Enhanced Iterative-Deepening Search*

Alexander Reinefeld
Paderborn Center for Parallel Computing
D-33095 Paderborn, Germany
<ar@Uni-Paderborn.DE>

T.A. Marsland
University of Alberta
Computing Science Dept.
Edmonton, Canada T6G 2H1
<tony@CS.UAlberta.CA>

October 20, 1993

Keywords

Heuristic search, A* algorithm, depth-first iterative-deepening, game trees, computer chess methods, Fifteen Puzzle, Traveling Salesman Problem.

Abstract

Iterative-deepening searches mimic a breadth-first node expansion with a series of depth-first searches that operate with successively extended search horizons. They have been proposed as a simple way to reduce the space complexity of best-first searches like A* from exponential to linear in the search depth.

But there is more to iterative-deepening than just a reduction of storage space. As we show, the search efficiency can be greatly improved by exploiting previously gained node information. The information management techniques considered here owe much to their counterparts from the domain of two-player games, namely the use of fast-execution memory functions to guide the search. Our methods not only save node expansions, but are also faster and easier to implement than previous proposals.

*2nd revision for IEEE-PAMI. For review only, please do not cite or copy.

1 Introduction

Of the brute-force searches, *depth-first iterative-deepening (DFID)* is the most practical, because it combines breadth-first optimality with the low space complexity of depth-first search. Its basic idea is as simple as conducting a series of independent depth-first (backtracking) searches, each with the look-ahead horizon extended by an additional tree level. With the iterative approach, DFID is guaranteed to find the shortest solution path, just as a breadth-first search would. But in contrast to the latter, DFID needs negligible memory space. Its space complexity grows only linearly with the search depth.

The origins of iterative-deepening search trace back to the late 1960s [24], when programmers sought a reliable mechanism to control the time consumption of the newly emerging tournament chess programs. Rather than blindly committing to one direct depth-d search of unpredictable duration, the total search task was subdivided into separate depth-first searches with successively deepened search horizons $1, 2, \ldots, n$. This allows the search process to halt with a best available answer as soon as some time limit is exceeded.

Even more important are the various *memory functions* that also build upon the iterative-deepening approach. They use node information from previous iterations to increase the cutoffs in the current iteration. Among the data that can be reused, move ordering and node scoring information is of special importance. Various memory functions have been invented to store this and other information: *refutation* or *killer tables* [1], *transposition tables* [30, 26] and *history tables* [23]. Taken together, the memory functions not only pay for themselves by yielding better frontier node evaluations, but also produce searches that are faster than a direct depth-d search [13].

In the mid 1980s, iterative-deepening was refined for *heuristic single-agent searches* like A* and AO*. Here, the successive iterations do not correspond to increased search depth, but to increased cost bounds of the currently investigated path. But again, iterative-deepening reduces the space complexity to linear while preserving optimality. As a consequence, Korf's *Iterative-Deepening A* (IDA*)* [8] can be applied in domains where excessive space requirements cause A* to fail. One such application is the 15-puzzle.

The better space efficiency is paid for by an increased number of node expansions. Because IDA* does not retain path information from one iteration to the next, the shallow tree parts are re-examined several times. Following the same lines as in multi-agent search, IDA* (like any iterative search) should be improved by using node information of previous iterations.

In this paper, we show how to adapt search enhancements, that have been found effective in the domain of two-player games to single-agent heuristic search. The techniques include *node pre-sorting*, the use of *principal variations*, *transposition* and *refutation tables* and other memory functions [13, 19]. With the best combination of these techniques optimal solution paths for the 15-puzzle can be found, while visiting less than half the nodes seen by pure IDA*. This is better than can be achieved with a perfectly informed (and hence non-deterministic) IDA* algorithm, one that performs an iterative depth-first search up to the penultimate iteration and finds a solution node right at the beginning of the last (goal) iteration.

In practice, speed of computation is more important than the number of node expansions. Since memory tables are accessed in unit time, the running time of the proposed algorithms is almost proportional to the node count. Maximal speedups are achieved in applications with time-consuming heuristic estimation functions. One such example is the traveling salesman problem. Here a 73% node reduction (as compared to IDA*) speeds up the total runtime by 72%, giving an almost linear improvement. This is a remarkable result, considering that unsuccessful table accesses must be compensated for by even greater savings elsewhere.

2 Applications

Heuristic single-agent search techniques can be found in applications where a decision tree/graph is built to determine the best of several alternatives by searching. Typical applications include perception problems, theorem proving, robot control, pattern recognition, expert systems and some combinatorial optimization problems of Operations Research. For our experiments we selected two problem domains that build large search graphs and are easy to implement: the *15-puzzle* and the *traveling salesman problem*.

2.1 The Fifteen-Puzzle

The 15-puzzle is simple, but has combinatorially large problem space of $16!/2 \approx 10^{13}$ states. It consists of fifteen square tiles $1, 2, \ldots, 15$, located in a square tray of size 4×4. One square, the *blank square*, is kept empty so that an orthogonally adjacent tile can slide into its position – thus leaving a blank square at its origin. The problem is to re-arrange some given initial configuration into a goal configuration without lifting one tile over another.

Although it would seem easy to find any solution to this problem, it is much harder to determine a mapping of the given initial configuration to the goal configuration with the fewest moves. Using IDA*, it takes some hundred millions of node generations to solve a random problem instance, when using the most popular heuristic estimate function, the *Manhattan* or *city-block distance*. This estimate is a sum of the minimum displacement of each tile from its goal position. As can be proved by induction, the Manhattan distance is *admissible*: It never overestimates the distance to the goal configuration. This is an important requirement if a heuristic search algorithm is to find an optimal (=shortest) path to a goal state.

2.2 The Traveling Salesman Problem

The *traveling salesman problem (TSP)* refers to the task of finding a shortest (or least cost) tour that returns to the starting point after visiting all cities in the n-city network only once. The TSP is known to be NP-hard, and exact solutions can only be obtained for tours involving some hundred cities.

While the well-known branch-and-bound algorithms of Held and Karp [5] or Little *et al.* [10] would be among the preferred solution techniques for the TSP in practice[1], we have chosen the method described in Pearl's book [16, p. 10ff], because it builds a graph rather than a tree. It does so by successively adding unvisited cities to the end of a temporary partial contiguous tour for as long as their cost estimates do not exceed the given bound. For our experiments, we randomly generated the coordinates of n cities and computed a complete symmetric euclidean cost matrix C with components c_{ij} denoting the (air-) distances between cities i and j.

As is customary, we used the cost of the minimum spanning tree (MST) covering the cities not yet visited as a bounding function for the completion cost of the current partial tour. More precisely, a *1-tree* [4] is computed, which is connected via two extra edges to the first and the last city of the partial tour. Using Prim's algorithm, a 1-tree of n cities is computed in $O(n^2)$ operations. Hence, the node expansion time is substantial, making the TSP an ideal test suite supplement to the 15-puzzle.

[1] As pointed out by Sen and Bagchi [25], the depth-first node expansion strategy of Little's method can also be adapted to best-first or depth-first iterative-deepening. But since the search graph is small and the node expansion time is appreciable, there is no point in using IDA* or any of its memory variants.

```
algorithm IterativeDeepening;
begin
    bound := h(root);                                   { initial bound is heuristic estimate }
    repeat
        bound := DepthFirstSearch (root, bound);        { perform iterative-deepening DFS }
    until solved;
end.

function DepthFirstSearch (n, bound): integer;          { returns next cost bound }
begin
    if h(n) = 0 then begin
        solved := true; return (0);                     { found a solution: return cost }
    end;
    new_bound := ∞;
    for each successor n_i of n do begin
        if c(n, n_i) + h(n_i) ≤ bound then
            b := c(n, n_i) + DepthFirstSearch (n_i, bound − c(n, n_i));   { search deeper }
        else
            b := c(n, n_i) + h(n_i);                    { cutoff }
        if solved then return (b);
        new_bound := min (new_bound, b);                { compute next iteration's bound }
    end;
    return (new_bound);                                 { return next iteration's bound }
end;
```

Figure 1: Iterative-Deepening A*

3 Iterative-Deepening A*

Iterative-Deepening A, IDA** for short, performs a series of cost-bounded depth-first searches with successively increased cost-thresholds. The total cost $f(n)$ of a node n is made up of $g(n)$, the cost already spent in reaching that node, plus $h(n)$, the estimated cost of the path to the nearest goal. At each iteration, IDA* does the search, cutting off all nodes that exceed a fixed cost bound. At the beginning, the cost bound is set to the heuristic estimate of the initial state, $h(root)$. Then, for each iteration, the bound is increased to the minimum path value that exceeds the previous bound.

Figure 1 gives a sketch of IDA*. The algorithm consists of a main IterativeDeepening routine, that sets up the cost bounds for the single iterations, and a DepthFirstSearch function, that actually does the search. The maximum search depth is controlled by the parameter *bound*. When the estimated solution cost $c(n, n_i) + h(n_i)$ of a path going from node n via successor n_i to a (yet unknown) goal node does not exceed the current *bound*, the search is deepened by recursively calling DepthFirstSearch. Otherwise, subtree n_i is cut off and the node expansion continues with the next successor n_{i+1}.

Of all path values that exceed the current bound, the smallest is used as a cost bound for the next iteration. It is computed by recursively backing up the cost values of all subtrees originating in the current node and storing the minimum value in the variable *new_bound*. Note, that these backed-up values are *revised cost bounds*, which are usually higher – and thus more valuable – than a direct heuristic estimate. In the simple IDA* algorithm shown in Figure 1, the revised cost bounds are only used to determine the cost threshold for the next iteration. In conjunction with a transposition table (see the Appendix), they can also serve to increase the cut offs.

With an admissible heuristic estimate function (i.e. one that never overestimates), IDA* is guaranteed to find the shortest solution path. Moreover, it has been proved [8, 11], that IDA* obeys the same asymptotic branching factor as A*, if the number of nodes grows exponentially with the solution depth. This growth rate is called the *heuristic branching factor* b_h (see Section 6.2). On the average IDA* requires $b_h/(b_h - 1)$ times as many operations as A* [27]. While the search overhead diminishes with increasing b_h (e.g., 11% overhead at $b_h = 10$, 1% at $b_h = 100$), IDA* benefits from the elimination of unnecessary node re-examinations in the shallow tree parts (all iterations before the last).

4 Related Limited-Memory Algorithms

Two algorithms have been proposed to fill the gap between the memory-intensive A* on one hand and the faster, but more node-intensive, IDA* on the other.

The recursive best-first search algorithm *MREC* of Sen and Bagchi [25] might best be described as an amalgamation of IDA* and A*. Like IDA*, MREC examines all nodes by iterative-deepening until a goal is found. Like A*, MREC grows an explicit search graph, that contains all nodes of the first few levels, until the available memory is exhausted. Unfortunately, the memory usage is static. Once occupied by an initial explicit sub-graph, the storage space cannot be re-used by other, more valuable, nodes that might be found at a later time. Moreover, MREC starts all iterations at the root node, irrespective of the explicit search graph that has already been built [25, p. 298]. The repeated traversal of the explicit graph is the price paid for the missing Open List[2]. Even so, one would expect a graph

[2]The repeated traversal of the explicit graph can be avoided by connecting the frontier nodes in a linked list, similar to A*'s Open list. But even then the savings would be negligible, because the list must be sorted before each new iteration. Only the backing up of the revised estimate

traversal to be much faster than generating new nodes and linking them to the explicit search graph. Unfortunately, this is not the case for the 15-puzzle with its cheap operator generation, and so Sen and Bagchi report poor CPU-time results [25, p. 299]. They also achieved only negligible (1%) node reductions as compared to IDA*, because their implementation builds a tree rather than a graph and does not check for duplicate nodes. On the other hand, MREC-implementations that eliminate transpositions were also found to be slow (again compared to IDA*), because of the costly maintenance of the explicit search graph.

Chakrabarti *et al.* [2] proposed *MA**, an iterative-deepening variant of Ibaraki's *Depth-m Search* [7]. Similar to MREC, MA* also grows an explicit search graph until the available memory space is filled, but dynamically re-assigns memory space to other states according to some merit value. When the storage space is exhausted, MA* is not confined to a pre-determined node expansion sequence, but starts a best-first search on the tip nodes of the explicit graph. The node selection is based on the backed-up cost values of the pruned nodes, which are more reliable than the direct heuristic estimates. Although the favorable results of Chakrabarti *et al.* were found to be erroneous (they "inadvertently compared IDA*'s node generation figures with MA*(0)'s node expansion figures" [12, p. 2]), other researchers built successfully on the basic ideas of MA*. *Iterative Threshold Search (ITS)* by Mahanti *et al.* [12] employs a fast node generation scheme (like IDA*) while making use of the available memory (like MA*). Another proposal, *SMA** by Russell [21], uses the "pathmax" node information of the backed up f-values.

Still, these methods are much slower than the memory-functions proposed here, while generating a comparable amount of nodes. This is because the others all operate on an explicit search graph, whose construction, maintenance and traversal is a time-consuming task. In each step, a tip node n with lowest $f(n)$-value is selected for further expansion. Since the explicit graph must be large to be effective, the node selection time dominates the runtime of the algorithm. From experiments with Stockman's best-first SSS*-algorithm [28] it is known that a reduced node count seldomly pays for the increased memory management costs [19]. Our hash transposition techniques, in contrast, are easier to implement and operate in unit time while retaining a similar node-count performance.

Aside from these memory-bound variants, there has been a flurry of proposals, that attempt to reduce the search overhead by allowing a more liberal increase of the cost bound between iterations. Such methods include Stickel and Tyson's

values in the explicit search graph can be saved.

evenly bounded depth-first search [27], Sarkar *et al.*'s iterative-deepening search with controlled re-expansion *IDA*_CR* [22], and the hybrid iterative-deepening depth-first branch-and-bound variants *DFS** [18] by Rao *et al.* and Wah's *MIDA** [29]. All these schemes attempt to reduce the search overhead by increasing the cost bound by more than the minimal value. As a consequence, node expansion cannot be stopped at the first solution, but must continue (possibly with a reduced cost bound) until all shorter paths have been checked for cheaper solutions. However, these systems can be modified to return quickly with a (possibly non-optimal) solution, one that is known to lie within an ϵ-range from optimality.

5 Improved Information Management

The enhancements that exploit node information gathered in the process of iterative-deepening follow two different schemes: (1) node ordering and (2) avoidance of re-expansions.

5.1 Strategies for Trees: Node Ordering Heuristics

Node ordering refers to the dynamic re-ordering of node successors. It speeds up the last iteration (where the goal is found) by investigating the most plausible successors first, but no savings are achieved in the shallower iterations. There are three ordering schemes of interest:

SORT: The simplest type of node ordering works without node information from previous iterations and has little space overhead of $O(wd)$. It is based on re-arranging the successors n_i of interior nodes n in increasing order of their heuristic estimates $h(n_i)$. Successors with low estimates are visited first, with the intention of reducing the distance to the goal. Like the well-known *hill climbing* techniques, SORT adds a local best-first component to the otherwise random heuristic search. In the 15-puzzle, SORT works much like a human player, who initially tries to shift tiles as near as possible to their destination positions.

Although this scheme helps humans in their search for non-optimal solutions, the savings achieved in (optimal) IDA* search rarely compensate for the additional overhead [17, p. 471]. This is because of the limited information horizon that the successor pre-sorting is based on. More sophisticated variants of SORT work with revised cost values of deeper tree levels (see the TRANS+MOVE variant in Section 5.2), or re-arrange the nodes of a whole search frontier [17].

PV: When searching adversary game trees like chess or checkers, each iteration yields w best paths starting at the root node. One of them, the *principal variation*, is the move sequence actually chosen if the players follow the minimax principle. The other $w-1$ paths are called *refutation lines* [1, 13]; they serve to prove the inferiority of their particular root move. Current principal variation and refutation lines are re-expanded first during each new iteration.

In single-agent search problems, the refutation line idea is not directly applicable, because there are no opponent moves that could be refuted. Only the principal variation line (PV) can be employed to investigate the most promising path first. We extend the PV heuristic by saving a whole subtree of paths from the root, instead of only the best available continuation. The leaf nodes of this subtree all lie at the same maximum distance from the start configuration. Because the search is cost-bounded, these leaves lie closest to the goal, that is, they have the largest g- and consequently lowest h-values.

History: The *history heuristic* [23] proved useful in the domain of two player games. It achieves its performance by maintaining a "score" table, called the *history table*, for every move seen in the search graph. Note, that History is the only heuristic that is based on sorting moves (operators) rather than nodes (states). All moves that are applicable in a given position are examined in order of their previous success. Compared to Sort, the history heuristic is less sensitive to the current context, yet it provides more reliable information on the success of the operators. In addition, History does not depend on domain specific knowledge (like heuristic estimate functions). It simply accumulates success scores from the previously expanded subtrees.

For the 15-puzzle, one needs a three dimensional array that holds a measure of the goodness of a move for each possible tile, each source position and each move direction. This gives 16 (tiles) $\times$ 16 (positions) $\times$ 4 (max. move directions) = 1024 move scores. In the traveling salesman problem, a two dimensional history table of size $n \times n$ is needed, where n is the number of cities on the tour. As a measure for the goodness of a move, we counted the number of occurrences the specific move led to the deepest subtree (i.e. the subtree that came closest to the goal).

5.2 Advanced Techniques in Graphs: Avoiding Re-Expansions

Most applications spawn a decision graph (with multiple paths ending in the same position) rather than a tree. In such cases, memory functions should be employed to avoid unnecessary re-expansions of previously visited nodes. The utilization of memory tables is twofold: First, they are used to eliminate cycles and transpositions within single iterations, and second, they serve to cache node

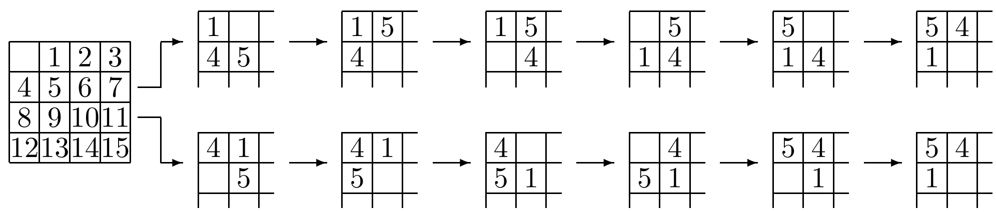


Figure 2: Shortest move transposition in the N-puzzle

information from one iteration to the next.

A *move cycle* is a sequence of operators, which, after going through some intermediate states, finally returns to the starting state. In general, move cycles can be eliminated with a stack of size g that holds all nodes on the path from the root to the current node. In the 15-puzzle, however, cycle elimination does not pay off, because closed move cycles occur only seldomly (less than 0.03% of the nodes lie on cycles, after the trivial 2-move cycle is removed by the move generator). As an example, the shortest cycle (see Figure 2, which can be viewed as a cycle when when reversing one line of arrows) consists of 12 moves. Since cycles contain inferior nodes with high goal distances h, the total expansion cost $g + h$ usually exceeds the cost threshold before completion. Note, that in the traveling salesman problem all cycles are automatically eliminated by the move generator.

TRANS: Move transpositions are more common. They arise when different paths end in the same position, see Figure 2. In the 15-puzzle, transpositions occur in search depths ≥ 6. They can be traced with a *transposition table* [30] that (ideally) holds a representation of every visited position, plus the cost bound to which the position has been searched. When the current position is found in the table, its subtree can be pruned if the remaining cost bound is less or equal to the corresponding bound retrieved from the table. Pseudo code in the Appendix illustrates the use of a transposition table in iterative-deepening search. Note that *revised* cost values (back-up values of deeper tree levels) are stored in the transposition table, sometimes allowing cut offs, even when the remaining search depth is deeper than that given in the table.

Because of its fast access time, a hashing technique is customarily used for implementing large transposition tables. The initial hash access index is a function of the board configuration with all redundant information removed. In the 15-puzzle, it includes the positions of all tiles on the board, whereas in the traveling salesman problem the index is a function of the subset of the remaining cities plus the last visited city. Note, that this scheme allows *pruning by dominance* [6], that

Algorithm	*Nodes*		*Time*
	mean	σ	
IDA*	100		100
Sort	99	42	105
PV	86	52	87
History	94	48	108
Trans	53	6	76
Trans+Move	46	28	63
Trans+Move+History	46	32	68
IDA*, iter. $1, \ldots, n-1$	54	26	–

Table 1: Empirical results on the 15-puzzle, 100 problems by Korf [8]

is, other partial tours covering the same cities in a different order (but with the same first and last city) are cut off.

Transposition tables should be allocated as much space as possible. (We used 256 K entries in both the 15-puzzle and TSP applications.) As the table gets filled, collisions occur. But old information is only overwritten if the current position has been searched more deeply.

Trans+Move: When the current position is found in the transposition table, but has been searched to an insufficient depth, the formerly best move (the one yielding the longest path) is retrieved from the table and tried first. Apart from selecting promising moves first, this approach has the additional advantage that information about the next position will probably also be held in the table. Thus, complete sub-variations are descended with minimal effort.

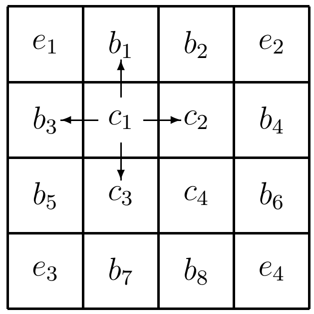
In the traveling salesman problem, move pre-sorting is based on the successor values stored in the table, because a table access is faster than the computation of the minimum spanning tree (our heuristic estimate function).

6 Experimental Results

The performance of the algorithms has been empirically evaluated using the 15-puzzle and the traveling salesman problem.

6.1 The Fifteen-Puzzle

For the 15-puzzle, we used Korf's selection of one hundred randomly generated problem instances as a test suite [8]. To ensure that the hard problems with high



$e_1, \ldots, e_4$: edge position
$b_1, \ldots, b_8$: border position
$c_1, \ldots, c_4$: center position

Figure 3: Tile positions in the 15-puzzle

node counts do not dominate the results, we computed the mean of the percentage difference relative to Korf's published solutions[3].

In all, ten different combinations of enhancements were tried and the results from six of them are presented. Table 1 gives the average number of node generations (with standard deviation σ) and the relative CPU time consumption of our implementation. All data is normalized to that of pure IDA*.

As expected, the node ordering heuristics (SORT, PV and HISTORY) are of limited use, because they only reduce the search effort of the final iteration. Table 1 shows, that the pre-sorting of successor nodes according to increasing heuristic estimates (SORT) does not pay off – neither in terms of node expansions, nor in terms of CPU time. A quick calculation reveals that SORT favors board configurations with the blank square being either in an edge or border position (Figure 3), because these configurations enjoy (statistically) lower h-values:

Let the blank be located in the center position c_1. For each adjacent field b_3, b_1, c_2, c_3, we calculate the probability that SORT will first move the blank to that field, because the resulting configuration enjoys a lower heuristic estimate. Or, the other way around, we enumerate for each source field all tiles that reduce the heuristic distance:

- b_3: When a tile moves from b_3 to c_1 the heuristic distance reduces by 1 in 12 out of 15 cases (because the tile's goal square is in the rightmost 3×4 block).

- b_1: When a tile moves from b_1 to c_1 the heuristic distance reduces by 1 in 12 out of 15 cases (because the tile's goal square is in the lower 4×3 block).

[3]Our replication of Korf's experiment identified three cases of differing node counts (presumably due to typographical errors in the original presentation [8, p. 106]):

Nr.	*Korf*	*Our Version*	*Difference*
22	750,746,755	750,745,755	-1,000
88	6,009,130,748	6,320,047,980	+310,917,232
89	166,571,097	166,571,021	-76

c_2: When a tile moves from c_2 to c_1 the heuristic distance reduces by 1 in 7 out of 15 cases (because the tile's goal square is in the left 2×4 block).

c_3: When a tile moves from c_3 to c_1 the heuristic distance reduces by 1 in 7 out of 15 cases (because the tile's goal square is in the upper 4×2 block).

In summary, there are 24 (=12+12) out of a total of 38 (=12+12+7+7) cases, where the blank will be moved from c_1 to a boarder position (b_3 or b_1) first. This gives a total of $24/38 = 63\%$. The chances vary slightly for all four center positions (because of the asymmetry caused by the destination square of the blank), but they are all between 58% and 63%, which is well above average. Likewise, we calculated chances between 44% and 55% for a blank to be first moved from a boarder position b_i into the adjacent edge e_j, which is also significantly higher than the expected random 33% chance.

On one hand, configurations with a blank tile in an outer position have lower mobility and are thus less desirable. But on the other, fewer moves are possible in such configurations, which reduces the size of the emanating subtree. It seems that the positive and negative effects of SORT just compensate for each other, leaving no net gain [17, p. 471]. This is no surprise when considering the limited information horizon that node ordering is based on. We therefore implemented an extended sorting scheme that works on a deeper (two level) lookahead. But it gave only marginal additional improvements while requiring more CPU-time. Better results are achieved when the pre-sorting is based on previously acquired node values of deeper tree levels, see TRANS+MOVE.

The PV heuristic is more effective than SORT: On the average, 14% of the node expansions are saved by searching the longest paths first, which confirms recent results on an exhaustive evaluation of the 8-puzzle [20]. However, the savings exhibit high variability. In some instances, the principal variation subtrees lead directly to the goal, whereas in other cases the PV-variant examines more nodes than the original IDA*. Note, that the PV heuristic does not involve time-consuming operations. It comes as a by-product of the search for an optimal path. Thus, any savings in the number of node expansions directly speeds up the execution time.

The HISTORY heuristic saves only a meager 6% of the node expansions, irrespective of the problem size. Considering its remarkable success in the domain of chess [23], one would have expected a much better result. But the two domains differ in several respects. First, in chess, only a small fraction of the total game tree is searched, so that the examined positions obey similar properties. Hence, a chess move that once caused a cutoff, will probably be effective whenever it can be applied in the future. This is not the case in the 15-puzzle, where board configurations are widely different, because the search depths (average of 53 moves) are greater.

Second, the 15-puzzle lacks clear criteria for measuring the merit of a move, thus taking the path lengths seems to be an obvious choice. But in our experiments, it turned out that many paths end at the same length, and hence a finer grained secondary measure – like a chess evaluation function – is needed. For example, a function that retains some secondary good values, even though this might reduce the effectiveness of IDA*.

With a transposition table (Trans), IDA* consistently examines fewer nodes in every single problem instance, yielding an average node count reduction of 47%. This is more than the 35% savings achieved in the 8-puzzle [20], because the pruned subtrees are deeper. More interestingly, no signs of table overloading were spotted in the hard 15-puzzle problems with large search trees. On the contrary: The performance of the transposition table seems to increase with growing problem size. This is because, on the one hand, there are more transpositions and cycles in deeper search trees, and on the other, many more nodes are eliminated by every single cutoff. In practice, the low standard deviation is another favorable aspect of Trans, because one can expect an almost constant efficiency gain by nearly one half for every problem.

Additional savings can be achieved by first expanding the best move stored in the transposition table (Trans+Move). Generally, the best move is a good choice. In six problem instances, however, the best move failed so miserably, that slightly more nodes were searched than with the original IDA*. The erratic behavior of these few cases results in a high standard deviation, and is a typical property of tree pruning systems. Adding the history heuristic to Trans+Move does not yield further benefit. In practice, one would avoid the history heuristic, with its additional program complexity and minor storage overhead, but retain a simple transposition table which holds the previously best move, and the value of the position.

The last line of Table 1 gives the average number of node expansions in all iterations excluding the last. This number corresponds to the best performance, that could be achieved with a perfectly informed node ordering mechanism, one that finds the goal node right at the beginning of the last iteration. Viewed in this light, the combinations involving Trans look even more favorable since they search fewer nodes than even this optimally informed IDA*.

These results are telling enough, but Figure 4 presents the data in a graphical

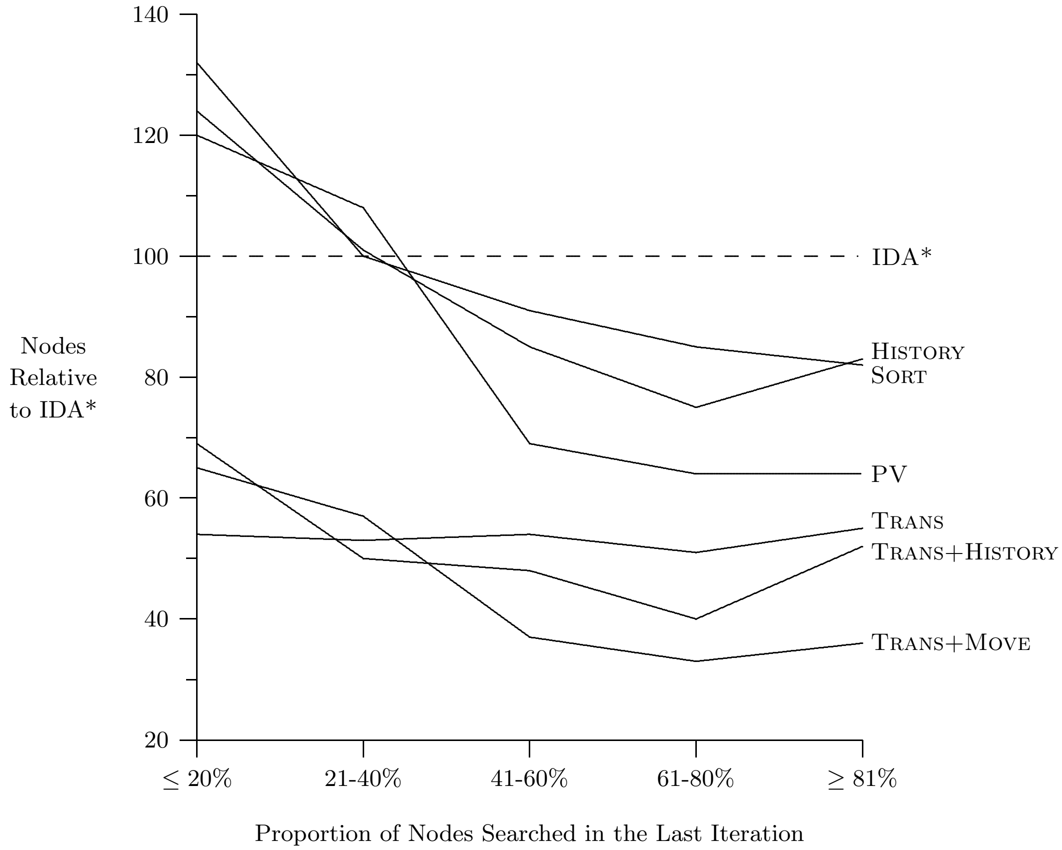


Figure 4: Relative performance of IDA* enhancements on the 15-puzzle

form and shows more clearly how the use of a transposition table is the one mechanism that is consistently effective. Here, Korf's hundred random problem instances are grouped into five sets (of increasing order of difficulty), defined by the proportion of the nodes searched in the goal iteration. The trees in the first problem set (0-20%) are already relatively well ordered for the simple IDA* and it seems hard to achieve further savings with any of the move ordering heuristics. On the contrary: in their attempt to improve the expansion order, History, Sort and PV often expand more nodes in the end. Only when the proportion of the goal iteration nodes is above 40% do these techniques become effective.

Schemes involving a hash table are almost equally effective over the whole range of problems. A simple transposition table (Trans) saves about half the node expansions, while the successor ordering techniques Trans+Move and History become even more effective when the tree is poorly ordered. In practice, based on Figure 4, one would use the combined version Trans+Move.

6.2 The Traveling Salesman Problem

At first sight, the TSP seems to be better suited for iterative-deepening search, because more successor-cities must be considered in the interior nodes of the TSP search graph than there are move choices in the 15-puzzle. From this, one should expect the node count to grow faster between iterations, which in turn should reduce the overhead incurred by re-expanding the shallow tree parts. But, as it turns out, the opposite is true.

In the following, we distinguish two types of branching factors. First, the *edge branching factor* b_e is defined as the average number of operators (edges) that are applicable to a state (node) of the search graph. It can be determined by computing the ratio of the total move generations to the number of interior (=non-terminal) nodes.

For the n-city TSP, we derive a lower bound of the edge branching factor by counting the node successors of an arbitrary path in the search graph. At the root node, there exist $n-1$ successors, at the first level $n-2$, at the second $n-3$, and so on, up to k successors at the last (cut off) level, where k is the number of the still unvisited cities. For the longest path (the solution path) we have $[(n-1)+(n-2)+\cdots+3+2+1+1]/n \approx n/2$. Since all other paths in the search graph are incomplete, $n/2$ gives a lower bound on the edge branching factor of the n-city TSP.

For the 15-puzzle, the edge branching factor is $b_e \approx 2$. This number is derived directly from Figure 3 by summing over all possible tile positions the number of move choices and dividing by 16 (that is, $48/16 = 3$), and then adjusting for the back move by subtracting 1. Hence, $b_e \approx 2$. In practice, b_e is marginally higher, because there is no back move in the initial position and because the blank is more likely to be located in one of the four center positions.

The second important parameter, the *heuristic branching factor* b_h, measures how many new nodes are generated when searching with the next larger cost bound. It is defined as the average node ratio of two consecutive iterations, $b_h = nodes_i/nodes_{i-1}$. We include in the computation only the shallow iterations $i_1, \ldots, i_{n-1}$, because in the last iteration (where the goal is found) the node count depends much on the expansion order and is therefore highly variable. Clearly, b_h depends on the quality of the heuristic estimate function and the efficiency of the search method.

For the 15-puzzle, we determined $b_h = 6.68$ (with $\sigma = 1.77$) by running IDA* on Korf's selection of one hundred random problem instances. This value is sufficiently high to allow effective use of iterative-deepening techniques. Moreover, the 15-puzzle is one of the rare applications with $b_h > b_e$, which further increases the effectiveness of IDA* as compared to other search methods. The heuristic branching factor is this big, because an increase of the cost bound by 2 (which is the only possible increase between iterations) allows *all* nodes at a search frontier to be expanded by at least one extra level – and some of them much more.

In the TSP, in contrast, the increase in the cost bound between iterations is not fixed to a predetermined value. Most often the cost bound is raised by a small amount only, allowing extension of only few frontier nodes in the next iteration. This results in a heuristic branching factor that is much lower than the edge branching factor. The exact magnitude of b_h depends on the domain of the inter-city distances. In the extreme case, that is with inter-city distances drawn from the real numbers, only one frontier node (the one that gave rise to the temporary iteration's cost bound) is expanded in every new iteration. Then, the heuristic branching factor is close to 1 and iterative-deepening is not efficient [18, 22]. The problem might be overcome by increasing the cost bound by more than the minimum value that exceeded the previous bound. But this approach could return sub-optimal solutions, unless special provision is taken.

The heuristic branching factor can be controlled in the range $1 \leq b_h \leq b_e$ by choosing suitable domains, from which the inter-city distances are drawn. This makes the TSP an ideal vehicle for studying the effectiveness of the proposed IDA* enhancements under various b_h. In our experiments, we used city coordinates that have been randomly drawn from the integer intervals $[1, 25], [1, 50], [1, 75]$ and $[1, 100]$. This results in heuristic branching factors (of the simple IDA*) ranging from 1.71, 1.29, 1.20 to 1.13, respectively. A total of fifty 20-city problems were solved for each algorithm/interval combination. All interconnections are included in the network, and the traveling salesman problem is complete, symmetric and euclidean. As is customary, we used the minimum spanning tree [4] of the remaining cities to estimate the completion cost of the partial tour.

Table 2 shows the experimental results with city coordinates drawn from the intervals $[1, 50]$ and $[1, 100]$. The performance is given relative to IDA* in terms of node expansions and CPU time consumption. As can be seen, neither of the node ordering heuristics (PV+Sort or History) yields substantial performance improvements. This is not surprising, since only 8% of the total nodes are visited in the last iteration, yielding an upper bound on the maximal improvement that can be achieved by any kind of node ordering (see the last line in Table 2). The History results are based on a 2-dimensional history table that holds for every city pair the frequency it contributed to the longest tour. Experiments with chains

Algorithm	*Domain [1,50]*		*Domain [1,100]*	
	Nodes	*Time*	*Nodes*	*Time*
IDA*	100	100	100	100
PV+SORT	95	95	98	99
HISTORY	95	95	99	99
TRANS	36	38	27	28
TRANS+MOVE	36	37	27	28
TRANS+REHASH	28	30	19	20
IDA*, $i_1 \ldots i_{n-1}$	92	–	97	–

Table 2: Relative performance on the 20-city-TSP, 50 problems

of three cities gave only marginal additional improvements, while occupying more resources (a 3-dimensional array).

Much better results of up to 73% node savings are achieved with a transposition table. While TRANS uses the table entries only for pruning duplicated states, TRANS+MOVE sorts the successors of interior nodes according to the retrieved estimate values. Although this did not yield any further savings in terms of node expansions, we found TRANS+MOVE to be much faster, because the computation of the minimum spanning tree takes more CPU time than a simple table retrieval.

In the best case, TRANS examines only 27% of the nodes that are visited by IDA*. The savings are better than can be achieved in the 15-puzzle. This is especially interesting, since the table entries cannot be used as effectively as in the 15-puzzle, where further expansion is stopped as soon as an entry with a value greater or equal to the remaining cost bound is retrieved. Such immediate cut offs are not possible in the TSP, because care must be taken not to prune subtrees containing a new cost bound for the next iteration. As is always the case in applications where the cost bound increase is not known *a priori*, cut offs are only feasible when the retrieved cost bound is higher than the temporary candidate for the next cost bound.

Table 2 also shows how further savings are achieved with the more sophisticated hashing techniques. The TRANS+REHASH variant resolves storage collisions by giving preference to states encountered in the shallow graph levels near the root. In addition, it does limited re-hashing (up to a chain length of 3) by moving the lower priority entries to the end of the re-hashing chain. As a result, the mostly re-expanded nodes at the shallow tree levels enjoy early occupancy in the transposition table and a fast retrieval time (due to the shorter chain lengths).

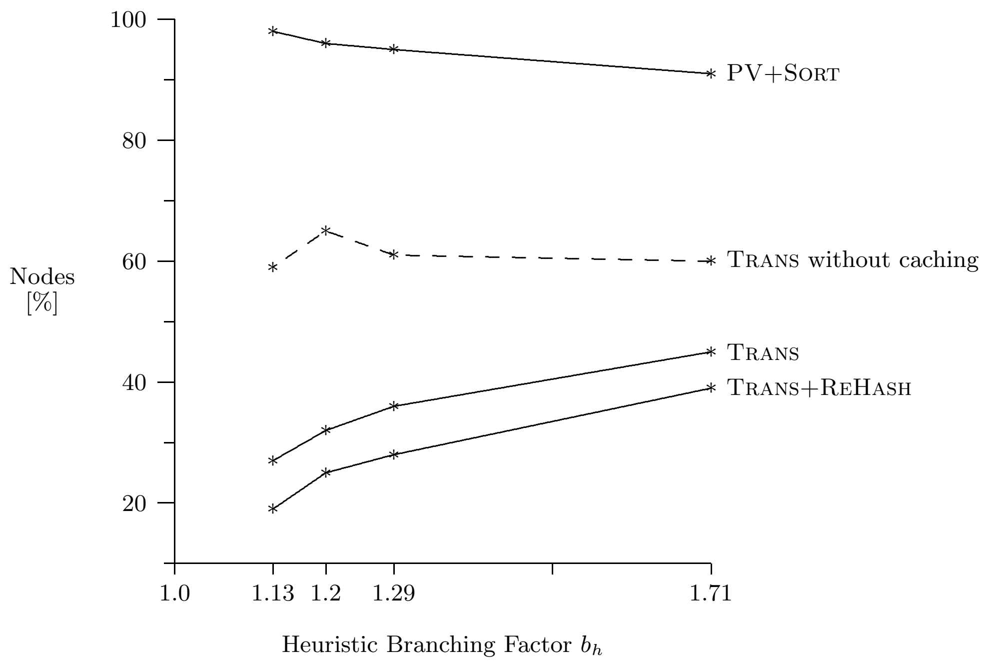


Figure 5: Relative performance on the 20-city TSP

Since these are also the nodes for which the MST computation is most costly, CPU time is saved even when the retrieved table value does not permit a cut off.

Figure 5 illustrates in a more general way the influence of the tree characteristics on the relative search efficiency. The data shown is that from Table 2, but expanded to include information from all four domain intervals considered. Instead of plotting the performance relative to the domain of the city coordinates, we took the heuristic branching factor achieved with the simple IDA* as a performance measure. (In other words, the shown b_h's of 1.13, 1.20, 1.29, 1.71 correspond to the city coordinate domains [1,100], [1,75], [1,50] and [1,25].)

The top graph in Figure 5 (PV+SORT) illustrates the growing importance of successor ordering schemes with increasing b_h. This is caused by the larger number of nodes in the last iteration, which rectify any additional effort invested in sorting the promising nodes to the beginning of the search. On the other hand, many of the nodes that are expanded deeper in graphs with large b_h are not contained in the transposition table, which reduces the relative performance of TRANS and TRANS+REHASH – see the two graphs at the bottom of Figure 5.

Interestingly, the additional transposition table savings in graphs with low heuristic branching factors are almost exclusively due to node information gathered in previous iterations. The amount of cycles and transpositions that are detected in the same iteration remains constant over the whole range of branching

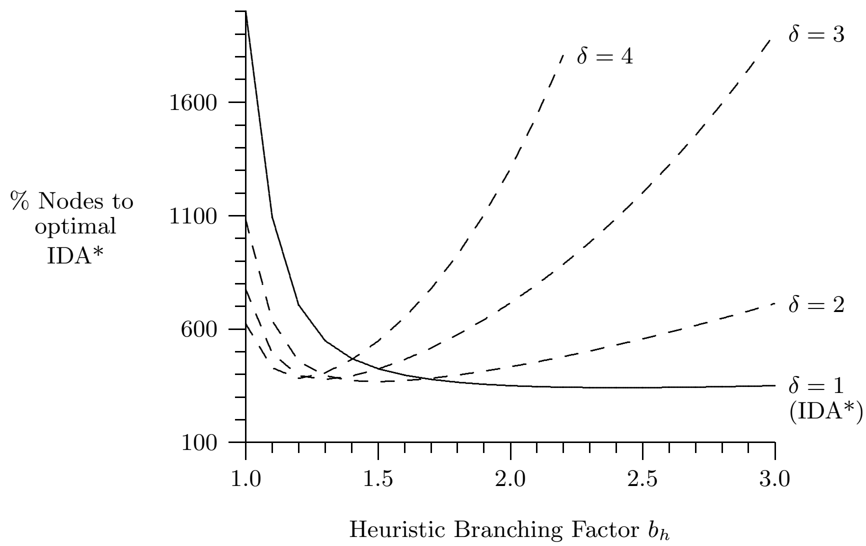


Figure 6: Effect of more liberal cost-bound increases δ

factors. This is confirmed by the dashed line in the middle of Figure 5 (at about 60%), which depicts the savings incurred by information gathered in the same iteration. For these data points, the transposition table has been cleared between iterations. In total, roughly 40% of the node generations can be saved by avoiding cycles and transpositions, while an additional 20 to 40% reduction can be achieved by exploiting information gathered in previous iterations.

In applications with low heuristic branching factors (like the TSP) iterative-deepening is clearly not the best solution method. To minimize repeated node expansions, the cost bound should be increased by more than the minimal amount [9, 15, 18, 22, 29]. But by how much should the cost bound be increased and up to which branching factor is it beneficial to do so? Figure 6 presents a numerical evaluation of iterative-deepening search with various cost bound increments δ. We made the following simplifying assumptions:

- there is one goal node in the solution depth $g = 37$, [4]
- we assume unity arc costs,
- the solution density does not increase with the search depth,
- the heuristic branching factor b_h is constant over all iterations,
- the cost bound increments δ remain constant over the search (we did not investigate decreasing or increasing δ).

[4] This solution depth occurred often in the TSP with domain [1,75].

The data points in Figure 6 are plotted relative to the node expansions of an optimally informed IDA*, one that performs iterations $i_1, i_2, \ldots, i_{36}$ and detects a solution in the *first* leaf node in the 37^{th} iteration (the goal iteration). Clearly, the total node count depends much on the fact whether the solution depth is a multiple of the cost bound increment δ. If this is the case, an optimal solution will be found in the last iteration, while saving some intermediate iterations. Figure 6 shows a worst case situation, where the solution depth $g = 37$ is a prime. As can be seen, the larger cost-bound increases are only beneficial in trees with very small branching factors (e.g., $\delta = 4$ is advantageous in trees with $b_h \leq 1.4$).

In practice, one would use a method that dynamically adjusts the cost bound increments to the heuristic branching factor, so that a sufficient number of new nodes are expanded in successive iterations (e.g. IDA*_CR, [22]). Other practical alternatives include hybrid iterative-deepening and depth-first branch-and-bound algorithms like DFS* [18] and MIDA* [29].

7 Conclusions

We adapted commonly used search techniques from the domain of adversary game-tree searching to single-agent iterative-deepening search. We found that avoiding transpositions and cycles is more lucrative than any kind of operator pre-sorting. The best combination of the proposed techniques, namely a transposition table with node successor ordering information, reduces the size of the search graph by one half (in the 15-puzzle) or even by three quarters (in the TSP). This is possible because the saved information can be used to detect duplicate states and to guide the expansion process to the most promising direction in the search tree. In both applications, our Trans+Move enhancement generates fewer nodes than a perfectly informed (non-deterministic) IDA*, which runs through all iterations $i_1, i_2, \ldots, i_{n-1}$ and finds a goal node at the very first node expansion in the final iteration i_n.

From a CPU-time performance standpoint, the 15-puzzle has proved to be an especially difficult application to improve, because of cheap operator costs and low branching factors. Although the simple successor ordering of Sort did not pay off, the other heuristics, namely PV, Trans and Trans+Move, reduce the search time by 13, 24 and 37%, respectively. These results compare favorably to those of others [25, Table 2], [2, 12, 21, 22, 29].

In practice, one would first include the PV-heuristic, because of its negligible space and time overheads. It simply uses standard information on the best subtree that is needed to determine the solution path. If memory space is available, one would then include a transposition table that holds all states seen during the search. Since a table access needs only unit time, it does not affect the time complexity of the program.

Transposition tables are most beneficial in applications with measurable operator costs, like the traveling salesman problem. Depending on the range of inter-city distance values, a transposition table of 256 K entries reduces the search time by as much as 72%. The CPU time saving corresponds to a node reduction of 73%, which justifies our assumption that unsuccessful table accesses are easily compensated by the fast successful retrievals.

Another favorable aspect of the hashing technique is that it can be efficiently applied in parallel environments. Although with tree structured data types, a whole path must be sent to identify a single node, hashing techniques need only transfer one hash key (that usually consists of one memory word only). Thus, hashing techniques make it possible to profit from the computations of the other processes.

Ease of implementation and maintenance is also a key issue. In our experience [19], hashing tables are much easier to implement and debug than the tree-structured data types of A* [3] and other IDA* variants [2, 21, 25]. In some way the transposition table plays a role similar to A*'s Open and Closed lists, with greater flexibility and speed, but with some risk of omission. When space restrictions are tight, table overloading might become a problem. It is then customary to overwrite the older information from deeper tree levels. The rationale is to give preference to the precious information on nodes near the root, where more CPU-time has been spent to search the emanating subtree.

Acknowledgments

Foundations to this work were laid during the first author's stay at the University of Alberta as a Killam Postdoctoral Fellow. There he benefited from discussions with Jonathan Schaeffer and his joint work on parallel IDA* implementations. We are also indebted to Alan Sharpe for improvements to our hash-transposition algorithm, and to the referees for their constructive comments.

Without the financial support of the Natural Sciences and Engineering Research Council of Canada under Grants OPG36952 and OPG07902, this work would not have been possible.

[1] S.G. Akl and M.M. Newborn, "The principal continuation and the killer heuristic", *Procs. ACM Nat. Conf.*, Seattle, pp. 466–473, 1977.

[2] P.P. Chakrabarti, S. Ghose, A. Acharya and S.C. de Sarkar, "Heuristic search in restricted memory", *Art. Intell.*, vol. 41, pp. 197–221, 1989/90.

[3] P.E. Hart, N.J. Nilsson and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", *IEEE Trans. Sys. Sci. Cybern.*, vol. SSC-4, no 2, pp. 100–107, 1968.

[4] M. Held and R.M. Karp, "The traveling salesman problem and minimal spanning trees", *Operations Research*, vol. 18, pp. 1138–1162, 1970.

[5] M. Held, R.M. Karp. "The traveling salesman problem and minimal spanning trees: Part II", *Mathematical Progr.*, vol. 1, 6–25, 1971.

[6] T. Ibaraki, "The power of dominance relations in branch-and-bound algorithms", *J. Ass. Comput. Mach.*, vol. 24, no 2, pp. 264–279, 1977.

[7] T. Ibaraki, "Depth-m search in branch-and-bound algorithms", *Intl. J. of Comp. and Inf. Sc.*, vol. 7, no 4, pp. 315–343, 1978.

[8] R.E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search", *Art. Intell.*, vol. 27, no 1, pp. 97–109, 1985.

[9] R.E. Korf, "Optimal path-finding algorithms", in *Search in Artificial Intelligence*, L. Kanal and V. Kumar, Eds., Springer-Verlag, New York, 1988.

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

[11] A. Mahanti, S. Ghosh, D.S. Nau, A.K. Pal and L. Kanal, "Performance of IDA* on trees and graphs", *10th Nat. Conf. on Art. Int., AAAI-92*, San Jose, CA, pp. 539–544, 1992.

[12] A. Mahanti, D.S. Nau, S. Ghosh, and L. Kanal, "An efficient iterative threshold heuristic search algorithm", Univ. of Maryland, College Park, Tech. Rep. CS-TR-2853, 1992.

[13] T.A. Marsland, "Computer chess methods", in *Encyclopedia of Art. Intell.*, 1st Edition, E. Shapiro, Ed., Wiley, pp. 159–171, 1987. See also, "Computer chess and search", 2nd Edition, pp. 224–241, 1992.

[14] T.A. Marsland, A. Reinefeld and J. Schaeffer, "Low overhead alternatives to SSS*", *Art. Intell.*, vol. 31, pp. 185–199, 1987.

[15] B.G. Patrick, "Binary iterative-deepening A*: An admissible generalization of IDA* search", *Procs. 9th Canadian Conf. on Art. Intell. AI'92*, Vancouver, BC, pp. 54–59, 1992.

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

[17] C. Powley and R.E. Korf, "Single-agent parallel window search", *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-13, no 5, pp. 466–477, May 1991.

[18] V.N. Rao, V. Kumar and R.E. Korf, "Depth-first vs. best-first search", *Procs. 9th Nat. Conf. on Art. Intell. AAAI-91*, Anaheim, CA, pp. 434–440, 1991.

[19] A. Reinefeld, J. Schaeffer and T.A. Marsland, "Information acquisition in minimal window search", *Procs. 9th Intl. Joint Conf. on AI*, pp. 1040–1043, 1985.

[20] A. Reinefeld. "Complete solution of the Eight-Puzzle and the benefit of node ordering in IDA*", *Procs. 13th IJCAI-Conference*, Chambéry, France, 1993.

[21] S. Russell, "Efficient memory-bounded search methods", *Procs. European AI Conf.*, Vienna, pp. 1–5, 1992.

[22] U.K. Sarkar, P.P. Chakrabarti, S. Ghose and S.C. de Sarkar, "Reducing re-expansions in iterative-deepening search by controlling cutoff bounds", *Art. Intell.*, vol. 50, pp. 207–221, 1991.

[23] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice", *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-11, no 11, pp. 1203–1212, 1989.

[24] J.J. Scott, "A chess-playing program", in *Machine Intelligence 4*, B. Melzer and D. Michie, Eds., Edinburgh Univ. Press, pp. 255–265, 1969.

[25] A.K. Sen and A. Bagchi, "Fast recursive formulation for best-first search that allow controlled use of memory", *Procs. 11th Intl. Joint Conf. on AI*, pp. 297–302, 1989.

[26] D.J. Slate and L.R. Atkin, "Chess 4.5 — The Northwestern University chess program", in *Chess Skill in Man and Machine*, P.W. Frey, Ed., Springer-Verlag, New York, pp. 82–118, 1977.

[27] M.E. Stickel and W.M. Tyson, "An analysis of consecutively bounded depth-first search with applications in automated deduction", *Procs. 9th Intl. Joint Conf. on AI*, pp. 1073–75, 1985.

[28] G.C. Stockman, "A minimax algorithm better than alpha-beta?", *Art. Intell.*, vol. 12, no 2, pp. 179–196, 1979.

[29] B.W. Wah, "MIDA*: An IDA* search with dynamic control", Univ. of Illinois, Champaign, Tech. Rep. UILU-ENG-91-2216, CRHC-91-9, 1991.

[30] A.L. Zobrist, "A new hashing method with applications for game playing", Univ. of Wisconsin, Madison, Tech. Rep. 88, 1970. Reprinted in *Intl. Comp. Chess Assoc. J.*, vol. 13, no 2, pp. 69–73, 1990.

Appendix

```
function DepthFirstSearch (n, bound): integer;          { returns next cost bound }
var
    new_bound, tt_bound, i, t: integer;
    next: node;
    succ: array [1..max_width] of node;                          { successor nodes }
    b: array [1..max_width] of integer;                   { successor's cost bounds }

begin
    if h(n) = 0 then begin
        solved := true; return (0);                  { found a solution: return cost }
    end;
    new_bound := ∞;

    for each successor n_i of n do begin
        succ[i] := n_i;
        if retrieve_tt (n_i, tt_bound) then           { if n_i is in transposition table }
            b[i] := c(n, n_i) + tt_bound;             { ... then use revised cost value }
        else
            b[i] := c(n, n_i) + h(n_i);               { ... else use heuristic estimate }
    end;

    sort (succ[], b[]);              { sort succ and b to increasing bound values n[] }

    for i = 1 to last successors of n do begin                             { recurse }
        next := succ[i];
        if b[i] ≤ bound then                                          { search deeper }
            t := c(n, next) + DepthFirstSearch (next, bound − c(n, next));
        else
            t := b[i];                                                        { cutoff }
        if solved then return (t);
        new_bound := min (new_bound, t);               { compute next iteration's bound }
    end;
    save_tt (n, new_bound);             { save lowest bound of n in transposition table }
    return (new_bound);                          { return next iteration's cost bound }
end;
```

Figure 7: Iterative-Deepening A* with transposition table and cost revision

DepthFirstSearch is called by IterativeDeepening, see Fig. 1. Note that this pseudo code depicts the 15-puzzle case. In the TSP, fewer cut-offs exist, as explained in Section 6.2.