Heuristic Search

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90095

October 2, 2006

Contents

Chapter 6

Design of Heuristic Functions

Chapters 3, 4, and 5 described algorithms that use heuristic evaluation functions to estimate the cost of reaching a goal from a given state. Here we address the question of how to design such heuristic functions.

6.1 Heuristics from Relaxed Models

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem[71]. As we will see below, heuristics derived in this way are both admissible and consistent.

6.1.1 Examples

For example, consider the problem of navigating in a network of roads from an initial location to a goal location. A good heuristic to estimate the cost in the road network between two points is the Euclidean or straight-line distance between the points, which can be computed in constant time. If we remove the constraint that we have to travel along the roads, and allow unrestricted travel between any two points, we get a relaxation of the original problem, which we might call helicopter navigation. The exact cost of an optimal solution to a helicopter navigation problem is the Euclidean distance between the two points. Thus, the exact cost of solving the simplified helicopter navigation problem is a good heuristic function for the original road navigation problem.

As another example, consider the travelling salesman problem (TSP). We can describe a legal TSP tour as a graph having three properties: 1) it covers every node, 2) every node has degree two, an edge entering the node, and an edge leaving it, and 3) the graph is connected. In addition, an optimal TSP tour is a graph of lowest cost that satisfies the above properties. If we delete any of these three constraints, we get a relaxed or simplified problem. In two cases, the optimal solution cost of the simplified problem is a good heuristic function for TSP.

For example, if we remove the constraint that every node has degree two, this leaves the requirement of a connected graph that covers all the nodes. This is called a spanning graph. The optimal solution to this simplified problem is a spanning tree of lowest cost, or a minimum spanning tree (MST). A minimum spanning tree of n nodes can be computed in $O(n^2)$ time, and can be used as a heuristic function for the TSP, for which all known optimal algorithms have exponential complexity.

Alternatively, if we remove the constraint that the graph be connected, we are left with a graph that covers every node, and every node has degree two. Each connected set of nodes in such a graph form a simple loop or cycle, but there can be multiple cycles that aren't connected to each other. Another way to view such a graph is that every city has another city assigned to it, the one that follows it in its cycle. Finding a lowest-cost set of such assignments is known as the assignment problem. Its exact solution can be computed in $O(n^3)$ time for n nodes[67]. The solution to the assignment problem is a heuristic function for TSP. It is particularly effective for the asymmetric TSP (ATSP), in which the cost of the edge from city i to city j is not necessarily the same as the cost of the edge from city j to city i.

If we delete the constraint that the graph has to cover every node, then the optimal solution is the empty graph, which has cost zero and hence is not an effective heuristic function.

Finally, consider a sliding-tile puzzle. One of the constraints on this problem is that a tile can only be slid into the blank position. If we remove this restriction, we allow any tile to be moved to any horizontally or vertically adjacent position. Note that this allows multiple tiles to occupy the same position. The optimal solution cost for an instance of this relaxed problem is simply the Manhattan distance, since every tile must be moved its Manhattan distance to its goal location, and each move can only move one tile to an adjacent position. Thus, once again we see that the optimal solution cost to a simplified version of a problem can be used as a heuristic evaluation function for the original problem.

6.1.2 The STRIPS Problem Formulation

Can we formalize this idea, and derive such heuristics automatically? To some extent, the answer is yes. Since generating a simplified problem involves manipulating the entire description of a problem, rather than simply individual states, we need a formal problem-description language that is richer than the problem-space graph.

One such language is called the *STRIPS* representation[24]. In this representation, the current state of a problem is described by a set of atomic formulas in predicate calculus. For a sliding-tile puzzle, for example, we can give the location of each tile by a set of predicate formulas of the form $On(x, y)$, meaning that tile x is in location y. In addition, we can specify the position of the blank or empty position by a predicate of the form $Clear(z)$, where z is a particular location. Finally, the structure of the tile locations, and in particular their adjacency, can be specified by a set of predicates of the form $Adj(y, z)$, meaning that location y is horizontally or vertically adjacent to location z. Note that the adjacency predicates *Adj* are fixed for a given size puzzle, but the *On* and *Clear* predicates will differ depending on the current state.

In addition to describing a state of the problem, we also have to describe the legal operators. This is done with a predicate of the form $Move(x, y, z)$, meaning that it is legal to move tile x from location y to location z. In order to define which operators are legal in a given state, and their effect on that state, there are three lists associated with each operator predicate: a *precondition list*, an *add list*, and a *delete list*.

The precondition list tells us what conditions must be true in order for $Move(x, y, z)$ to be a legal move. In particular, tile x must be in location y, $On(x, y)$, location z must be empty, $Clear(z)$, and locations y and z must be adjacent, $Adj(y, z)$. The add list tells us what predicates that were not true before the operator was applied will be true after it is applied. In particular, tile x will be in location z, $On(x, z)$, and location y will be empty, $Clear(y)$. Finally, the delete list tells us what predicates that were true before the operator is applied will no longer be true after it is applied. In particular, tile x will no longer be in location y, $On(x, y)$, and location z will no longer be empty, $Clear(z)$. In general, the delete list will be a subset of the precondition list.

The idea here is that the state is described by a set of predicates that are true at a given point in time. The precondition list is compared to this set to determine what operators can be legally applied, and the application of an operator adds and deletes predicates from the list of true predicates. All predicates that were true before an operator is applied, and that are not on the add or delete lists of the operator, continue to be true.

Once we have formulated our problem in the STRIPS representation, constructing simplified or relaxed problems is easy. We simply remove preconditions on the operators from the precondition list. For example, if we remove the $Clear(z)$ precondition, we can move any tile to any adjacent location, regardless of the position of the blank. This is the relaxed problem described above, for which the exact cost of an optimal solution is the Manhattan distance.

6.1.3 Admissibility and Consistency

The heuristics derived by this method are both admissible and consistent. To see why, consider the effect of removing operator preconditions on the problem-space graph. All the operators that were applicable before the precondition deletion are still applicable. Thus, all previous edges still exist in the new graph. In addition, by deleting some preconditions, we add new operator applications, which is equivalent to adding new edges to the graph. Furthermore, in some cases we add new states as well. For example, by removing the precondition that the destination location of a tile be empty, we effectively add new states where multiple tiles occupy the same location. The key property, however, is that we don't remove any states or edges from the original graph, but simply add new ones.

To see that the resulting heuristic functions are admissible, note that the heuristic is the lowest cost of any path between two states in the problem-space graph of the relaxed problem, while the exact solution cost is the lowest cost of any path between the states in the problem-space graph of the original problem. Since we only add new states and operators in going from the original graph to the graph of the relaxed problem, the lowest-cost path in the new graph can only have an equal or lower cost than the lowest-cost path in the original graph. Thus, the heuristic function is admissible, or a lower bound on the optimal solution cost for the original problem.

To see that the heuristic is consistent as well, note that the value of the heuristic is the actual cost of the best solution path in the graph of the relaxed problem. Recall that a heuristic h is consistent if and only if for all nodes n and m, $h(n) \leq c(n, m) + h(m)$, where $c(n, m)$ is the cost of an optimal path from n to m. $h(n)$ is the actual optimal cost of reaching a goal from node n in the graph of the relaxed problem. This cost has to be less than or equal to the cost of going from n to any other node m, plus the actual cost of reaching a goal from node m, with equality for any nodes m on an optimal path to a goal. Thus, any heuristics based on the exact cost of optimal solutions of relaxed problems are guaranteed to be both admissible and consistent.

6.1.4 Automatically Deriving such Heuristics

Given a STRIPS representation of a problem, it is easy to create a description of a relaxed problem, simply by deleting preconditions. While this step is easily automated, automating the entire process of generating heuristic functions is still quite difficult. The hard part is identifying which relaxed problems have the property that their exact solution costs can be efficiently computed, and deriving an efficient algorithm to perform the computation. For example, once we remove the $Clear(z)$ condition from the above description of the sliding-tile puzzles, it is not immediately obvious, at least not to a computer, that the cost of an exact solution to the resulting relaxed problem is easy to compute, any more than it is easy to compute the exact cost of an optimal solution to the original problem. Furthermore, automatically figuring out how to compute what we call Manhattan distance efficiently is also a challenge. The state of the art in this area is represented by [76].

6.2 Pattern Database Heuristics

The Manhattan distance heuristic can be efficiently computed because of the assumption that the individual tiles do not interact with one another. The reason the sliding-tile puzzle is difficult, and why the Manhattan distance is only a lower bound on actual 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.

6.2.1 Non-Additive Pattern Databases

Pattern databases[13] are one way to do this, and were originally applied to the Fifteen Puzzle. Figure 6.1 shows the Fifteen Puzzle, with a subset of the tiles labelled, called the *fringe* tiles. For a given state, the minimum number of moves required to get the fringe tiles from their current locations to their goal positions, including any required moves of other tiles, is obviously a lower bound on the number of moves needed to solve the entire puzzle.

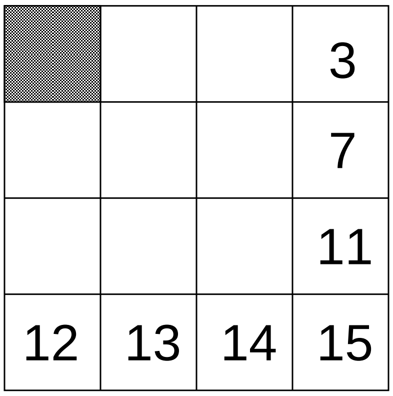

Figure 6.1: The Fringe Pattern for the Fifteen Puzzle

This number depends only on the positions of the fringe tiles, and the blank position, but is independent of the positions of the other tiles. While it would be too expensive to calculate the moves needed to solve the fringe tiles for each state in the search, we can precompute all of these values, store them in memory in a pattern database, and look them up when they are needed during the search. Since there are seven fringe tiles and one blank, and sixteen different locations, the total number of different possible configurations of these tiles and blank is $16!/(16-8)! = 518,918,400$. For each table entry, we store the number of moves needed to solve the fringe tiles from their corresponding locations, which takes less than a byte of storage. Thus, we can store the whole 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 6.1. In this search, the unlabelled tiles are all considered equivalent, and a state is uniquely determined by the positions of the labelled 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 database are filled. Note that this table need only be computed once for a given goal state, and its cost can be amortized over the solution of multiple problem instances with that same goal state.

Once the table is stored, we can 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, is used as the heuristic value for that state.

Using this pattern database, Culberson and Schaeffer 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 six, compared to Manhattan distance [13]. 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.

Rubik's Cube

Non-additive pattern databases were also used to find the first optimal solutions to Rubik's Cube. The standard $3\times3\times3$ Rubik's Cube, shown in Figure 1.1, contains about 4.3252×10^{19} different reachable states. Of the 27 possible $1 \times 1 \times 1$ cubies, 26 are visible, with one in the center. Of these, the six cubies in the center of a face rotate, but don't move. These six cubies form a fixed reference framework, disallowing rotations of the entire cube. Of the remaining 20 movable cubies, eight are on the corners, with three visible faces each, and twelve are on the edges, with two visible faces each. Corner cubies stay on the corners, and edge cubies stay on the edges. A corner cubie can be oriented in any of three ways, and an edge cubie can be oriented two different ways. Each twist moves four corner cubies and four edge cubies.

The obvious heuristic for Rubik's Cube is a three-dimensional version of Manhattan distance. For each cubie, compute the minimum number of moves required to correctly position and orient it, and sum these values over all cubies. Unfortunately, to be admissible, this value has to be divided by eight, since every twist moves eight cubies. A better heuristic is to take the maximum of the sum of the Manhattan distances of the corner cubies, divided by four, and the Manhattan distances of the edge cubies, divided by four. The expected value of the Manhattan distance of the edge cubies is $22/4 = 5.5$, while the corresponding value for the corner cubies is $12.333/4 \approx 3.08$, partly because there are twelve edge cubies, but only eight corner cubies.

If we consider just the eight corner cubies, they can be in any one of $88,179,840$ different possible permutations and orientations. Using a breadth-first search from the goal state, we can enumerate these states, and record in a table the number of moves required to solve each combination of corner cubies. Since this value ranges from 0 to 11, only four bits are required for each table entry. Thus, this table requires $44,089,920$ bytes of memory, or 42 megabytes, which is easily accommodated on modern workstations. Note that this table only needs to be computed once for each goal state, and its cost can be amortized over the solution of multiple problem instances with the same goal.

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

We can improve this heuristic by considering the edge cubies as well.

A single pattern database containing the number of moves required to solve all twelve edge cubies requires too much memory, but we can store pattern databases for subsets of the edge cubies. The number of possible combinations for six of the twelve edge cubies is $42,577,920$. The number of moves needed to solve them ranges from 0 to 10, with an expected value of about 7.668 moves. At four bits per entry, this table requires $21,288,960$ bytes, or 20 megabytes. Similarly, we can compute the corresponding pattern database for the remaining six edge cubies.

Unfortunately, the only way to combine these different pattern database heuristics, without overestimating the actual solution cost, is to take the maximum of their individual values. 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. The heuristic used for the experiments reported below is the maximum of these three values: all eight corner cubies, and two groups of six edge cubies each. The total amount of memory for all three tables is 82 megabytes. The total time to generate the heuristic tables was about an hour, and this cost is amortized over multiple problem instances with the same goal state. The expected value of the maximum of these three heuristics is about 8.898 moves. Even though this is only a small increase above the 8.764 expected value for just the corner cubies, it results in a significant performance improvement, because the heuristic distribution is more favorable.

IDA*, using the maximum of the three pattern database values described above as the heuristic, was used to find optimal solutions to ten random instances of Rubik's Cube[46]. The median optimal solution length is only 18 moves. One problem instance generated a trillion nodes, and required a couple of weeks to run. With improvements by Michael Reid and Herbert Kociemba, and larger pattern databases, most states can now be solved optimally in a day.

Limitations of Non-Additive Pattern Databases

The main limitation of non-additive pattern databases is that they don't scale up to larger problems. For example, the state space of the Twenty-Four Puzzle is more than 100,000 times larger than that of

Rubik's Cube. Since the Twenty-Four puzzle contains 25 different positions, a pattern database covering n tiles and the blank would require $25!/(25-n+1)!$ entries. For example, five tiles plus the blank generate almost 128 million possibilities. A pattern database of six tiles and the blank would require over 2.3 gigabytes. Furthermore, the values from a database covering only six tiles are smaller than the simple Manhattan distance of all the tiles. Even with multiple databases, the best way to combine them admissibly is to take the maximum of their values, even if the tiles in the different databases don't overlap. 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 idea of disjoint pattern databases.

6.2.2 Disjoint Pattern Databases

The easiest way to construct an additive pattern database for the sliding-tile puzzles is to 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 solve those tiles. We call the set of such tables, one per group of tiles, a *disjoint additive pattern database*, or a *disjoint database* for short[48]. Then, given a particular state in the search, for each group of tiles, we use the positions of the tiles in the group 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.

The key difference between these databases and the non-additive databases described in [13], is that their databases include all moves required to solve the pattern tiles, including moves of other 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 in an admissible heuristic, because moves counted in one database may be moving 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 seems remarkably simple, it eluded at least two groups of researchers who worked on this problem [13, 48].

A second difference between these two types of databases is that we don't consider the blank position, decreasing the size of the databases. A disjoint database contains the minimum number of moves over all possible blank positions. While this could also be done in a non-additive database, the position of the blank is less important in a disjoint pattern database.

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 by the following process. 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.

A non-trivial example of a disjoint database for Fifteen Puzzle is shown in Figure 6.2, 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. The eight-tile database contains $518,918,400$ entries, which range from 0 to 38. 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 are most likely to interact with one another.

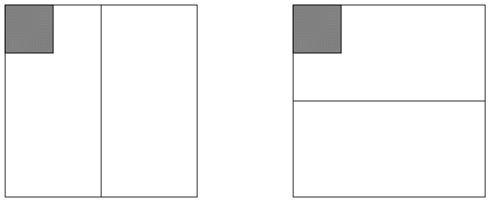

Figure 6.2: Disjoint Database for Fifteen Puzzle and its Reflection

Experiments on Fifteen Puzzle

As a first test, optimal solutions to 1000 random Fifteen Puzzle problem instances were found, using IDA* with a variety of heuristics. The average optimal solution length for these problems is 52.522 moves. Table 6.1 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. The third column displays the average speed of the algorithm, measured in nodes per second, on a 440 MegaHertz Sun Ultra10 workstation. The fourth column indicates the average running time, in seconds, to optimally solve a problem instance.

Heuristic Function	Value	Nodes	Nodes/sec	Seconds
Manhattan Distance	36.940	401,189,630	7,269,026	55.192
Linear Conflicts	38.788	40,224,625	4,142,193	9.710
Disjoint Database	44.752	136,289	2,174,362	.063
Disjoint+Reflected	45.630	36,710	1,302,233	.028

Table 6.1: Experimental Results on the Fifteen Puzzle

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[31]. 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 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 line of the table is for a heuristic which is the sum of the seven and eight-tile database values depicted on the left side of Figure 6.2. 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 1/2 the number of additional moves.

The heuristic for the fourth line is computed by first computing the heuristic of the third line. We then compute the sum of the seven and eight-tile database values shown on the right side of Figure 6.2. Finally, the overall heuristic is the maximum of these two different sums. Since the two different partitions in Figure 6.2 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.

The disjoint database heuristics provide dramatic reductions both in nodes generated and overall running time. The maximum of the original and reflected database values, shown in the fourth line, reduce the number of node generations by a factor of almost eleven thousand, and the running time by a factor of almost two thousand, compared to Manhattan distance. By contrast, the best non-additive pattern database heuristic used by Culberson and Schaeffer[13], using a similar amount of memory, generated almost 10 times more nodes then the disjoint database, and ran only 12 times faster than simple Manhattan distance.

Experiments on Twenty-Four Puzzle

Finding optimal solutions to the Twenty-Four Puzzle is so expensive that it is only practical with the most powerful heuristics. The heuristic used was based on a disjoint pattern database, and its reflection about the main diagonal, shown in Figure 6.3. There is nothing special about this decomposition of the problem, but we chose to group together tiles that are close to each other in the goal state, in the belief that such tiles are more likely to interact with one another. 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 each group 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 database range from 0 to 35, and for the irregular database they range from 0 to 34. At one byte per entry, the total amount of memory for both tables is 243 megabytes, but these could be compressed as well.

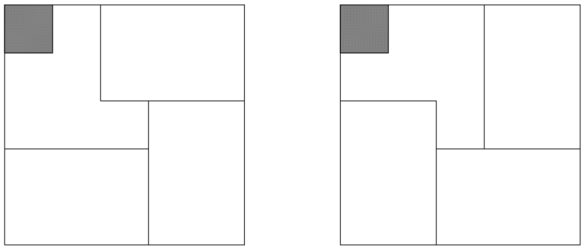

Figure 6.3: Disjoint Databases for TwentyFour Puzzle

Fifty random problem instances were solved optimally with IDA*, using this heuristic function. The average optimal solution length is 100.78 moves. The average number of nodes generated was $360,892,479,671$. The program was 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 ranged from 18 seconds to almost 23 days, with an average of two days per problem. The average Manhattan distance for a random sample of 10,000 initial states is 76.078 moves, while for the disjoint database heuristic it is 81.607 moves.

6.2.3 Pairwise and Higher-Order Distances

The main drawback of disjoint databases is that they don't capture interactions between tiles in different groups of the partition. This requires a different approach. 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. We call these values the *pairwise distances* of the tiles. For most pairs of tiles in most positions, their pairwise distance will equal the sum of their Manhattan distances. For some tiles is 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 a complete pairwise distance table. As with all pattern databases, this table is only computed once for a given goal state.

Given a complete pairwise distance table, and a given state of the puzzle, how do we compute the heuristic value for that state? We can't simply sum the table values for each pair of tiles, since each tile participates in many pairs, and the sum of all values will grossly overestimate the optimal solution length. Rather, we partition the n tiles into $n/2$ non-overlapping pairs, and then sum the pairwise distances for the pairs we chose. In order to get the most accurate admissible heuristic, we want to choose a set of non-overlapping pairs for which the sum of their pairwise distances is maximized. For each state of the search, the corresponding set of pairs may be different, requiring this computation to be performed for each heuristic evaluation.

To understand this problem more generally, 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 then 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 weights of the edges is maximized. This is called the *maximal weighted matching* problem, and can be solved in $O(n^3)$ time[67], where n is the number of nodes, or tiles in this case.

This technique can obviously be extended to triples of tiles, or even higher-order distances. Unfortunately, for triples the corresponding three-dimensional matching problem is NP-Complete [27], as is higher-order matching. This idea is the basis of the heuristic function used in [43], although it was implemented in a domain-specific fashion.

The advantage of this approach is that it produces more accurate heuristic values, and hence fewer node generations, compared to disjoint pattern databases. The reason is that if two tiles are in different groups in a disjoint database, their interactions are not captured by the disjoint database, but they can be captured by a pairwise-distance pattern database. The disadvantage of this approach is that to compute the heuristic value of each state in the search requires solving a matching problem, which is much more expensive than simply adding the values for each group in a disjoint database.

Extensive experiments were performed with pairwise distances and triple distances, on both the Fifteen and Twenty-Four puzzles. These heuristics resulted in fewer node generations, but the overhead of computing them resulted in longer running times, compared to the disjoint databases.