Finding Optimal Solutions to the Twenty-Four Puzzle

Richard E. Korf and **Larry A. Taylor**
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
korf@cs.ucla.edu, ltaylor@cs.ucla.edu

Abstract

We have found the first optimal solutions to random instances of the Twenty-Four Puzzle, the 5×5 version of the well-known sliding-tile puzzles. Our new contribution to this problem is a more powerful admissible heuristic function. We present a general theory for the automatic discovery of such heuristics, which is based on considering multiple subgoals simultaneously. In addition, we apply a technique for pruning duplicate nodes in depth-first search using a finite-state machine. Finally, we observe that as heuristic search problems are scaled up, more powerful heuristic functions become both necessary and cost-effective.

Introduction

The sliding-tile puzzles, such as the Eight and Fifteen Puzzle, have long served as testbeds for heuristic search in AI. A square frame is filled with numbered tiles, leaving one position empty, called the blank. Any tile that is horizontally or vertically adjacent to the blank can be slid into the blank position. The task is to rearrange the tiles from some random initial configuration into a particular goal configuration, ideally or optimally in a minimum number of moves. The state space for the Eight Puzzle contains over 10^5 nodes, the Fifteen Puzzle space contains about 10^{13} nodes, and the Twenty-Four Puzzle contains almost 10^{25} nodes.

Due to its small search space, optimal solutions to the Eight Puzzle can be found with breadth-first search. We first found optimal solutions to the Fifteen Puzzle using Iterative-Deepening-A* (IDA*) and the Manhattan distance heuristic function (Korf 1985). IDA* is a variant of the well-known A* algorithm (Hart, Nilsson, and Rafael 1968), which runs in space that is linear in the maximum search depth, rather than exponential. IDA* proceeds in a series of depth-first search iterations, starting from the initial state. Each path is explored until a node n is reached where the number of moves from the initial state, $g(n)$, plus the heuristic estimate of the number of moves necessary to reach the goal state, $h(n)$, exceeds a threshold for that iteration. The threshold for the first iteration is the heuristic estimate for the initial state, and the threshold for each succeeding iteration is the minimum total cost, $f(n) = g(n) + h(n)$, of all nodes on the frontier of the previous iteration. The algorithm continues until a goal node is chosen for expansion.

	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure 1: The Twenty-Four Puzzle in its goal state

The Manhattan distance heuristic is computed by taking each tile, counting the number of grid units to its goal location, and then summing these values for all tiles. Since only one tile can move at a time, Manhattan distance never overestimates the number of moves needed to solve a given problem. Given such an *admissible* heuristic function, IDA* is guaranteed to return an optimal solution, if one exists.

IDA* with the Manhattan distance heuristic can solve random instances of the Fifteen Puzzle (Korf 1985). In spite of considerable work on this problem in the last decade, however, nobody has solved a significantly larger version of the puzzle. Note that the state space of the Twenty-Four Puzzle is almost a trillion times larger than that of the Fifteen Puzzle.

We present the first random Twenty-Four Puzzle instances for which optimal solutions have been found. Ten random solvable instances were generated, and so far we have found optimal solutions to all but one.

Three factors have contributed to this limited success. The first is simply faster computers. The Sun Ultra Sparc workstation that these experiments were run on is about 70 times faster than the DEC 2060 that the Fifteen Puzzle was originally solved on. The second is a technique we developed for pruning duplicate nodes in depth-first search (Taylor and Korf 1993). Finally, we have discovered more powerful heuristic functions for this problem. The most important contribution of this paper, however, is a new theory that allows these heuristics to be automatically learned and applied. All examples in this paper refer to the Twenty-Four Puzzle, where positions are labelled by the tiles that occupy them in the goal state shown in Figure 1.

Improved Admissible Heuristics

Linear Conflict Heuristic

The first significant improvement to Manhattan distance was the linear-conflict heuristic (Hansson, Mayer, and Yung 1992). It applies when two tiles are in their goal row or column, but are reversed relative to their goal positions. For example, if the top row of the puzzle contains the tiles (2 1) in that order, to reverse them, one of the tiles must move down out of the top row, to allow the other to pass by, and then back up. Since these two moves are not counted in the Manhattan distance of either tile, two moves can be added to Manhattan distance without violating admissibility.

As another example, if the top row contains the tiles (3 2 1) in that order, four more moves can be added to the Manhattan distance, since every pair of tiles is reversed, and two tiles must move out of the row temporarily. Furthermore, a tile in its goal position may be in both a row and a column conflict. Since the extra moves required to resolve a row conflict are vertical moves, and those required by a column conflict are horizontal, both sets of moves can be added to the Manhattan distance, and still preserve admissibility.

This addition to the Manhattan distance heuristic reduces the number of nodes generated by IDA* on the Fifteen Puzzle by roughly an order of magnitude. The additional complexity of computing the linear conflicts slows down node generation by about a factor of two, however, for a net improvement of a factor of five. Efficiently computing this heuristic involves precomputing and storing all possible permutations of tiles in a row or column, and incrementally computing the heuristic value of a child from that of its parent.

Last Moves Heuristic

The next enhancement to the heuristic is based on the last moves of a solution, which must return the blank to its goal position, the upper-left corner in this case. Thus, the last move must either move the 1 tile right, or the 5 tile down. Therefore, immediately before the last move, either the 1 or 5 tile must be in the upper-left corner. Since the Manhattan distance of these tiles is computed to their goal positions, unless the 1 tile is in the left-most column, its Manhattan distance will not accommodate a path through the upper-left corner. Similarly, unless the 5 tile is in the top row, its Manhattan distance will not accommodate a path through the upper-left corner. Thus, if the 1 tile is not in the left-most column, and the 5 tile is not in the top row, we can add two moves to the Manhattan distance, and still preserve admissibility.

While two moves may seem like a small improvement, it can be added to about 64% of random Twenty-Four Puzzle states. The effect of two additional moves is to save an entire iteration of IDA*. Since each iteration of IDA* on the Twenty-Four Puzzle can generate up to ten times as many nodes as the previous iteration, saving an iteration can result in an order of magnitude savings in nodes generated.

We can extend the same idea to the last two moves. If the last move is made by the 1 tile, the next-to-last move must either move the 2 tile right, or the 6 tile down. Similarly, if the last move is made by the 5 tile, the next-to-last move must either move the 6 tile right, or the 10 tile down. Considering the last two moves can add up to four moves to the Manhattan distance. Extending this idea to the last three moves was not cost effective on the Twenty-Four Puzzle.

To benefit from both the linear conflict and last moves enhancements, and maintain admissibility, we must consider their interactions. For example, assume that the 1 tile is not in the left-most column, and the 5 tile is not in the top row. If the 1 tile is in its goal column, and in a column conflict with another tile, then the two additional moves added by the linear conflict could be used to move the 1 tile left, allowing it to pass through the upper-left corner. Similarly, if the 5 tile is in its goal row, and in a row conflict, the two additional linear conflict moves could be used to move it up and hence through the upper-left corner. Thus, if either of these conditions occur, we can't add two more moves for the last move, since that may count twice moves already added by the linear conflict heuristic. Similarly, any additional moves added for the last two moves must also be checked against linear conflicts involving the 2, 6, and 10 tiles. In general, whenever more than one heuristic is being used, we must compute their interactions to maintain admissibility.

Relation to Bidirectional Search The reader may notice that a heuristic based on the last moves in the solution is related to bi-directional search. The most effective form of bidirectional heuristic search is called perimeter search (Dillenburg and Nelson 1994) (Manzini 1995). A limited breadth-first search backward from the goal state is performed, and the nodes on the perimeter of this search are stored. IDA* is then run from the initial state, with heuristic calculations made to determine the minimum distance to any state on the perimeter. This heuristic value is then added to the distance from the initial state to the given node, plus the distance from the perimeter to the goal state, for a more accurate admissible heuristic.

In a unidirectional search, the heuristic function is always computed to a single goal state. As a result, the heuristic calculation can be optimized to take advantage of this. With any form of bidirectional search, however, the heuristic must be calculated between arbitrary pairs of states, reducing the opportunities for optimization. While (Manzini 1995) reports speedups of up to a factor of eight on the Fifteen Puzzle using his improved perimeter search, he uses only the Manhattan distance heuristic function. It's not clear if similar results could be achieved with a more complex heuristic such as linear conflict.

Corner-Tiles Heuristic

The next enhancement to our heuristic focuses on the corners of the puzzle. For example, if the 3 tile is in its goal position, but some tile other than the 4 is in the 4 position, the 3 tile will have to move temporarily to correctly position the 4 tile. This requires two moves of the 3 tile, one to move it out of position, and another to move it back. If the 3 tile is involved in a row conflict, then two moves will already be counted for it, and no more can be added. It can't be involved in a column conflict if it's in its goal position.

The same rule applies to the 9 tile, unless the 9 is involved in a column conflict. In fact, if both the 3 and 9 tiles are correctly positioned, and the 4 tile is not, then four moves can be added, since both the 3 and 9 tiles will have to move to correctly position the 4.

This rule also applies to the 15, 19, 21, and 23 tiles. It applies to the 1 and 5 tiles as well, but the interaction of this heuristic with the last moves heuristic is so complex that to avoid the overhead of this calculation, we exclude the 1 and 5 tiles from the corner heuristic.

The corner-tile heuristic can potentially add up to twelve additional moves to the Manhattan distance, two for each of the six tiles adjacent to three of the corners. These extra moves require that at least one of these six tiles be in its goal position, a situation that only occurs in about 22% of random states. A search for the goal, however, does not generate a random sample of states, but is biased toward states that are close to the goal, or at least appear to the heuristic to be close. In other words, the search is trying to correctly position the tiles, and hence this heuristic adds extra moves much more often than would be expected from a random sample of states.

In summary, we have considered three enhancements to the Manhattan distance heuristic, based on linear conflicts, the last moves, and the corner tiles. The last two are introduced here for the first time.

A New Theory of Admissible Heuristics

While these enhancements result in a much more powerful heuristic, they appear to be a collection of domain-specific hacks. Furthermore, integrating the enhancements together into an admissible heuristic seems to require even more domain-specific reasoning. However, all these heuristics can be derived from a general theory that is largely domain-independent, and the heuristics can be automatically learned and applied. While we would like to be able to claim that these heuristics were discovered from the general theory, in reality the theory was discovered after the fact.

The classic theory of admissible heuristic functions is that they are the costs of optimal solutions to simplified problems, derived by removing constraints from the original problem (Pearl 1984). For example, if we remove the condition that a tile can only be moved into the blank position, the resulting problem allows any tile to move to any adjacent position at any time, and allows multiple tiles to occupy the same position. The number of moves to optimally solve this simplified problem is the Manhattan distance from the initial state to the goal state. While this theory accounts for many heuristics for many problems, it doesn't explain any of the above enhancements to Manhattan distance.

Automatically Learning the Heuristics

An alternative derivation of Manhattan distance is based on the original problem, but focuses on only one tile at a time. For each possible location of each individual tile, we perform a search to correctly position that tile, ignoring all other tiles, and only counting moves of the tile in question. In this search, a state is uniquely determined by the position of the tile of interest and the position of the blank, since all other tiles are equivalent. Since the operators of the sliding-tile puzzle are invertible, we can perform a single search for each tile, starting from its goal position, and record how many moves of the tile are required to move it to every other position. Doing this for all tiles results in a table which gives, for each possible position of each tile, its Manhattan distance from its goal position. Then, noticing that each move only moves one tile, for a given state we add up the Manhattan distances of each tile to get an admissible heuristic for the state. Of course, we don't really need to do the search in this case, since we can easily determine the values from the problem, but we presented it in this way to eliminate as much domain-specific reasoning as possible, and replace it with domain-independent search.

The value of this reconstruction of Manhattan distance is that it suggests a further generalization. The above formulation considers each tile in isolation, and the inaccuracy of the resulting heuristic stems from ignoring the interactions between the tiles. The obvious next step is to repeat the above process on all possible pairs of tiles. In other words, for each pair of tiles, and each combination of positions they could occupy, perform a search to their goal positions, and count only moves of the two tiles of interest. We call this value the *pairwise distance* of the two tiles from their goal locations. A state of this search consists of the positions of the two tiles and the position of the blank, since all other tiles are equivalent. Again for efficiency, for each pair of tiles we can perform a single search starting from their goal positions, with the blank also in its goal position, and store the pairwise distances to all other positions. The goal of this search is to find the shortest path from the goal state to all possible positions of the two tiles, where only moves of the two tiles of interest are counted. We can do this with a best-first search, counting only these moves.

Since states of these searches are only distinguishable by the positions of the two tiles and the blank, the size of these search spaces is $O(n^3)$, where n is the number of tiles. There are $O(n^2)$ such searches to perform, one for each pair of tiles, for a time complexity of $O(n^5)$. The size of the resulting table is $O(n^4)$, for each pair of tiles in each combination of positions.

For almost all pairs of tiles and positions, their pairwise distances equal the sum of their Manhattan distances from their goal positions. However, there are three types of cases where the pairwise distance exceeds the combined Manhattan distance. The first is when the two tiles are in a linear conflict. The second is when the two tiles are 1) a tile in its goal position adjacent to a corner, and 2) the tile that either belongs in, or that happens to be in, the corresponding corner. The third case is tiles 1 and 5, which are adjacent to the blank position in the goal state. The reason their pairwise distance may exceed their combined Manhattan distances is that the backwards pairwise search starts from the goal state, and hence the first move is to move the 1 or the 5 tile into the corner. Thus, computing all the pairwise distances by a simple search "discovers" Manhattan distance along with all three of the heuristic enhancements described above, with very little domain-specific reasoning. No other enhancements are discovered by the pairwise searches.

Applying the Heuristics

The next question is how to automatically handle the interactions between these heuristics to compute an admissible heuristic estimate for a particular state. Assume that we have precomputed all the pairwise tile distances and stored them in a table. Given a particular state, we look up all the pairwise distances for the current positions of the tiles. To compute the overall heuristic, we then partition the tiles into groups of two, and sum the corresponding pairwise distances, in a way that maximizes the resulting heuristic value.

To see this problem more clearly, represent a state as a graph with a node for each tile, and an edge between each pair of tiles, labelled with their pairwise distance. We need to select a set of edges from this graph, so that no two edges are connected to a common node, and the sum of the labels of the selected edges is maximized. This problem is called the maximum weighted matching problem, and can be solved in $O(n^3)$ time, where n is the number of nodes (Papadimitriou and Steiglitz 1982). Thus, this approach to heuristic generation can be automated, and runs in polynomial time.

Higher-Order Heuristics

Unfortunately, the pairwise distances do not account for the full power of the heuristic enhancements described above. For example, consider the linear conflicts represented by the tiles (3 2 1), in that order in the top row. The linear conflict heuristic would add four moves to the Manhattan distance of these tiles, since all pairs are reversed, and two of the tiles must move out of the row. The pairwise distance of each pair of these tiles is two moves plus their Manhattan distances. The graph representation of this situation is a triangle of tiles, with each edge of the triangle having weight two, ignoring the Manhattan distances. The maximum matching on this graph only contains one edge, with a total weight of two, since any two edges have a node in common. Thus, the pairwise distances capture only part of the linear conflict heuristic.

As another example, consider the corner-tile heuristic, and a state in which the 3 and 9 tiles are correctly positioned, but the 4 tile is not. The corner heuristic would add four moves to the Manhattan distance of the 4 tile, since both the 3 and 9 tiles must move to correctly place the 4 tile. The graphical representation of this situation consists of an edge between the 3 and 4 tiles, and an edge between the 9 and 4 tiles, each with a label of two, if we ignore the Manhattan distance. Since both these edges include the 4 tile, we can only select one of them, for an addition of only two moves.

Finally, while the pairwise distances capture the enhancement due to the last move of the solution, it doesn't capture the last two moves, since these involve the 2, 6, and 10 tiles, in addition to the 1 and 5 tiles.

In order to capture the full power of these heuristics, we extend the idea of pairwise distances to include triples of tiles, quadruples, etc. The linear conflict example of (3 2 1) requires us to consider all three tiles together to get four additional moves. If we consider each corner tile together with both adjacent tiles, we get the full power of the corner-tile heuristic. Finally, the last-two-moves enhancement requires considering all five tiles that may be involved. The corresponding matching problem is hypergraph matching, where a single edge "connects" three or more nodes, and unfortunately is NP-Complete. Thus, we may have to rely on a greedy approach to the higher-dimensional matching problem, and a lower heuristic value. As we consider higher-order heuristics, the complexity of the learning search, the size of the lookup table, and the complexity of the matching all increase, in return for more accurate heuristic values.

We believe this is a general theory for the discovery and implementation of admissible heuristic functions. All combinatorial problems involve solving multiple subgoals. Many admissible heuristics are constructed by considering the solution to each individual subproblem in isolation, and ignoring the interactions with other subproblems. We are proposing heuristics based on the simultaneous consideration of two, three, or more subgoals. As another example, consider a job-shop scheduling problem. There are a set of jobs to be performed, and a collection of machines with which to accomplish them. Each machine can only process a single job at a time. One way to derive a lower bound on the optimal solution is to consider the resources required by each job individually, and sum this over all jobs, ignoring resource conflicts between the jobs. Following our approach, one would consider all pairs of jobs, compute the resources required for each pair, and then compute the total resources by summing these values for a pairwise partition of the jobs.

Pruning Duplicate Nodes

While the main concern of this paper is the heuristic functions, we also used another orthogonal technique to significantly speed up the experiments.

Any depth-first search, such as IDA*, will generate the same node multiple times on a graph with cycles. For example, consider a square grid problem space, with the moves Up, Down, Left, and Right, each moving one unit in the indicated direction. Since there are four moves from every state, the asymptotic complexity of a depth-first search to depth d is $O(4^d)$. However, there are only $O(d^2)$ distinct states at depth d in a grid, and a breadth-first search, which stores all nodes generated and checks for duplicates, will run in $O(d^2)$ time. The difference in complexity between the breadth-first and depth-first search in this example illustrates the magnitude of this problem.

In the grid example, the operator pairs Left-Right and Up-Down are inverses of each other. Any good depth-first search implementation will remember the last operator applied, and never immediately apply its inverse. This can be done by encoding the last operator applied as the state of a finite-state machine. The machine has five states, an initial transient state and four recurrent states, one for each last move. Each arc of the machine represents an operator, except that the inverse of the last move is excluded. This reduces the complexity of the depth-first search from $O(4^d)$ to $O(3^d)$, a significant reduction, but still far from the $O(d^2)$ complexity of the breadth-first search.

This idea can be carried further, and is described in detail in (Taylor and Korf 1993). Ideally, we would like to allow only one path to each node in the grid. This can be done by first making all Left or Right moves, if any, followed by a single turn, and then all Up moves or all Down moves, if any. These rules can also be enforced by a five-state finite-state machine. The initial state allows all four operators, and each resulting state encodes the last move applied. If the last move was to the Right, all moves are allowed except a move to the Left. Similarly, if the last move was to the Left, all moves are allowed except a move to the Right. If the last move was Up, however, the only allowable move is another Up move. Similarly, if the last move was Down, the only allowable move is another Down move. This finite-state machine can only generate a single path to each point of the grid, and hence a depth-first search controlled by this machine runs in time $O(d^2)$, which is the same as a breadth-first search.

These finite-state machines can be automatically learned, from a small breadth-first search to discover duplicate operator strings. In this case, a breadth-first search to depth two is sufficient to learn all the duplicate strings to construct the above machine. Once the machine is constructed, there is almost no overhead to using it to control the depth-first search.

This technique can be applied to other problems, such as the sliding tile-puzzles. After rejecting inverse operators, the next shortest cycle in the sliding-tile puzzles is twelve moves long, corresponding to rotating the tiles in a 2×2 square. Using a breadth-first search, a finite-state machine for the Twenty-Four Puzzle was constructed with over 619,000 states. This machine is then used to control a depth-first search, rejecting operators that lead to duplicate nodes. The effect of this duplicate pruning is to reduce the asymptotic complexity of a depth-first search from $O(2.368^d)$ to $O(2.235^d)$. While this may seem like a small improvement, in the two easiest problems reported below, duplicate pruning decreased the running time of IDA* by factors of 2.4 and 3.6, with the larger improvement coming on the harder problem.

Experimental Results

We implemented IDA*, taking full advantage of the Manhattan distance, linear conflict, last-two-moves, and corner-tile heuristics, as well as the finite-state machine pruning. Since we were concerned with efficiency, our implementation was specialized to these heuristics and their interactions, rather than using a general table lookup and matching algorithm.

As a first test of our program, we ran it on 100 randomly generated solvable instances of the Nineteen Puzzle. The Nineteen Puzzle is the 4×5 sliding-tile puzzle, and its state space contains about 10^{18} states. All the puzzle instances were solved optimally, and the average solution length was 71.5 moves, as compared to an average solution length of 52.6 moves for the Fifteen Puzzle. The average number of node generations per problem instance was almost a billion, which is comparable to those generated by IDA* on the Fifteen Puzzle using just the Manhattan distance heuristic. To our knowledge, these are the first random Nineteen Puzzle problem instances to be solved optimally.

We then turned our attention to the Twenty-Four Puzzle. Ten random solvable instances were generated. Since there is enormous variation in the time to solve these problems, different iterations of IDA* were interleaved on different problem instances, in order find and solve the easier ones first. To date, nine of these prob-

No.	Initial State	Nodes Generated	Optimal Sol.
1	17 1 20 9 16 2 22 19 14 5 15 21 0 3 24 23 18 13 12 7 10 8 6 4 11	8,110,532,608	100
2	14 5 9 2 18 8 23 19 12 17 15 0 10 20 4 6 11 21 1 7 24 3 16 22 13	18,771,430,922	95
3	7 13 11 22 12 20 1 18 21 5 0 8 14 24 19 9 4 17 16 10 23 15 3 2 6	82,203,971,683	108
4	18 14 0 9 8 3 7 19 2 15 5 12 1 13 24 23 4 21 10 20 16 22 11 6 17	83,573,198,724	98
5	2 0 10 19 1 4 16 3 15 20 22 9 6 18 5 13 12 21 8 17 23 11 24 7 14	221,769,436,018	101
6	16 5 1 12 6 24 17 9 2 22 4 10 13 18 19 20 0 23 7 21 15 11 8 3 14	523,772,060,498	96
7	21 22 15 9 24 12 16 23 2 8 5 18 17 7 10 14 13 4 0 6 20 11 3 1 19	792,795,062,385	104
8	6 0 24 14 8 5 21 19 9 17 16 20 10 13 2 15 11 22 1 3 7 23 4 18 12	1,415,436,865,760	97
9	3 2 17 0 14 18 22 19 15 20 9 7 10 21 16 6 24 23 8 5 1 4 11 12 13	3,033,449,077,924	113
10	23 14 0 24 17 9 20 21 2 18 10 13 22 1 3 11 4 16 6 5 7 12 8 15 19	> 3,000,000,000,000	≥ 112

Table 1: Twenty-four puzzle problem instances, nodes generated, and optimal solution lengths

lems have been solved optimally, with a lower bound established for the remaining one. Table 1 shows all ten problem instances, sorted by difficulty. For the solved problems, we give the number of nodes generated and the optimal solution length, and for the unsolved one we give lower bounds on these values. The tiles are listed from left to right and top to bottom, with 0 representing the blank. In this notation, the tiles of the goal state in Figure 1 would be listed in numerical order. The average optimal solution length for these ten problems is at least 102.4 moves. The code was written in C, runs on a Sun Ultra Sparc workstation, and generates about a million nodes per second. The easiest problem took about two hours and 15 minutes to solve, and the most difficult solved problem took over a month. To date, the remaining unsolved problem has run for over a month. These are the first random Twenty-Four Puzzle instances to be solved optimally.

Conclusions

We have found the first optimal solutions to random instances of the Twenty-Four Puzzle, a problem with almost 10^{25} states. The branching factor is 2.368, and the optimal solutions average over 100 moves long. We implemented IDA* on a state-of-the-art workstation, with a more powerful admissible heuristic function, and a method for pruning duplicate nodes in depth-first search. The most important contribution of this paper is a new general theory for the automatic discovery and application of admissible heuristics. Instead of considering individual subgoals in isolation, our approach considers two or more subgoals simultaneously. This theory allows one to automatically discover Manhattan distance, along with the linear conflict, last moves, and corner-tile enhancements to it, with nothing more than small searches of the problem space. By considering three or more subgoals at a time, even more powerful heuristics can be derived.

A more powerful heuristic function increases the time per node generation by a polynomial amount. On the other hand, it generally decreases the effective branching factor by a small amount, yielding an asymptotic improvement. For small problems, more powerful heuristics may not be cost effective, since one doesn't search deep enough to overcome the polynomial overhead. As machines get faster and larger problems are addressed, however, seeming small improvements in a heuristic function eventually become cost effective. Thus, as problem size increases, it becomes both necessary and cost-effective to encode more knowledge of the problem in the form of improved heuristics. We have developed an approach to doing this automatically.

Acknowledgements

This work was supported by NSF Grant IRI-9119825, and a grant from Rockwell International.

References

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

Hansson, O., A. Mayer, and M. Yung, Criticizing solutions to relaxed models yields powerful admissible heuristics, *Information Sciences*, Vol. 63, No. 3, 1992, pp. 207-227.

Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, 1968, pp. 100-107.

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

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

Papadimitriou, C.H., and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

Pearl, J. *Heuristics*, Addison-Wesley, Reading, MA, 1984.

Taylor, L., and R.E. Korf, Pruning duplicate nodes in depth-first search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-93)*, Washington D.C., July 1993, pp. 756-761.