Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

ROBERT CLAUSECKER

Notes on the Construction of Pattern Databases

ZIB Report 17-59 (November 2017)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Notes on the Construction of Pattern Databases

Robert Clausecker
<clausecker@zib.de>
Zuse Institute Berlin

November 2017

Abstract

This report aims to explain and extend *pattern databases* as a tool in finding optimal solutions to combination puzzles. Using *sliding tile puzzles* as an example, an overview over techniques to construct and use pattern databases is given, analysed, and extended. As novel results, *zero aware additive pattern databases* and *PDB catalogues* are introduced. This report originally appeared as a bachelor thesis at Humboldt University of Berlin under the title *Notes on the Construction of Tablebases.*

Notation

We write $A{:=}B$ for *"A is defined by $A = B$"* and $A{:\leftrightarrow}B$ for *"A is defined by $A \leftrightarrow B$."* Furthermore, we denote *true* with $\top$ and *false* with $\bot$. We denote with $\mathbf{N} = \{0, 1, \ldots\}$ the set of natural numbers including zero and with $n^{\underline{k}}{:=} n(n-1)\cdots(n-k+1)$ the *falling factorial*.

S_k denotes the *symmetric group of order k* whose members are the *permutations of k items*. Its neutral element is id. Given such a permutation $\sigma \in S_k$, $\sigma(i)$ with $0 \leq i < k$ is the element at position i in σ. Given two permutations $\sigma, \tau \in S_k$, $\sigma \circ \tau$ is the *composition* of σ and τ with $(\sigma \circ \tau)(i){:=}\tau(\sigma(i))$ for all $0 \leq i < k$. The notation $(a\ b)$ denotes the *transposition of a and b* which is a permutation defined by

$$(a\ b)(i){:=}\begin{cases} a & \text{if } i = b \\ b & \text{if } i = a \\ i & \text{otherwise.} \end{cases}$$

Given an equivalence relation $\approx\ \subset S \times S$ over some set S, the notation $[a]_{\approx}{:=}\{\, x \mid x \in S, a \approx x \,\}$ indicates the *equivalence class* of a under $\approx$. When the relation is $\simeq_T$, indicating equivalence under the APDB or ZPDB for tile set T, we write $[v]_T$ instead of $[v]_{\simeq_T}$ for convenience to denote the set of all configurations that map to the same partial puzzle configuration as v.

We denote an undirected graph $G = (V, E)$ as a set of vertices V and a set of edges $E \subset V \times V$ with E symmetric. Given two vertices $v, w \in V$, the relation $v \sim_G w$ indicates whether v is connected to w. The *distance function* $d_G(v, w) : V \times V \mapsto \mathbf{N} \cup \{\infty\}$ yields the number of edges in the shortest path from v to w in G or ∞ if no such path exists:

$$d_G(v, w){:=}\begin{cases} \min\limits_{n \in \mathbf{N}} v \sim_G^n w & \text{if } v \sim_G^* w \\ \infty & \text{otherwise.} \end{cases}$$

The *neighbourhood* $N_G(v){:=}\{\, w \mid w \in V, v \sim_G w \,\}$ of a vertex v is the set of all vertices it is connected to. If the graph we consider is obvious from the context, the subscripts are dropped.

We denote by $G/{\approx}$ the *quotient graph* of G with respect to some equivalence relation $\approx\ \subset V \times V$. The vertices of $G/{\approx}$ are the equivalence classes of V under $\approx$ denoted by $V/{\approx}{:=}\{\, [v]_{\approx} \mid v \in V \,\}$, furthermore it is $[v]_{\approx} \sim_{G/\approx} [w]_{\approx}$, $v, w \in V$ iff there are some $\tilde{v} \in [v]$, $\tilde{w} \in [w]$ such that $\tilde{v} \sim_G \tilde{w}$. Again, we denote by $G/T{:=}G/{\simeq_T}$ the quotient graph of the state transition graph of an $(nm-1)$ puzzle under the equivalence class induced by the APDB or ZPDB for tile set T.

1 Sliding Tile Puzzles

The first sliding tile puzzle was the *15 puzzle* invented in 1874 by Noyes Chapman, a postmaster in Canastota, New York [2]. The puzzle comprises 15 square tiles numbered 1 to 15 arranged randomly in a 4×4 square tray with one spot empty. The configuration of the puzzle can be changed by sliding any tile adjacent to the empty grid location into the empty grid location. The objective of the puzzle is to transition the puzzle configuration into the *solved configuration** shown in figure 1.1.

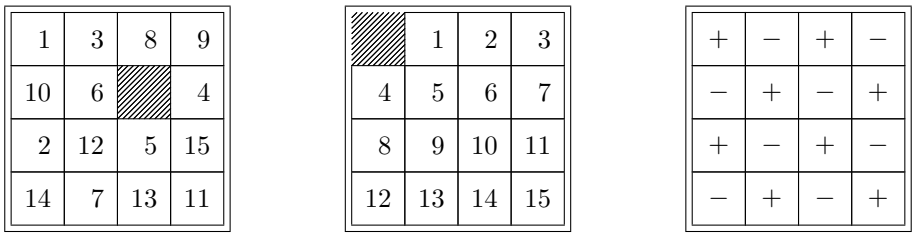


Figure 1.1 *a permuted and a solved 15 puzzle, square parity*

By varying the size of the tray and consequentially varying the number of tiles in the puzzle, the 15 puzzle can be generalised to the $(nm-1)$ *puzzle* played on an $n \times m$ tray. For this thesis the author has mainly researched the *24 puzzle* played on a 5×5 tray, but the results are applicable to arbitrary puzzle sizes. The solved configuration for an $(nm-1)$ puzzle is formed by leaving the upper left corner empty and then arranging the tiles 1, 2, ..., $nm-1$ from left to right, top to bottom.

1.1 The Invariant Property

Since their invention, sliding tile puzzles have been subject to abundant research.

Let's label the empty grid spot with 0 and call it *zero tile*. If we number the grid locations such that in the solved configuration (cf. figure 1.1), each tile is on the grid location with the same number, we can model an $(nm-1)$ puzzle configuration as a permutation σ of the tiles $\{0, 1, \ldots, nm-1\}$ with $\sigma(t)$ representing the location of tile t. The sign of this permutation alternates with every move as a move is a composition of σ with a transposition of the zero tile with some adjacent tile. Similarly, when the grid locations are assigned a parity as shown in figure 1.1, the parity of the grid location on which the zero tile resides changes every move. If these two are combined, we get the *parity of a puzzle configuration* $P(\sigma)$, shown in (1.1.1).

$$P(\sigma) := \operatorname{sgn}\sigma \cdot \operatorname{sgn}\sigma(0) \tag{1.1.1}$$

Because both factors of $P(\sigma)$ change with every move, $P(\sigma)$ stays unchanged when transitioning into a different state, forming an invariant.

This invariant was found in an initial analysis in 1879 by Johnson and Story [4] who showed that it exists in 15 puzzle configurations; pairs of configurations can be reached from each other iff they have the same parity. Because the puzzle dimensions are not used in the invariant's definition, the result readily generalises to all $(nm-1)$ puzzles.

* Traditionally, the solved configuration has tile 1 in the top left corner and the bottom right corner left empty. The solved configuration used here is more common in research due to the slightly easier programming. Nevertheless, some authors [11] prefer the more traditional solved configuration.

2 Informed Graph Search: The A* Family of Algorithms

An interesting problem on sliding tile puzzles and other combination puzzles is finding the shortest solution for a given configuration. This can be modelled as a *single-pair shortest path problem* on the puzzles state transition graph and can be solved efficiently using an algorithm from the *A* family of algorithms.* Indeed, sliding tile puzzles have for a long time been popular test cases for heuristic search research [4].

As an improvement over earlier graph search algorithms, the key idea of the A* family of algorithms is to use an externally supplied heuristic function $h(v)$ to disregard paths leading away from the goal, searching primarily along paths that achieve progress. In many real world use cases such a heuristic function obtains readily. For example, for graphs embedded into Euclidean space, Euclidean distance can generally be used with good performance. §§3–4 of this thesis deal with the construction of good heuristic functions for sliding tile puzzles.

Given a graph $G = (E, V)$ and a goal vertex $z \in V$, a function $h(v) : V \mapsto \mathbf{N}$ is called an *admissible heuristic function* for path searches to z if

$$h(v) \leq d(v, z) \tag{2.1}$$

for all $v \in V$. Such a heuristic function provides a h *value* which is a lower bound for the distance from v to z, informally allowing the search algorithm to receive a notion of how close vertex v is to the goal.

A heuristic function is called *consistent*, if

$$v \sim w \to h(v) - h(w) \leq 1 \tag{2.2}$$

for all $v, w \in V$, a stronger property than admissibility:

Lemma 2.1 *Every consistent heuristic function h with $h(z) = 0$ is admissible.*

Proof Let $v_0 \in V$ such that $v_0 \sim^* z$ and let $v_0 v_1 \dots z$ be a shortest path from v_0 to z. This path comprises $d(v_0, z) + 1$ vertices. We can now expand $h(v_0)$ using the consistency property:

$$\begin{aligned} h(v_0) &= h(v_0) + \big(h(v_1) - h(v_1)\big) + \big(h(v_2) - h(v_2)\big) + \ldots + \big(h(z) - h(z)\big) \\ &= \big(h(v_0) - h(v_1)\big) + \big(h(v_1) - h(v_2)\big) + \ldots + \big(h(v_{d(v_0,z)-1}) - h(z)\big) + h(z) \\ &\leq 1 + 1 + \ldots + 1 + 0 = d(v_0, z), \end{aligned}$$

proving the lemma. *q. e. d.*

Algorithm 2.1 presents the A* algorithm invented in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael [5]. The algorithm keeps a priority queue *open* of vertices whose neighbourhood has not yet been expanded and a set *closed* of vertices which have already been visited. For each vertex v in the open list, we keep track of the distance g at which we encountered v. If in addition to the distance the shortest path is desired, too, we can keep track of that as well.

The priority queue *open* is ordered by $f = g + h(v)$, the sum of the exact length of the path from v_0 to v and the estimated length of the path from v to z. As each vertex v we add to the open list is added at the shortest distance we encounter it, $g = d(v_0, v)$ and thus $f \leq d(v_0, z)$ for all vertices in the open list, guaranteeing that the shortest path is found.

In each step, we take a vertex v with minimal f from *open*, mark it as closed, add the neighbourhood of v to *open* and finally remove all closed nodes from the open list. If the goal node is encountered in the open list, we immediately terminate the search. If we find *open* to be empty, z must be unreachable from v_0, so we return ∞.

A detailed analysis of the algorithm as well as a rigorous proof of correctness can be found in [5].

Algorithm 2.1 *The A* algorithm* $\mathrm{A}^*(G, h, v_0, z)$. *Find a shortest path from v_0 to z on graph $G = (V, E)$ using the admissible heuristic $h(v)$. The algorithm returns $d(v_0, z)$.*

1 $open \leftarrow \{(v_0, 0)\}$
2 $closed \leftarrow \emptyset$
3 **while** $open \neq \emptyset$ **do**
  3.1 **if** $\exists g \in \mathbf{N}.\ (z, g) \in open$ **then return** g
  3.2 $(v, g) \leftarrow \min_f open$
  3.3 $closed \leftarrow closed \cup \{v\}$
  3.4 $open \leftarrow open \setminus \{(v, g)\}$
  3.5 $open \leftarrow open \cup \big\{\, (w, g+1) \;\big|\; w \in N_G(v), w \notin closed \,\big\}$
4 **return** ∞

While A* can be proven to have optimal run time for a given admissible heuristic $h(v)$, its practical use is often limited by the requirement to keep track of the open and closed lists. When trying to find shortest paths in very large graphs (for example, the state transition graphs of sliding tile puzzles), list size quickly exceeds available storage.

To solve this problem, Richard Korf in 1985 [6] introduced algorithm 2.2 named *Iterative Deepening A** (IDA*), a variant of *Depth-First Iterative Deepening Search* (IDFS, ibid.). IDA* removes the need for open and closed lists by performing a depth-first search over the graph. To ensure that an optimal result is obtained, only vertices with an f value below a bound b are expanded. If no path has been found, the bound is raised to the lowest f value encountered that was just out of reach and the search started anew. Analysis and a proof of correctness can be found in [6].

This algorithm performs well on large graphs, but has a number of drawbacks. If the graph contains cycles and no path to z exists, IDA* does not terminate. If the graph contains cycles, IDA* expands the same vertex multiple times. If $h(v)$ is not very accurate and the graph contains many cycles, this can quickly lead to almost all expanded vertices being duplicates as [7] noted. This effect can be reduced using pruning techniques that detect duplicate vertex expansions without having to keep tab of the already expanded vertices. Korf and Taylor developed *Finite State Machine Pruning* in [7], §6.2 tries to expand this idea.

Algorithm 2.2 *The IDA* algorithm* IDA*(G, h, v_0, z). *Find a shortest path from* v_0 *to* z *on graph* $G = (V, E)$ *using the admissible heuristic* $h(v)$.

1 $(found, b) \leftarrow (\bot, 0)$
2 **until** $b = \infty \vee found$ **do** $(found, b) \leftarrow search\_to\_bound(G, h, v_0, z, b)$
3 **return** b

function $search\_to\_bound(G, h, v_0, z, b_0) \mapsto (found, b)$:
1 **if** $h(v_0) > b_0$ **then return** $(\bot, h(v_0))$
2 **if** $v_0 = z$ **then return** $(\top, 0)$
3 $b \leftarrow \infty$
4 **for each** $v \in N_G(v_0)$ **do**
4.1 $(found, b_v) \leftarrow search\_to\_bound(G, h, v_0, z, b_0 - 1)$
4.2 **if** *found* **then return** $(found, b_v + 1)$
4.3 **if** $b > b_v + 1$ **then** $b \leftarrow b_v + 1$
5 **return** $(\bot, b)$

3 Pattern Databases as Heuristic Functions

Since the introduction of sliding tile puzzles as test vehicles for heuristic search, a number of ways to build good heuristics have been developed; see [4] for an overview. Currently, the best heuristics are produced by *additive pattern databases* (APDB).

The *pattern database* (PDB) was introduced in 1998 by Culberson and Schaeffer [1] to find solutions for the 15 puzzle. The key idea developed by the two was to simplify the state of the puzzle, disregarding the identity of some tiles and then tabulating $d(v, z)$ for each state of the simplified puzzle. Figure 3.1 shows how such a pattern database for the tile set $\{2, 3, 4, 7, 8, 9\}$ views a configuration of the 24 puzzle.

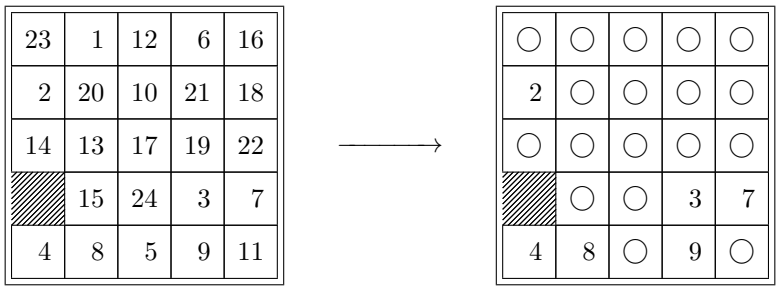


Figure 3.1 *a 24 puzzle configuration as seen by the* $\{2, 3, 4, 7, 8, 9\}$ *PDB*

While a full table for the 15 puzzle has $16!/2 \approx 10.4614 \times 10^{12}$ entries and is infeasible for use as a lookup table in most applications, a PDB considering just 7 out of the 15 tiles has merely $16!/8! = 518\,918\,400$ entries. The distances stored in the pattern database underestimate the distances in the complete puzzle and form a good admissible and consistent heuristic if a reasonable large set of tiles is chosen. However, as the identity of some tiles is lost, nothing accounts for their entanglement, leading to poor h values if the unaccounted tiles are badly permuted.

This problem was addressed by Korf and Felner in 2002 with the introduction of *disjoint pattern databases* [8], here called *additive pattern databases* as in Korf's later papers. Instead of disregarding just the label of some tiles, those tiles are entirely ignored in additive pattern databases. Only moves that touch a tile in the tile set observed by the pattern database are accounted for, as displayed in figure 3.2.

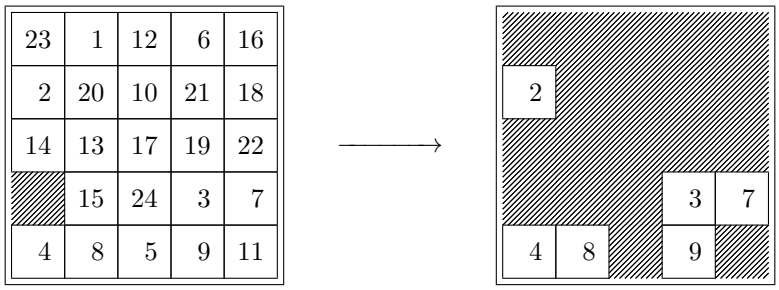


Figure 3.2 *a 24 puzzle configuration as seen by the* $\{2, 3, 4, 7, 8, 9\}$ *APDB*

An APDB for some tile set $T \subset \{1, \ldots, nm - 1\}$ can be described through the equivalence relation $\simeq_T$ it induces on puzzle configurations. Two puzzle configurations map to the same APDB entry iff all tiles in T are in the same spot:

$$\pi \simeq_T \sigma \colon \leftrightarrow \forall t \in T.\, \pi(t) = \sigma(t) \tag{3.1}$$

The heuristic function h_T provided by the APDB for tile set T could then simply be $h_T(v) = d_{G/T}(v, z)$, the distance between v and z in the quotient graph induced by $\simeq_T$. However, Korf et al. [10] use a slightly more sophisticated construction yielding significantly better distances while still maintaining the admissibility. See also [9] for some discussion on the APDB construction procedure.

By counting the number of ways the k tiles of a tile set T can be placed inside an $n \times m$ grid, it is easy to see that an APDB for the $(nm - 1)$ puzzle APDB has

$$s_{\text{APDB}} \colon = (nm)^{\underline{k}} = k! \binom{nm}{k} \tag{3.2}$$

entries. By including the zero tile in T, the same formula (3.1) could be used to describe the equivalence relation forming a Culberson & Schaeffer PDB, but we would like to use $\simeq_T$ for some tile set T with $0 \in T$ to denote the equivalence relation induced by ZPDBs introduced below.

Even though this leads to a worse heuristic on its own, additive pattern databases have the property that the h value of multiple additive pattern databases for disjoint tile sets can be added to an admissible heuristic that captures the entanglements in each tile set used. This solves the issues observed with Culberson and Schaeffer's pattern databases and is proven below.

Lemma 3.1 *Given a set $S = \{T_1, T_2, \ldots, T_k\}$ of pairwise disjoint tile sets $T_i \subset \{1, \ldots, nm-1\}$ and the heuristic functions $h_1, h_2 \ldots, h_k$ constructed from the additive pattern databases for these tile sets, $h = h_1 + h_2 + \ldots + h_k$ is an admissible and consistent heuristic.*

Proof Let $G = (V, E)$ be the state transition graph of the $(nm-1)$ puzzle we consider. Because for $i = 1, 2, \ldots, k-1$ we have $h_i(v) = d_{G/T_i}(v, z) \leq d_G(v, z)$ equal to the actual distance between v and z in the quotient graph induced by $\simeq_{T_i}$, all heuristic functions h_i are both admissible and consistent.

For each edge $(v, w) \in E$ there is at most one $T \in S$ such that $v \not\simeq_T w$: Considering the corresponding tile permutations σ_v and σ_w, we see that each move transposes the zero tile with some tile t giving $\sigma_v(u) = \sigma_w(u)$ for all $u \neq t$ and $v \simeq_T w$ for all $T \in S$ with $t \notin T$ by (3.1). However, as all $T_1, T_2, \ldots, T_k$ are pairwise disjoint, t can be part of at most one T_i.

Hence there are two possibilities for the difference $h(v) - h(w)$ between the h values of two adjacent vertices $v, w \in V$: Either it is $v \simeq_T w$ for all $T_i \in S$ and $h_i(v) = h_i(w)$ for all $1 \leq i \leq k$ yielding $h(v) - h(w) = 0$, or there is exactly one $T_i \in S$ with $v \not\simeq_T w$ and $h(v) - h(w) = h_i(v) - h_i(w) \leq 1$ using the consistency of h_i.

This shows the consistency of h and using lemma 2.1 implies that h is admissible. *q. e. d.*

While additive pattern databases are a very good improvement over Culberson and Schaeffer's pattern databases, they fail to account both for the entanglement between two tile sets in an additive pattern database (as shown in figure 3.3) as well as the usually very slight distance increase given by the location of the zero tile, which is discarded by additive pattern databases.

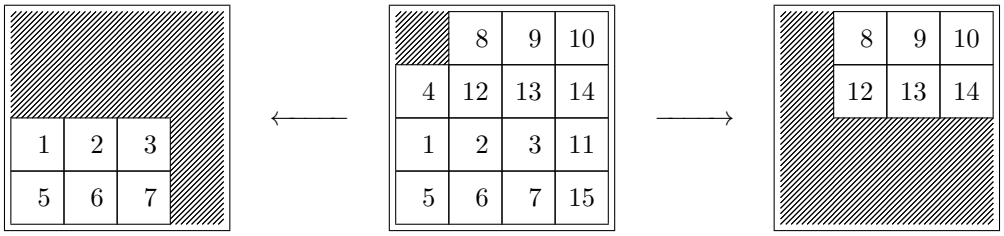


Figure 3.3 *a 15 puzzle configuration as seen by the $\{1, 2, 3, 5, 6, 7\}$ and $\{8, 9, 10, 12, 13, 14\}$ additive pattern databases. Note how the pattern databases do not account for the cost of moving the two tile groups around each other.*

To address the first deficiency, we discuss *PDB catalogues* in §5.3. To deal with the second deficiency, we introduce *zero aware additive pattern databases* (ZPDB) whose construction is detailed in §4.

The idea behind a ZPDB for some tile set T is to keep track of the *zero tile region*. This is the connected region of grid locations not covered by a tile in T in which the zero tile is located. This restricts the possible moves in the partial puzzle configuration observed by the ZPDB to the moves moving a tile adjacent to the zero tile region into it, thus mirroring the true distance to a solved configuration closer while maintaining the additive property of the APDB. Figure 3.4 shows how a partial puzzle configuration is partitioned into regions, and compares the resulting ZPDB with the APDB for the same puzzle configuration and tile set.

By refining definition (3.1), we can describe a ZPDB with the same equivalence relation $\simeq_T$ for some tile set $T \subset \{0, 1, \ldots, nm-1\}$, allowing $0 \in T$ to indicate a zero-tile aware pattern database. In the definition below, $R_T(\sigma)$ is the zero tile region of σ with respect to tile set T. If $0 \notin T$, this is $\{0, 1, \ldots, nm-1\} \setminus T$, otherwise it's the largest connected region of grid locations not occupied by tiles in $T \setminus \{0\}$ which contains the square $\sigma(0)$.

$$\pi \simeq_T \sigma :\leftrightarrow R_T(\pi) = R_T(\sigma) \wedge \forall t \in T \setminus \{0\}.\, \pi(t) = \sigma(t) \tag{3.3}$$

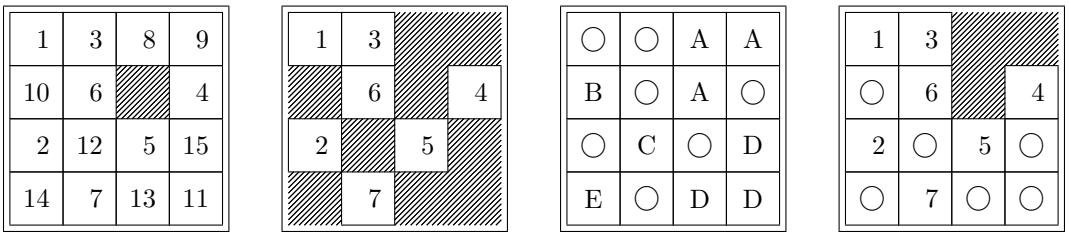


Figure 3.4 *(a) a 15 puzzle configuration, (b) as seen by the* $\{1, 2, 3, 4, 5, 6, 7\}$ *APDB, (c) its possible zero-tile regions, and (d) as seen by the corresponding ZPDB*

The proof for lemma 3.1 also holds for zero-aware pattern databases as no assumption about zero-tile regions is made; a ZPDB forms an admissible and consistent heuristic just as an APDB does. Furthermore, because the ZPDB refines the information in the APDB for the same tile set, its heuristic is never worse than the corresponding APDB heuristic. Compared to additive pattern databases, a simple formula for the number of entries in a zero-aware pattern database does not obtain easily. The number of entries in a ZPDB is given by

$$s_{\text{ZPDB}} := \bar{\rho}(nm)^{\underline{k}} \tag{3.4}$$

where $\bar{\rho}$ is the average number of zero tile regions for a given tile arrangement. Table 3.5 gives $\bar{\rho}$ values for different tile set sizes in the 24 puzzle.

k	s_{APDB}	s_{ZPDB}	$\bar{\rho}$	$\rho_{\max}$
2	600	608	1.01	2
3	13 800	14 472	1.04	2
4	303 600	339 048	1.12	3
5	6 375 600	7 871 280	1.23	4
6	127 512 000	181 008 000	1.42	5
7	2 422 728 000	4 066 655 040	1.68	6
8	43 609 104 000	87 358 400 640	2.00	7
9	741 354 768 000	1 759 513 674 240	2.37	8
10	11 861 676 288 000	32 787 717 580 800	2.76	10
11	177 925 144 320 000	560 680 553 664 000	3.15	11
12	2 490 952 020 480 000	8 749 801 518 796 800	3.51	13

Table 3.5 *The size of a k-tile APDB for the 24 puzzle, the size of the corresponding ZPDB, and the average and maximal number of zero tile regions per partial tile configuration.*

The storage required is always larger than the storage required for an APDB with an extra tile, but no extra tile is "consumed" with respect to the additive property. The extra storage requirement is rather low in practical cases, making ZPDBs a useful tool to gain a better approximation while spending little extra storage compared to the usage of a larger APDB.

In previous research [10], zero tile region information was collected during database construction and then discarded by storing for each partial puzzle configuration just the minimum h value for all zero tile regions. This is useful to get a much better approximation than a naïvely generated APDB without using additional storage.

In §4.2 we show how the region-tracking property of a ZPDB can be used to generate it easily, avoiding the costly open and closed lists used in the Döbbelin et al. algorithm [9]. In §5.1 we empirically compare the quality of the heuristic provided by a ZPDB with the heuristic provided by the corresponding APDBs.

4 The Construction of Pattern Databases

In this section, we discuss the representation, construction, and verification of APDBs and ZPDBs for $(nm-1)$ puzzles. The focus lies on algorithms and data structures that perform well in practice.

4.1 Constructing an Index Function

In [11], Korf et al. introduce two ways to store the h values of an additive pattern database. First, a *sparse mapping* is introduced in which a k-tile APDB is represented as a k-dimensional array with $(nm)^k$ entries. While this representation wastes a lot of space as it does not account for no pair of tiles being allowed to occupy the same square on the grid, indices into the table are very fast to compute. Second, a *compact mapping* is described where one array entry is reserved for each partial tile permutation in the APDB, reducing space usage to $(nm)^{\underline{k}}$ at a significantly higher indexing cost.

In this thesis we do not consider the sparse mapping any further as its space requirements are prohibitive for large tile sets. Instead we enhance the compact mapping by indexing an entry $e = (\mu, \pi)$ in the APDB for tile set T as a pair of a *tile map rank* μ describing the grid locations occupied by tiles in T, and a *tile permutation index* π describing the permutation of the tiles in T within the grid locations given by μ. Similarly, as shown in figure 4.1.1, a ZPDB entry is indexed as a triple $e = (\mu, \pi, \rho)$ keeping track of the *zero tile region index* ρ in addition to μ and π.

The APDB is stored as an array of *cohort* arrays, where each cohort contains the entries for one μ. This slightly complicated scheme is needed as the number of zero tile regions differs between partial configurations. But as this number only depends on μ, we can get away with a rather small lookup table for the cohort offsets by grouping entries by μ first and then by π and ρ.

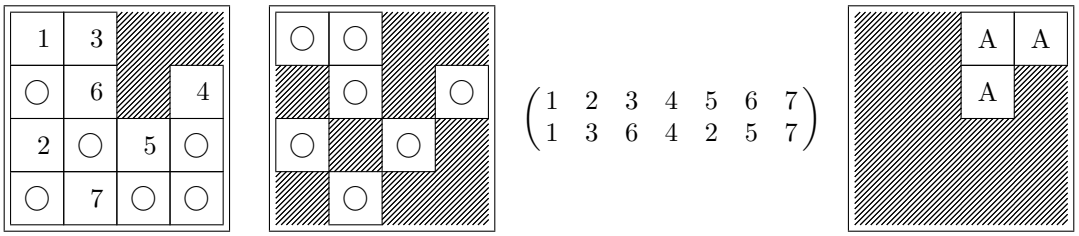


Figure 4.1.1 *A 15 puzzle configuration as seen by the* $\{1, 2, 3, 4, 5, 6, 7\}$ *ZPDB and the components of the corresponding index, see also figure 3.4*

The tile map rank μ is a number between 0 and $\binom{nm}{k} - 1$ representing the choice of the grid locations on which we place the tiles in T. The combinatorial number system [12] can be used to map these combinations to integers. In the author's implementation, lookup tables were used to implement these *rank* and *unrank* operations.

Similarly, tile permutation index π is a number between 0 and $k! - 1$ representing the permutation of the tiles in T within the squares selected by μ. Computing this is slightly more involved, the standard approach involves computing the permutation's *inversion table* [13] and then encoding it into a single integer using the *factorial number system* (ibid.). However, without hardware support, both computing inversion tables and reconstructing the permutation from the inversion table have run time $\Theta(k^2)$ and slow performance in practice. See appendix C for some implementation ideas using the aforementioned hardware support.

As a practical solution, algorithm 4.1.1, a variant of the *Fisher-Yates Shuffle* [14], can be used instead. The idea behind this algorithm is to interpret the random values used in the Fisher-Yates Shuffle as entropy that goes into selecting a permutation. By retrieving these random values from a single integer using the factorial number system, we can quickly compute a permutation belonging to a given π.

Algorithm 4.1.1 $\text{shuffle}_k(\pi)$ *returns a permutation* σ *of* k *items using* $0 \le \pi < k!$ *as an entropy source.*

1 $\sigma \leftarrow \text{id}$
2 $\pi_1, \pi_2, \ldots, \pi_k \leftarrow \pi$ with $0 \le \pi_i < i$ using the factorial number system
3 **for** $i \leftarrow 0, 1, \ldots, k-1$ **do** transpose $\sigma(i)$ and $\sigma(i + \pi_{k-i})$
4 **return** σ

As seen easily, the run time of algorithm 4.1.1 is $\Theta(k)$. To show that $\text{shuffle}_k(\pi)$ is suitable both as a shuffling function and as an inverse index function, we prove that it is a bijection.

Lemma 4.1.1 $\text{shuffle}_k(\pi) : \{0, 1, \ldots, k! - 1\} \mapsto S_k$ *implemented by algorithm 4.1.1 is a bijection.*

Proof Assume there are $\pi, \tilde{\pi} \in \{0, 1, \ldots, k! - 1\}$ such that $\pi \neq \tilde{\pi}$ but $\text{shuffle}_k(\pi) = \text{shuffle}_k(\tilde{\pi})$. Let us observe the execution of shuffle_k for π and $\tilde{\pi}$ with the variables in $\text{shuffle}_k(\tilde{\pi})$ named $\tilde{\sigma}$ and $\tilde{\pi}_{k-i}$. If $\pi_i = \tilde{\pi}_i$ for all i, then $\pi = \tilde{\pi}$ contradicting the assumption. Therefore there must be an i such that $\pi_{k-i} \neq \tilde{\pi}_{k-i}$. Let $i_{\min}$ be the smallest such i.

Right before iteration $i_{\min}$ of step 3, it is $\sigma = \tilde{\sigma}$ because $\pi_{k-i} = \tilde{\pi}_{k-i}$ for all $i < i_{\min}$ but right after this iteration $\sigma(i_{\min}) \neq \tilde{\sigma}(i_{\min})$ as we transposed $\sigma(i_{\min})$ with $\sigma(i_{\min} + \pi_{k-i_{\min}})$ in $\text{shuffle}_k(\pi)$ but with $\sigma(i_{\min} + \tilde{\pi}_{k-i_{\min}}) = \tilde{\sigma}(i_{\min} + \tilde{\pi}_{k-i_{\min}})$ in $\text{shuffle}_k(\tilde{\pi})$ and $\pi_{k-i_{\min}} \neq \tilde{\pi}_{k-i_{\min}}$ by construction.

As $i + \pi_{k-i} > i$ for all i, the values of $\sigma(i_{\min})$ remain fixed after iteration $i_{\min}$, hence $\sigma(i_{\min}) \neq \tilde{\sigma}(i_{\min})$ and therefore $\sigma \neq \tilde{\sigma}$ at the end of algorithm 4.1.1, contradicting the assumption.

This shows that $\text{shuffle}_k(\pi)$ is injective. Because $|S_k| = k!$, domain and codomain of $\text{shuffle}_k(\pi)$ have the same finite size and $\text{shuffle}_k(\pi)$ is total, it is a bijection by the pigeonhole principle. *q. e. d.*

By keeping track of the inverse permutation as done in algorithm 4.1.2, we can just as quickly compute the π that generates a given permutation (cf. [14], exercise 12).

Algorithm 4.1.2 $\text{shuffle}_k^{-1}(\sigma)$ *computes an integer* $0 \leq \pi < k!$ *such that* $\text{shuffle}_k(\pi) = \sigma$.

1 $\tau \leftarrow \tau^{-1} \leftarrow \text{id}$
2 **for** $i \leftarrow 0, 1, \ldots, k - 1$ **do**
  2.1 $\pi_{k-i} \leftarrow \tau^{-1}(\sigma(i)) - i$
  2.2 transpose $\tau^{-1}(\tau(i))$ and $\tau^{-1}(\tau(i + \pi_{k-i}))$
  2.3 transpose $\tau(i)$ and $\tau(i + \pi_{k-i})$
3 $\pi \leftarrow \pi_1, \pi_2, \ldots, \pi_k$ using the factorial number system
4 **return** π

Implementations of algorithm 4.1.2 can be optimised using the invariant $\tau^{-1} \circ \tau = \text{id}$ and the observation that neither $\tau^{-1}(\sigma(i))$ nor $\tau(i)$ are read after iteration i, allowing us to replace step 2.2 with $\tau^{-1}[\tau[i]] \leftarrow i + \pi_{k-i}$ and step 2.3 with $\tau[i + \pi_{k-i}] \leftarrow \tau[i]$ in an implementation where τ and τ^{-1} are arrays, saving two of the five assignments in the loop.

Algorithm 4.1.2 works by tracing in τ the intermediate values σ had when trying to generate the permutation using algorithm 4.1.1 and reconstructing the choice of π_{k-i} that would cause $\sigma(i)$ to have the desired value. To do this efficiently, we trace the inverse permutation of τ in τ^{-1}, allowing us to reconstruct each "random value" π_{k-i} in constant time for a total run time of $\Theta(k)$.

Lemma 4.1.2 *Algorithm 4.1.2's* $\text{shuffle}_k^{-1}(\sigma) : S_n \mapsto \{0, 1, \ldots, k! - 1\}$ *is the inverse of* $\text{shuffle}_k(\pi)$.

Proof The invariant $\tau \circ \tau^{-1} = \text{id}$ which holds initially and at the end of each iteration of step 2 is of use here. Step 2.2 mutates τ^{-1} to $(i\ i + \pi_{k-i}) \circ \tau^{-1}$ and τ is mutated to $\tau \circ (i\ i + \pi_{k-i})$ in step 2.3 to restore the invariant.

After each iteration of step 2, it is $\tau(j) = \sigma(j)$ for all $j \leq i$ by induction: in the first iteration, this is ensured for $j = 0$ by exchanging $\tau(0)$ with $\tau(\tau^{-1}(\sigma(0)))$. In iteration i the induction hypothesis gives us $\tau(j) = \sigma(i)$ for all $j < i$. Before step 2.2 we have $\tau^{-1}(\sigma(i)) \geq i$ and therefore $0 \leq \pi_{k-i} < k - i$ as otherwise $\sigma(\tau^{-1}(\sigma(i))) = \tau(\tau^{-1}(\sigma(i)))$ using the induction hypothesis and

$$i = (\sigma \circ \sigma^{-1})(i) = (\sigma \circ \tau^{-1} \circ \tau \circ \sigma^{-1})(i) = (\sigma \circ \tau^{-1} \circ \sigma \circ \sigma^{-1})(i) = (\sigma \circ \tau^{-1})(i)$$

contradicts $\tau^{-1}(\sigma(i)) < i$. Steps 2.2 and 2.3 ensure $\tau(i) = \sigma(i)$ by mutating τ to $\tau \circ (i\ i + \pi_{k-i})$. Because $i \leq i + \pi_{k-i}$, this does not destroy $\tau(j) = \sigma(j)$ for $j < i$, concluding the induction.

If we write down the mutations performed on τ as a composition of transpositions, we get

$$\sigma = \tau = (0\ \pi_k) \circ (1\ 1 + \pi_{k-1}) \circ \ldots \circ (k - 1\ k - 1 + \pi_1)$$

with $0 \leq \pi_i < i$ as shown above. This is the same sequence of transpositions performed by $\text{shuffle}_k(\pi)$ to yield σ, proving $\text{shuffle}_k\left(\text{shuffle}_k^{-1}(\sigma)\right) = \sigma$. *q. e. d.*

As the last piece of the index function, we need to compute ρ. To do so, one can create a lookup table with $nm\binom{nm}{k}$ entries containing the correct ρ for each cohort and grid location to use. In practice, this table consumes a few megabytes of storage for common puzzle and tile set sizes and it can be shared among all pattern databases with the same k. The same table can be used to store the offset from the beginning of the ZPDB where a cohort's entries are stored, improving cache locality.

4.2 Computing the Pattern Database by Iterated Neighbour Expansion

Algorithm 4.2.1, a variant of *breadth-first search*, is used to generate pattern databases. This algorithm can be used both for Culberson & Schaefer's non-additive PDBs, APDBs, and ZPBDs.

When an APDB is generated directly using this algorithm, the resulting heuristic has poor quality because it does not account for some configurations in the partial puzzle configuration appearing to be easier than they could possibly be as illustrated in figure 4.2.1. To improve the result one can first generate a ZPDB and then identify the entries of all equivalence classes, storing only the minimum. This yields the same result as the Korf et al. algorithm [10] but consumes less storage because $\bar{\rho}$ table entries have to be stored for each (μ, π) instead of one entry and a bitmap for $\rho_{\max}$ possible classes as often $\bar{\rho} < 1 + \lceil \rho_{\max}/8 \rceil$, see also table 3.5.

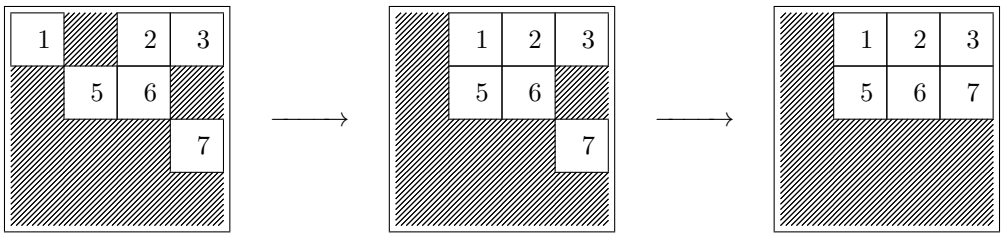


Figure 4.2.1 *A partial puzzle configuration in the naïvely constructed* $\{1, 2, 3, 5, 6, 7\}$ *APDB with* $h = 2$*. Regardless of where the zero tile is located, more than the two moves depicted in this impossible yet technically shortest move sequence are needed to actually transition into the solved partial configuration.*

Algorithm 4.2.1 *The algorithm* make_pdb(G, T, z, idx) *creates a pattern database pdb for the puzzle with state transition graph* $G = (V, E)$ *using the bijective index function* $idx(v) : V/T \mapsto \{0, \ldots, s-1\}$ *leading to the solved configuration* z *where* $s = |V/T|$ *is the number of entries in the pattern database.*

1 $pdb[0, \ldots, s-1] \leftarrow \infty$
2 $pdb[idx(z)] \leftarrow 0$
3 $r \leftarrow 1$
4 **while** $\exists 0 \leq e < s.\, pdb[e] = r$ **do**
  4.1 $V_r \leftarrow \left\{ idx^{-1}(e) \;\middle|\; 0 \leq e < s, pdb[e] = r \right\}$
  4.2 $V_{r+1} \leftarrow \left\{ w \;\middle|\; v \in V_r, w \in N_{G/T}(v), pdb[idx(w)] = \infty \right\}$
  4.3 **for** $v \in V_{r+1}$ **do** $pdb[idx(v)] \leftarrow r + 1$
  4.4 $r \leftarrow r + 1$
5 **return** *pdb*

Steps 4.1–4.3 of algorithm 4.2.1 can be executed in parallel easily. No synchronisation when writing to *pdb* is necessary as all writes overwrite entries valued ∞ with $r + 1$, an idempotent operation. Similarly, algorithm 4.2.1 can be implemented with *pdb* in background storage by fetching one cohort and its neighbourhood at a time, updating the entries and then writing the cohort back to background storage.

Lemma 4.2.1 *After* make_pdb(G, T, z, idx) *has executed,* $pdb[idx(v)] = d_{G/T}(v, z)$ *for all* $v \in V$*.*

Proof First, let us prove that algorithm 4.2.1 always terminates: In each iteration, the number of entries e with $pdb[e] = \infty$ either decreases or $V_{r+1} = \emptyset$. In the latter case, the loop terminates because no members of *pdb* have been set to $r \leftarrow r + 1$. As *pdb* has finitely many entries, eventually V_{r+1} must be empty. An infinite descent is impossible and algorithm 4.2.1 terminates.

After the initialisation in step 1, each entry $pdb[e_r]$ is assigned up to once. When $pdb[e_r]$ with $e_r \neq idx(z)$ is assigned in step 4.3, there has to be some entry e_{r-1} with $idx^{-1}(e_{r-1}) \sim_{G/T} idx^{-1}(e_r)$ that caused e_r to be assigned. Iterating, we find a chain of entries $e_r e_{r-1} \ldots e_0$ with $e_0 = idx(z)$

describing the path through which e_r was assigned "reachable in r steps." For each e_i in this path, it is $e_i \sim_{G/T} e_{i-1}$, so the corresponding vertices $v_i = idx^{-1}(e_i)$ form a path from e_r to z.

It follows that for all $v \in V$, if $d_{G/T}(v, z) = \infty$ then $pdb[idx(v)] = \infty$ must be as otherwise a path could be obtained as explained in the previous paragraph, leading to a contradiction. Similarly, if $d_{G/T}(v, z) < \infty$, then $pdb[idx(v)] \leq d_{G/T}(v, z)$ because the shortest path from v to z in G/T has such a chain of entries and would cause assignment to $idx(v)$ in round $d_{G/T}(v, z)$ and an earlier assignment is impossible as no path shorter than $d_{G/T}(v, z) + 1$ vertices from v to z can exist by definition. *q. e. d.*

4.3 The Pattern Database as a Witness of Its Own Correctness

Algorithm 4.3.1 verifies if an APDB or ZPDB is *internally consistent*, that is, all entries in the database are consistent with all entries in their neighbourhoods. As shown in Lemma 4.3.1, this is equivalent to the database storing $d_{G/T}(v, z)$ for all $v \in V/T$, allowing us to easily verify that no errors have caused the database to contain wrong values. This is useful both to verify the correctness of an algorithm 4.2.1 implementation and to catch any hardware errors that caused bits to flip during generation, potentially helping to reduce the problems noted in [15] when using disks to store large amount of state.

Algorithm 4.3.1 *Algorithm* pdb_consistent(pdb, G, T, z, idx) *verifies if* $pdb[e]$ *is internally consistent for all* $0 \leq e < s$ *where* $s = |V/T|$ *is the number of entries in the pattern database.*

1 **if** $pdb[idx(z)] \neq 0$ **then return** $\bot$
2 **for** $0 \leq e < s$ **do**
  2.1 $v \leftarrow idx^{-1}(e)$
  2.2 **if** $\exists w \in N_{G/T}(v).\, pdb[e] - pdb[idx(w)] > 1$ **then return** $\bot$
  2.3 **if** $v \neq z \wedge \neg\exists w \in N_{G/T}(v).\, pdb[idx(w)] = pdb[e] - 1$ **then return** $\bot$
3 **return** $\top$

In steps 2.2 and 2.3 we handle pattern databases for disconnected graphs by assuming

$$\infty - n := \begin{cases} 0 & \text{if } n = \infty \\ \infty & \text{otherwise.} \end{cases}$$

This allows us to avoid special cases for disconnected graphs where $pdb[e] = \infty$ for some e. The following invariants on the heuristic h_T represented by pdb are checked:

a) In step 1, we verify that $h_T(z) = d_{G/T}(z, z) = 0$
b) In step 2.2, we verify that $h_T(v)$ is consistent for each v
c) In step 2.3, we verify that we can progress towards z from each $v \neq z$ iff $v \sim_{G/T} z$

Note that algorithm 4.3.1 cannot be applied to verify an APDB generated by first generating the corresponding ZPDB and then identifying all entries with the same μ and π as such an APDB does not represent $d_{G/T}(v, z)$ but rather some value $d_{G/T}(v, z) \leq h_T(v) \leq d_{G/T \cup \{0\}}(v, z)$.

Lemma 4.3.1 *It is* pdb_consistent(pdb, G, T, z, idx) *iff* $pdb[idx(v)] = d(v, z)$ *for all* $v \in V/T$.

Proof In this proof, we write $d(v, w)$ for $d_{G/T}(v, w)$, $v \sim w$ for $v \sim_{G/T} w$, etc. for easier reading.

Let us first prove that pdb_consistent(pdb, G, T, z, idx) if $pdb[idx(v_0)] = d(v_0, z)$ for all $v_0 \in V/T$. Invariant (a) holds by assumption, (b) holds because $h(v) = d(v, z)$ is a consistent heuristic and for (c) observe that if $v_0 \sim z$ with $v_0 \neq z$, a shortest path $v_0 v_1 \ldots z$ with $d(v_0, z)$ edges exists and thus $pdb[idx(v_1)] = d(v_1, z) = d(v_0, z) - 1 = pdb[idx(v_0)] - 1$. Conversely, if $pdb[idx(v_0)] = \infty$ we have $v_0 \not\sim^* z$ and $pdb[idx(w)] = \infty$ for all $w \in N(v_0)$ as otherwise $v_0 \sim w \sim^* z \to v_0 \sim^* z$ contradicting the assumption.

For the other direction, observe that if invariants (a) and (b) hold, h_T represented by pdb is consistent by (2.2) and admissible by lemma 2.1. Now suppose $h_T(v) \neq d(v, z)$ for some $v \in V/T$. Without loss of generality, assume v_0 is a counterexample such that $h_T(v_0)$ is minimal. By invariant (c) there must be some $w \in N(v)$ such that $h_T(w) = h_T(v_0) - 1$. Because $h_T(w) < h_T(v_0)$ and $h_T(w)$ has been chosen to be the least counterexample, it is $h_T(w) = d(w, z)$ and

$$h_T(v_0) = 1 + h_T(w) = 1 + d(w, z) \geq d(v_0, w) + d(w, z) \geq d(v_0, z)$$

by the triangle inequality. The case $h_T(v_0) > d(v_0, z)$ contradicts the admissibility of h_T and case $h_T(v_0) > d(v_0, z)$ contradicts the existence of a counterexample. *q. e. d.*

5 Heuristic Quality Analysis

In this section, we empirically compare zero-aware pattern databases with other heuristics and then explore the effects of choosing different pattern databases on heuristic quality.

5.1 A Measure of Heuristic Quality

Previous research [20] found that the number of expanded nodes $E(N, d, P)$ in an IDA* iteration to depth d with $N(i)$ nodes at level i and $P(i)$ being the fraction of nodes with h value less than or equal to i can be described by (5.1.1).

$$E(N, d, P) = \sum_{i=0}^{d} N(i)P(d-i) \tag{5.1.1}$$

Assuming $N(i) = b^i$ where b is the *brute force branching factor* with $b \approx 2.36761$ for the 24 puzzle, Korf shows that

$$\frac{E(N, d, P)}{E(N, d-1, P)} \approx b \tag{5.1.2}$$

from which we can conclude that the time complexity of an IDA* search for the shortest path from v to z is in $\mathcal{O}(b^{d(v,z)-h(v)})$ compared to $\mathcal{O}(b^{d(v,z)})$ for a brute-force search. This indicates that the number of node expansions exponentially depends on the h value of the initial position in our search tree.

We can capture this notion of heuristic quality by computing $\bar{h}$, the average h value for a given heuristic by random sample and compare it with $\bar{d}$, the average distance to the solved configuration. The average number of node expansions can then be predicted by $b^{\bar{d}-\bar{h}}$. Furthermore the quality of two heuristics h_1 and h_2 can be compared through the ratio of their expected node expansions. It is

$$\frac{b^{\bar{d}-\bar{h}_1}}{b^{\bar{d}-\bar{h}_2}} = b^{\bar{h}_2-\bar{h}_1} \tag{5.1.3}$$

allowing us to compare the quality of two heuristics just through the difference of their $\bar{h}$ values and the brute force branching factor b. For the 15 puzzle, $d_{\max} = 80$ and $\bar{d} = 52.59$ are known [17] but due to computational limitations, these figures have not yet been computed for the 24 puzzle.

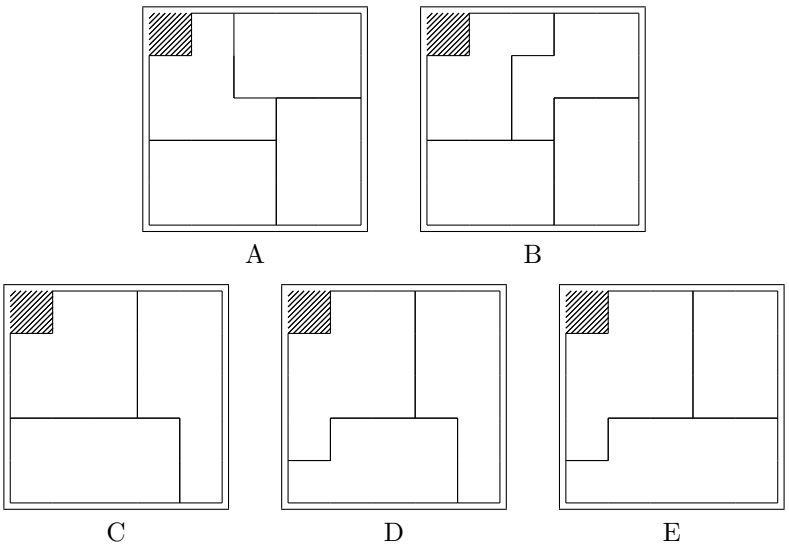


Figure 5.2.1 *Two 6-6-6-6 partitionings and three partitionings from [9]*

5.2 ZPDBs and APDBs in Comparison

To empirically evaluate the quality advantage gained from a ZPDB in comparison to an APDB, the author has solved the 50 instance of the 24 puzzle from Korf [8] using an implementation of IDA* without pruning using partitioning B (cf. figure 5.2.1). The results are listed in appendix B and show that using ZPDBs results in a reduction of node expansions by a factor of 1.6065 in Korf's test cases on

average. In these searches, the h value for a node in the search tree was formed by looking up both the puzzle configuration and its transposition in the A/ZPDB set and taking the maximum.

Partitioning B was found to be an improvement over Korf's partitioning A (ibid.). For Korf's partitioning we get $\bar{h}_{\text{APDB}} = 81.57$ and $\bar{h}_{\text{ZPDB}} = 81.82$ with a sample size of $n = 100\,000\,000$ while partitioning B has $\bar{h}_{\text{APDB}} = 81.81$ and $\bar{h}_{\text{ZPDB}} = 82.06$ with an expected reduction by $b^{82.06-81.81} = 1.24$, further showing that the analysis from §5.1 is not very accurate.

5.3 The Influence of Pattern Size

In [9], the effectiveness of differently sized APDBs was researched. The node expansion figures are not directly comparable as Döbbelin et al. used BF-IDA* [18] instead of IDA* as used in this paper, leading to slightly lower node expansion figures due to pruning of duplicate nodes. However, we can compare the average and maximal h values given in table 5.3.1.

pattern	x	s	$\bar{h}$	$h_{\min}$	$h_{\max}$	$\bar{r}_{A/x}$	$b^{\bar{h}_A - \bar{h}_x}$
6-6-6-6	A	510 048 000	81.85	40	115	1.00	1.00
8-8-8	C	130 827 312 000	82.84	40	116	3.00	2.35
9-8-7	D	787 386 600 000	83.10	43	116	5.74	2.94
9-9-6	E	1 482 837 048 000	84.56	44	116	8.37	10.34

Table 5.3.1 *PDB statistics from [9] for partitionings A, C, D, and E in figure 5.2.1*

In the second to last column $\bar{r}_{A/x}$, the ratio between node expansions in partitioning A and the current line is provided. This is contrasted with the node expansion ratio predicted by (5.1.3) in column $b^{\bar{h}_A - \bar{h}_x}$. The numbers are in the same region but not very close, indicating that using $\bar{h}$ to compare node expansions is a possible indicator for the real performance of a heuristic but not sufficient for accurate predictions.

Of note are the diminishing returns from an increase in pattern size. For example, partitioning C occupies 256.5 times more space than partitioning A, yet it only reduces node expansions threefold, suggesting that raising the PDB size is even less effective than for Culberson & Schaeffer PDBs where Korf [16] showed that a PDB with s entries in a problem with branching factor b reduces the number of expanded IDA* nodes by a factor of at least

$$\frac{\log_b s + 1}{s} \tag{5.3.1}$$

compared to the number of nodes in a brute-force search. In his paper, Korf exclaims a desire to extend this analysis to additive pattern databases. As to my knowledge, no such analysis has been performed.

In the following section we show how instead of raising the tile set size, a better heuristic can be constructed just as well by using a catalogue of multiple small PDBs.

5.4 Pattern Database Catalogues

In [19], Holte et. al. researched how using the maximum h value from a *catalogue* of heuristics can be used to reduce node expansions. In their paper, they make two propositions to explain this effect: (a) using multiple heuristics reduces the number of configurations with low h values and (b) eliminating low h values is more important for improving search performance than retaining large h values.

The first proposition is explained as a direct result of taking the maximum, fixing "weaknesses" one heuristic has which another one lacks. The second proposition is explained through (5.1.1) with $N(i) = b^i$ assumed, causing $P(d-i)$ for high i to have an exponentially higher influence on the number of node expansions than $P(d-i)$ for low i.

This does not contradict the analysis in §5.1 but is not supported by it either as increasing the h value for any configuration has an equal effect on $\bar{h}$. The author has continued to primarily rely on $\bar{h}$ as a measure of heuristic quality and tried to find a reasonable small catalogue of ZPDBs with high $\bar{h}$, resulting in a catalogue with $\bar{h} = 83.31$ comprising 20 ZPDBs @ 6 tiles each, forming 14 partitionings and a subset of 14 ZPDBs forming 7 partitionings with $\bar{h} = 83.08$ but still good performance in practice due to the reduced lookup time. The partitionings used are shown in appendix A.

To find a good PDB catalogue with high $\bar{h}$ and a modest number of ZPDBs, the author used the following empirical method: first, a set of known good partitionings was used as a starting point. Then,

new partitionings were added by deliberately selecting partitionings whose tile sets are both reasonably compact and group tiles together that were not grouped together before. If possible, an attempt is made to reuse existing tile sets to keep the number of PDBs low. After this step, $\bar{h}$ is sampled with $n = 100\,000\,000$. For each partitioning, we count the number of samples whose maximal h value was only predicted by this partitioning. Then, partitionings that rarely lead to an improvement are removed from the catalogue. This procedure is iterated until no further improvements can be found within the desired catalogue size.

5.5 The Cost of PDB Lookups

The run time of IDA* using one or more APDB partitionings as a heuristic function is dominated by the time needed to compute PDB indices and lookup configurations. Lookup time is severely penalised by the random-access behaviour, rendering the CPU cache ineffective. On an Intel Xeon E3-1290 v2 processor @ 3.70 GHz, the author's implementation performs 8.88×10^6 APDB lookups and 8.16×10^6 ZPDB lookups per second on random puzzle configurations. APDB lookups are slightly faster as their layout saves one memory access due to the lack of the auxiliary array outlined in §4.1.

With this knowledge, the reductions in node expansions of a PDB catalogue must be contrasted with the number of PDB lookups performed per expanded node. Note that as with each move only one tile is moved and, all h values of PDBs not involving this tile can be copied from the previous configuration. In a search with one partitioning, this means that one lookup per expanded node and one lookup per unexpanded node have to be performed. In a search on the $(nm - 1)$ puzzle with a PDB catalogue comprising pattern databases for tile sets $T_1, T_2, \ldots, T_k$, the number of lookups per configuration is estimated by (5.5.1).

$$\sum_{i=1}^{k} \frac{|T_i|}{nm - 1} \tag{5.5.1}$$

This formula assumes that when searching through the graph, the tile moved is distributed equally, allowing us to average the number of PDBs that involve each tile to get an easy formula.

Applying this to the PDB catalogues in appendix A, we get 5 lookups per configuration for the large catalogue and 3.5 lookups for the small catalogue. To outperform a single PDB, the small catalogue needs to reduce node expansions at least threefold. To outperform the small catalogue, the large catalogue needs to reduce the number of expanded nodes by at least a factor of $5/3.5 \approx 1.43$. Both have been annotated in the sample instances in appendix B, showing that while the small catalogue outperforms a single ZPDB partitioning in 34 out of 50 sample configurations, the large catalogue outperforms the small one in only 11 instances and only once outperforms the single ZPDB partitioning were the small catalogue has not.

These empirical results show that using a small catalogue of PDBs can improve heuristic search performance in the average case, but as with many other heuristics, returns quickly diminish.

6 Outlook

In this section we list ideas for further research that appeared while working on this thesis but were either ignored due to lack of time or not fitting into the scope of this thesis.

6.1 Pattern Databases and Distributed Search

In [9], Döbbelin et al. built large 9-tile APDBs for the 25-puzzle to perform a distributed search for optimal solutions. In their implementation, a hash function is used to distribute APDB entries over the compute nodes to avoid an imbalance in the workload. However, this causes most table lookups to require network communication, slowing down the search. By observing that whether two vertices $v, w \in V/T$ are connected only depends on μ_v and μ_w, we can represent the problem of distributing an APDB over the network as a graph partitioning problem and potentially gain good performance with minimal traffic.

6.2 Finite State Machine Pruning

In [7], Taylor and Korf introduce *Finite State Machine Pruning*, a technique to prune duplicate nodes in an IDA* search tree by keeping track of the path to the current node using a finite state machine and pruning edges that lead more than half-way through a cycle in the state transition graph. This method is very effective because the state transition graph of an $(nm - 1)$ puzzle is highly self-similar; a cycle of length $2l$ can be detected solely by looking at the last l moves we did to reach the current position, allowing for a fairly good state machine to occupy little space.

Building these state machines is a fairly elaborate process: to detect cycles up to length $2l$, all paths of length l and the nodes they end in have to be enumerated and checked for duplicates. By finding a more efficient way to build finite state machines for pruning, this approach might be made practical.

Another interesting question is if a comprehensive pruning finite state machine could be build. Such a finite state machine would account for all cycles in the graph. With this machine, a polynomial time algorithm for solving any instance of the $(nm - 1)$ puzzle the machine was built for obtains: Find an arbitrary solution for the instance, then use the machine to repeatedly find segments that can be shortened by flipping them over to the other half of the loop. When no loop is found, the shortest solution has been obtained.

6.3 Tail Databases

In [11], Donald Knuth mentions, how IDA* could be sped up using a database of all puzzle configurations v for which $h(v) = d(v, z)$. When IDA* reaches a point where $f = b$, we can query this *tail database*. If the configuration is not present, we immediately know that this vertex cannot lead to a solution within the current bound b. If the configuration is found, we know that a solution can be reached from this vertex within the current bound b. Knuth further mentions how such a database for the 15 puzzle under the Manhattan heuristic contains just 88 728 779 entries, suggesting that this approach might be feasible.

Clearly, this could be generalised to *n-tail databases* storing all configurations with $h(v) \geq d(v, z) - n$. Since a false positive in the tail database merely causes less aggressive pruning, a probabilistic data structure such as a bloom filter could be used to implement *probabilistic tail databases* with sufficient accuracy and modest storage requirements.

6.4 Finding an Optimal Partitioning

The analysis in §5.1 suggest that the performance of IDA* for a specific configuration v is exponential in $h(v)$, suggesting that finding the PDB set with $h(v)$ maximal and using just that PDB might lead to good search performance while avoiding the performance impact from having to do lookups into multiple non-additive heuristics.

As observed in [8], partitioning the puzzle into compact regions is generally a good idea since "those tiles interact the most with each other." The same rule could be formalised and used to generate partitionings based on which tiles are close to each other in the puzzle instance we want to solve as those should interact just as much with each other as tiles close to each other in the solved configuration.

We could also generate all 6 tile ZPDBs in advance and then find the optimal partitioning for v by brute force. The author estimates that all ZPDBs would occupy about 1 TB storage with general-purpose compression, making this approach just feasible with modern disk sizes. This idea is similar to the high-order heuristics outlined in [4] but avoids its performance problems by determining an optimal partitioning once instead of for every node in the search tree.

7 Literature

[1] Joseph C. Culberson, Jonathan Schaeffer, *Pattern Databases*, Computational Intelligence, vol. 14 no. 3 p. 318–334, August 1998.

[2] Jerry Slocum, Dic Sonneveld, *The 15 Puzzle*, The Slocum Puzzle Foundation, ISBN 1-890980-15-3, 2006.

[3] Woolsey Johnson, William E. Story, *Notes on the "15" Puzzle*, American Journal of Mathematics, vol. 2, no. 4, p. 397–404, December 1879.

[4] Richard E. Korf, Larry A. Taylor, *Finding Optimal Solutions to the Twenty-Four Puzzle*, AAAI-96 Conference Proceedings,, p. 1202–1207, 1996.

[5] Peter E. Hart, Nils J. Nilsson, Bertram Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE transactions of Systems Science and Cybernetics, vol. 4, no. 2, p. 100–107, July 1968.

[6] Richard E. Korf, *Depth-First Iterative-Deepening: An Optimal Admissible Tree Search*, Artificial Intelligence, vol. 27, no. 1, p. 97–109, September 1985.

[7] Larry A. Taylor, Richard E. Korf, *Pruning Duplicate Nodes in Depth-First Search*, AAAI-93 Conference Proceedings, p. 756–761, 1993.

[8] Richard E. Korf, Ariel Felner, *Disjoint Pattern Database Heuristics*, Artificial Intelligence, vol. 134, no. 1, p. 9–22, January 2002.

[9] Robert Döbbelin, Thorsten Schütt, Alexander Reinefeld, *Building Large Compressed PDBs for the Sliding Tile Puzzle*, Workshop on Computer Games 2013 Proceedings, p. 16–27, April 2014.

[10] Ariel Felner, Richard E. Korf, Sarit Hanan, *Additive Pattern Database Heuristics*, Journal of Artificial Intelligence Research, vol. 22, April 2004.

[11] Donald E. Knuth, `15PUZZLE-KORF1`, August 2015.

[12] Donald E. Knuth, *Generating All Combinations*, The Art of Computer Programming, vol. 4A, §7.2.1.3, p. 360, Addison Wesley, ISBN 0-201-03804-8, 2011.

[13] Donald E. Knuth, *Inversions*, The Art of Computer Programming, vol. 3, §*5.1.1, p. 11–22, Addison Wesley, ISBN 0-201-89685-0, 1998.

[14] Donald E. Knuth, *Random Sampling and Shuffling*, The Art of Computer Programing, vol. 2, §3.4.2, p. 142–148, Addison Wesley, ISBN 0-201-89684-2, 1998.

[15] Richard Korf, Peter Schultze, *Large-Scale Parallel Breadth-First Search*, AAAI-05 Conference Proceedings, p. 1380–1385, July 2005.

[16] Richard E. Korf, *Analyzing the Performance of Pattern Database Heuristics*, AAAI-07 Conference Proceedings, p. 1164–1170, 2007.

[17] Hugo Pfoertner, *Number of configurations of the sliding block 15-puzzle that require a minimum of n moves to be reached, starting with the empty square in one of the corners*, The On-Line Encyclopedia of Integer Sequences, seq. A089484, November 2003.

[18] Rong Zhou, Eric A. Hansen, *Breadth-first heuristic search*, Artificial Intelligence, vol. 170, no. 4–5, p. 385–408, April 2006.

[19] R. C. Holte, Jack Newton, A. Felner, R. Meshulam, David Furcy, *Multiple Pattern Databases*, ICAPS-04 Proceedings, p. 122–131, 2004.

[20] Richard E. Korf, Michael Reid, Stefan Edelkamp, *Time Complexity of iterative-deepening-A**, Artificial Intellgience, vol. 129, no. 1, p. 199–218, February 2000.

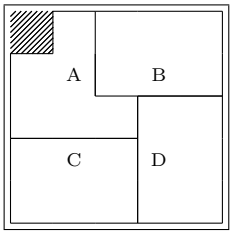
[21] Intel Corporation, *Intel 64 and IA-32 architectures software developer's manual*, vol. 2B, p. 4-270, October 2017.

[22] User chqrlie, answer to question *How can I effectively encode/decode a compressed position description?*, Stack Overflow, `https://stackoverflow.com/a/39627199/417501`, September 2016.

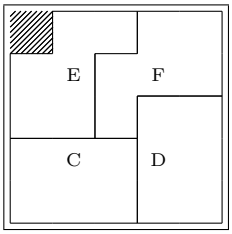
[23] Thorsten Schütt, Robert Döbbelin, Alexander Reinefeld, *Forward Perimeter Search with Controlled Use of Memory*, Proceedings of The Twenty-Third International Joint Conference on Artificial Intelligence, p. 659–665, August 2013.

A Sample PDB Catalogue

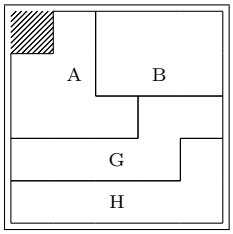
The following catalogue was created using the empirical method explained in §5.4. For each partitioning in the catalogue, the fraction of samples for which that partitioning's ZPDB heuristic applied to the sample or its transposition was the only one to yield the maximal h value is listed. A subset of these partitionings was chosen to form a smaller catalogue with better performance due to the reduced number of PDB lookups. These partionings have two such fractions, the first one for the big catalogue, the second one for the small catalogue. Most tile sets are used multiple times. To make identification easier, each tile set is given a letter.



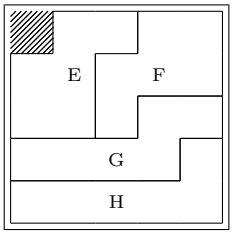
1.42%/2.21%



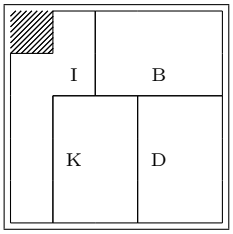
0.95%/4.49%



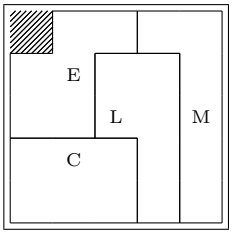
0.60%/0.83%



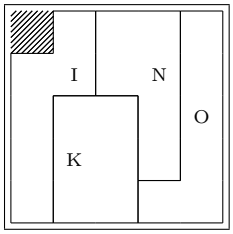
0.91%/1.72%



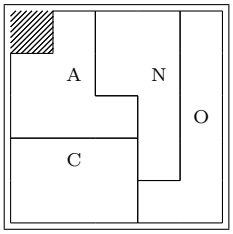
2.48%/5.18%



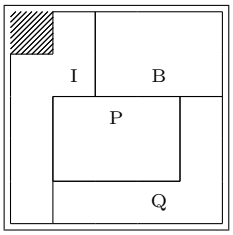
1.50%/7.15%



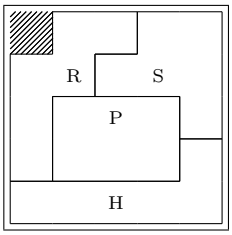
0.95%



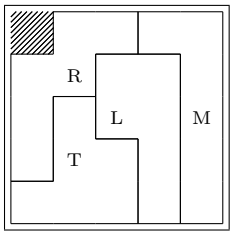
0.73%



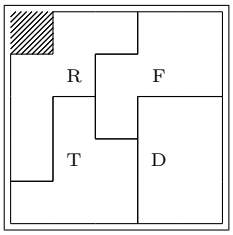
1.78%/4.77%



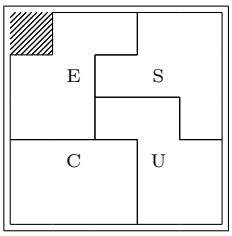
2.29%



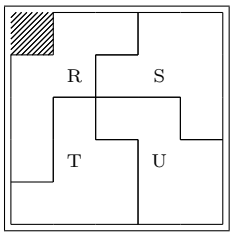
0.73%



0.64%



0.83%



0.29%

B Node Expansions

The following table contains node expansion figures for Korf's 50 instances [8] using IDA* without pruning. For columns *APDB* and *ZPDB*, partitioning B from figure 5.2.1 was used, first as an APDB and then as a ZPDB. The APDB was generated by identifying equivalence classes of the corresponding ZPDB. The columns *large* and *small* contain node expansion figures for the large and small PDB catalogues shown in appendix A. For both catalogues, ZPDBs were used. In all searches, the heuristics for both configuration and its transposition were evaluated and the maximum taken. Column *A/Z* contains the ratio of APDB node expansions to ZPDB node expansions, giving the node expansion reduction factor.

Instances for which the large catalogue outperforms the small catalogue when accounting for PDB lookup time are marked with an asterisk * in the *large* column. Similarly, configurations for which the small catalogue outperforms the *ZPDB* column when accounting for PDB lookup time are marked with a dagger † in the *small* column. Instance 29 is marked with a dagger in the *large* column to indicate that the large catalogue outperforms the *ZPDB* column while the small catalogue does not.

	instance	*dst*	*APDB*	*ZPDB*	*large*	*small*	*A/Z*
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	3 141 570 558	1 519 342 878	108 703 674	117 467 064†	2.068
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	247 550 334 056	192 822 200 142	25 135 947 124	29 264 736 068†	1.284
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	17 615 736 271	11 413 683 106	3 999 342 793	5 434 702 977	1.543
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	10 986 155 299	5 864 537 433	935 567 772*	1 400 503 448†	1.873
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	12 001 345 718	9 315 037 634	273 410 647	390 578 091†	1.288
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	54 535 765 242	42 030 291 877	4 796 719 621	6 698 966 111†	1.298
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	77 231 244 201	60 961 985 243	13 194 614 396	17 906 078 537	1.267
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	47 525 839 166	29 697 758 251	10 376 569 382	13 535 201 029	1.600
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	2 236 415 582 740	1 672 298 149 150	187 175 422 518*	309 130 241 762†	1.337
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	924 368 172 083	676 392 893 522	134 940 105 049	167 583 392 153†	1.367
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	1 902 486 069 006	1 351 698 389 608	49 573 504 121*	116 474 549 412†	1.407
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	1 197 561 998 218	790 809 277 878	141 565 811 281	171 571 908 021†	1.514
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	2 402 824 683	1 884 837 825	616 447 004	701 223 515	1.275
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	835 033 892 309	604 143 294 406	150 994 599 974	200 822 010 034	1.382
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	160 297 159 212	119 978 837 751	20 200 259 355	26 267 090 496†	1.336
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	4 425 035 169	2 948 432 041	553 571 538	662 629 439†	1.501
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	342 545 189 161	276 985 020 457	22 863 713 753*	33 631 896 485†	1.237
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	1 630 815 047 031	951 396 884 026	216 061 645 298	285 537 534 331	1.714
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	175 567 621 968	143 446 839 323	21 754 367 511*	32 889 968 312†	1.224
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	441 263 822 738	198 843 123 714	18 204 163 668	22 735 477 601†	2.219
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	399 664 301 385	239 491 639 509	44 703 839 402	58 584 164 217†	1.669
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	4 242 972 533	2 040 103 620	358 982 248*	548 311 618†	2.080
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	93 524 754 833	54 505 056 714	10 548 597 086	13 045 533 560†	1.716
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	588 001 863 444	334 268 324 895	33 428 251 235	44 201 568 857†	1.759
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	287 284 728	226 092 925	69 184 229	84 227 356	1.271

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	15 971 930 421	9 507 576 552	2 139 369 616	2 659 110 319†	1.680
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	65 129 544 138	49 270 386 612	2 046 198 636*	4 216 102 078†	1.568
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	2 527 026 959	1 648 150 241	277 237 937	372 429 469†	1.533
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	3 627 610 303	1 843 325 968	328 724 666†	600 979 319	1.968
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	1 856 058 898	1 193 365 165	269 745 193	344 637 494	1.555
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	34 260 502 816	25 189 694 913	8 155 230 997	9 053 104 297	1.360
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	1 771 683 541	1 309 878 844	110 429 060	132 190 231†	1.356
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	1 732 619 843 486	1 128 931 796 105	212 585 689 793	242 636 097 372†	1.535
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	779 613 628 794	363 395 428 336	24 601 164 074	35 005 528 703†	2.145
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	79 111 763 362	50 830 832 599	7 657 225 682	10 634 620 002†	1.519
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	2 583 250 416	1 279 740 693	448 117 068	539 330 132	2.019
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	1 602 136 899	833 616 086	217 671 293	241 609 604	1.922
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	41 008 133	34 805 331	5 569 572	7 625 016†	1.178
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	180 338 759 227	114 074 886 931	13 479 027 313*	21 307 377 603†	1.581
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	63 460 870	45 178 716	15 211 386	19 188 906	1.405
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	30 362 260 471	17 880 754 992	7 118 193 532	7 573 206 864	1.698
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	323 637 323 917	237 833 366 277	39 893 389 786	50 231 887 608†	1.361
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	48 240 581 391	35 565 936 934	8 704 466 404	11 222 700 070	1.356
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	765 221 908	207 281 923	54 511 645	58 563 520†	3.691
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	83 844 370 278	61 023 573 559	14 386 085 329	16 291 382 555†	1.374
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	53 061 069 552	39 494 530 901	1 721 078 711*	3 323 229 228†	1.344
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	30 693 099 481	13 924 939 866	4 208 353 106	5 381 998 824†	2.204
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	335 529 724 182	229 804 892 114	97 237 792 233	120 724 359 750	1.460
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	86 995 188 013	48 316 661 736	8 357 503 208*	12 414 119 411†	1.801
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	4 410 965 020 136	2 973 799 264 599	451 651 234 220	493 791 380 323†	1.483
	averages	101	394 294 072 986	263 644 437 998	40 362 051 222	52 160 056 183	1.607

C Computing Inversion Tables

The following algorithms to compute inversion vectors were found to perform well but require special hardware support. In the following descriptions, $a \,\&\, b$ and $a \mid b$ indicate *bitwise and* as well as *bitwise or* as in the C programming language. Likewise, $\sim a$ indicates the bitwise complement and $a \,\%\, b$ is the remainder.

Algorithm C.1 *Given a permutation σ of k items, compute the inversion table $b_0, b_1, \ldots, b_{k-1}$ using the* popcount *function.*

1 $m \leftarrow 2^k - 1$
2 **for** $i \leftarrow 0, 1, \ldots, k-1$ **do**
  2.1 $b_i \leftarrow \text{popcount}(m \,\&\, 2^{\sigma(i)} - 1)$
  2.2 $m \leftarrow m \,\&\, \sim 2^{\sigma(i)}$
3 **return** $b_1, b_2, \ldots, b_{k-1}$

Algorithm C.1 works by keeping in m a bitmask of all items already encountered. Using popcount on m with an appropriate mask, we can quickly determine how many of the spots to the left of the current item have already been filled. This algorithm is attractive as popcount is available as an instruction on many modern computers.

For other computers, algorithm C.2 [22] can be used. It does not need popcount but requires an integer type with $k\lceil \log_2 k \rceil$ bits to be available instead. For the 15 puzzle, this is just a 64 bit type, but for the 25 puzzle, a type with at least 125 bits is needed already.

Algorithm C.2 *Given a permutation σ of k items, compute the inversion table $b_0, b_1, \ldots, b_{k-1}$ using integers with at least $k\lceil \log_2 k \rceil$ bits.*

1 $w \leftarrow 2^{\lceil \log_2 k \rceil}$
2 $m \leftarrow 1w + 2w^2 + \ldots + (k-1)w^{k-1}$
3 **for** $i \leftarrow 0, 1, \ldots, k-1$ **do**
  3.1 $m \leftarrow m - w^{\sigma(i)}(w + w^2 + \ldots + w^{k-1}) \,\%\, w^k$
  3.2 $b_i \leftarrow \lfloor m / w^{\sigma(i)} \rfloor \,\%\, w$
4 **return** $b_1, b_2, \ldots, b_{k-1}$

This algorithm is easy to understand if you view m as a vector with k entries. Then, the algorithm initialises m with $m_i \leftarrow i$ and in each iteration decreases all m_i to the right of $\sigma(i)$, setting $b_i \leftarrow m_{\sigma(i)}$.

The same method can be used to reconstruct σ from the inversion table as well (ibid.):

Algorithm C.3 *Given an inversion table $b_0, b_1, \ldots, b_{k-1}$ compute the corresponding permutation σ using integers with at least $k\lceil \log_2 k \rceil$ bits.*

1 $w \leftarrow 2^{\lceil \log_2 k \rceil}$
2 $m \leftarrow 1w + 2w^2 + \ldots + (k-1)w^{k-1}$
3 **for** $i \leftarrow 0, 1, \ldots, k-1$ **do**
  3.1 $l \leftarrow w^{b_i} - 1$
  3.2 $\sigma(i) \leftarrow \lfloor m / w^{b_i} \rfloor \,\%\, w$
  3.3 $m \leftarrow m \,\&\, l \mid \lfloor m / w \rfloor \,\&\, \sim l$
4 **return** σ

Using the `pdep` instruction from the BMI2 instruction set [21] and the widely available intrinsic function ctz, an even faster reconstruction procedure obtains. However, availability of `pdep` is limited to the amd64 architecture as of October 2017.

Algorithm C.4 *Given an inversion table $b_0, b_1, \ldots, b_{k-1}$ compute the corresponding permutation σ using the* `pdep` *instruction.*

1 $m \leftarrow 2^k - 1$
2 **for** $i \leftarrow 0, 1, \ldots, k-1$ **do**
  2.1 $\sigma(i) \leftarrow \text{ctz}\left(\texttt{pdep}(2^{b_i}, m)\right)$
  2.2 $m \leftarrow m \,\&\, \sim 2^{\sigma(i)}$
3 **return** σ

The function $\text{ctz}(x) = \log_2(x \,\&\, -x)$ computes the number of trailing zeros in the binary representation of x with $\text{ctz}(0)$ undefined. The behaviour of the instruction $\texttt{pdep}(x, y)$ is explained in [21].

D Difficult Instances

In [23], Schütt et. al. try to find difficult 25 puzzle instances by starting with a 15 puzzle instance of maximal distance in the upper left corner of the 5×5 puzzle tray and then finding the hardest assignment of the remaining tiles using a breadth-first search. The hard instance they constructed is given on the left in figure D.1. While Schütt et. al. were not able to solve the instance optimally, they found the instance's distance to be either 140 or 142 moves.

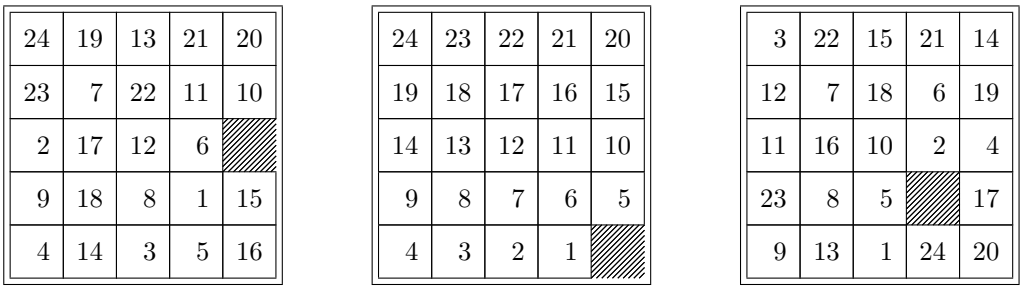


Figure D.1 *hard 25 puzzle instances*

During the work on this thesis, the author found that the configuration obtained by rotating the solved configuration by 180° is hard to solve, too. The small catalogue gives $h = 128$ for this configuration and no solution is obtained withinin a bound of 144 moves. The configuration solved by the author's program requiring the highest number of vertex expansions at the time of this writing is given on the right of figure D.1, has distance 110 and required the expansion of 3 288 483 999 260 IDA* nodes using the small catalogue given in appendix A.