Disjoint Pattern Database Heuristics

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

Ariel Felner
Dept. of Mathematics and Computer Science
Bar-Ilan University
Ramat-Gan, 52900 Israel
felner@macs.biu.ac.il

September 5, 2001

Abstract

We describe a new technique for designing more accurate admissible heuristic evaluation functions, based on *pattern databases* [1]. While many heuristics, such as Manhattan distance, compute the cost of solving individual subgoals independently, pattern databases consider the cost of solving multiple subgoals simultaneously. Existing work on pattern databases allows combining values from different pattern databases by taking their maximum. If the subgoals can be divided into disjoint subsets so that each operator only affects subgoals in one subset, then we can add the pattern-database values for each subset, resulting in a more accurate admissible heuristic function. We used this technique to improve performance on the Fifteen Puzzle by a factor of over 2000, and to find optimal solutions to 50 random instances of the Twenty-Four Puzzle.

1 Introduction and Oveview

The sliding-tile puzzles are a classic challenge for search algorithms in AI. The key to finding optimal solutions to these problems is an accurate admissible heuristic function. We describe a generalization of the Manhattan distance heuristic that considers the interactions among multiple tiles, while allowing the moves of different groups of tiles to be added together without violating admissibility. This results in a much more accurate admissible heuristic.

1.1 Sliding-Tile Puzzles

The 4×4 Fifteen, and 5×5 Twenty-Four Puzzles are shown in Figure 1. A square frame is filled with tiles, except for one empty or blank position. Any tile horizontally or vertically adjacent to the blank can be slid into that position. The task is to rearrange the tiles from a given initial configuration into a particular goal configuration, ideally or optimally in a minimum number of moves. The state space for the Fifteen Puzzle space contains about 10^{13} states, and the Twenty-Four Puzzle contains almost 10^{25} states.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

	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 Fifteen and Twenty-Four Puzzles in their Goal States

The Fifteen Puzzle was invented by Sam Loyd in the 1870s [13], and appeared in the scientific literature shortly thereafter [7]. 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." The reason for the 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, since the state space was divided into even and odd permutations, with no way to transform one into the other by legal moves.

1.2 Search Algorithms

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

To address the Fifteen Puzzle requires a heuristic search algorithm, such as A* [6]. 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 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 the goal. If $h(n)$ is *admissible*, or never overestimates distance to the goal, then A* is guaranteed to find a shortest solution, if one exists.

The classic heuristic function for the sliding-tile puzzles is Manhattan distance. For each tile we count the number of grid units between its current and goal locations, and sum 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 one square.

Unfortunately, A* can't solve random instances of the Fifteen Puzzle, because it stores every node generated, and exhausts the available memory in minutes on most problems. Iterative-Deepening-A* (IDA*) [8] is a linear-space version of A*. It performs a series of depth-first searches, pruning a path when the cost $f(n) = g(n) + h(n)$ of the last node n on the path exceeds a threshold for that iteration. The threshold is initially set to the heuristic estimate of the initial state, and increases in each iteration to the lowest cost of all nodes pruned on the last iteration, until a goal node is expanded. IDA* also 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* with the Manhattan distance heuristic was the first algorithm to find optimal solutions to random instances of the Fifteen Puzzle [8]. An average of about 400 million nodes are generated per problem instance, requiring almost five hours on a DEC 2060 in 1984.

1.3 Overview

On larger problems, IDA* with Manhattan distance takes too long, and more accurate heuristic functions are needed. While Manhattan distance sums the cost of solving each tile independently, we consider the costs of solving several tiles simultaneously, taking into account the interactions between them. Our main contribution is to show how heuristic values for different groups of tiles can be added together, rather than taking their maximum.

We first present existing heuristics, including non-additive pattern databases, using the example of Rubik's Cube. Next, we describe disjoint pattern databases, showing how they can be precomputed, and combined into an admissible heuristic. We then present experimental results on the Fifteen and Twenty-Four puzzles, finding optimal solutions to the Fifteen Puzzle 2000 times faster than with Manhattan distance, and finding optimal solutions to 50 random Twenty-Four Puzzles. Initial results of this work first appeared in [11].

2 Existing Heuristics

2.1 Manhattan Distance

Where did the Manhattan distance heuristic come from? In addition to the standard answer to this question, we present an alternative that suggests the disjoint pattern database extension.

The standard explanation for the origin of admissible heuristic functions is that they represent the cost of exact solutions to simplified versions of the original problem [15]. For example, in a sliding-tile puzzle, to move a tile from position x to position y, x and y must be adjacent, and position y must be empty. If we ignore the empty constraint, we get a simplified problem where any tile can move to any adjacent position, and multiple tiles can occupy the same position. In this new problem, the tiles are independent of each other, and we can solve any instance optimally by moving each tile 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 exactly the Manhattan distance from the initial to the goal state. Since we removed a constraint on the moves, any solution to the original problem is also a solution to the simplified problem, and 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.

Alternatively, we can derive Manhattan distance by observing that a sliding-tile puzzle contains subproblems of getting each tile to its goal location. This suggests considering the cost of solving each subproblem independently, assuming no interactions between them. In other words, we could search for the minimum number of moves needed to get each tile to its goal location, ignoring the other tiles, which is the Manhattan distance of that tile. Since each move only moves one tile, we can add these individual distances together to get an admissible heuristic for the problem. While the first derivation requires a problem description in terms of constraints on the legal moves, the second only requires recognizing a single tile as a subproblem.

The key idea here, which makes it possible to efficiently compute Manhattan distance, is the assumption that the individual tiles do not interact with one another. The reason the problem is difficult, and why Manhattan distance is only a lower bound on actual solution cost, is that the tiles get in each other's way. By taking into account some of these interactions, we can compute more accurate admissible heuristic functions.

2.2 Non-Additive Pattern Databases

Pattern databases[1], originally applied to the Fifteen Puzzle, are one way to do this. Figure 2 shows a subset of the Fifteen Puzzle tiles, called the *fringe* tiles. For a given state, the minimum number of moves needed to get the fringe tiles to their goal positions, including required moves of other tiles, is a lower bound on the number of moves needed to solve the entire puzzle.

This number depends on the current positions of the fringe tiles and the blank, but is independent of the positions of the other tiles. Thus, we can precompute all these values, store them in memory, and look them up as they are needed during the search. Since there are seven fringe tiles and one blank, and sixteen different locations, the total number of possible permutations of fringe tiles and blank is $16!/(16-8)! = 518,918,400$. For each permutation, we store the number of moves needed to move the fringe tiles and the blank to their goal locations, which takes less than one byte. Thus, we can store the whole pattern database table in less than 495 megabytes of memory.

We can compute this entire table with a single breadth-first search backward from the goal state shown in Figure 2. The unlabelled tiles are all

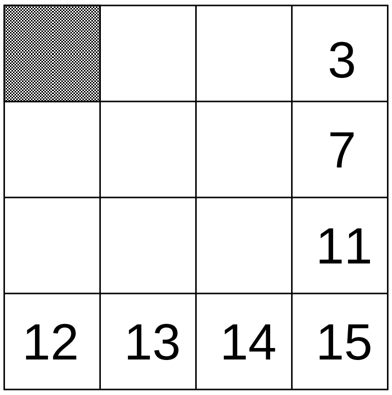

Figure 2: The Fringe Pattern for the Fifteen Puzzle

equivalent, and a state is uniquely determined by the positions of the fringe 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 table, until all entries are filled. Note that this table is only computed once for a given goal state, and its cost is amortized over the solution of multiple problem instances with the same goal state.

Once the table is stored, we use IDA* to search for an optimal solution to a particular problem instance. As each state is generated, the positions of the fringe 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 fringe tiles and blank, is used as the heuristic value for that state.

Using this pattern database, Culberson and Schaeffer reduced the number of nodes generated to solve random Fifteen Puzzles by a factor of 346, and reduced the running time by a factor of six, compared to Manhattan distance [1]. Combining this with another pattern database, and taking the maximum of the two database values as the overall heuristic value, reduced the nodes generated by about a thousand, and the running time by a factor of twelve.

2.2.1 Rubik's Cube

Non-additive pattern databases were also used to find the first optimal solutions to Rubik's Cube [10]. 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, making it the best-known combinatorial puzzle of all time. Each 3×3 plane of the cube can be rotated independently, and the task is to rearrange the individual pieces so that each side shows only one color (see Figure 3).

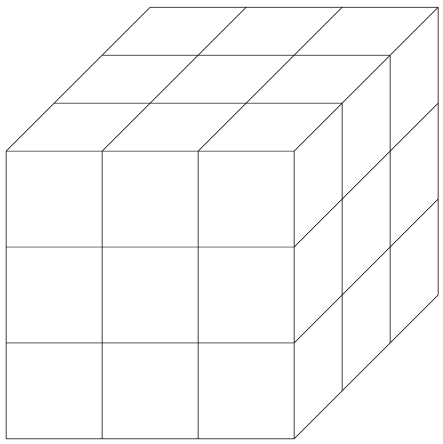

Figure 3: $3 \times 3 \times 3$ Rubik's Cube

The $3 \times 3 \times 3$ Rubik's Cube contains about 4.3252×10^{19} different reachable states. There are 20 movable subcubes, or *cubies*, which can be divided into eight *corner cubies*, with three faces each, and twelve *edge cubies*, with two faces each. There are $88,179,840$ different positions and orientations of the corner cubies, and the number of moves needed to solve just the corner cubies ranges from zero to eleven. At four bits per entry, a pattern database for the corner cubies can be stored in about 42 megabytes of memory. Six of the twelve edge cubies generate $42,577,920$ different possibilities, and a corresponding pattern database occupies about 20 megabytes of memory. The remaining six edge cubies generate another database of the same size.

Given a state of an IDA* search, we use the configuration of the corner cubies to compute an index into the corner-cubie pattern database, whose value tells us the number of moves needed to solve just the corner cubies. We also use each of the two sets of six edge cubies to compute indices into the corresponding edge-cubie databases, yielding the number of moves needed to solve each set of six edge cubies. Given these three different heuristic values, the best way to combine them, without overestimating actual solution cost, is to take their maximum, even though each cubie belongs to only one database. The reason is that every twist of the cube moves four edge cubies and four corner cubies, and moves that contribute to the solution of cubies in one pattern database may also contribute to the solution of the others. IDA*, using the maximum of the three pattern-database heuristic values described above, will find optimal solutions to random instances of Rubik's Cube [10]. The median optimal solution length is only 18 moves. One problem instance generated a trillion nodes, and required a couple weeks to run. With improvements by Michael Reid and Herbert Kociemba, larger pattern databases, and faster computers, most states can now be solved optimally in about an hour.

2.2.2 Limitations of Non-Additive Pattern Databases

The main limitation of non-additive pattern databases is that they can't solve larger problems. For example, since the Twenty-Four puzzle contains 25 different positions, a pattern database covering n tiles and the blank requires $25!/(25-n-1)!$ entries. A database of only six tiles and the blank would require over 2.2 billion entries. Furthermore, the values from a database of only six tiles would be smaller than the Manhattan distance of all the tiles. With multiple databases, the best way to combine them admissibly is to take the maximum of their values, even if the sets of tiles are disjoint. The reason is that non-additive pattern database values include all moves needed to solve the pattern tiles, including moves of other tiles.

Instead of taking the *maximum* of different pattern database values, we would like to be able to *sum* their values, to get a more accurate heuristic, without violating admissibility. This is the main idea of disjoint pattern databases.

3 Disjoint Pattern Databases

To construct a *disjoint pattern database* for the sliding-tile puzzles, we partition the tiles into disjoint groups, such that no tile belongs to more than one group. We then precompute tables of the minimum number of moves of the tiles in each group that are required to get those tiles to their goal positions. We call the set of such tables, one per group of tiles, a *disjoint* *pattern database*, or a *disjoint database* for short. Then, given a particular state in the search, for each group of tiles, we use the positions of those tiles to compute an index into the corresponding table, retrieve the number of moves required to solve the tiles in that group, and then add together the values for each group, to compute an overall heuristic for the given state. This value will be at least as large as the Manhattan distance, and usually larger, since it accounts for interactions between tiles in the same group.

The key difference between disjoint databases and the non-additive databases described above is that the non-additive databases include all moves required to solve the pattern tiles, including moves of tiles not in the pattern set. As a result, given two such databases, even if there is no overlap among their tiles, we can only take the maximum of the two values as an admissible heuristic, because moves counted in one database may move tiles in the other database, and hence these moves would be counted twice. In a disjoint database, we only count moves of the tiles in the group. While this idea is very simple, it eluded at least two groups of researchers who worked on this problem [1, 9].

A second difference between these two types of databases is that our disjoint databases don't consider the blank position, decreasing their size. A disjoint database contains the minimum number of moves needed to solve a group of tiles, for all possible blank positions.

Manhattan distance is a trivial example of a disjoint database, where each group contains only a single tile. While Manhattan distance was initially discovered by hand, it could also be "discovered" automatically as follows. For each tile in each position, perform a search until it reaches its goal location, in which all other tiles are indistinguishable. A state of this search is uniquely determined by the position of the tile in question and the position of the blank, and only moves of the tile of interest are counted. 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 set of tables which give, for each possible position of each tile, its Manhattan distance from its goal position. Since we only counted moves of the tile of interest, and each move only moves a single tile, we can sum the Manhattan distances to get an admissible heuristic.

Two non-trivial examples of disjoint databases for Fifteen Puzzle are shown in Figure 4, where we have divided the tiles into a group of seven and a group of eight. The seven-tile database contains $57,657,600$ entries, which range from 0 to 33 moves. The eight-tile database contains 518,918,400 entries, which range from 0 to 38 moves. In neither case is the blank position part of the index to the database. As a general rule, when partitioning the tiles, we want to group together tiles that are near each other in the goal state, since these tiles will interact the most with one another.

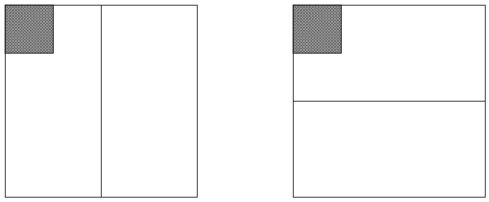

Figure 4: Disjoint Database for Fifteen Puzzle and its Reflection

4 Experimental Results

4.1 Fifteen Puzzle

We found all optimal solutions to 1000 random Fifteen Puzzle problem instances, using IDA* with a variety of heuristics. The average optimal solution length of these instances is 52.522 moves, and the average number of optimal solutions is 15.9. Table 1 shows the results. The first data column shows the average value of the heuristic function over the 1000 initial states. The second column gives the average number of nodes generated per problem instance to find the first optimal solution. The third column displays the average speed of the algorithm, in nodes per second, on a 440 MegaHertz Sun Ultra10 workstation. The fourth column indicates the average running time, in seconds, to find the first optimal solution. The last column gives the average number of nodes generated to find all optimal solutions to a problem instance.

The first row gives results for the Manhattan distance heuristic. The second row is for Manhattan distance enhanced by *linear conflicts*. Historically, the linear-conflict heuristic was the first significant improvement over Manhattan distance [5]. It applies to tiles in their goal row or column, but reversed relative to each other. For example, assume the top row of a given state contains the tiles (2 1) in that order, but in the goal state they appear

Heuristic Function	Value	Nodes	Nodes/sec	Seconds	All Solutions
Manhattan Distance	36.940	401,189,630	7,269,026	55.192	1,178,106,819
Linear Conflicts	38.788	40,224,625	4,142,193	9.710	144,965,491
Disjoint Database	44.752	136,289	2,174,362	.063	472,595
Disjoint+Reflected	45.630	36,710	1,377,630	.027	130,367

Table 1: Experimental Results on the Fifteen Puzzle

in the order (1 2). To reverse them, one of the tiles must move out of the top row, to allow the other tile to pass by, and then move back into the top row. Since these two moves are not counted in the Manhattan distance of either tile, two moves can be added to the sum of the Manhattan distances of these two tiles without violating admissibility. The same idea can be applied to tiles in their goal column as well. In fact, a tile in its goal position may participate in both a row and a column conflict simultaneously. Since the extra moves required to resolve a row conflict are vertical moves, and those required to resolve a column conflict are horizontal moves, both sets of moves can be added to the Manhattan distance, without violating admissibility. The linear-conflict heuristic reduces the number of nodes generated by an order of magnitude, at a cost of almost factor of two in speed, for an overall speedup of over a factor of five, compared to Manhattan distance.

The next two rows are for disjoint pattern database heuristics. The third row represents the heuristic which is the sum of the seven and eight-tile database values depicted on the left side of Figure 4. We used one byte per entry for efficiency, occupying a total of 550 megabytes, but these databases could be compressed. For example, we could store only the additional moves exceeding the Manhattan distances of the pattern tiles, and separately compute the Manhattan distances during the search. Furthermore, since the parity of the additional moves is the same as that of the Manhattan distance, we could store only half the number of additional moves, and multiply by two.

The fourth row represents a heuristic computed by starting with the heuristic of the third row. We then compute the sum of the seven and eight-tile database values shown on the right side of Figure 4. Finally, the overall heuristic is the maximum of these two sums. Since the two different partitions are reflections of one another, we use the same pair of tables for both databases, and simply reflect the tiles and their positions about the main diagonal to obtain the reflected values.

This last heuristic reduces the number of node generations by over four orders of magnitude, and the running time by a factor of over two thousand, compared to Manhattan distance. This comes at the cost of 550 megabytes of memory. By contrast, the best non-additive pattern database heuristic used by Culberson and Schaeffer [1], using a similar amount of memory, generated almost ten times more nodes then our best disjoint database, and ran only twelve times faster than simple Manhattan distance.

4.2 Twenty-Four Puzzle

Finding optimal solutions to the Twenty-Four Puzzle is only practical with our most powerful heuristics. We optimally solved fifty random problem instances with IDA*, using a disjoint pattern database heuristic, and its reflection about the main diagonal, shown in Figure 5. Each group consists of six tiles, requiring $127,512,000$ entries each. For a given state, the value from each database is the sum of the number of moves needed to solve all four groups of tiles from that state. The overall heuristic is the maximum of the values from the original and reflected databases. Since there are only two different-shaped groups, a 2×3 block of tiles, and an irregular block surrounding the blank, we only have to store two different tables, with the remaining values obtained by mapping the tiles and their positions into one of these two tables. The heuristic values for the 2×3 group range from 0 to 35 moves, and for the irregular group they range from 0 to 34 moves. At one byte per entry, the total amount of memory for both tables is 243 megabytes, but these tables could be compressed as well.

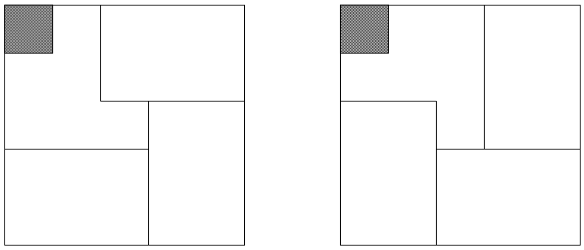

Figure 5: Disjoint Databases for TwentyFour Puzzle

Table 3 shows the initial state, length of the optimal solution, and number of nodes generated by IDA* to find the first solution to each problem instance. The initial state is represented by listing the tiles from left to right and top to bottom, with zero 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 of these 50 problems is 100.78 moves. The average number of nodes generated is $360,892,479,671$. The program is written in C, and generates about 2,110,000 nodes per second on a 440 MegaHertz Sun Ultra10 workstation. The running times on individual problems range from 18 seconds to almost 23 days, with an average of two days per problem. Using the analytic results developed in [12], we can predict that solving the Twenty-Four Puzzle with Manhattan distance alone would take an average of about 50,000 years per problem! The average Manhattan distance for a random sample of 10,000 initial states is 76.078 moves, while for our disjoint database heuristic it is 81.607 moves.

4.2.1 Comparison to Previous Results

Previously, the only program to find optimal solutions to the Twenty-Four Puzzle solved the first ten problems in table 3 [9], and differed from this in two important respects. First, the heuristic used in [9] was much more complex. Secondly, [9] used a technique, based on finite-state machines (FSMs), to prune duplicate nodes representing the same state arrived at via different paths in the graph [16]. FSM pruning reduced the number of nodes generated by IDA* on five of the problems by a factor that ranged from 2.4 to 3.6.

For this work, we did not use FSM pruning, because the technique is complex, and the results depend on the particular FSM used, making it difficult for other researchers to reproduce the same results. Thus, we solved all 50 problems without duplicate pruning, except for eliminating the parent of a node as one of its children.

No	Sol	FSM pruning	no FSM pruning	Disjoint Database	Speedup
1	95	18,771,430,922	67,189,320,726	2,031,102,635	21.764
4	98	83,573,198,724	234,662,490,010	10,991,471,966	14.046
5	100	8,110,532,608	19,865,967,282	2,899,007,625	4.508
6	101	221,769,436,018	745,218,119,072	103,460,814,368	4.739
8	108	82,203,971,683	211,917,514,087	116,202,273,788	1.199

Table 2: Comparison to Previous Results

Table 2 shows comparative results for the five easiest problems solved in [9], which we also solved using the same heuristic as in [9], but without duplicate pruning. The first column gives the corresponding problem number from Table 3, the second the length of the optimal solution, the third the number of node generations from [9] with FSM pruning, the fourth column the number of nodes generated using the same heuristic but without FSM pruning, and the fifth column gives the number of nodes generated with our disjoint pattern database heuristic without FSM pruning. The program of [9] generates about $3,207,000$ nodes per second without FSM pruning, compared to about $2,110,000$ nodes per second for our disjoint database program on the same machine. Taking this speed difference into account, the last column gives the speedup factor of our program over that in [9], without FSM pruning, which ranges from a factor of 1.2 to a factor of 21.8. Since these comparisons are based on the easiest problems for the program of [9], they may significantly underestimate the average speedup. On the other hand, the program of [9] uses very little memory, which is the main reason it runs faster, since it has much better cache performance.

5 Pairwise and Higher-Order Distances

The main drawback of disjoint database heuristics is that they don't capture interactions between tiles in different groups of the partition. This requires a different approach, developed independently by Gasser[4] and Korf[9]. Consider a table for a sliding-tile puzzle which contains for each pair of tiles, and each possible pair of positions they could occupy, the number of moves required of both tiles to move them to their goal positions. Gasser refers to this table as the *2-tile pattern database*, while we call these values the *pairwise distances*. For most pairs of tiles in most positions, their pairwise distance will equal the sum of their Manhattan distances. For some tiles in some positions however, such as two tiles in a linear conflict, their pairwise distance will exceed the sum of their Manhattan distances. Given n tiles, there are $O(n^4)$ entries in the complete 2-tile database, but only those pairwise distances that exceed the sum of the Manhattan distances of the two tiles need be stored. This table is only computed once for a given goal state.

Given a 2-tile database, and a state of the puzzle, we can't simply sum the database values for each pair of tiles to compute the heuristic, since each tile participates in many pairs, and this sum will grossly overestimate the optimal solution length. Rather, we must partition the n tiles into $n/2$ non-overlapping pairs, and then sum the pairwise distances for each of the chosen pairs. To get the most accurate admissible heuristic, we want a partition that maximizes the sum of the pairwise distances. For each state of the search, the corresponding partition may be different, requiring this computation to be performed for each heuristic evaluation.

For a given state, define a graph where each tile is represented by a node, and there is an edge between each pair of nodes, labelled with the pairwise distance of the corresponding tiles in that state. The task is to choose a set of edges from this graph so that no two chosen edges are incident to the same node, such that the sum of the labels of the chosen edges is maximized. This is called the *maximal weighted matching* problem, and can be solved in $O(n^3)$ time [14], where n is the number of nodes, or tiles in our case.

This technique can obviously be extended to triples of tiles, generating a 3-tile database, or even higher-order distances. Unfortunately, for even three tiles the corresponding three-dimensional matching problem is NP-Complete [3], as is higher-order matching. For the tile puzzles, however, if we only include tiles whose pairwise or triple distances exceed the sum of their Manhattan distances, this graph is very sparse, and the corresponding matching problem can be solved relatively efficiently.

The main advantage of this approach is that it can potentially capture more tile interactions, compared to a disjoint database that only captures interactions between tiles in the same group. Another advantage of this approach is its modest memory requirements. The 2- and 3-tile databases required only three megabytes of memory, compared to the 500 megabytes we used for the disjoint databases. The disadvantage of this approach is that computing the heuristic value of a given state requires solving a matching problem, which is much more expensive than simply adding the values for each group in a disjoint database.

Gasser implemented the 2-tile and 3-tile pattern database heuristics for the Fifteen Puzzle, and reported node expansions, but not actual running times. We performed similar experiments on both the Fifteen and Twenty-Four Puzzles. With the 2- and 3-tile databases, it took an average of five seconds to solve a random Fifteen Puzzle problem instance, and generated an average of $700,000$ nodes. This compares to 53 seconds for Manhattan distance but only 27 milliseconds for the disjoint databases, which both generated fewer nodes and incurred less overhead per node. For the Twenty-Four Puzzle, the 2- and 3-tile databases usually generated fewer nodes than the disjoint database. Again, however, since this heuristic is more complex to compute, it incurred a larger constant time per node, and thus the actual running time was greater than for the disjoint databases. Since the disjoint database heuristics are both simpler and perform better, the details of our experiments with the 2- and 3-tile databases are omitted here, but can be found in [2].

The performance difference between the disjoint databases and the 2- and 3-tile databases was greater for the Fifteen Puzzle than the Twenty-Four Puzzle, probably because we could store half the Fifteen Puzzle tiles in a single database, but only a quarter of the Twenty-Four Puzzle tiles. For these two problems, the disjoint database heuristics are both simpler and more effective, but they may not be for larger versions of the problem or other domains.

6 Summary, Conclusions, and Further Work

We have found optimal solutions to fifty random instances of the Twenty-Four Puzzle, a problem with almost 10^{25} states. The branching factor of the problem is 2.3676, and optimal solutions average about 100 moves. We also find optimal solutions to the Fifteen Puzzle in 27 milliseconds on average. This is by far the best performance on these problems to date.

To achieve this, we implemented IDA* with new admissible heuristic functions, based on pattern databases [1]. Rather than computing the costs of solving individual subgoals independently, a pattern database heuristic considers the costs of solving several subgoals simultaneously, taking into account the interactions between them. Culberson and Schaeffer [1] combined heuristics from different pattern databases by taking the maximum of their values. This is the most general approach, since the maximum of any two admissible heuristics is always another admissible heuristic.

We introduced disjoint pattern databases to permit the values from different databases to be added together, resulting in more accurate heuristic values. A disjoint pattern database partitions the set of subgoals into disjoint groups, and then adds together the costs of solving all the subgoals in each group. This requires that the groups be disjoint, and that a single operator only affect subgoals in a single group. For example, in the sliding-tile puzzle, each operator only moves one tile. This is just as efficient as taking the maximum of different values, but much more accurate, and still admissible.

Pattern database heuristics are more expensive to evaluate during search, mostly due to the latency of randomly accessing the large database in memory. This overhead is more than compensated for by the decrease in the number of nodes generated to solve a problem.

It remains to be seen how general this approach is to the discovery and implementation of admissible heuristic functions. The obvious next step is to apply it to other problems. All combinatorial problems involve solving multiple subgoals. This work suggests heuristics based on the simultaneous consideration of multiple subgoals, in such a way that their values can be added together to create a more accurate admissible heuristic.

7 Acknowledgements

This work was supported by NSF Grant IRI-9619447. Thanks to Eitan Yarden and Moshe Malin for their work on pairwise and triple distances.

References

[1] Culberson, J., and J. Schaeffer. Pattern Databases, *Computational Intelligence*, Vol. 14, No. 3, pp. 318-334, 1998.

[2] Felner, A., Improving Search Techniques and Using them on Different Environments, Ph.D. Thesis, Dept. of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, Israel, 2001.

[3] Garey, M.R., and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[4] Gasser, R. U., Harnessing Computational Resources for Efficient Exhaustive Search, Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1995.

[5] 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.

[6] 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.

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

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

[9] 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.

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

[11] Korf, R.E., Recent Progress in the Design and Analysis of Admissible heuristic functions, *Proceedings of AAAI-2000*, Austin, TX, July, 2000, pp. 1165-1170.

[12] Korf, R.E., M. Reid, and S. Edelkamp, Time complexity of iterative-deepening-A*, *Artificial Intelligence*, Vol 129, No. 1-2, June 2001, pp. 199-218.

[13] Loyd, S., *Mathematical Puzzles of Sam Loyd*, selected and edited by Martin Gardner, Dover, New York, 1959.

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

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

[16] Taylor, L., and R.E. Korf, Pruning duplicate nodes in depth-first search, *Proceedings of AAAI-93*, Washington D.C., July 1993, pp. 756-761.

No	Initial State	Sol	Nodes
1	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	95	2,031,102,635
2	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	96	211,884,984,525
3	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	97	21,148,144,928
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	98	10,991,471,966
5	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	100	2,899,007,625
6	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	101	103,460,814,368
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	104	106,321,592,792
8	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	108	116,202,273,788
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	113	1,818,005,616,606
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	114	1,519,052,821,943
11	15 11 8 18 14 3 19 16 20 5 24 2 17 4 22 10 1 13 9 21 23 7 6 12 0	106	1,654,042,891,186
12	12 23 9 18 24 22 4 0 16 13 20 3 15 6 17 8 7 11 19 1 10 2 14 5 21	109	624,413,663,951
13	21 24 8 1 19 22 12 9 7 18 4 0 23 14 10 6 3 11 16 5 15 2 20 13 17	101	1,959,833,487
14	24 1 17 10 15 14 3 13 8 0 22 16 20 7 21 4 12 9 2 11 5 23 6 18 19	111	1,283,051,362,385
15	24 10 15 9 16 6 3 22 17 13 19 23 21 11 18 0 1 2 7 8 20 5 12 4 14	103	173,999,717,809
16	18 24 17 11 12 10 19 15 6 1 5 21 22 9 7 3 2 16 14 4 20 23 0 8 13	96	3,803,445,934
17	23 16 13 24 5 18 22 11 17 0 6 9 20 7 3 2 10 14 12 21 1 19 15 8 4	109	367,150,048,758
18	0 12 24 10 13 5 2 4 19 21 23 18 8 17 9 22 16 11 6 15 7 3 14 1 20	110	987,725,030,433
19	16 13 6 23 9 8 3 5 24 15 22 12 21 17 1 19 10 7 11 4 18 2 14 20 0	106	218,284,544,233
20	4 5 1 23 21 13 2 10 18 17 15 7 0 9 3 14 11 12 19 8 6 20 24 22 16	92	312,016,177,684
21	24 8 14 5 16 4 13 6 22 19 1 10 9 12 3 0 18 21 20 23 15 17 11 7 2	103	724,024,589,335
22	7 6 3 22 15 19 21 2 13 0 8 10 9 4 18 16 11 24 5 12 17 1 23 14 20	95	3,592,980,531
23	24 11 18 7 3 17 5 1 23 15 21 8 2 4 19 14 0 16 22 6 9 13 20 12 10	104	171,498,441,076
24	14 24 18 12 22 15 5 1 23 11 6 19 10 13 7 0 3 9 4 17 2 21 16 20 8	107	357,290,691,483
25	3 17 9 8 24 1 11 12 14 0 5 4 22 13 16 21 15 6 7 10 20 23 2 18 19	81	292,174,444
26	22 21 15 3 14 13 9 19 24 23 16 0 7 10 18 4 11 20 8 2 1 6 5 17 12	105	12,397,787,391
27	9 19 8 20 2 3 14 1 24 6 13 18 7 10 17 5 22 12 21 16 15 0 23 11 4	99	53,444,360,033
28	17 15 7 12 8 3 4 9 21 5 16 6 19 20 1 22 24 18 11 14 23 10 2 13 0	98	2,258,006,870
29	10 3 6 13 1 2 20 14 18 11 15 7 5 12 9 24 17 22 4 8 21 23 19 16 0	88	4,787,505,637
30	8 19 7 16 12 2 13 22 14 9 11 5 6 3 18 24 0 15 10 23 1 20 4 17 21	92	1,634,941,420
31	19 20 12 21 7 0 16 10 5 9 14 23 3 11 4 2 6 1 8 15 17 13 22 24 18	99	26,200,330,686
32	1 12 18 13 17 15 3 7 20 0 19 24 6 5 21 11 2 8 9 16 22 10 4 23 14	97	428,222,507
33	11 22 6 21 8 13 20 23 0 2 15 7 12 18 16 3 1 17 5 4 9 14 24 10 19	106	1,062,250,612,558
34	5 18 3 21 22 17 13 24 0 7 15 14 11 2 9 10 1 8 6 16 19 4 20 23 12	102	481,039,271,661
35	2 10 24 11 22 19 0 3 8 17 15 16 6 4 23 20 18 7 9 14 13 5 12 1 21	98	116,131,234,743
36	2 10 1 7 16 9 0 6 12 11 3 18 22 4 13 24 20 15 8 14 21 23 17 19 5	90	2,582,008,940
37	23 22 5 3 9 6 18 15 10 2 21 13 19 12 20 7 0 1 16 24 17 4 14 8 11	100	1,496,759,944
38	10 3 24 12 0 7 8 11 14 21 22 23 2 1 9 17 18 6 20 4 13 15 5 19 16	96	38,173,507
39	16 24 3 14 5 18 7 6 4 2 0 15 8 10 20 13 19 9 21 11 17 12 22 23 1	104	161,211,472,633
40	2 17 4 13 7 12 10 3 0 16 21 24 8 5 18 20 15 19 14 9 22 11 6 1 23	82	65,099,578
41	13 19 9 10 14 15 23 21 24 16 12 11 0 5 22 20 4 18 3 1 6 2 7 17 8	106	26,998,190,480
42	16 6 20 18 23 19 7 11 13 17 12 9 1 24 3 22 2 21 10 4 8 15 14 5 0	108	245,852,754,920
43	7 4 19 12 16 20 15 23 8 10 1 18 2 17 14 24 9 5 0 21 6 3 11 13 22	104	55,147,320,204
44	8 12 18 3 2 11 10 22 24 17 1 13 23 4 20 16 6 15 9 21 19 5 14 0 7	93	867,106,238
45	9 7 16 18 12 1 23 8 22 0 6 19 4 13 2 24 11 15 21 17 20 3 10 14 5	101	79,148,491,306
46	1 16 10 14 17 13 0 3 5 7 4 15 19 2 21 9 23 8 12 6 11 24 22 20 18	100	65,675,717,510
47	21 11 10 4 16 6 13 24 7 14 1 20 9 17 0 15 2 5 8 22 3 12 18 19 23	92	30,443,173,162
48	2 22 21 0 23 8 14 20 12 7 16 11 3 5 1 15 4 9 24 10 13 6 19 17 18	107	555,085,543,507
49	2 21 3 7 0 8 5 14 18 6 12 11 23 20 10 15 17 4 9 16 13 19 24 22 1	100	108,197,305,702
50	23 1 12 6 16 2 20 10 21 18 14 13 17 19 22 0 15 24 3 7 4 8 5 9 11	113	4,156,099,168,506

Table 3: Twenty-Four Puzzle Data