# Early Work on Optimization-Based Heuristics for the Sliding Tile Puzzle

**Ariel Felner**
ISE Department
Ben-Gurion University
Israel
felner@bgu.ac.il

## Abstract

Optimization-based heuristics may offer very good estimates. But, calculating them may be time consuming, especially if the optimization problem is intractable. This raises the question of their applicability. This paper summarizes early work from the year 2000 on optimization-based heuristics in the context of PDBs for the Tile-Puzzle. We show that an admissible heuristic based on Vertex-Cover (VC) can be calculated in reasonable time over a large collection of small PDBs. When larger PDBs are involved we suggest the idea of using another lookup table that precalculates and stores all possible relevant VC values. This table can be later looked up in a constant time during the search. We discuss the conditions under which this idea can be generalized. Experimental results demonstrate the applicability of these two ideas on the 15- and 24-Puzzle. The first idea appeared in (Felner, Korf, and Hanan 2004) but the second idea is presented here for the first time.

## Introduction and overview

The main difference between *informed search* and *uninformed search* is the usage of *heuristic functions* that guide the search towards the goal while avoiding non-promising paths. Informed search algorithms such as A* (Hart, Nilsson, and Raphael 1968) and and its variants are guided by the cost function $f(n) = g(n)+h(n)$, where $g(n)$ is the cost of the current path from the start node to node $n$ and $h(n)$ is a heuristic function estimating the cost from $n$ to a goal node. If $h(n)$ is *admissible* (i.e., is always a lower bound) these algorithms are guaranteed to find optimal paths. More informed heuristics (i.e., with tighter lower bounds) will find the goal faster but this imposes a tradeoff between the quality of the heuristic and CPU time as more informed heuristic take longer to compute. Thus, when developing a heuristic one should balance between accuracy of the heuristic and the running time of its calculation.

One of the topics of this workshop is *optimization-based search heuristics*. In the past few years there has been a significant thread of papers on such heuristics, mainly in the area of domain-independent cost-optimal planning. A number of researchers show that it is feasible and beneficial to obtain heuristic estimates by using linear programming at the nodes seen during the search (Pommerening et al. 2014; Seipp and Helmert 2014; Pommerening, Röger, and Helmert 2013; Katz and Domshlak 2010; Bonet and van den Briel 2014).[1] Constraints are written to guarantee admissibility and the purpose of the linear program is to find a strong heuristic that satisfies all the constraints. Linear programming can be solved rather efficiently, but the time overhead for doing this at every node is still a concern.

This concern is more severe when calculating an admissible heuristic for a node requires solving an optimization problem which is NP-hard. Even if the obtained heuristic is very accurate and the number of generated nodes is rather small, the execution time of the search process might be significantly weakened by the large overhead needed to solve the optimization problem. One way of remedying this is to relax the optimization problem such that it can be solved in polynomial time at a tradeoff of a weaker heuristic. For example, this was done for the *Hitting Set* problem (Bonet and Helmert 2010).

In this paper I would like to show that there are circumstances where this undesirable situation of large overhead due to the optimization problem does not occur. Despite the fact that an NP-hard problem should be solved at each node, the time overhead for doing this may be reasonable given the savings produced by providing a strong heuristic.

For showing this I would like to go back in time to the year 2000. In these early days of working on PDBs, the Sliding-tile Puzzle (shown in Figure 1) was a dominant research domain. This puzzle is very easy to describe and encode but on the other hand it provides a very large state-space. Thus, it serves as a great experimental domain for search problems and heuristics for decades.

The main question on the table was how to compute an admissible $h$-value from a number of PDBs. One idea that emerged in that research was that of *disjoint additive PDBs* (Korf and Felner 2002; Felner, Korf, and Hanan 2004) (referred to here as *plain additivity* (PA)). We showed that if the PDBs are disjoint, their values can be admissibly added. Nevertheless, during that time we tried other ideas for gathering information from PDBs.

In another direction we built PDBs for *all* possible pairs, triples and quadruples of tiles. The tiles were transformed



[1] A survey can be found in (Pommerening et al. 2014).

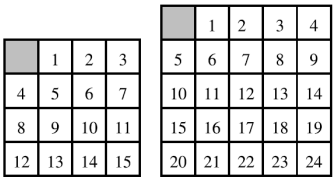

Figure 1: The 15- and 24-Puzzles

into vertices of a hypergraph and the PDB values for a given search node were transformed to weights of hyperedges. Then, we solved the Maximal Matching or Minimal (weighted) Vertex-Cover on this hypergraph to find admissible heuristics. In this paper we denote this method as the Weighted Vertex Cover method (WVC). It turns out that the time needed for calculating WVC at every node was not a limiting factor. In a third direction, we generate a small number of large but overlapping (non disjoint) PDBs. Here, we again want to calculate the weighted vertex cover for every node but doing this was time consuming because the PDBs were large. We overcame this with the following method. Similar to PDBs, the WVC for all relevant possibilities of weights is calculated in a preprocessing phase and stored in a lookup-table. Then, during the search, WVC values can be retrieved directly from this table in constant time. We denote this method as the *vertex cover table* method (VCT).

The PA idea is simple to grasp and implement and it provided a tremendous speedup over previous approaches for optimally solving the puzzle. Thus, it received much attention and appeared in later publications. The WVC and VCT methods are more complex and achieved moderate or no speedup over PA in our experiments. The WVC method was described in (Felner, Korf, and Hanan 2004) but is less remembered. The VCT method was never published except in my Ph.D dissertation (Felner 2002). Nevertheless, since there is a great interest now in optimization-based heuristics, this paper summarizes these early directions and introduces VCT for the first time. As we show below, VCT outperformed both PA and WVC in some cases. More important, when applicable, the idea of VCT might be used by others who solve hard or complex optimization problems for calculating heuristics. Both WVC and VCT show that when used with caution, NP-hard problems can be fully exploited.

A paper very related to WVC is titled: "Getting the Most Out of Pattern Databases for Classical Planning" (Pommerening, Röger, and Helmert 2013). Both papers share a number of core ideas but the later paper describes them in a general way that may fit domain independent planning.

We now turn describe our three directions (PA, WVC and VCT). Some parts of the text were modified from (Felner, Korf, and Hanan 2004; Felner 2002). All the experiments reported here were performed during the year 2000.

## Direction 1: Plain additive PDBs

Plain additive PDBs (PA or PA-PDBs) were called *statically-partitioned additive pattern databases* or *disjoint pattern databases* and were introduced in (Korf and Felner 2002). To construct a PA-PDB for the Tile Puzzle, we partition the tiles into disjoint groups. For each group we precompute the minimum number of moves that are required to get the tiles in the group from all possible locations to their goal locations. This information is stored in a lookup table - the PDB. Then, given a node $s$ in the search, for each group of tiles, we use the locations of those tiles to compute an index into the corresponding PDB and retrieve the number of moves required to solve the tiles in that group. Since the groups are disjoint and every operator only moves a single tile we can then *add* together the values from the different groups, to compute an admissible heuristic for $s$. This value will be *larger than or equal to* the Manhattan Distance (MD) of the state since it also accounts for interactions between tiles in the same group.

MD can be easily computed for every node. Thus, for simplicity and efficiency, PA-PDBs may only store additions above MD and MD can be added to the heuristic on the fly. We use such PDBs in the rest of this paper. If the puzzle is reflected about the main diagonal then we get a symmetric puzzle. We showed that given a set of disjoint PDBs, the same PDBs can be used for the reflected partition at no additional memory cost. Thus, the maximum between the regular and the reflected PDBs can be taken as an admissible heuristic. The 7-8 partition of the 15-Puzzle and the 6-6-6-6 partition of the 24-Puzzle (both shown in figure 3) received most of the attention in (Korf and Felner 2002; Felner, Korf, and Hanan 2004) and in later papers.

## Direction 2: Maximal-Matching and Minimum-Vertex-Cover PDBs

The main limitation of PA-PDBs is that they fail to capture interactions between tiles in different groups of the partition. This can be remedied by a different approach described next. First, we store PDBs for all groups of size up to $k$ (e.g., pairs, triples and quadruples). Then we need to extract admissible information from these PDBs. We have two ways to do this. The first one is a general approach and was called *dynamically-partitioned PDBs* in (Felner, Korf, and Hanan 2004). The exact partition is chosen dynamically during the search by solving a *maximal matching* problem (MM). The second direction is specific for the tile puzzle. In this puzzle we can do better than MM by solving a variant of the *vertex-cover* problem (VC). We cover both these methods next.

### Maximum matching for pairs

Consider a set of 2-tile PDBs which contain for each pair of tiles, and each pair of positions they could occupy, the number of moves required by those two tiles to move to their goal positions. We call these values the *pairwise distances*. For most pairs of tiles in most positions, the pairwise distance will equal their MD. For some tiles in some positions, such as two tiles in a linear conflict, their pairwise distance will exceed their MD. Given $n$ tiles, there are $O(n^4)$ entries in the complete set of 2-tile PDBs, but only those pairwise distances that exceed the MD of the two tiles need be stored. For example, out of $(24 \cdot 23/2) \cdot 25 \cdot 24 = 165,600$ entries of pairs for the 24-Puzzle, only about 3000 of these exceed their MD and are stored.

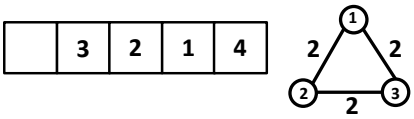


Figure 2: Mutual cost graph for 3-way linear conflict

To get an admissible heuristic for a particular node of the puzzle we 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 node of the search, this maximizing partition may be different, requiring the partitioning to be performed again for each heuristic evaluation.

To compute this heuristic for a node $s$, define a graph where each tile is represented by a vertex, and there is an edge between each pair of vertices, labeled with the pairwise distance of the corresponding pair of tiles in $s$. We call this graph the *mutual-cost graph* (MCG). The task is to choose a set of edges from this graph so that no two chosen edges are incident to the same vertex, such that the sum of the labels of the chosen edges is maximized. This is called the *maximum weighted matching* problem (MM), and can be solved in $O(n^3)$ time (Papadimitriou and Steiglitz 1982), where $n$ is the number of vertices, or tiles in this case.[2]

For example, consider the state shown in Figure 2 where tiles 1, 2, and 3 are in their goal row, but in reversed order. Each pair of these tiles is in a linear conflict. Tile 4 is in its goal position and has no conflicts. The MCG for this 3-way linear conflict is also shown in Figure 2 where the weight of each edge is two moves to resolve the linear conflict (the MD is omitted and will be added on the fly). All edges connecting Tile 4 have weight of 0 because there is nothing to add over MD for such pairs. Thus, Tile 4 and its connecting edges can be deleted from the MCG. MM of this MCG would pick one edge of weight two. Thus, two can be admissibly added to the MD of the state.

**Triple and Higher-Order Groups** This idea can be generalized to groups larger than 2. The full set of $k$-tile PDBs include a PDB for each of the $\binom{n}{k}$ different groups of $k$ tiles. Each PDB includes the number of moves of these $k$ tiles that are required to get them to their goal locations from each possible set of $k$ locations they could occupy. Potentially there are $(n+1)n(n-1)(n-2)\dots(n-k+2)$ different entries in each table, since $n$ is the number of tiles while $n+1$ is the number of locations. In practice, in a $k$-tile PDB we only need to store a value if it exceeds the sum of any of the partitions of the $k$ tiles to smaller size groups. For example, for 3-tile PDBs, we only need to store those values that exceed both the sum of the individual MD values, as well as any of the three 2-1 partitions, i.e., where we add the pairwise distance to the MD of the third tile.[3]

We then build all $k$-tile PDBs for all values of $k \in \{2..r\}$ (where $r > 2$). The corresponding MCG contains a vertex for each tile, an edge for each pairwise distance, and a hyperedge connecting $k$ vertices for each $k$-tile distance (for $k > 2$). In practice, we only store additions above MD. This means that edges or hyperedges with weights of 0 are omitted from the graph.

The task is to choose a set of edges and hyperedges of the MCG that have no vertices in common, so that the sum of the weights of the edges and hyperedges is maximized. Unfortunately, the MM problem for hypergraphs (i.e., for $k > 2$) is NP-complete (Garey and Johnson 1979). For the Tile Puzzle, however, if we only include tiles whose pairwise or triple distances exceed the sum of their MD the MCG becomes very sparse, and the corresponding matching problem can be solved relatively efficiently. For problems where the MCG is not sparse, we might have to settle for a suboptimal matching which is still an admissible heuristic.

## A better heuristic based on constraints

MM is an elegant direction but we can do better. Consider again the state shown in Figure 2 and its MCG. What is the largest admissible heuristic for this situation? The MM of this MCG can only contain one edge with a cost of two. However, for this state four moves can be added to the MD of the three tiles, because two of the three tiles must temporarily move out of the goal row. Thus, while MM is clearly admissible, it doesn't always yield the largest possible admissible heuristic. If the pairwise distance of tiles $X$ and $Y$ in a given node is $a$, there will be an edge in the corresponding MCG between vertices $X$ and $Y$, weighted by $a$. If $x$ is the number of moves of tile $X$ in a solution, and $y$ is the number of moves of tile $Y$, then their pairwise distance represents a constraint that $x + y \geq a$. Each edge of the MCG represents a similar constraint on any solution.

The general problem here is to assign a number of moves to each vertex such that all the pairwise constraints are satisfied. The heuristic is the sum of numbers over all tiles (plus MD). Since the constraints are all lower bounds, assigning large values to each vertex will satisfy all the constraints. In order for the resulting heuristic to be admissible, however, the sum of the values of all vertices must be the minimum sum that satisfies all the constraints. In the case of the triangle MCG of Figure 2, assigning a value of one to each vertex or tile, for an overall heuristic of three, is the minimum value that will satisfy all the pairwise constraints.

In formal, we are solving the following optimization problem. The tiles are labeled $t_1, t_2, \dots t_q$ (for the 15 puzzle $q = 15$). We are given a list of PDBs in the from: $P^i = \{t_{i_1}, t_{i_2}, \dots, t_{i_k}\}$. For a given search node $s$ each of these PDBs returns a heuristic value: $h_{P^i}(s)$. We want to assign an integer value $c_i(s) \geq 0$ for each tile $t_i$. The constraints are that for each $P^i$: $\sum_{j=1}^{k} c_{i_j}(s) \geq h_{P^i}(s)$. The objective function is to minimize $h(s) = \sum_{j=1}^{q} c_j$. $h(s)$ is then used

[2]The main idea of MM on the MCG was independently mentioned before by (Gasser 1995) and (Korf and Taylor 1996). But, we were the first to investigate and implement this idea. A similar idea is that of the *Canonical Heuristic* (Haslum et al. 2007) for cost-optimal planning.

[3]A very related idea is that of *"Interesting Patterns"* described by (Pommerening, Röger, and Helmert 2013).

as an admissible heuristic for search node $s$.[4]

## Weighted Vertex Cover Heuristic

While this constraint-based approach is useful for the general case, we can take it further for the Tile Puzzle domain. In this domain any path between any two locations must have the same even-odd parity as the MD between the two locations. Therefore, if the pairwise distance of $X$ and $Y$ is larger than their MD by two, then at least one of the tiles must move at least two moves more than its MD to satisfy the pairwise conflict. Such an edge of weight 2 in the MCG represents a constraint (not only that $x + y \geq 2$ but) that

$$x \geq 2 \ \ or \ \ y \geq 2.$$

In order to satisfy all such constraints, one of the vertices incident to each edge must be set to at least two.

The problem is to assign a number of moves to each vertex such that all the constraints are satisfied. In order for the resulting heuristic to be admissible, however, the sum of the values of all vertices must be the minimum sum that satisfies all the constraints. This sum is then the maximum admissible heuristic for the given state. If all edges included in the MCG have cost two, then the minimal assignment is two times the number of vertices needed such that each edge is incident to one of these vertices. Such a set of vertices is called a *vertex cover* (VC), and the smallest such set of vertices is called a *minimum vertex cover*. A minimum VC of the graph in Figure 2 must include two vertices, for an overall heuristic of four (plus MD).

## Vertex Cover for $k > 2$

For hyperedges corresponds to $k > 2$ tiles, the corresponding constraint is that the sum of the costs assigned to each of the $k$ tiles (in units of 2) must be *greater than or equal to* the weight of the hyperedge but we want to cover all possibilities to do so. There are two possibilities to cover a pairwise edge $(X, Y) = 2$, i.e., assigning two moves either to $X$ or to $Y$. However, for hyperedges ($k > 2$) the cost can be more than two. For example, some 3-tile PDB values were 4. When we have a 3-tile hyperedge of $(X, Y, Z) = 4$ then the following constraint should be added:

$$\begin{array}{l}(X \geq 2 \ \ and \ \ Y \geq 2) \ \ or \\ \quad (Y \geq 2 \ \ and \ \ Z \geq 2) \ \ or \\ \quad\quad (X \geq 2 \ \ and \ \ Z \geq 2) \ \ or \\ \quad\quad\quad (X \geq 4) \ \ or \ \ (Y \geq 4) \ \ or \ \ (Z \geq 4)\end{array}$$

For $k > 3$, values larger than 4 occur in the corresponding PDBs but the principle is the same. The optimization problem to solve is similar to the optimization problem defined above. The difference is that now the constraint we add for a hyperedge must allow all possibilities to split the cost among the tiles in units of 2.

The general problem here can be called "weighted vertex cover" (WVC). Given a hypergraph with integer-weighted edges, assign an integer value to each vertex, such that for each hyperedge, the sum of the values assigned to each vertex incident to the hyperedge is at least as large as the weight of the hyperedge.[5]

To be admissible, we are interested in the *minimum WVC*, i.e., we are looking for a WVC for which the sum of the vertex values is the lowest possible. It is easy to prove that WVC is NP-complete since ordinary VC is a special case where we only allow regular edges (between two vertices) and we only allow weights of 1. See (Felner, Korf, and Hanan 2004) for a longer proof.

It is important to note that our optimization problem of WVC is NP-complete because we are solving it as an integer problem. We can relax this optimization problem to allow real values. It can solved in polynomial time but the resulting heuristic will be less informed.

## Computing the WVC heuristic

In practice, since increases of moves assigned to a tile in the tile-puzzle come in units of two we can divide all edges by 2 before solving the WVC problem and then multiple the result by 2. In our experiments, we performed a simple branch and bound with simple pruning on the space of possible assignments and returned the minimum assignment. Naturally, encoding this as a CSP, SAT, or Integer Programming might speed up this process.

Since WVC is NP-complete, and we have to solve this problem to compute the heuristic for each search node, it may seem that this is not a practical approach for computing heuristics. Our experimental results show the contrary, for several reasons. First, in our experiments we only included pairs, triples and quadruples. By only including those hyperedges that exceed MD, the resulting MCG is extremely sparse as many of the hyperedges were 0 and omitted. Because the MCG is sparse, it is likely to be composed of two or more disconnected components and we can solve the problem for each of these independently. Second, by only including pairs, triples and quadruples, the values of the nonzero hyperedges were rather small: most of them were 2, some were 4 and very few were 6. The number of possibilities to split the weight $w$ of an hyperedge among the vertices that make up that hyperedge is exponential in $w$; in our case this was small too. Third, in the course of the search, we can compute the heuristic incrementally. Given the heuristic value of a parent node, we compute the heuristic value of each child by focusing only on the difference between the parent and child nodes. In our case, only one tile moves between a parent and child node, so we incrementally compute the WVC based on this small change to the MCG.

[4]The $h^{PhO}$ heuristic by (Pommerening, Röger, and Helmert 2013) uses linear programming to solve a very similar set of constraints. $h^{PhO}$ may return non integer values while we are restricted to only use integer values.

[5]The term *weighted vertex cover* is also used in the literature to denote another problem (Bar-Yehuda and Even 1981). We are given a graph $G = (V, E)$ with weights on the vertices of $V$. We want to find a set $V' \in V$ of vertices that cover all edges such that sum of the weights of vertices in $V'$ is minimized. In this paper we stick with our definition of *weighted vertex cover* so as to be consistent with our previous paper (Felner, Korf, and Hanan 2004) which used that term.

## Direction 3: WVC for a set of large PDBs

We now describe yet a third method to retrieve data from different PDBs in an admissible manner. This method was never published before but appeared in my Ph.D thesis (Felner 2002). It is a hybrid between the two directions described above, namely PA and WVC (for small groups). Similar to PA, we select a small number of groups of tiles but each group can be large and can contain up to 8 tiles. The groups chosen should potentially have high values of additional cost above MD. However, unlike PA but similar to WVC, these groups are not required to be disjoint and may overlap; a given tile can belong to more than one group. As in PA we first calculate the additional cost above MD for all different combinations within a group and store them in a PDB.

However, since a tile may belong to more than one group, we cannot add up these values. Alternatively, as done for the collection of small PDBs in Direction 2, here too we build a MCG for this fixed set of PDBs, calculate the minimum WVC and use it as an admissible heuristic. WVC is NP-complete but for small sizes of groups used in Direction 2 (up to quadruples) this was not a limitation even when we included all possible groups. In that case, most values of the MCG were 0 and omitted from the graph and those who were present had small values. Thus the MCG was sparse and we could optimally solve the WVC problem rather fast. This is not the case for larger PDBs (where we stored up to 8 tiles in a PDB). Here, larger values of up to 16 are stored, hyperedges that weigh zero are rare and the resulting MCG is not sparse and usually contains a single connected component. Furthermore, since hyperedges have large values we have an exponentially large number of possible partitions that split these large values along the vertices (tiles) that make up the hyperedge. Thus, optimally solving WVC for the resulting MCG drastically increases the CPU time. In our experiments this was not practical.

### Storing all VC values in a table

This run time problem can be solved as follows. When we have a fixed set of PDBs, the structure of the MCG (its vertices and hyperedges) is also fixed throughout the search process as it is composed of exactly the same set of hyperedges, each corresponds to a given PDB. Nevertheless, the values of these hyperedges may change from node to node. But, since we have the entire set of PDBs at hand, we know the exact set of values that are possible for each PDB.

For example, for a 6-tile PDB the values vary from 0 to 12 with increases of 2, i.e., we can get 7 different values for each hyperedge. If we have $q$ different 6-tile PDBs then we will have $7^q$ different combinations for the weights of the graph (not including symmetries). We can therefore precompute the WVC for all these $7^q$ combinations of values and store them in a lookup table. We call this table the *vertex-cover table* (VCT). For example, for the 24-Puzzle we had 8 different 6-tile PDBs and the resulting table was of size $7^8 = 5,764,801$. In a preprocessing phase we iterate over all these combinations, calculate the minimum WVC for each and store it in the VCT. Similar to the generation of a PDB, this preprocessing phase might take a long time. However, as in PDBs it is only calculated once and then can be amortized over solving many problem instances. In fact, building the entire set of PDBs and the corresponding VCT for the 24-puzzle took a few days. But, these tables are still present and can be used today with zero overhead.

To summarize, in the VCT method, in order to get the heuristic of a node during the actual search, perform the following three steps, each can be done in constant time:

**Step 1:** Retrieve the PDB values for all groups.

**Step 2:** Use these values as indexes and lookup the relevant entry in the VCT to retrieve the value of WVC.

**Step 3:** Add this value to MD.

### Generality of the idea

This idea of storing a lookup table for all possible combinations can be used by others who need to solve hard combinatorial problems for every node. We make the following observations on the general applicability of this idea:

**(1)** When we have a fixed goal and/or a fixed set of PDBs the entire lookup table can be calculated once and then be used for a large number of problem instances. In this case the cost of building the table can be omitted as it is amortized over all these instance. For example, I still have the VCT for the 6- and 7-tile PDBs that I generated in 2000.

**(2)** If the number of generated nodes is expected to be much larger than the number of entries in the lookup table it might be worthwhile to calculate this table even for solving a single instance only.

**(3)** The table can be built up lazily during the search as values are being requested. In this way, no value will ever be calculated without a specific need.

**(4)** Finally, this method is applicable for other optimizations problems that are needed to be solved at each node. A basic requirement is that the different combinations of parameters are fixed and known beforehand.

## Experimental Results

We performed extensive experiments with our three directions on the 15- and 24-Puzzle. All experiments were performed during the year 2000.

### 15 puzzle

Table 1 presents the results of running IDA* with different heuristics averaged over the same 1000 random instances of the 15-Puzzle used in (Korf and Felner 2002). The average optimal solution length was 52.552 moves. Each row corresponds to a different heuristic. The **Value** column shows the average heuristic value of the 1000 initial states. The **Nodes** column shows the average number of nodes generated to find an optimal solution. The **Seconds** column gives the average amount of CPU time that was needed to solve a problem on a 500 megahertz PC. The next column presents the speed of the algorithm in nodes per second. Finally, the last column shows the amount of memory in kilobytes that was needed for the PDBs, plus the VCT when applicable.

The rows are ordered according to the chronological order that we experimented and received these results. In general,

| # | Heuristic Function | Value | Nodes | Sec. | Nodes/sec | Memory |
|---|---|---|---|---|---|---|
| 1 | Manhattan | 36.940 | 401,189,630 | 53 | 7,509,527 | 0 |
| 3 | MM: pairs | 39.411 | 21,211,091 | 13 | 1,581,848 | 1,000 |
| 4 | MM: pairs+triples | 41.801 | 2,877,328 | 8 | 351,173 | 2,300 |
| 5 | WVC: pairs | 40.432 | 9,983,886 | 10 | 959,896 | 1,000 |
| 6 | WVC: pairs+triples | 42.792 | 707,476 | 5 | 139,376 | 2,300 |
| 7 | WVC: pairs+triples+quadruples | 43.990 | 110,394 | 9 | 11,901 | 78,800 |
| 8 | PA: 5-5-5 | 41.560 | 3,090,405 | .540 | 5,722,922 | 3,145 |
| 9 | PA: 6-6-3 | 42.924 | 617,555 | .163 | 3,788,680 | 33,554 |
| 10 | PA: 7-7-1 | 44.586 | 116,985 | .047 | 2,489,042 | 268,437 |
| 11 | VCT: 7 6-tile PDBs | 43.211 | 397,107 | .134 | 2,963,485 | 34,377 |
| 12 | VCT: 10 6-tile PDBs | 43.485 | 242,186 | .115 | 2,105,965 | 332,806 |
| 13 | VCT: 5 7-tile PDBs | 44.563 | 97,730 | .044 | 2,221,136 | 402,669 |
| 14 | VCT: 6 7-tile PDBs | 44.531 | 76,634 | .037 | 2,071,189 | 419,548 |
| 15 | PA: 7-8 | 45.630 | 36,710 | .028 | 1,377,630 | 576,575 |

Table 1: Experimental results on the 15-Puzzle.

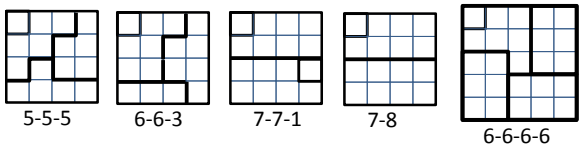


Figure 3: Different PA-PDBs for the Tile Puzzles

with exceptions, they are also ordered by decreasing CPU time and by increasing amount of memory. The first row presents the results of MD. The next two rows present the results of the dynamically-partitioned additive PDBs when the heuristic was MM of the MCG for pairs, and for pairs plus triples. Clearly, a significant speedup is shown over MD. The next three rows present the results of the WVC heuristic over the same MCG. WVC is more accurate than MM and this results in better performance for WVC. With WVC, the number of nodes decrease when moving from pairs to triples and from triples to quadruples. But, the best CPU time is that of pairs+triples. We have found that the bottleneck here was keeping the MCG accurate for each new node of the search tree. For every tile that moves we need to check all the pairs, triples and quadruples that include this tile to see whether edges were added or deleted from the MCG.

Lines 8-10 are for PA-PDBs of 5-5-5, 6-6-3 and 7-7-1 shown in Figure 3. Each of these was also reflected about the main diagonal and the numbers correspond the maximum of the two possibilities. As the PDBs get larger more memory is needed but the number of nodes and the CPU time decreases. The best variant in this set is the 7-7-1.

Next, we report results for a number of variants of VCT. Lines 11-12 report results for 7 different (but overlapping) 6-tile PDBs, and 10 6-tile PDBs, respectively. Lines 13-14 show 5 different 7-tile PDBs and 6 different 7-tile PDBs, respectively. A very important observation is that the number of nodes per second for the VCT variants is very similar to that of PA. This shows that performing VCT lookups is very efficient. Our machine only had half a giga byte of memory and given this limit, the VCT variants were clearly better in terms of nodes. Line 14 (6 groups of 7-tile PDBs) is the best variant both in nodes and in CPU time in this set. In fact, it was the winner variant in my lab for a few months.

However, then comes Line 15 where the 7-8 partition (also shown in figure 3) is provided. All the previous PDBs were stored in a simple multi-dimensional array (called *sparse mapping* in (Felner, Korf, and Hanan 2004)). By contrast, the 7-8 PDBs were stored in a more sophisticated single array (called *compact mapping* in (Felner, Korf, and Hanan 2004)). Still, it needed more than half a giga byte of memory and was executed much later on a different machine with larger memory. Clearly, this partition is most elegant and provides the best results. Since the VCT variants were only slightly better than the PA variants for up to 7 tiles we decided to omit VCT from our previous papers.

## 24 puzzle

For the 24-Puzzle we compare the pairs+triples WVC system, the 6-6-6-6 PA partitioning (shown in Figure 3) plus their reflection about the main diagonal (a total of 8 PDBs) and the VCT system for the same 8 6-tile PDBs used by PA. Table 2 shows the results for the first ten randomly-generated problem instances from (Korf and Felner 2002). The **Sol** column gives the optimal solution length. The next three columns give the number of nodes for WVC, PA and VCT. The last three columns give the CPU time for these options. The memory needs for the PA and VCT methods was around 250 Megabytes.

For the 24-Puzzle, WVC usually results in fewer node generations. Problem instance 3 in our set is the only case where WVC did not win in nodes. VCT consistently generates 20% less nodes than PA. On average, all three systems are within 7% of each other in terms of CPU time. It is important to note that the fastest method, however, was VCT.

We also built the WVC and PA systems for the 35-puzzle. Due to the large size of this puzzle optimal solutions are not known even today (2014). In (Felner, Korf, and Hanan 2004) we provided an analysis based on the prediction formula by (Korf, Reid, and Edelkamp 2001) and predicted that VC will outperform PA by a factor of 1.8.

| | | Nodes | | | Time (seconds) | | |
|---|---|---|---|---|---|---|---|
| # | Sol | WVC | PA | VCT | WVC | PA | VCT |
| 1 | 95 | **306,958,148** | 2,031,102,635 | 1,377,159,819 | 1,757 | 1,446 | **1,063** |
| 2 | 96 | **65,125,210,009** | 211,884,984,525 | 158,889,554,781 | 692,829 | 147,493 | **123,018** |
| 3 | 97 | 52,906,797,645 | 21,148,144,928 | **14,448,309,001** | 524,603 | 14,972 | **11,294** |
| 4 | 98 | **8,465,759,895** | 10,991,471,966 | 9,262,519,107 | 72,911 | 7,809 | **7,016** |
| 5 | 100 | **715,535,336** | 2,899,007,625 | 2,480,350,516 | 3,922 | 2,024 | **1,894** |
| 6 | 101 | **10,415,838,041** | 103,460,814,368 | 86,134,496,298 | 151,083 | 74,100 | **65,252** |
| 7 | 104 | **46,196,984,340** | 106,321,592,792 | 85,774,231,083 | 717,454 | 76,522 | **66,491** |
| 8 | 108 | **15,377,764,962** | 116,202,273,788 | 83,209,058,152 | 82,180 | 81,643 | **64,424** |
| 9 | 113 | **135,129,533,132** | 1,818,055,616,606 | 1,476,665,302,180 | 747,443 | 3,831,042 | **3,222,608** |
| 10 | 114 | **726,455,970,727** | 1,519,052,821,943 | 1,331,681,205,551 | 4,214,591 | **3,320,098** | 3,390,445 |
| Avg | 102.6 | **106,109,635,224** | 391,204,783,118 | 309,119,152,126 | 720,877 | 752,698 | **695,351** |

Table 2: 24-Puzzle results for static vs. dynamic databases

## Summary

We demonstrated that NP-hard problems may be efficiently solved and exploited at every node, either directly or through the use of a lookup table.

The results clearly show that as the puzzle gets larger the relative advantage of PA over WVC decreases. The reason is that PA systems are most effected by the constant amount of available memory. We can only build PDBs on our available memory and their relative size compared to the puzzle size decreases on bigger puzzles. The WVC method is more modest in its memory requirements and can still be fully built when the size of the puzzle increases. Despite the fact that it is NP-complete, it can be solved rather fast. This was our conclusion in (Felner, Korf, and Hanan 2004).

Our implementation of VCT is complex to implement and was only moderately better than PA. Therefore, we omitted it from our former papers. Nevertheless, it provides a motivation for others who perform hard optimization problems to try and build a lookup table for all their possible values if they are initially known. This will incur a large overhead when generating this table but it can be later used an infinite number of times. In many settings this might pay off.

## Acknowledgements

I thank Richard E. Korf for his willingness to work with me in those early days.

## References

Bar-Yehuda, R., and Even, S. 1981. A linear-time approximation algorithm for the weighted vertex cover problem. *J. Algorithms* 2(2):198–203.

Bonet, B., and Helmert, M. 2010. Strengthening landmark heuristics via hitting sets. In *ECAI-2010*, 329–334.

Bonet, B., and van den Briel, M. 2014. Flow-based heuristics for optimal planning: Landmarks and merges. In *ICAPS-2014*.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Felner, A. 2002. *Improving search techniques and using them on different environments*. Ph.D. Dissertation, available at http://felner.wix.com/home#!publications.

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman.

Gasser, R. 1995. *Harnessing computational resources for efficient exhastive search*. Ph.D. Dissertation, Swiss Federal Institute of Technology, Zurich, Switzerland.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SCC-4(2):100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI-2007*, 1007–1012.

Katz, M., and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *Artif. Intell.* 174(12-13):767–798.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1-2):9–22.

Korf, R. E., and Taylor, L. 1996. Finding optimal solutions to the twenty-four puzzle. In *National Conference on Artificial Intelligence (AAAI-96)*, 1202–1207.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening-A*. *Artificial Intelligence* 129(1–2):199–218.

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

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-based heuristics for cost-optimal planning. In *ICAPS-2014*.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning. In *IJCAI-2013*.

Seipp, J., and Helmert, M. 2014. Diverse and additive cartesian abstraction heuristics. In *ICAPS-2014*.