Mitteilung Nr. 198

Memory Functions in Iterative-Deepening Search

A. Reinefeld T.A. Marsland[†]

FBI-HH-M-198/91

Mai 1991

Universität Hamburg
Fachbereich Informatik
Bodenstedtstraße 16
D-2000 Hamburg 50

[†] University of Alberta, Computing Science Dept., Edmonton, Canada T6G 2H1

Contents

Abstract

Depth-first iterative-deepening mimics a breadth-first search with a series of depth-first searches that operate with successively extended search horizons. It has been proposed as a simple way to reduce the space complexity of best-first searches like A*, thereby making the space complexity linear instead of exponential.

But there is more to iterative-deepening than just a reduction of storage space. As we show, the search overhead can be greatly reduced by exploiting node information gained in previous iterations. The information management techniques considered here owe much to their counterparts from the domain of two-player games.

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 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 [Scott 1969], when chess programmers sought for 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 allowed the search process to halt with a best available answer as soon as the 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* [Akl & Newborn 1977], *transposition tables* [Zobrist 1970, Slate & Atkin 1977] and *history tables* [Schaeffer 1989]. 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 [Marsland 1987].

In the mid 1980s, iterative-deepening was rediscovered 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*)* [Korf 1985] can be applied in all domains where excessive space requirements cause A* to fail. One such application domain is the 15-puzzle.

The space efficiency is paid for by an increased search overhead. Because IDA* does not retain path information, the shallow parts of the tree are re-examined several times. Following the same lines as above, IDA* (like any iterative search) should be improved by using node information of previous iterations.

In this paper, we show how to adapt the most commonly used memory functions from 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 the *history heuristic* [Marsland 1987, Reinefeld et al. 1985a]. 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 finds the solution node at the start of the last 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 node count reduction of 73% (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 by even greater savings.

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 former 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 an average of 363 million node expansions to solve a randomly generated problem instance, even when using the best-known heuristic estimate function, the *Manhattan* or *city-block distance*. This estimate is a sum of the 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 for any heuristic search algorithm 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 only a few (some hundreds) cities.

While the well known branch-and-bound algorithm of Little et al. [Little et al. 1963] would be the preferred solution technique for the TSP in practice[1], we have chosen the method described in Pearl's book [Pearl 1984, p. 10ff], because it builds a graph rather than a tree. In our implementation, we used a complete, symmetric euclidean cost matrix C, that has been computed by randomly generating x and y coordinates for the cities and storing the distance in the component c_{xy}.

[1]As pointed out by Sen and Bagchi [Sen & Bagchi 1989], the depth-first node expansion strategy of Little's method can be adapted to best-first or depth-first iterative-deepening. But since the node expansion time is appreciable, there is no point in using IDA* or one of its memory variants.

As is customary, we used the cost of the minimum spanning tree covering the cities not yet visited to estimate the completion cost of the current partial tour. More precisely, a *1-tree* [Held & Karp 1979] is computed, which is connected via two edges (the first and the last) to the cities 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 supplement to the 15-puzzle for a test suite.

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 the cost already spent in reaching that node $g(n)$, plus the estimated cost of the path to the goal $h(n)$. 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 exceeded the previous bound.

Figure 1 gives a sketch of IDA*. The algorithm consists of a main routine `iterative_deepening`, that sets up the cost bounds for the single iterations, and a function `depth_first_search`, 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 `depth_first_search`. 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 minimum is taken for the 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 Fig. 5 in the Appendix), however, they also serve to increase the cut offs.

When a goal node has been found, the global variable `solved` is set `true` and the nodes lying on the optimal solution path are stored in the array `path`, during

```
PROGRAM iterative_deepening;
VAR solved: BOOLEAN,                             { global termination flag }
    path:   ARRAY [1..MAX_DEPTH] of NODE;        { global solution path }
    bound:  INTEGER;           { local cost bound for single iterations }
BEGIN
   solved := FALSE; bound := h (root);      { bound is initial estimate }
   REPEAT
      bound := depth_first_search (root, 1, bound);
   UNTIL solved;
   OUTPUT "path" as optimal solution path with cost "bound";
END;


FUNCTION depth_first_search (n: NODE; depth, bound: INTEGER): INTEGER;
                               { returns cost bound for next iteration }
VAR b, new_bound: INTEGER;
BEGIN
   IF h(n) = 0 THEN BEGIN
      solved := true; RETURN (0);                { return solution cost }
   END;
   new_bound := INF;
   FOR each successor n.i of n DO BEGIN
      IF c(n,n.i) + h(n.i) <= bound THEN BEGIN          { search deeper }
         b := c(n,n.i)
               + depth_first_search (n.i, depth + 1, bound - c(n,n.i));
         IF solved THEN BEGIN
            path[depth] := n.i; RETURN (b); END;
      END ELSE
         b := c(n,n.i) + h(n.i);                               { cutoff }
      IF b < new_bound THEN
         new_bound := b;          { determine bound for next iteration }
   END;
   RETURN (new_bound);                { return next iteration's bound }
END;
```

Figure 1: Iterative-Deepening A*

the recursive backing up the final solution cost to the root node.

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 [Korf 1985], that IDA* obeys the same asymptotic branching factor as A*, if the number of nodes grows exponentially with the solution depth. IDA* requires on the average only $\frac{w}{w-1}$ times as many operations as A* in a tree of width w [Stickel & Tyson 1985]. The search overhead diminishes with increasing width. Even so, in practice IDA* wastes an intolerable amount of effort in re-examining the shallow tree parts (all iterations before the last). Depending on the average branching factor of the given application, the overhead may account for as much as 54% (as in the 15-puzzle) or even 73% (in the TSP).

4 Related Algorithms

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

One scheme, MREC [Sen & Bagchi 1989], is a recursive best-first search algorithm that 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.

This approach has two disadvantages. First, the construction and maintenance of an explicit search graph takes more CPU time, and its implementation is more error prone than the simpler memory functions proposed in this paper. Second – and more important – MREC starts all iterations at the root node, irrespective of the size of the explicit search graph that has already been built [Sen & Bagchi 1989, p. 298]. It turns out, that the repeated traversal of the explicit graph is the price paid for the missing OPEN-list[2]. As a result, MREC saves only 1% of the node expansions in the 15-puzzle, even when as many as 600,000 positions are stored in the explicit search graph [Sen & Bagchi 1989, p. 299].

More efficient – at least in terms of node expansions – is a proposal of Chakrabarti et al., named *MA** [Chakrabarti et al. 1989]. In essence, MA* is an iterative-deepening variant of Ibaraki's *Depth-m Search* [Ibaraki 1978]. Similar to MREC, MA* also grows an explicit search graph until the available memory space is filled. When the storage space is exhausted, MA* is (unlike MREC) not confined to a pre-determined node expansion sequence, but it 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. Combined with the cost-revision idea, the best-first approach saves node expansions, even when only a little memory space is available. For the 15-puzzle, it is claimed that MA* examines only 57% of the nodes that are searched by IDA* [Chakrabarti et al. 1989, p. 205].

On the negative side, however, are the increased node selection costs. In each step, MA* selects a node n in OPEN with the smallest $f(n)$-value. Since

[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 values in the shallow part of explicit graph can be saved.

the OPEN list is usually very long, the node selection time dominates the running time of the algorithm. From experiments with Stockman's SSS*-algorithm [Stockman 1979], it is known that the reduced node count seldomly pays for the increased memory management costs [Reinefeld et al. 1985b].

In addition, MA* maintains a CLOSED list that holds all expanded nodes with some other information. When memory is in short supply, the CLOSED entries are removed after only a couple of new expansions. It follows that MA* cannot compete with other algorithms on a CPU time basis. (Unfortunately, no CPU time results are given by Chakrabarti et al. [Chakrabarti et al. 1989].)

Both algorithms operate on the same data structure: an explicit search graph. Its construction, maintenance and traversal is a time-consuming task. Our hashing techniques, in contrast, are easier to implement and they operate in unit time while retaining a similar performance.

5 Improved Information Management

The enhancements that exploit the information acquired in the process of iterative-deepening follow two different schemes: node ordering and the 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. No savings are achieved in the shallower iterations.

SORT: The simplest type of node ordering requires neither information from previous iterations nor extra storage space. 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* technique, 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.

While this scheme helps humans in their search for non-optimal solutions, the savings achieved in (optimal) IDA* search rarely compensate for the additional overhead [Powley et al. 1990, p. 54]. 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 TRANS+MOVE), or re-arrange the nodes of a whole search frontier [Powley et al. 1990].

PV: When searching adversary game trees like chess or checkers, each iteration yields w (=width) 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* [Marsland 1987]; 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* [Schaeffer 1989] also 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 tree. 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, but 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 learns from the success in 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 Advances Techniques in Graphs: Avoiding Re-Expansions

Most applications spawn a decision graph (with multiple paths ending in the same position) rather than a tree. Here, memory functions can be employed to avoid

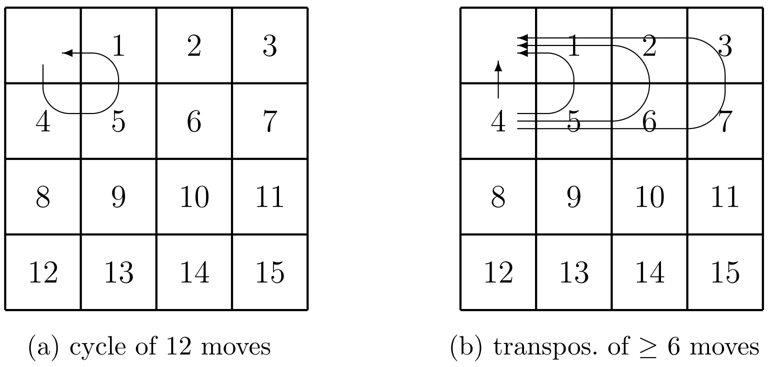

Figure 2: Closed move cycle and transpositions

the re-expansion of previously visited nodes. They usually yield higher profits than the node ordering techniques, because savings are achieved in all iterations. In the following, we distinguish between cycles and transpositions.

A *move cycle* is a sequence of operators, which, after going through some intermediate nodes, finally returns to the starting node. Move cycles are 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, cycle elimination yields modest savings, because closed move cycles occur seldomly, after the trivial 2-move cycle is removed by the move generator. The obvious cycle, shown in Fig. 2a, consists of 12 moves, and the next longer cycle with six tiles has a path length of $g = 30$. Since every cycle contains inferior nodes with high goal distances h, the total expansion cost $g + h$ usually exceeds the cost threshold before completing a cycle. Note that in the traveling salesman problem, all move 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 Fig. 2b. In the 15-puzzle, transpositions occur in search depths $d \geq 6$. They can be traced with a *transposition table* [Zobrist 1970] 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, it can be pruned if the remaining cost bound is less or equal to the corresponding bound given in the table, i.e. when $\texttt{bound} \leq \texttt{bound}_{Trans}$.[3]

Pseudo code in the Appendix illustrates 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. This allows cut offs, even when the

[3]In domains with reversible operators and low branching factors (like the 15-puzzle) subtrees may sometimes be pruned even if $\texttt{bound} > \texttt{bound}_{Trans}$. Research is under way to investigate cases where such "back cutoffs" are possible.

remaining search depth is deeper than that given in the table.

Because of its minimal 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* [Ibaraki 1977], that is, other partial tours covering the same cities in a different order (but with the same last city) are cut off.

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

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 the next position will probably also be contained in the transposition 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 [Korf 1985, p. 106]. To ensure that the hard problems with high node counts do not dominate the results, we computed the mean of the percentage difference relative to Korf's published solutions[4].

[4]Our replication of Korf's experiment identified two cases of differing node counts: In problem #88 our IDA* algorithm examined 6,320,047,980 nodes (vs. 6,009,130,748 in [Korf 1985, p. 107]) and in problem #89 we saved 76 node expansions.

Search algorithm	*Nodes examined [%]*				*CPU-time*
	50 easy problems	*50 hard problems*	*all 100 problems*		*50 easy problems*
			mean	*std*	
IDA*	100.00	100.00	100.00		100
SORT	106.17	92.63	99.40	42.22	114
PV	84.22	87.76	85.99	51.81	85
PV+SORT	86.65	85.09	85.87	58.94	98
HISTORY	93.50	94.11	93.81	47.99	107
TRANS	55.51	50.46	52.98	5.75	82
TRANS+HISTORY	53.49	48.40	50.95	24.64	87
TRANS+MOVE	45.97	45.31	45.64	27.80	64
TRANS+MOVE+HISTORY	46.60	45.04	45.82	31.96	72
IDA*, iter. $1, \ldots, n-1$	52.68	54.82	53.75	26.01	–

Table 1: Empirical results on the 15-puzzle

In all ten different combinations of enhancements were tried, the results from nine of them are presented in Table 1. Thus a total of 1000 computations of Korf's problems were compared, making this the most comprehensive study of the 15-puzzle to date. The results in Table 1 are grouped into a set of fifty easy and fifty hard problem instances. A problem is deemed "harder" if IDA* makes more than 50 million node expansions during its search. In addition, data on the mean performance and the standard deviation on the whole problem set is given. The last column of the table shows the relative CPU time consumption of our implementation.

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. As the table shows, the pre-sorting of successor nodes according to increasing estimate values (SORT) did not pay off – neither in terms of node expansions, nor in terms of CPU time. A mathematical investigation reveals that SORT favors board configurations with the blank square being either in an edge or border position (Fig. 3), because these configurations enjoy (statistically) lower h-values. As an example, chances are $24/40 = 60\%$ that the blank will first be moved to a border position $\{b_1, \ldots, b_8\}$ when it previously was in a center position $\{c_1, \ldots, c_4\}$. Likewise, when the blank is located in a border position, chances are 50% that it will be moved to the adjacent edge and only 33% and 17% that it will be moved

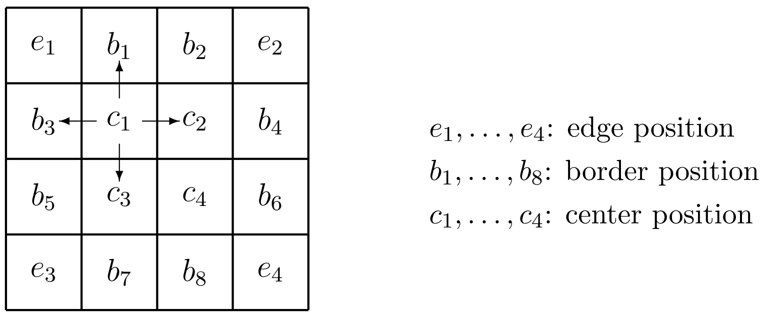
e_1
b_1
b_2
e_2
b_3
c_1
c_2
b_4
b_5
c_3
c_4
b_6
e_3
b_7
b_8
e_4
$e_1, \ldots, e_4$: edge position
$b_1, \ldots, b_8$: border position
$c_1, \ldots, c_4$: center position

Figure 3: Tile positions in the 15-puzzle

to the next boarder or center positions. (These probabilities have been derived by computing, for all possible sliding tiles, the average increase in the Manhattan distance function.)

On one hand, configurations with a blank tile in an outer position have lower mobility and are thus less desirable. But on the other hand, fewer moves are possible in such configurations, which reduces the size of the subtree to be searched. It seems that the positive and negative effects of SORT just compensate for each other, leaving no net gain [Powley et al. 1990, p. 54].

This is no surprise when considering the limited information horizon, the 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. Best 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, but this scheme exhibits 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 speed up the execution time.

In most of the individual problem instances one of either PV or SORT has a low node count. This led us to the assumption, that these two heuristics might be ideal supplements to each other. As can be seen in Table 1, however, the combined version PV+SORT is dominated by the performance of PV, which is executed first. If the principal variation proves to be a bad choice, too many node expansions have already been wasted, so that it is too late for the pre-sorting heuristic to improve the overall performance.

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 [Schaeffer 1989], 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 cut-off, 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. Comparing the fifty easy problems to the hard problems, no signs of table overloading were spotted. On the contrary: The performance of the transposition table seems to increase with growing problem size. This is because, on one hand, there are more transpositions and cycles in deeper search trees, and on the other hand, many more nodes are eliminated by each single cut-off. In practice, the low standard deviation is another favorable aspect of Trans, because one can expect an almost constant efficiency gain of nearly 50% for every problem.

An additional 7% can be saved 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 in total 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 the table gives the average number of node expansions in all iterations excluding the last. This number corresponds to the best performance,

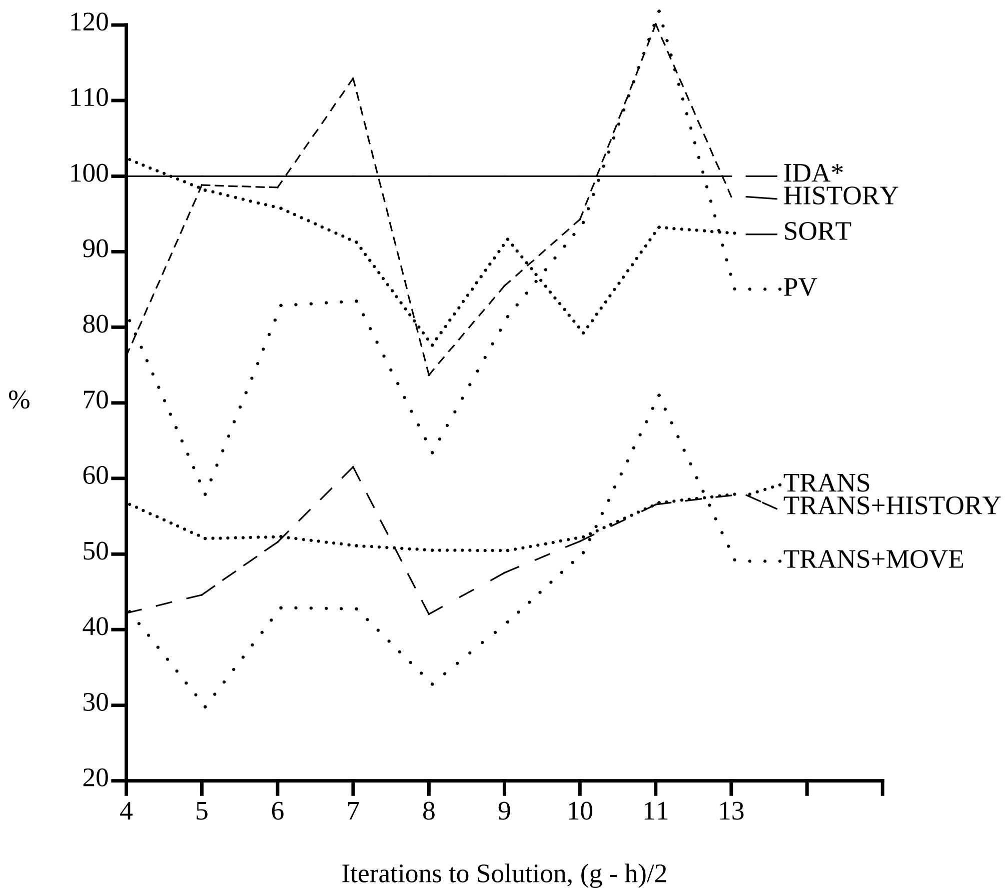

Figure 4: Relative performance of IDA* enhancements

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 Fig. 4 presents the data in a graphical form and shows more clearly how the use of a transposition table is the one mechanism that is consistently effective. In Fig. 4, various enhancements to IDA* are considered, based on an examination of their average performance on increasingly difficult groups of problems. Here difficulty is proportional to the number of iterations required to find a solution[5]. That is, the optimal solution length minus initial Manhattan distance $(g - h)$. All the methods that involve a transposition table are effective, with the combination of cycle detection (Trans) and prior op-

[5]Interestingly, none of Korf's 100 random problems were solvable in 24 iterations.

erator choice (MOVE) being especially good across all problems. Even in the worst average cases a reduction to 70% of IDA* is achieved, with best average reduction being 45%. Not all the available data is plotted here. In particular, on the $g - h$ basis, there is insignificant difference between PV and PV+SORT performance, and also between TRANS+MOVE and TRANS+MOVE+HISTORY, and so only one plot of each is shown.

6.2 The Traveling Salesman Problem

On 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.

While the *brute-force branching factor* [Korf 1988], which is defined as the average ratio of the number of nodes at a given depth to the number of nodes at the next shallower depth, is indeed higher in the TSP, the total node count does not increase as fast when entering a new iteration. In other words, the TSP obeys a lower *heuristic branching factor* than the 15-puzzle.

The explanation for this phenomenon is, that in the 15-puzzle, an increase of the cost bound by 2 (which is the only possible increase between iterations) allows *all* nodes of a search frontier to be expanded by at least one extra level – and sometimes by much more. In the TSP, on the other hand, the smallest of the cost values, that exceed the current threshold may affect only few frontier nodes to be further extended in the next iteration. All other nodes are cut off at the same depth as before.

The exact magnitude of the new node expansions depends on the domain of the inter-city distances. In the extreme case, that is with real valued inter-city distances, only one frontier node gets expanded in every new iteration. Clearly, iterative-deepening is then of no use in this case. The problem might be overcome by increasing the cost bound by more than the minimum value that exceeded the previous bound [Korf 1988, p. 240]. But this approach could result in sub-optimal solution paths and will therefore not be discussed here.

Table 2 shows the relative performance of the algorithms on fifty randomly generated 20-city problems with inter-city domains of 50, 75 and 100 different integer values. As expected, neither of the node ordering heuristics (SORT, PV or HISTORY) yields substantial performance improvements. This is not surprising,

Search algorithm	*50 diff. values*		*75 diff. values*		*100 diff. values*	
	nodes	*time*	*nodes*	*time*	*nodes*	*time*
IDA*	100.00	100	100.00	100	100.00	100
SORT	95.55	96	97.11	97	99.14	99
PV	96.03	96	96.54	97	98.84	99
PV+SORT	94.93	95	96.29	96	98.32	99
HISTORY	95.22	95	96.77	97	99.37	99
TRANS	35.72	38	32.17	34	26.81	28
TRANS+MOVE	36.04	37	32.28	33	26.96	28
IDA*, it. $1, \ldots, n-1$	92.08	–	94.35	–	96.53	–

Table 2: Empirical results on the TSP (20 cities)

since only 8, 6 and 4% (resp.) of the total nodes are examined in the last iteration, which gives an upper bound on the maximal improvement that can be achieved (see last line in the table).

The HISTORY results are based on a two dimensional history table that holds for every city pair the frequency it contributed to the longest tour. Experiments with chains of three cities gave only marginal additional improvements while occupying more resources (a 3-dimensional array).

As in the 15-puzzle, the best results are achieved with a transposition table. Here the table entries have been used to pre-sort all successors before expanding further. We found this the best usage of the table, since a table access is faster than the computation of the minimum spanning tree (which is used as an estimate for the remaining path cost).

In total, TRANS examines only 36, 32 and 27% of the nodes that are visited by IDA*. This is more than expected, because the single table entries are less valuable in the TSP than in the 15-puzzle. While in the 15-puzzle, cut offs occur as soon as the retrieved cost bound exceeds the current bound, in the TSP, care must be taken not to prune subtrees containing a new cost bound for the next iteration. Here, a subtree can only be pruned when the remaining cost bound is higher than the current `new_bound`. (The problem does not occur in the 15-puzzle, because the cost bound is always incremented by 2.)

Despite this disadvantage, the transposition table is more useful in the TSP than in the 15-puzzle. Not only does it reduce the node count by almost three quarters, but it also speeds up the execution time by the same amount. This is because a hash table access is much faster than the computation of the minimum spanning tree, so that losses during unsuccessful retrievals are easily paid off.

7 Conclusions

In this paper, we adapted techniques known 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 successor ordering information, reduces the size of the search graph by one half (in the 15-puzzle) or even by about three quarters (in the TSP). This is possible because the saved information can be used to detect duplicate states. The savings are greater than is possible with a perfectly informed (and hence non-deterministic) IDA* algorithm, which finds the solution node at the first node expansion of the last iteration.

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 15, 18 and 36%, respectively. This compares favorably to the results given in [Sen & Bagchi 1989, table 2].

In practice, one would first include the PV-heuristic, because it needs no extra resources in terms of space or time. 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 runs in unit time, it does not affect the time complexity of the program.

This is especially true for applications with non-neglectable operator costs, like the traveling salesman problem. Depending on the range of inter-city distance values, a transposition table of 256k 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. While with tree structured data types, a whole path must be sent to identify a single node, hashing techniques need only one hash key (that usually consists of one word only) to be transferred. 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 [Reinefeld et al. 1985a], hashing tables are much easier to implement and debug than the tree-structured data types of A* [Hart et al. 1968] and other IDA* variants [Chakrabarti et al. 1989, Sen & Bagchi 1989]. 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.

This is especially true for trees with some irreversible operators (like chess). If all the operators are reversible (like in the 15-puzzle), some of the branches can be pruned even when the remaining search depth is greater than the depth of the retrieved node information. Research is under way to investigate possible further gains through operator reversal.

Acknowledgments

Foundations to this work were laid during the first author's stay at the University of Alberta in the years 1987/88. Jonathan Schaeffer's explanation of the history heuristic and his general support is greatly appreciated. Also, Sam Farrage and Randal Kornelson helped in re-running some of the 15-puzzle experiments to obtain more accurate timings and in developing some scripts for post-processing the node-count data.

Without the Killam Post Doctoral Fellowship awarded to the first author, and financial support of the National Sciences and Research Council under Grants OPG36952 and OPG07902, this work would not have been possible.

References

[Akl & Newborn 1977] S.G. Akl, M.M. Newborn. *The principal continuation and the killer heuristic.* Procs. ACM Nat. Conf., Seattle (1977), 466-473.

[Chakrabarti et al. 1989] P.P. Chakrabarti, S. Ghose, A. Acharya, S.C. de Sarkar. *Heuristic search in restricted memory.* Artificial Intelligence 41 (1989/90), 197-221.

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

[Held & Karp 1979] M. Held, R.M. Karp. *The traveling salesman problem and minimal spanning trees.* Operations Research 18(1979), 1138-1162.

[Ibaraki 1977] T. Ibaraki. *The power of dominance relations in branch-and-bound algorithms.* Jour. of the ACM 24,2(Apr. 1977), 264-279.

[Ibaraki 1978] T. Ibaraki. *Depth-m search in branch-and-bound algorithms.* Intl. Jour. of Comp. and Inf. Sc., 7,4(1978), 315-343.

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

[Korf 1988] R.E. Korf. *Optimal path-finding algorithms.* In: L. Kanal, V. Kumar (eds.), Search in Artificial Intelligence, Springer-Verlag, New York (1988).

[Little et al. 1963] J.D.C. Little, K.G. Murty, D.W. Sweeney, G. Karel. *An algorithm for the traveling salesman problem.* Operations Research 11(1963), 972-989.

[Marsland 1987] T.A. Marsland. *Computer chess methods.* In E. Shapiro (ed.), Encyclopedia of Artificial Intelligence, Wiley (1987), 159-171.

[Marsland et al. 1987] T.A. Marsland, A. Reinefeld, J. Schaeffer. *Low overhead alternatives to SSS*.* Artificial Intelligence 31(1987), 185-199.

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

[Powley et al. 1990] C. Powley, Ch. Ferguson, R.E. Korf. *Parallel heuristic search: Two approaches.* In V. Kumar, P.S. Gopalakrishnan, L.N. Kanal (eds.), Parallel Algorithms for Machine Intelligence and Vision, Springer-Verlag, New York (1990), 42-65.

[Reinefeld et al. 1985a] A. Reinefeld, J. Schaeffer, T.A. Marsland. *Information acquisition in minimal window search.* 9th IJCAI Conf. Procs. (1985), 1040-1043.

[Reinefeld et al. 1985b] A. Reinefeld, T.A. Marsland, J. Schaeffer. *Is best first search really best?* Tech. Rep. TR85-16, Dept. of Comp. Science, University of Alberta, Edmonton, Canada (1985).

[Schaeffer 1989] J. Schaeffer. *The history heuristic and alpha-beta search enhancements in practice.* IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-11,11(Nov. 1989), 1203-1212.

[Scott 1969] J.J. Scott. *A chess-playing program.* B. Melzer, D. Michie (eds.), Machine Intelligence 4, Edinburgh Univ. Press (1969), 255-265.

[Sen & Bagchi 1989] A.K. Sen, A. Bagchi. *Fast recursive formulation for best-first search that allow controlled use of memory.* 11th IJCAI Conf. Procs. (1989), 297-302.

[Slate & Atkin 1977] D.J. Slate, L.R. Atkin. *Chess 4.5 — The Northwestern University chess program.* In P.W. Frey (ed.), Chess Skill in Man and Machine, Springer-Verlag, New York (1977), 82-118.

[Stickel & Tyson 1985] M.E. Stickel, W.M. Tyson. *An analysis of consecutively bounded depth-first search with applications in automated deduction.* 9th IJCAI Conf. Procs. (1985), 1073-75.

[Stockman 1979] G.C. Stockman. *A minimax algorithm better than alpha-beta?* Artificial Intelligence 12,2(1979), 179-196.

[Zobrist 1970] A.L. Zobrist. *A new hashing method with applications for game playing.* Techn. Report 88, Comp. Sciences Dept., Univ. of Wisconsin, Madison (Apr. 1970). Reprinted in the ICCA Journal 13,2(1990), 69-73.

Appendix

```
FUNCTION depth_first_search (n: NODE; depth, bound: INTEGER): INTEGER;
                               { returns cost bound for next iteration }
VAR new_bound, tt_bound, next, i, t: INTEGER;
    succ: ARRAY [1..MAX_WIDTH] OF NODE;              { successor nodes }
    b: ARRAY [1..MAX_WIDTH] OF INTEGER;          { cost bounds of succ }

BEGIN
   IF h(n) = 0 THEN BEGIN
      solved := true; RETURN (0);               { return solution cost }
   END;
   new_bound := INF;

   FOR each successor n.i of n DO BEGIN
      succ[i] := n.i;
      IF retrieve_tt (n.i, tt_bound) THEN      { if n.i in trans-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[] according increasing bounds b[] }

   FOR i := 1 TO number of successors DO BEGIN               { recurse }
      next := succ[i];
      IF b[next] <= bound THEN
         t := c(n,next)
              + depth_first_search (next, depth + 1, bound - c(n,next));
      ELSE
         t := c(n,next) + h(next);                            { cutoff }
      IF solved THEN BEGIN
         path[depth] := next; RETURN (t); END;                  { done }
      IF t < new_bound THEN
         new_bound := t;         { determine bound for next iteration }
   END;
   save_tt (n, new_bound);  { save lowest bound in transposition table }
   RETURN (new_bound);           { return next iteration's cost bound }
END;
```

Figure 5: IDA* with a transposition table and cost revision.
The function `depth_first_search` is called by the `iterative_deepening` routine, see Fig. 1.