Recent Progress in the Design and Analysis of Admissible Heuristic Functions

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

Abstract

In the past several years, significant progress has been made in finding optimal solutions to combinatorial problems. In particular, random instances of both Rubik's Cube, with over 10^{19} states, and the 5×5 sliding-tile puzzle, with almost 10^{25} states, have been solved optimally. This progress is not the result of better search algorithms, but more effective heuristic evaluation functions. In addition, we have learned how to accurately predict the running time of admissible heuristic search algorithms, as a function of the solution depth and the heuristic evaluation function. One corollary of this analysis is that an admissible heuristic function reduces the effective depth of search, rather than the effective branching factor.

Introduction

The Fifteen Puzzle consists of fifteen numbered square tiles in a 4×4 square grid, with one position empty or blank. Any tile horizontally or vertically adjacent to the blank can be moved into the blank position. The task is to rearrange the tiles from some random initial configuration into a desired goal configuration, ideally or optimally using the fewest moves possible.

The Fifteen Puzzle was invented by Sam Loyd in the 1870s (Loyd, 1959), and appeared in the scientific literature shortly thereafter (Johnson and Story, 1879). The editor of the journal added the following comment to the paper: "The '15' puzzle for the last few weeks has been prominently before the American public, and may safely be said to have engaged the attention of nine out of ten persons of both sexes and of all ages and conditions of the community."

One reason for the world-wide Fifteen Puzzle craze was that Loyd offered a $1000 cash prize to transform a particular initial state to a particular goal state. Johnson and Story proved that it wasn't possible, that the entire state space was divided into even and odd permutations, and that there is no way to transform one into the other by legal moves.

Rubik's Cube was invented in 1974 by Erno Rubik of Hungary, and like the Fifteen Puzzle a hundred years earlier, became a world-wide sensation. More than 100 million Rubik's Cubes have been sold, and it is the best-known combinatorial puzzle of all time.

In the remainder of this paper, we'll use these example problems to illustrate recent progress in heuristic search. In particular, the design of more accurate heuristic evaluation functions has allowed us to find optimal solutions to random instances of both the 5×5 Twenty-Four puzzle, and Rubik's Cube for the first time. In addition, we'll present a theory that allows us to accurately predict the running time of admissible heuristic search algorithms from the solution depth and the heuristic evaluation function. One consequence of this theory is that an admissible heuristic function decreases the effective depth of search, relative to a brute-force search, rather than the effective branching factor.

Search Algorithms

The 3×3 Eight puzzle contains only 181,440 reachable states, and hence can be solved optimally by a brute-force breadth-first search in a fraction of a second.

To solve the 4×4 Fifteen Puzzle however, with about 10^{13} states, we need a heuristic search algorithm, such as A* (Hart, Nilsson, and Raphael 1968). A* is a best-first search in which the cost of a node n is computed as $f(n) = g(n) + h(n)$, where $g(n)$ is the length of the current path from the start to node n, and $h(n)$ is a heuristic estimate of the length of a shortest path from node n to a goal. If $h(n)$ is *admissible*, meaning it never overestimates the distance to a goal, A* is guaranteed to find a shortest solution, if one exists.

The classic heuristic function for the sliding-tile puzzles is Manhattan distance. It is computed by taking each tile, counting the number of grid units between its current location and its goal location, and summing these values for all tiles. Manhattan distance is a lower bound on actual solution length, because every tile must move at least its Manhattan distance, and each move only moves one tile.

Unfortunately, A* can't solve the Fifteen Puzzle, because it stores every node it generates, and exhausts the available memory on most problems before finding a solution. Iterative-Deepening-A* (IDA*) (Korf, 1985) is a linear-space version of A*. It performs a series of depth-first searches, pruning a path and backtracking when the cost $f(n) = g(n) + h(n)$ of a node n on the path exceeds a cutoff threshold for that iteration. The initial threshold is set to the heuristic estimate of the initial state, and increases in each iteration to the lowest cost all the nodes pruned on the last iteration, until a goal node is expanded. Like A*, IDA* guarantees an optimal solution if the heuristic function is admissible. Unlike A*, however, IDA* only requires memory that is linear in the maximum search depth. IDA*, using the Manhattan distance heuristic, was the first algorithm to find optimal solutions to random instances of the Fifteen Puzzle (Korf, 1985). An average of about 400 million nodes are generated per problem instance, requiring about 6 hours of running time in 1985.

Design of Heuristic Functions

Classical Explanation

The standard explanation for the origin of heuristic functions is that they compute the cost of exact solutions to a simplified version of the original problem (Pearl, 1984). For example, in the sliding-tile puzzles, if we ignore the constraint that we can only move a tile into the empty position, we get a new problem where any tile can be moved to any adjacent position, and multiple tiles can occupy the same position. In this simplified problem, we can solve any instance by taking each tile one at a time, and moving it along a shortest path to its goal position, counting the number of moves made. The cost of an optimal solution to this simplified problem is just the Manhattan distance of the original problem. Since we simplified the problem by removing a constraint on the moves, any solution to the original problem is also a solution to the simplified problem, and hence the cost of an optimal solution to the simplified problem is a lower bound on the cost of an optimal solution to the original problem. Thus, any heuristic derived in this way is admissible.

What makes it possible to efficiently compute the Manhattan distance is that in the simplified problem, the individual tiles can move independently of each another. The reason the original problem is difficult, and why the Manhattan distance is only a lower bound on actual cost, is that the tiles interact. By taking into account some of these interactions, we can compute more accurate admissible heuristic functions.

Pattern Databases

Pattern databases (Culberson and Schaeffer, 1998) are one way to do this. Consider any subset of tiles, such as the seven tiles in the right column and bottom row of the Fifteen Puzzle, which they called the *fringe pattern*. The minimum number of moves required to get the fringe tiles from their initial positions to their goal positions, including any required moves of other tiles as well, is obviously a lower bound on the minimum number of moves needed to solve the entire problem.

It would be too expensive to calculate the moves needed to solve the fringe tiles for each state in the search. This number, however, depends only on the positions of the fringe tiles and the blank position, but not on the positions of the other tiles. Since there are only a limited number of such configurations, we can precompute all of these values, store them in memory in a table, and look them up as needed during the search. Since there are seven fringe tiles and one blank, and sixteen different locations, the total number of possible configurations of these tiles is $16!/(16-8)! = 518,918,400$. For each table entry, we can store the number of moves needed to solve the fringe tiles from their corresponding locations, which takes only a byte of storage. Thus, we can store the whole table in less than 500 megabytes of memory.

We can compute this table by a single breadth-first search backward from the goal state. In this search, the non-pattern tiles are all considered equivalent, and a state is uniquely determined by the positions of the pattern tiles and the blank. As each configuration of these tiles is encountered for the first time, the number of moves made to reach it is stored in the corresponding entry of the pattern database. The search continues until all entries of the table are filled. Note that this table is only computed once for a given goal state, and its cost can be amortized over the solution of multiple problem instances with the same goal state.

Once the table is built, we use IDA* to search for an optimal solution to a problem instance. As each state is generated, the positions of the pattern tiles and the blank are used to compute an index into the pattern database, and the corresponding entry, which is the number of moves needed to solve the pattern tiles, is used as the heuristic value for that state.

Using the fringe pattern database, (Culberson and Schaeffer, 1998) reduced the number of nodes generated to solve the Fifteen Puzzle by a factor of 346, and reduced the running time by a factor of 6. Combining this with another pattern database, and taking the maximum of the two database values as the heuristic value, reduced the nodes generated by about a thousand, and the running time by a factor of 12, compared to Manhattan distance.

Rubik's Cube Pattern databases have also been used to find optimal solutions to Rubik's Cube (Korf, 1997). The standard $3 \times 3 \times 3$ Rubik's Cube contains about 4.3252×10^{19} different reachable states. Of the 27 subcubes, or *cubies*, 20 of them move. These can be divided into eight *corner cubies*, with three faces each, and twelve *edge cubies*, with two faces each. There are only $88,179,840$ different configurations of the corner cubies, and the number of moves to solve just the corner cubies ranges from zero to eleven moves. At four bits per entry, a pattern database for the corner cubies requires about 42 megabytes of memory. Six of the twelve edge cubies generate $42,577,920$ different possibilities, and a corresponding pattern database requires about 20 megabytes of memory. Similarly, the remaining six edge cubies generate another pattern database of the same size.

Given multiple pattern databases, the best way to combine them without overestimating the actual solution cost, is to take the maximum of their values, even if the cubies in the different databases don't overlap. The reason for this is that every twist of the cube moves eight different cubies, and hence moves that contribute to the solution of the cubies in one pattern may also contribute to the solution of the others. Taking the maximum of the values in all three pattern databases described allowed IDA* to find the first optimal solutions to random instances of Rubik's Cube (Korf, 1997). The median optimal solution length is 18 moves. At least one problem instance generated a trillion nodes, and required a couple weeks to run. With further improvements by Michael Reid, Herbert Kociemba, and others, most states can now be solved optimally in a day.

Disjoint Pattern Databases

The main limitation of Culberson and Schaeffer's pattern databases is that the only way to combine the values from different databases without overestimating actual cost is to take their maximum value. Returning to the Fifteen Puzzle, even if we compute a separate pattern database for the remaining eight tiles not in the fringe pattern, the best admissible combination of these two heuristic values is their maximum. The reason is that Culberson and Schaeffer counted all moves required to solve the pattern tiles, including moves of tiles not in the pattern. As a result, moves used to solve tiles in one pattern may also be used to solve tiles in another pattern.

One way to improve on this is when computing the heuristic value for a pattern of tiles, only count the moves of the tiles in the pattern. Then, given two or more patterns that have no tiles in common, we can add together the heuristic values from the different databases, and still get an admissible heuristic. This is because in the sliding-tile puzzle, each operator only moves a single tile. We call such a set of databases a *disjoint pattern database*, or a disjoint database for short. Summing the values of different heuristics results in a much larger value than taking their maximum, and thus greatly reduces the amount of search that is necessary.

A trivial example of a disjoint pattern database is Manhattan distance. Manhattan distance can be viewed as the sum of a set of individual pattern database values, each representing only a single tile. It could be "discovered" by running a pattern search for each tile, recording the number of moves required to get that tile to each location from its goal location.

A non-trivial example of a disjoint database divides the Fifteen Puzzle in half horizontally, into a group of seven tiles on top, and eight tiles on the bottom, assuming the goal position of the blank is the upper-left corner. We precompute the number of moves required to solve the tiles in each of these two patterns, from all possible combinations of positions, but only counting moves of the tiles in the given pattern. Instead of explicitly representing the blank position in the database, we store the minimum value for all possible positions of the blank. The eight-tile pattern contains $16!/(16-8)! = 518,918,400$ entries, each of which requires a byte, or 495 megabytes of memory. The 7-tile pattern contains only $16!/(16-7)! = 57,657,600$ entries, or 55 megabytes of storage.

The memory requirement can be reduced by only storing in the database the number of moves needed in addition to the sum of the Manhattan distances of the pattern tiles, which only takes four bits. Then, during the search, we compute the Manhattan distances of the pattern tiles, and add the database value to the Manhattan distance to get the overall heuristic.

Once these pattern databases are computed and stored, we get another set of heuristic values by reflecting all the tiles and their positions about the main diagonal of the puzzle. This gives us a 7-tile database on the left side of the puzzle, and an 8-tile pattern database on the right. The values from these two different sets of databases can only be combined by taking their maximum, since their individual tiles overlap.

This heuristic can be used to optimally solve random Fifteen Puzzle instances, generating an average of about 37,700 nodes, and taking about 43 milliseconds per problem instance on a 440 Megahertz Sun Ultra 10 workstation with 640 megabytes of memory. This is in comparison to 400 million nodes and about 75 seconds per problem on the same machine for simple Manhattan distance. This is a factor of over 10,000 in nodes generated, and over 1700 in actual running time.

Pairwise Distances

The original pattern database idea allows the most general combination rule, since the maximum of any set of admissible heuristics is always an admissible heuristic. Conversely, disjoint pattern databases admit the most powerful combination rule, by allowing the values from different heuristics to be added together, but are not very general, since they require each operator to effect only subgoals within a given pattern. Disjoint databases cannot be used on Rubik's Cube, for example, since each twist moves eight different cubies. Between these two extremes lies a technique that combines the two ideas.

Consider a database that contains the number of moves required to correctly position every pair of tiles, from every possible pair of positions they could be in. In most cases, this will be the sum of their Manhattan distances. In some cases, however, this *pairwise distance* will exceed the sum of the Manhattan distances. For example, if two tiles are in the same row, which is also their goal row, but they are reversed with respect to each other, one tile will have to move vertically out of the row, to allow the other to pass by, and then move back into the row. This adds two moves to the sum of their Manhattan distances, which only reflects the moves within their goal row. This is the idea behind the "linear conflict" heuristic function (Hansson, Mayer, and Yung, 1992), the first significant improvement to Manhattan distance. There are also other situations where the pairwise distance of two tiles from their goal location exceeds the sum of their Manhattan distances (Korf and Taylor, 1996).

The difficulty with the pairwise distance heuristic comes in applying it to a given state. We can't simply sum the pairwise distances of all pairs of tiles, because moves of the same tile will be counted repeatedly. Rather, we must partition the tiles into non-overlapping groups of two, and then sum the pairwise distances of each of the disjoint groups. Ideally, we want to choose a grouping for each state that maximizes the heuristic value. This is known as a maximal matching problem, and must be solved for each state in the search. Thus, heuristics based on pairwise distances are relatively expensive to compute. The idea of pairwise distances can obviously be generalized to distances of triples or quadruples of tiles as well.

Twenty-Four Puzzle An admissible heuristic based on linear conflicts and other pairwise distances lead to the first optimal solutions to random instance of the 5×5 Twenty-Four Puzzle (Korf and Taylor, 1996), containing almost 10^{25} states. Some of these problems generated trillions of nodes, and required weeks to run. Currently, we are applying disjoint databases to this problem, using patterns of six tiles, with significant reductions in nodes generated and running times.

Time Complexity of Heuristic Search

We now turn our attention to the time complexity of heuristic search algorithms. The central difficulty is that the running time depends on the quality of the heuristic function, which has to be characterized in some way. We begin with computing the brute-force branching factor, and then consider heuristic search.

Brute-Force Branching Factor

The running time of a brute-force search is $O(b^d)$, where b is the branching factor of the search space, and d is the solution depth of the problem instance. In the sliding-tile puzzles, the branching factor of a node depends on the position of the blank. If the blank is in a corner, there are two places it can go, if it's on a side it can go to three places, and from a center position it can to to four places. If we assume that all possible positions of the blank are equally likely, we get a branching factor of $4 \cdot 2 + 8 \cdot 3 + 4 \cdot 4/16 = 3$ for the Fifteen Puzzle. Subtracting one to eliminate the move back to the parent node yields a branching factor of 2.

Unfortunately, the blank is not equally likely to be in any position in a deep search. In particular, the more central location of the middle positions causes those positions to be over-represented in the search space. To compute the asymptotic branching factor, we need to compute the equilibrium fraction of nodes with the blank in the different types of positions at a given depth of the search tree, in the limit of large depth. When this is done correctly (Edelkamp and Korf, 1998), we get an asymptotic branching factor of about 2.13 for the Fifteen Puzzle.

A similar situation occurs in Rubik's Cube, even though all operators are always applicable. In this case, we restrict the operators applied to avoid redundant states. For example, if we allow any twist of a single face as a primitive operator, we don't want to twist the same face twice in a row, since the same effect can be achieved by a single twist. Furthermore, since twists of opposite faces are independent, these operators commute, and we only allow two consecutive twists of opposite faces to occur in one particular order. These considerations result in a branching factor of about 13.34847 for Rubik's Cube, compared to $6 \cdot 3 = 18$ for the naive problem space.

Conditions for Node Expansion

We now turn our attention to heuristic search. The running time of a heuristic search is proportional to the number of nodes expanded. Both A* and IDA* expand all nodes n whose total cost is less than the optimal solution cost, i.e. $f(n) = g(n) + h(n) < c*$, where $c*$ is the optimal solution cost (Pearl, 1984). An easy way to understand this node expansion condition is that any admissible search algorithm must continue to expand every partial solution path, until its cost equals or exceeds the cost of an optimal solution, lest it lead to a better solution.

Characterization of the Heuristic

As mentioned above, the central difficulty in analyzing the time complexity of heuristic search lies in characterizing the heuristic. Previous work on this problem (Pearl, 1984) characterized the heuristic by its accuracy as an estimator of optimal solution cost, and relied on an abstract analytic model of the search space. There are several problems with this approach. The first is that to determine the accuracy of a heuristic function on even a single problem instance, we have to determine the optimal solution cost, which is computationally very expensive on large problems. Secondly, most real problems don't fit the restrictive assumptions of the abstract model, namely that the problem space contain only a single solution path to the goal. Finally, the results obtained are only asymptotic results in the limit of large depth. As a result, this previous work cannot predict the actual performance of heuristic search on real problems such as the sliding-tile puzzles or Rubik's cube.

In our analysis (Korf and Reid, 1998), we characterize the heuristic function by the distribution of heuristic values over the problem space. In other words, we only need to know the fraction of states with each different heuristic value. Equivalently, let $P(x)$ be the fraction of total states in the problem space with heuristic value less than or equal to x. In other words, $P(x)$ is the probability that a randomly chosen state in the problem space has heuristic value less than or equal to x. More precisely, we need the distribution of heuristic values at a given depth of the brute-force search tree, in the limit of large depth, but we ignore this detail here. Note that the heuristic distribution says nothing directly about the accuracy of the heuristic function, except that distributions shifted toward larger values are more accurate, since we assume that our heuristics are admissible.

For heuristics based on a pattern database, we can compute the heuristic distribution exactly, simply by scanning the database. If the heuristic is based on several different pattern databases, we assume that the different heuristic values are independent. For heuristics based on functions, such as Manhattan distance, we can randomly sample states from the problem space, and use the heuristic values of the samples to approximate the heuristic distribution. Note that in either case, we don't have to solve any problem instances to get the heuristic distribution.

Main Theoretical Result

Here's the main result of our analysis (Korf and Reid, 1998). Let N_i be the number of nodes at depth i in the brute-force search tree. For example, N_i might be b^i, where b is the brute-force branching factor. In a heuristic search to depth d, the number of nodes expanded by A* or IDA* at depth i is simply $N_i \cdot P(d-i)$. At one level, the argument for this is simple. The nodes n at depth i have $g(n) = i$, and $P(d-i)$ is the fraction of nodes n for which $h(n) \leq d-i$. Thus, for these nodes, $f(n) = g(n) + h(n) \leq i + d - i = d$, which is the condition for node expansion in a search to depth d.

The key property that makes this work is consistency of the heuristic function. We say that h is consistent if for all nodes n and their neighbors n', $h(n) \leq c(n, n') + h(n')$, where $c(n, n')$ is the cost from node n to its neighbor n'. This is akin to the triangle inequality of metrics, and almost all admissible heuristics are consistent. If our heuristic is consistent, then the pruning that occurs in the tree doesn't effect the heuristic distribution of the nodes that are expanded. Given the number of nodes expanded at a given depth, we sum these values for all depths up to the optimal solution depth to determine the total number of nodes expanded, and hence the running time of the algorithm.

Experimental Results

We have experimentally verified this analysis on Rubik's Cube, the Eight Puzzle, and the Fifteen Puzzle. In each case, for N_i we used the actual numbers of nodes in the brute-force tree at each depth. For Rubik's cube, we determined the heuristic distribution from the pattern databases, assuming the values from different databases are independent. For the Eight Puzzle, we computed the heuristic distribution of Manhattan distance exactly by exhaustively generating the space, and for the Fifteen Puzzle, we approximated the Manhattan distance distribution by a random sample of ten billion states. We then compared the number of node expansions predicted by our theory to the average number of nodes expanded by IDA* on different random initial states. For Rubik's cube, we got agreement to within one percent, and for Fifteen puzzle we got agreement to within 2.5 percent at typical solution depths. For the Eight Puzzle, our theoretical predictions agreed exactly with our experimental results, since we could average the experimental results over all states in the problem space. This indicates that our theory accounts for all the relevant factors of the problem.

The "Heuristic Branching Factor"

From previous analyses, it was thought that the effect of an admissible heuristic function is to reduce the effective branching factor of a heuristic search relative to a brute-force search. The effective branching factor of a search is the limit at large depth of the ratio of the number of nodes generated at one level to the number generated at the next shallower level. One immediate consequence of our analysis, however, is that the effective branching factor of a heuristic search is the same as the brute-force branching factor of the problem space. The effect of the heuristic is merely to decrease the effective depth of search, by a constant based on the heuristic function. This prediction is also verified by our experimental results.

Conclusions

Pattern databases (Culberson and Schaeffer, 1998) automate the design of more effective lower-bound heuristics. We have used them to find optimal solutions to Rubik's cube. We have also extended the original idea to disjoint databases, which allow the values from different pattern databases to be added together, rather than just taking their maximum. Disjoint databases reduce the time to find optimal solutions to the Fifteen Puzzle by over three orders of magnitude, relative to the Manhattan distance heuristic. In addition, pairwise and higher order distances can also be used to compute more effective heuristics, but at greater cost per node evaluation. We have used both disjoint databases and pairwise distances to find optimal solutions to the 5×5 Twenty-Four puzzle.

We have also developed a new theory that allows us to predict the running time of heuristic search algorithms. The heuristic is characterized simply by the distribution of heuristic values over the problem space. Our theory accurately predicts our experimental results on the sliding-tile puzzles and Rubik's Cube. One consequence of our theory is that the effect of a heuristic is to reduce the effective depth of search, rather than the effective branching factor.

Acknowledgements

I would like to thank my collaborators in this work, including Stefan Edelkamp, Ariel Felner, Michael Reid, and Larry Taylor. This research was sponsored by NSF grant No. IRI-9619447.

References

Culberson, J., and J. Schaeffer. Pattern Databases, *Computational Intelligence*, Vol. 14, No. 4, 1998, pp. 318-334.

Edelkamp, S. and R.E. Korf, The branching factor of regular search spaces, *Proceedings of AAAI-98*, Madison, WI, July, 1998, pp. 299-304.

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. SSC-4, No. 2, July 1968, pp. 100-107.

Johnson, W.W. and W.E. Storey, Notes on the 15 puzzle, *American Journal of Mathematics*, Vol. 2, 1879, pp. 397-404.

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., and L.A. Taylor, Finding optimal solutions to the twenty-four puzzle, *Proceedings of AAAI-96*, Portland, OR, Aug. 1996, pp. 1202-1207.

Korf, R.E., Finding optimal solutions to Rubik's Cube using pattern databases, *Proceedings of AAAI-97*, Providence, RI, July, 1997, pp. 700-705.

Korf, R.E., and M. Reid, Complexity analysis of admissible heuristic search, *Proceedings AAAI-98*, Madison, WI, July, 1998, pp. 305-310.

Loyd, S., *Mathematical Puzzles of Sam Loyd*, Selected and Edited by Martin Gardner, Dover, New York, 1959.

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