*, it makes dynamic use of memory. Our theoretical analysis shows that, ITS never does more node generations than IDA* and there are trees where it generates fewer nodes than IDA*. Our experimental results indicate that with additional memory, ITS can significantly reduce the number of node generations and run faster on problems for which the node-generation time is sufficiently high.

[2] Oddly, Figure 7 shows a relative improvement for ITS at the two largest problem sizes. However, we suspect that these data are spurious, because on these two problem instances, we exceeded the maximum integer size of some of our counters and also encountered thrashing.

References

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

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

[3] 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.

[4] S. Ghosh. *Heuristic Search with Limited Resources.* PhD thesis, Department of Computer Science, U. of Maryland, 1994 (forthcoming).

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

[6] R. E. Korf. Depth first iterative deepening: An optimal admissible tree search. *Artif. Intel.*, 27:97–109, 1985.

[7] 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.

[8] B. G. Patrick, M. Almulla, and M. M. Newborn. An upper bound on the complexity of iterative-deepening-A*. In *Symp. on Artif. Intel. and Math.*, Fort Lauderdale, FL, 1989.

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

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

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

[12] 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.

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