Finding Optimal Solutions to Rubik's Cube Using Pattern Databases

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90095
korf@cs.ucla.edu

Abstract

We have found the first optimal solutions to random instances of Rubik's Cube. The median optimal solution length appears to be 18 moves. The algorithm used is iterative-deepening-A* (IDA*), with a lower-bound heuristic function based on large memory-based lookup tables, or "pattern databases" (Culberson and Schaeffer 1996). These tables store the exact number of moves required to solve various subgoals of the problem, in this case subsets of the individual movable cubies. We characterize the effectiveness of an admissible heuristic function by its expected value, and hypothesize that the overall performance of the program obeys a relation in which the product of the time and space used equals the size of the state space. Thus, the speed of the program increases linearly with the amount of memory available. As computer memories become larger and cheaper, we believe that this approach will become increasingly cost-effective.

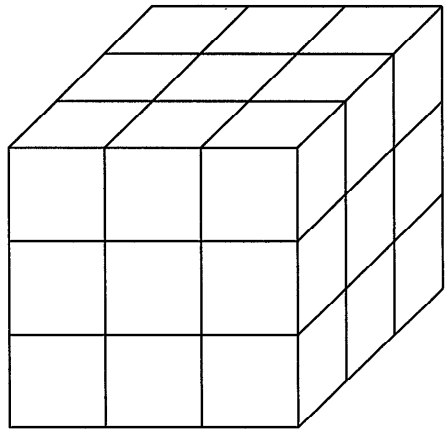

Figure 1: Rubik's Cube

Introduction

Rubik's Cube, invented in the late 1970s by Erno Rubik of Hungary, is the most famous combinatorial puzzle of its time. The standard version (See Figure 1) consists of a $3 \times 3 \times 3$ cube, with different colored stickers on each of the exposed squares of the subcubes, or *cubies*. Any $3 \times 3 \times 1$ plane of the cube can be rotated or twisted 90, 180, or 270 degrees relative to the rest of the cube. In the goal state, all the squares on each side of the cube are the same color. The puzzle is scrambled by making a number of random twists, and the task is to restore the cube to its original goal state.

The problem is quite difficult. The Saganesque slogan printed on the package, that there are billions of combinations, is a considerable understatement. In fact, there are 4.3252×10^{19} different states that can be reached from any given configuration. By comparison, the 4×4 Fifteen Puzzle contains 10^{13} states, and the 5×5 Twenty-Four Puzzle generates 10^{25} states.

To solve Rubik's Cube, one needs a general strategy, which usually consists of a set of move sequences, or macro-operators, that correctly position individual cubies without violating previously positioned ones. Such strategies typically require 50 to 100 moves to solve a randomly scrambled cube. It is believed, however, that any cube can be solved in no more than 20 moves.

We address the problem of finding a shortest sequence of moves required to solve a given problem instance. As far as we have been able to determine, optimal solutions to random instances have not been found previously. We generated ten random problem instances, and solved them all optimally. One was solved in 16 moves, three required 17 moves, and the remaining six were solved in 18 moves.

The Problem Space

The first step is to translate the physical puzzle into a symbolic problem space to be manipulated by a computer. Of the 27 possible $1 \times 1 \times 1$ cubies, 26 are visible, with one in the center. Of these, the six cubies in the center of a face rotate, but don't move. These six cubies form a fixed reference framework, disallowing rotations of the entire cube. Of the remaining 20 movable cubies, 8 are on the corners, with three visible faces each, and 12 are on the edges, with two visible faces

Copyright ©1997, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

each. Corner cubies only move among corner positions, and edge cubies only move among edge positions. A corner cubie in a given position can be oriented in any of three ways, and an edge cubie can be oriented two different ways. Thus, we can uniquely specify the state of a cube by giving the position and orientation of each of the 8 edge cubies and 12 corner cubies. This is represented as an array of 20 elements, one for each cubie, the contents of which encode the position and orientation of the cubie as one of 24 different values, $8 \cdot 3$ for the corners, and $12 \cdot 2$ for the edges. The total number of possibilities is thus $8! \cdot 3^8 \cdot 12! \cdot 2^{12}$. Additional constraints reduce this by a factor of 12, since the entire problem space consists of 12 separate but isomorphic subgraphs, with no legal moves between them. Thus, the total number of states reachable from a given state is $8! \cdot 3^8 \cdot 12! \cdot 2^{12}/12 = 43,252,003,274,489,856,000$.

The next question is how to define the primitive operators, for which there are two approaches. The first is that a primitive operator is any 90 degree, or quarter turn, of a face. A 180 degree turn is represented by two quarter turns in the same direction. Alternatively, a primitive operator may be any twist of a single face, either 90 degrees clockwise or counterclockwise, or 180 degrees. We adopt this latter formulation since it leads to a search tree with fewer duplicate nodes. For example, two consecutive clockwise twists of the same face leads to the same state as two counterclockwise twists.

Counting any of the three different twists of the six different faces as a primitive operator leads to a search tree with a branching factor of 18. Since twisting the same face twice in a row is redundant, ruling out such moves reduces the branching factor to 15 after the first move. Furthermore, twists of opposite faces of the cube are independent and commutative. For example, twisting the front face, then twisting the back face, leads to the same state as performing the same twists in the opposite order. Thus, for each pair of opposite faces we arbitrarily chose an order, and forbid moves that twist the two faces consecutively in the opposite order. This results in a search tree with an asymptotic branching factor of about 13.34847. Table 1 shows the number of nodes in the search tree as a function of depth, up to depth 18. All these nodes do not correspond to unique states of the cube, since there are duplicate paths to the same node. At depth 18, the number of nodes in the tree exceeds the number of states in the problem space for the first time. This implies that some states are at least 18 moves from the goal, but doesn't guarantee that no states are further away.

Depth	Nodes
1	18
2	243
3	3,240
4	43,254
5	577,368
6	7,706,988
7	102,876,480
8	1,373,243,544
9	18,330,699,168
10	244,686,773,808
11	3,266,193,870,720
12	43,598,688,377,184
13	581,975,750,199,168
14	7,768,485,393,179,328
15	103,697,388,221,736,960
16	1,384,201,395,738,071,424
17	18,476,969,736,848,122,368
18	246,639,261,965,462,754,048

Table 1: Nodes in search tree as a function of depth

A First Attempt

To find optimal solutions, we need an admissible search algorithm. Exponential-space algorithms like A* are impractical on large problems. Since it is difficult to find any solution, or a tight upper bound on the optimal solution length, depth-first branch-and-bound is also not feasible. This suggests iterative-deepening-A* (IDA*) (Korf 1985a). IDA* is a depth-first search that looks for increasingly longer solutions in a series of iterations, using a lower-bound heuristic to prune branches once their estimated length exceeds the current iteration bound. Given the branching factor of this space, the time overhead of the non-goal iterations is only 8%.

Next we need a heuristic function. The obvious heuristic is a three-dimensional version of the well-known manhattan distance for sliding-tile puzzles. For each cubie, compute the minimum number of moves required to correctly position and orient it, and sum these values over all cubies. Unfortunately, to be admissible, this value has to be divided by eight, since every twist moves four corner and four edge cubies. This result can be rounded up. A better heuristic is to take the maximum of the sum of the manhattan distances of the corner cubies, and the edge cubies, each divided by four. The expected value of the manhattan distance of the edge cubies is 5.5, while the expected value of the manhattan distance of the corner cubies is only about 3, partly because there are 12 edge cubies, but only 8 corner cubies. As a result, if we only compute the manhattan distance of the edge cubies, ignoring the corner cubies, the additional node generations are made up for by the lower cost per node.

The expected value of 5.5 for this heuristic gives an estimate of its effectiveness, as we will see later. IDA* running on our Sun Ultra-Sparc Model 1 workstation with this heuristic can search to depth 14 in about three days, but solving problem instances at depth 18 would take over 250 years. We need a better heuristic.

Pattern Databases

While we normally think of a heuristic as a function computed by an algorithm, any function can also be

computed by a table lookup, given sufficient memory. In fact, for reasons of efficiency, heuristic functions are commonly precomputed and stored in memory. For example, the manhattan distance function above is computed with the aid of a small table that contains the manhattan distance of each individual cubie from all possible positions and orientations. (Culberson and Schaeffer 1996) have carried this idea much further.

If we consider just the eight corner cubies, the position and orientation of the last cubie is determined by the remaining seven, so there are exactly $8! \cdot 3^7 = 88,179,840$ possible combinations. Using a breadth-first search from the goal state, we can enumerate these states, and record in a table the number of moves required to solve each combination of corner cubies. Since this value ranges from zero to eleven, only four bits are required for each table entry. Thus, this table requires 44,089,920 bytes of memory, or 42 megabytes, which is easily accommodated on modern workstations. Note that this table only needs to be computed once for each goal state, and its cost can be amortized over the solution of multiple problem instances with the same goal. The use of such tables, called "pattern databases" is due to (Culberson and Schaeffer 1996), who applied it to the Fifteen Puzzle.

The expected value of this heuristic is about 8.764 moves, compared to 5.5 moves for the manhattan distance of the edge cubies. During the IDA* search, as each state is generated, a unique index into the heuristic table is computed, followed by a reference to the table. The stored value is the number of moves needed to solve the corner cubies, and thus a lower bound on the number of moves needed to solve the entire puzzle.

We can improve this heuristic by considering the edge cubies as well. The number of possible combinations for six of the twelve edge cubies is $12!/6! \cdot 2^6 = 42,577,920$. The number of moves needed to solve them ranges from zero to ten, with an expected value of about 7.668 moves. At four bits per entry, this table requires 21,288,960 bytes, or 20 megabytes. A table for seven edge cubies would require 244 megabytes of memory. The only admissible way to combine the corner and edge heuristics is to take their maximum.

Similarly, we can compute the corresponding heuristic table for the remaining six edge cubies. The heuristic used for the experiments reported below is the maximum of all three of these values: all eight corner cubies, and two groups of six edge cubies each. The total amount of memory for all three tables is 82 megabytes. The total time to generate all three heuristic tables was about an hour, and this cost is amortized over multiple problem instances with the same goal state. The expected value of the maximum of these three heuristics is 8.878 moves. Even though this is only a small increase above the 8.764 expected value for just the corner cubies, it results in a significant performance improvement. Given more memory, we could compute and store even larger tables.

Experimental Results

We generated ten solvable instances of Rubik's Cube, by making 100 random moves each, starting from the goal state. Given the high dimensionality of the problem space based on the fact that each state has 18 neighbors, and the conjecture that the diameter of the space is only 20 moves, we believe that 100-move random walks generate effectively random problem instances. We then ran IDA* using the above heuristic, solving all the problems optimally. The results are summarized in Table 2, sorted by problem difficulty. One problem was solved in 16 moves, three required 17 moves, and six were solved in 18 moves. The actual initial states and their solutions are available from the author on request.

No.	Depth	Nodes Generated
1	16	3,720,885,493
2	17	11,485,155,726
3	17	64,837,508,623
4	17	126,005,368,381
5	18	262,228,269,081
6	18	344,770,394,346
7	18	502,417,601,953
8	18	562,494,969,937
9	18	626,785,460,346
10	18	1,021,814,815,051

Table 2: Optimal solution lengths and nodes generated for random Rubik's Cube problem instances

Running on a Sun Ultra-Sparc Model 1 workstation, our program generates and evaluates about 700,000 nodes per second. At large depths, the number of nodes generated per iteration is very stable across different problem instances. For example, of the six completed iterations at depth 17, the number of nodes generated ranged only from 116 to 127 billion. Complete searches to depth 16 require an average of 9.5 billion nodes, and take less than four hours. Complete searches to depth 17 generate an average of 122 billion nodes, and take about two days. A complete depth 18 search should take less than four weeks. The heuristic branching factor, which is the ratio of the number of nodes generated in one iteration to the number generated in the previous iteration, is roughly the brute-force branching factor of 13.34847.

Performance Analysis

The consistent number of nodes generated by IDA* on a given iteration across different problem instances suggests that the performance of the algorithm is amenable to analysis. Most analyses of single-agent heuristic search algorithms have been done on simple analytic models, and have not predicted performance on any real problems. The discussion below is not a formal analysis, but rather a set of observations of regularities in our data, supported by intuitive arguments.

Most analyses of heuristic evaluation functions relate the performance of a search algorithm to the accuracy of the heuristic as an estimate of the exact distance to the goal. One difficulty with this approach is that heuristic accuracy is hard to measure, since determining the exact distance to the goal for a given problem instance is computationally difficult for large problems.

Our first observation is that we can characterize the effectiveness of an admissible heuristic function by its expected value over the problem space. This is easily determined to any desired degree of accuracy by randomly sampling the problem space, and computing the heuristic for each state. In our case, the heuristic values are enumerated in the heuristic table, and their expected value can be computed exactly as the average of the table values. Furthermore, the expected value of the maximum of two such heuristics can be computed exactly from their heuristic tables, assuming that the values are independent, a reasonable assumption here. We conjecture that the larger the expected value of an admissible heuristic, the better the performance of an admissible search algorithm using it.

If the heuristic value of every state equaled its expected value e, then IDA* searching to depth d would be equivalent to brute-force depth-first iterative deepening searching to depth $d - e$, since the $f = g + h$ value of every state would be its depth plus e. Thus it is tempting to conjecture that the number of nodes generated by IDA* searching to depth d is approximately the same as a brute-force search to depth $d - e$. In our experiments, our heuristic has an expected value of 8.878, which we round up to 9. A complete IDA* search to depth 17 generates approximately 122 billion nodes. However, a brute-force search to depth $17-9 = 8$ would generate only about 1.37 billion nodes, which is almost two orders of magnitude less.

The reason for this discrepancy is that the states encountered in an IDA* search are not a random sample of the problem space. States with large heuristic values are pruned, and states with small heuristic values spawn more children in the same iteration. Thus, the search is biased in favor of small heuristic values, which lowers the average heuristic values of states encountered by IDA*, causing more nodes to be generated than predicted by the expected value of the heuristic.

The next step is to predict the expected value of a pattern database heuristic from the amount of memory used, and the branching factor b of the problem space. To see how to do this, note that Table 1 implies that most Rubik's Cube problem instances must be at least 18 moves from the goal state, since this is the depth at which the number of nodes in the tree first exceeds the number of states in the space, which agrees with our experimental results. In general, a lower bound on the expected value of a pattern database heuristic is the log base b of the number of states stored in the table, since with d moves, one can generate at most b^d states. In general, one will generate fewer unique states since there are duplicate nodes in the search tree.

As another example, since there are about 88 million different states of the corner cubies, Table 1 suggests that the average number of moves required to solve them should be about seven. In fact, the average number of moves is 8.764. Again, the reason for the discrepancy is that not all the nodes at depth seven in the tree correspond to unique states of the corner cubies, requiring us to go deeper to generate them all.

In other words, estimating the expected value of a heuristic from the branching factor of the space and the number of states gives a value that is too low or pessimistic. On the other hand, estimating the number of nodes generated by IDA* from the expected value of the heuristic gives a value that is also too low, which is optimistic. This suggests that by combining the two, to estimate the number of nodes generated by IDA* from the branching factor of the space and the number of states in the heuristic table, the two errors may cancel each other to some extent.

More formally, let n be the number of states in the entire problem space, let b be the brute-force branching factor of the space, let d be the average optimal solution length for a random problem instance, let e be the expected value of the heuristic, let m be the amount of memory used, in terms of heuristic values stored, and let t be the running time of IDA*, in terms of nodes generated. The average optimal solution length d of a random instance, which is the depth to which IDA* must search, can be estimated as $\log_b n$, or $d \approx \log_b n$. As argued above, $e \approx \log_b m$, and $t \approx b^{d-e}$. Substituting the values for d and e into this formula gives

$$t \approx b^{d-e} \approx b^{\log_b n - \log_b m} = n/m.$$

Thus, the running time of IDA* may be approximated by $O(n/m)$, the size of the problem space divided by the memory available. Using the data from our experiments, $n \approx 4.3252 \cdot 10^{19}, m = 173,335,680, n/m = 249,527,409,904$, and $t = 352,656,042,894$, which is only off by a factor of 1.4.

Given this hypothesized linear relationship between the available memory and the speed of IDA*, an obvious thing to try is to store a larger heuristic table on disk. Unfortunately, the long latency of disks makes this approach impractical. The access pattern for the heuristic tables doesn't exhibit any locality of reference, and we can expect that each access will take as long as the latency time of the disk. If we have a single disk with a ten millisecond latency, this gives only 100 accesses per second. Even assuming only one heuristic calculation per node, this yields a speed of only 100 nodes per second. Our current implementation, which stores the heuristic tables in memory, runs at about 700,000 nodes per second, a factor of 7000 times faster. Given the formula $t \approx n/m$, the disk capacity would have to be about 7000 times that of main memory to make up for the latency of the disk accesses. Currently, this ratio is only about a factor of ten.

Related Work

Most of the ideas in this paper have appeared in the literature. Our main contribution is to bring them together to solve a new problem for the first time.

The first appearance of Rubik's Cube in the published AI literature was (Korf 1985b). The focus of that work was on learning to solve the cube, and the solutions generated by that program averaged 86 moves.

Schroeppel, Shamir, Fiat et al

The first paper to address finding optimal solutions to Rubik's Cube was (Fiat et al. 1989), which is based on (Schroeppel and Shamir 1981). To understand this work, we first discuss the idea of bidirectional search.

Bidirectional search searches forward from the initial state, and backwards from the goal state simultaneously, until a common state is reached from both directions. Then the forward search path, combined with the reverse of the backward search path, is a solution to the problem. Given a problem with branching factor b and optimal solution depth d, the time complexity of bidirectional search is $O(b^{d/2})$, since the two searches need only proceed to half the depth. The drawback, however, is that at least one of the search frontiers must be stored, requiring $O(b^{d/2})$ space as well. For Rubik's cube, this would require storing all the states at depth nine for an 18-move solution, which isn't feasible since there are over 18 billion states at depth nine. Even if we could store this many states on disk, the latency of the disk would make the algorithm impractical in terms of time, for the reasons given above.

(Schroeppel and Shamir 1981) improved on bidirectional search for many problems. If the states in the two search frontiers can be generated in some order defined on the state description, then a match between the two frontiers can be found without storing either of the frontiers. We simply generate the two frontiers synchronously, keeping the current frontier nodes as close as possible to each other in the order. To find a solution of depth d, we generate and store in memory all states at depth $d/4$ from both initial and goal states. Then, by combining all possible pairs of paths of length $d/4$, all states at depth $d/2$ are generated, in order, from both initial and goal states, and a match is found if one exists. The space complexity of this method is $O(b^{d/4})$, and the time complexity is $O(b^{d/2})$.

(Fiat et al. 1989) showed how to apply this technique to permutation groups, such as Rubik's Cube. Their main contribution was to show how to generate compositions of permutations in lexicographically sorted order. They claimed that their algorithm was the first that could feasibly find optimal solutions to Rubik's Cube. They only reported results from their implementation up to depth 16, however, and speculated about what they could do with a depth 18 search.

(Bright, Kasif, and Stiller 1994) showed how to parallelize the Schroeppel and Shamir, and Fiat el al. algorithms, and give a clearer exposition of both.

We implemented the Fiat algorithm, and its speed is roughly comparable to our implementation of IDA* with pattern database heuristics, using 82 megabytes of memory, on depth 18 searches. Currently, our Fiat code is about a factor of two slower than our IDA* code, but we believe that we could narrow this gap with further optimization of our Fiat code.

If n is the size of the problem space, the Fiat algorithm has an asymptotic time complexity of $O(n^{1/2})$, and an asymptotic space complexity of $O(n^{1/4})$. This is superior to our hypothesized space-time tradeoff of $t \approx n/m$ for IDA*. There are two caveats to this, however. First, the constant time factor for our Fiat implementation is about 70 times what it is for our IDA* implementation. Our Fiat code generates about 10,000 permutations per second, while our IDA* code generates about 700,000 per second. Secondly, our Fiat code requires over 500 bytes per permutation, while our heuristic table requires only half a byte per permutation, a factor of 1000 in space. As a result, depth 19 searches with the Fiat algorithm would require well over 300 megabytes of memory, but shouldn't run any faster than our IDA* implementation at that depth, using only 82 megabytes.

While the Fiat algorithm allows less memory to be used at the cost of increased time, it doesn't run any faster with more memory, as does our approach. Given additional memory, we can compute and store larger pattern databases, resulting in larger heuristic values, which would speed up the IDA* search. Given the hypothesized relation $t \approx n/m$, one could argue that in order to reduce the time below $O(n^{1/2})$, would require more than $O(n^{1/2})$ space, at which point the time to build the heuristic tables would become the dominant time cost. However, the heuristic tables only need to be built once for a given goal state, and hence this time cost can be amortized over the solution of multiple problem instances with a common goal state. Furthermore, note that purely asymptotic comparisons ignore the differences of several orders of magnitude in the constant factors, both in time and space.

Since the running time of IDA* with pattern databases decreases with increasing memory, we believe that as memories get larger, it will continue to outperform the Fiat algorithm on this problem. In addition, IDA* is much simpler and easier to implement, and we believe more general as well. Even though the Fiat algorithm has been around since 1987, and widely discussed in an electronic mailing list devoted to Rubik's Cube, we could find only one other reported implementation of it (Moews 1995), and it has not been used to find optimal solutions to random problem instances, as far as we know.

Prieditis, Culberson, Schaeffer

The idea of using the optimal solution to the corner cubies as a heuristic for Rubik's Cube was first described by (Prieditis 1993). He also precomputed this

value up to depth six. Although he provides no detailed performance results, and doesn't report solving any problem instances, he claims that IDA* using this heuristic results in eight orders of magnitude speedup over brute-force search. However, this appears to be on a much less efficient problem space, which includes rotations of the entire cube as operators.

The idea of using large tables of precomputed optimal solution lengths for subparts of a combinatorial problem was first proposed by (Culberson and Schaeffer 1996). They studied these pattern databases in the context of the 15-puzzle, achieving significant performance improvements over simple heuristics like manhattan distance. This paper can be viewed as simply applying their idea to Rubik's Cube.

Large tables of heuristic values originated in the area of two-player games, where such tables are used to store the exact value (win, lose, or draw) of endgame positions. This technique has been used to great effect by (Schaeffer et al. 1992) in the game of checkers.

Finally, ever since the rise of linear-space heuristic search algorithms, there has been an ongoing debate about how to speed up such algorithms by using the large memories available on current machines. One of the best options is known as perimeter search (Dillenburg and Nelson 1994) (Manzini 1995) (Kaindl et al. 1995), and is related to bidirectional search described above. In perimeter search, a breadth-first search is performed backwards from the goal state until memory is nearly exhausted, and the states on the resulting perimeter around the goal state are stored in memory. Then, a linear-space forward search is performed, until a state on the perimeter is reached. The real value of this technique is that the states on the perimeter can be used to improve the heuristic values in the forward search. This paper provides an alternative use of memory in heuristic search, and it remains to be seen if methods such as perimeter search can perform as well on problems such as Rubik's Cube.

Conclusions

As far as we have been able to determine, we have found the first optimal solutions to random instances of Rubik's Cube, one of the most famous combinatorial puzzles of its time. The median optimal solution length appears to be 18 moves. The key idea, due to (Culberson and Schaeffer 1996), is to take a subset of the goals of the original problem, and precompute and store the exact number of moves needed to solve these subgoals from all possible initial states. Then the exact solution to the subgoals is used as a lower bound heuristic for an IDA* search of the original problem. We characterize the effectiveness of an admissible heuristic simply by its expected value. We also present an informal analysis of the technique, and hypothesize that its performance is governed by the approximation $t \approx n/m$, where t is the running time, m is the amount of memory used, and n is the size of the problem space. This approximation is consistent with the results of our experiments, and suggests that the speed of the algorithm increases linearly with the amount of memory available. As computer memories become larger and cheaper, we expect that this algorithm will become increasingly cost-effective.

Acknowledgements

This work was supported by NSF Grant IRI-9119825.

References

Bright, J., S. Kasif, and L. Stiller, Exploiting algebraic structure in parallel state space search, *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, July 1994, pp. 1341-1346.

Culberson, J.C., and J. Schaeffer, Searching with pattern databases, *Proceedings of the 11th Conference of the Canadian Society for the Computational Study of Intelligence*, published in *Advances in Artificial Intelligence*, Gordon McCalla (Ed.), Springer Verlag, 1996.

Dillenburg, J.F., and P.C. Nelson, Perimeter search, *Artificial Intelligence*, Vol. 65, No. 1, Jan. 1994, pp. 165-178.

Fiat, A., S. Moses, A. Shamir, I. Shimshoni, and G. Tardos, Planning and learning in permutation groups, *Proceedings of the 30th A.C.M. Foundations of Computer Science Conference (FOCS)*, 1989, pp. 274-279.

Kaindl, H., G. Kainz, A. Leeb, and H. Smetana, How to use limited memory in heuristic search, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, Aug. 1995, pp. 236-242.

Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

Korf, R.E., Macro-operators: A weak method for learning, *Artificial Intelligence*, Vol. 26, No. 1, pp. 35-77, 1985.

Manzini, G., BIDA*: An improved perimeter search algorithm, *Artificial Intelligence*, Vol. 75, No. 2, June 1995, pp. 347-360.

Moews, D., Shamir's method on the super-flip, posting to Cube-Lovers mailing list, Jan. 23, 1995.

Prieditis, A.E., Machine discovery of effective admissible heuristics, *Machine Learning*, Vol. 12, 1993, pp. 117-141.

Schaeffer, J., J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, A world championship caliber checkers program, *Artificial Intelligence*, Vol. 53, No. 2-3, 1992, pp. 273-290.

Schroeppel, R., and A. Shamir, A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-Complete Problems, *SIAM Journal of Computing*, Vol. 10, No. 3, Aug. 1981, pp. 456-464.