University of Basel

Solving the Sliding Tile Puzzle with Post-Hoc Optimization

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Malte Helmert
Supervisor: Florian Pommerening

Benedikt Heuser
ben.heuser@unibas.ch
2021-050-257

13.7.2024

Acknowledgments

First and foremost, I would like to thank my supervisor Florian Pommerening for accompanying me with endless patience and readiness to help. Thank you as well to Prof. Nathan Sturtevant at the Univeristy of Alberta for allowing me to use his HOG2 research code and for taking the time to help me get familiar with it. I also extend my gratitude to the habitual dwellers of the ZG at the department, who made writing this thesis a much less solitary task. Finally, I greatly appreciate that I was given the opportunity to work on this topic by my examiner Prof. Malte Helmert.

Abstract

Solving the sliding tile puzzle is an important benchmark problem for testing informed search algorithms as it provides a large state space and is straightforward to implement. The key to finding solutions for the sliding tile puzzle efficiently, is to use high quality heuristic functions, that guide the state space search. Popular search algorithms such as the IDA* algorithm use these functions in combination with the path cost from the start node to decide which nodes to expand. The post-hoc optimization heuristic is one such function, that promises particularly high quality by combining the combination of many overlapping pattern database heuristics into a single value with the use of linear programming. Our thesis explores the impact of various inputs to the heuristic in the form of pattern database collections and compares their performance with the state-of-the-art.

Table of Contents

1 Introduction

The sliding tile puzzle has been a popular benchmark for comparing search algorithms for more than 50 years. It continues to provide an intuitive platform for testing and serves as a representative model for finite state spaces. Modern state-of-the-art sliding tile solvers employ heuristic search to find solutions for instances of the puzzle. In particular, algorithms like IDA^* have established themselves as the most popular. They find a deterministic sequence of actions to get from an initial state to a predefined goal state guided by the function $f(s) = g(s) + h(s)$. This function indicates to the algorithm which candidate nodes to expand next during search. The function $g(s)$ maps each state to its distance from the start state and $h(s)$ is a heuristic function. A heuristic function is function that takes a state and computes an estimated cost required to get from that state to the goal state. The solution that IDA* finds is optimal if the heuristic is admissible, which means that the estimate is a lower bound on the cost of an optimal solution.
Consequently, one of the main ways heuristic search algorithms can be improved is by designing more accurate heuristic functions. However, the trade-off for such improvements often lies in an increasing amount of required computational work and memory to compute these heuristics. Therefore, a balance has to be found between these two aspects.
Considering this, this thesis attempts to evaluate the viability of a particular heuristic called the post-hoc optimization heuristic for the sliding tile puzzle domain and tries to find configurations that yield a good performance in a series of experiments. The post-hoc optimization heuristic belongs to the class of abstraction heuristics, that compute heuristics based on a smaller abstract version of the state space instead of the full state space itself. The abstract state space is typically constructed in such a way that an optimal solution is easier to compute and that solution's length is then used as the heuristic. The common approach to abstraction heuristics for the sliding tile puzzle is to use so-called pattern databases [2]. Pattern databases are abstractions that abstract away all but a select number of tiles, which are collectively called patterns. Pattern databases map abstract states to the length of an optimal solution within that abstract space where only the tiles in their respective pattern are considered. Modern solvers typically employ multiple pattern databases and combine them to compute new heuristics. Korf and Felner explored this by experimenting with disjoint pattern databases [12]. As the name implies, this method only allows collections of

pairwise disjoint patterns, which makes their heuristics additive. This leads us to consider new methods that can extract more information by allowing overlapping patterns to obtain even better cost estimates. This is where post-hoc optimization comes in.

Post-hoc optimization is a novel approach to pattern databases that allows us to use previously unavailable collections of pattern databases and use them to compute better heuristics by including information from more interactions between tiles. It is an extension of ideas used for the sliding tile puzzle, but has not been tested on it specifically outside of a similar bachelor thesis which applied post-hoc optimization to the sliding tile puzzle and compared its performance to previous methods [10]. This thesis attempts to improve on these results by re-implementing the post-hoc optimization solver and testing its performance for a wider variety of configurations.

We are motivated by the potential post-hoc optimization shows with regards to the high quality of heuristics it can provide and the further refinement of the algorithm to the sliding tile puzzle in particular. Beyond the academic realm, understanding and improving heuristic search algorithms has practical implications across various domains, from robotics to video game AI.

In the following chapters we first outline background information and notation for the sliding tile puzzle, pattern databases and post-hoc optimization for an easy familiarization of the related theory and the notation we will use in the rest of the thesis. We then discuss the application of post-hoc optimization to the sliding tile puzzle in particular and describe its problem specific aspects. Finally, we present the results of our experiments and interpret their implications.

2 Background

2.1 Sliding Tile Puzzle

The classic sliding tile puzzle (or STP) is a two dimensional combinatorial puzzle composed of 15 square numbered tiles and one missing tile arranged in a four by four square frame. The tiles that are orthogonally adjacent to the space of the missing tile can be moved into that space, creating a new empty space where that tile was previously. The task of the puzzle is to reach a state where the numbered tiles are in row-wise ascending order starting from the top left and the empty space is in the top left position. This is a common convention used in scientific papers and digital implementations of the puzzle. For physical STPs, however, the goal state is traditionally considered as shown in Figure 2.1.

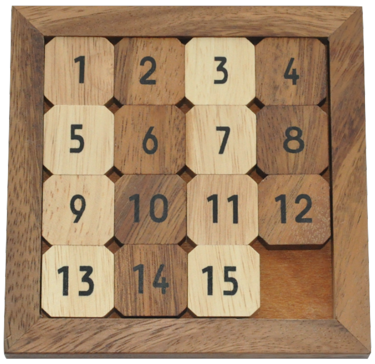


Figure 2.1: A sliding tile puzzle made of wood.

Formally, we refer to tiles by their labels $t \in \{0, ..., 15\}$, where the blank is represented by the number 0. A state of the sliding tile puzzle is a bijective function $s : \{0, ..., 15\} \rightarrow \{0, ..., 15\}$. It maps each tile to a position $s(t)$ going row-wise from the top left square. The puzzle is in a goal state when $t = s(t)$ for every tile in the puzzle.

To represent states we introduce the following notation. A state can be described by writing the numbers corresponding to the tiles in ascending order of their positions as a tuple. It should not be confused with the shorthand notation for the permutation matrix. For the

1	5	2	3
4	9	7	11
8	13	10	15
12	0	6	14

(a) Unsolved

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(b) Solved

Figure 2.2: Examples of a solved 2.2b and unsolved 2.2a STP state

state depicted in Figure 2.2a we would therefore write:

$$(1, 5, 2, 3, 4, 9, 7, 11, 8, 13, 10, 15, 12, 0, 6, 14) \tag{2.1}$$

For the actions of the puzzle we construct a set of operators $\mathcal{O} = \{\langle t, d\rangle \mid t \in \{1, ..., 15\}, d \in \{\text{left}, \text{right}, \text{up}, \text{down}\}\}$. Each operator $o \in \mathcal{O}$ corresponds to sliding a tile t in the direction d. For any state s the set of allowed moves is exactly the set of operators that slide a tile t that is orthogonally adjacent to the blank into the empty space. Consequently, applying an operator to a state is equivalent to swapping the position of that tile with the blank. For example, moving the tile 6 to the left in the state shown in (2.1) would result in the new state:

$$(1, 5, 2, 3, 4, 9, 7, 11, 8, 13, 10, 15, 12, 6, 0, 14) \tag{2.2}$$

We say the *distance* between two states is exactly the minimum number of moves required to get from the first state to the other. A *path* from one state to another is a sequence of operators $\langle o_1, ..., o_n\rangle$ that, when applied sequentially, transform the first state to the second state. In this thesis we often care about optimal solutions to the sliding tile puzzle. A *solution* to the puzzle is a path from an initial state to the goal state. We say that a path incurs a cost equal to its length where $cost(o) = 1$ for all $o \in \mathcal{O}$. Hence, an *optimal solution* is any solution for which there is no other solution with a lower cost.

The full state space of the STP has one state for every permutation of the tiles 0 to 15. However, the state space consists of two connected components with equal cardinality, where the one that contains the goal state is solvable and the other is not because there is no way to reach the goal state. From any solvable state, when correctly applying the allowed actions in the puzzle, only other solvable states can be reached, and vice versa. This was shown by Johnson and Story using a parity argument [9]. The cardinality of the solvable component is equal to $\frac{16!}{2}$.

The sliding tile puzzle is a specific instance of the more general n-puzzle, which can feature a square grid of any size greater than two. As a result, the sliding tile puzzle is often referred to as the 15-puzzle. Other common n-puzzles include the 24-puzzle, with a five-by-five grid, and the 8-puzzle, with a three-by-three grid. A more general variation is the mn-puzzle, which does not restrict the grid to a square and includes rectangular grids in addition to the standard variations. The properties described in the previous section apply to all variants, and the solving approaches we will discuss are equally applicable to all these variations.

2.2 Early Abstraction Heuristics for the STP

State space searches are guided by what we call a *heuristic function*, which is a function that maps states to a numeric value that represents an estimate of the distance of a given state a goal state. For the STP this means calculating an estimate of the number of moves required to solve the puzzle. The principle idea is that by guiding the search in the direction of operators that lead to a state with a lower heuristic value, the search algorithm will reach a goal state faster than by exploring the state space blindly. There are many ways of calculating heuristics, but abstraction heuristics have been the prevailing approach for puzzles such as the STP. *Abstraction heuristics* are a class of heuristics that utilize abstractions. An abstraction is a surjective function mapping states to a (usually smaller) state space. The key idea here being, that the number of moves to solve the abstract puzzle can then be used as the heuristic for the full puzzle.

A heuristic h is termed *admissible* when it maps each state to a value that is less than or equal to the value given by the perfect heuristic. The *perfect heuristic* h^* maps each state to the minimum cost of an optimal solution from that state to the goal state. Importantly, search algorithms such as the popular IDA* algorithm are guaranteed to yield optimal solutions when used with an admissible heuristic.

An example of an admissible abstraction heuristic for the sliding tile puzzle is obtained by ignoring the constraint that tiles can only be moved into an empty space. Counting the number of moves required to move each tile to its goal position yields a value known as the *Manhattan distance*. The Manhattan distance is an admissible heuristic because it results from removing a constraint from the problem, making the new abstract problem "easier" to solve since each tile can be moved while ignoring other tiles. The Manhattan distance has been a popular heuristic for the sliding tile puzzle due to its ease of implementation. However, as we will see in the following sections, better performance can be achieved.

2.3 Pattern Databases

Modern sliding tile puzzle solvers commonly employ pattern databases (PDBs), a widely used type of abstraction heuristic in search and planning [2]. They are what first allowed for Rubik's Cube to be solved optimally [11]. For the sliding tile puzzle, they were first introduced in 1996 by Culberson and Schaeffer [1, 2].

We define a *pattern* as a subset of the set containing all 16 tiles, including the blank. For any such pattern $P \subseteq \{0, \dots, 15\}$, we consider an abstract state space $\mathcal{S}_P$. Each abstract state $s_P \in \mathcal{S}_P$ is a function $P \to \{0, \dots, 15\}$ that maps each tile in the pattern to a position, analogous to the previous definition of states of the puzzle in section 2.1.

The abstraction provided by the pattern lies in the inclusion or exclusion of tiles. The behavior of the included tiles in the abstract puzzle is similar to their behavior in the full puzzle. Swapping the position of the blank with a tile not in the pattern simply changes the position of the blank without incurring costs. We sometimes represent the excluded tiles in the pattern with the $\square$ symbol or do not write them at all, rendering them indistinguishable from one another. The size of the abstract state space for a pattern P is given by $\frac{16!}{(16-|P|)!}$.

A *pattern database* is a mapping similar to a lookup table that assigns each abstract state a

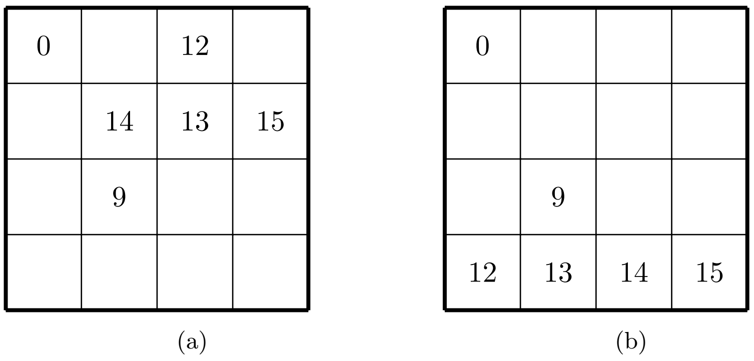


Figure 2.3: Examples two abstract states for the pattern $\{9, 12, 13, 14, 15\}$. 2.3a shows an arbitrary abstract state and 2.3b shows the abstract goal state for the same pattern.

heuristic value and are typically pre-computed using a breadth first search over the abstract state space. That value is exactly that state's distance from its respective goal state. The distance between states $s_1, s_2 \in \mathcal{S}_P$ in the context of the abstract state space is the cost of the shortest path α_{s_1,s_2}, where for every $o \in \alpha_{s_1,s_2}$:

$$cost_P(o) = \begin{cases} 1 & \text{if } t \in P \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

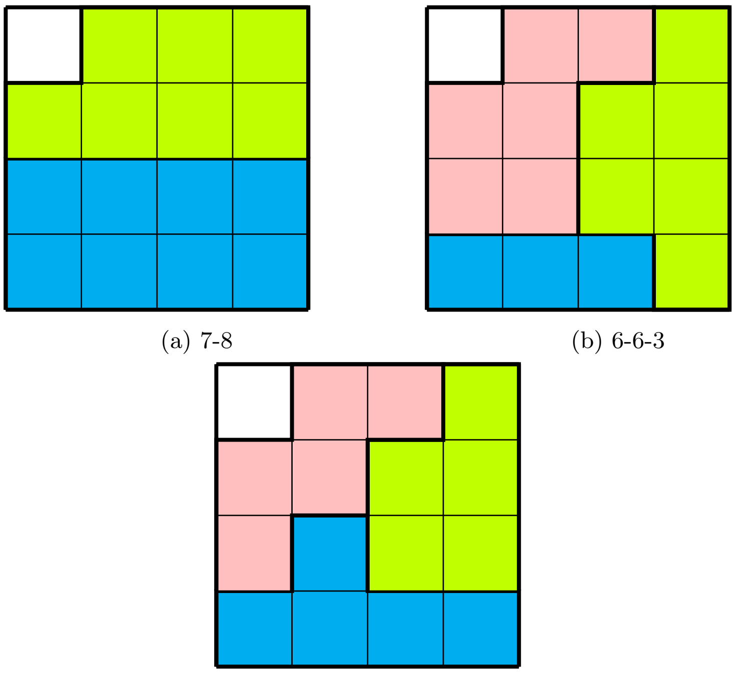
A *pattern database heuristic* is a function $h_P : \mathcal{S} \to \mathbb{N}$ that maps a full state to the value that the pattern database for the pattern P mapped to the abstract state that corresponds to this full state. All pattern database heuristics are admissible because every solution to the full puzzle is also a solution to the abstract puzzle. Therefore, the lowest cost solution to the abstract puzzle can only have lower or equal cost to the lowest cost solution to the full puzzle.

2.4 Combining Multiple Pattern Databases

Individual PDB heuristics are admissible, but they do not incorporate any information pertaining to excluded tile's positions. While it is therefore possible to perform a search using only a single pattern database heuristic, we would like our heuristic to be more informed. Therefore, in order to construct informed heuristics modern solvers typically use multiple pattern databases and combine their respective heuristics. In the following we will describe such methods for combining pattern databases.

Since each individual pattern database heuristic is admissible, any algorithm that always chooses any single one of these heuristics and discards the rest will be admissible as well. In particular, it makes sense to choose the heuristic that yielded the highest value, as it will be the closest estimate to the real cost. In this thesis we refer to this as the *maximum heuristic*.

Due to the way we construct PDBs, for any set of *disjoint patterns* (patterns, where each tile apart from the blank is only in one of the patterns) we can use the sum of all pattern database heuristics, as each tile's costs only count towards a single heuristic. We will refer to this heuristic the *plain additive database heuristic* PA. The patterns used in the sum can be determined during execution or be constant for every iteration of the search. If they are constant we say the pattern databases are *statically-partitioned* and *dynamically-partitioned* if the choice of patterns is redetermined for each iteration. At the time of this thesis, statically-partitioned database heuristics outperform dynamically-partitioned heuristics because of the increased overhead evaluating the collection of patterns causes. The best-performing static pattern collections for the sliding-tile puzzle in the literature have been (in order) the 7-8, 6-6-3, and 5-5-5 partitions shown in Figure 2.4 [4]. We interpret these images such that each set of tiles with same color is a single pattern. The tiles they represent are the tiles that would be at that location in the goal state. In the case of disjoint patterns we can display them all in the same image. However, in later parts of this thesis we use multiple images to represent overlapping patterns.



(a) 7-8 (b) 6-6-3

(c) 5-5-5

Figure 2.4: Visual representations of the three disjoint pattern collections with the best performance for the sliding tile puzzle. Each tile's color corresponds to a single pattern that it belongs to.

2.5 Maximum Matching Pairs

The idea for the *maximum matching pairs* comes from an extension on the idea of the Manhattan distance. It involves incorporating information about linear conflicts [6]. A *linear conflict* occurs when two tiles are adjacent and already in their goal row or column but are not in the correct order. The Manhattan distance heuristic would have these tiles move through each other, which is impossible without one tile moving out of the way for the other to pass. We can therefore improve the heuristic by adding two to the heuristic value for the moves needed to resolve each conflict.

Consider a pattern that consists of two tiles. There are two possible scenarios: either the two tiles can move to their goal positions without getting in each others' way or they get into a linear conflict where one needs to move out of the way for the other to pass. In the first of these cases the heuristic that the corresponding pattern database will map to will be exactly the sum of the Manhattan distances of the two tiles. In the second the heuristic would be exactly two higher because of the linear conflict, which is advantageous because it means the quality of the heuristic is better. We call this the *pairwise distance* of these two tiles. But now consider the state where a row of the puzzle contains the tiles (3, 1, 2, 4). In this case tiles 3 and 1 are in a conflict as well as tiles 3 and 2. We are able to resolve both linear conflicts if the tile 3 moves out of the way, meaning we only need to add two to the heuristic once.

From this observation comes the idea for the *maximum matching pairs heuristic* (MM) which was described by Felner in 2015 [3]. To calculate it we construct a graph called a *mutual-cost graph* where we represent each tile with a vertex and connect each pair of vertices with an edge labeled with their pairwise distance. The task is then to select edges in such a way that no vertex is connected to two of the selected edges while maximizing the sum of their labels, which can be done in $O(n^3)$ [14]. This sum is then exactly the value of the maximum matching pairs heuristic. For simplicity, the labels are only assigned the values exceeding their Manhattan distance, which helps with clarity and has other advantages we will see in the next section.

Felner develops the idea of MM further by generalizing the maximum matching pairs to higher-order groups of tiles, which transforms the graph into a hypergraph, for which the problem is NP-complete [5]. The task is then to choose a set of hyperedges of the MCG without vertices in common, so that the sum of the weights of the edges and hyperedges is maximized.

2.6 Weighted Vertex Cover Heuristic

Similar to the maximum matching pairs heuristic, Felner describes the *weighted vertex cover* heuristic. Consider the example state (3, 2, 1, 4) where each of the tile 1, 2 and 3 are in a linear conflict with each other. In this case the mutual-cost graph can be drawn as shown in Figure 2.5.

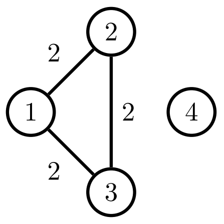


Figure 2.5: A mutual-cost graph for the state (3, 2, 1, 4). Nodes that are in a linear conflict are connected by an edge with a weight of two.

The idea is that if there is an edge with a label of 2 connecting two vertices in the mutual-cost graph, we know that either the first tile has to use two moves more or the other tile does. So to receive an admissible heuristic, we need to assign a number of moves to each vertex such that all constraints are met, and the total number of moves is minimized. In other terms, the problem that must be solved here is called the *weighted vertex cover* problem. A *vertex cover* is a set of vertices whose combined set of edges to which they are incident to are the set of all edges of a graph. Consequently, the *minimum vertex cover* is the smallest set of vertices that form a vertex cover. For example, if the weights are all equal to two, as in Figure 2.5, multiplying the number of nodes in our minimum vertex by two directly gives us the value of the weighted vertex cover heuristic for this state. When this problem is then translated into the context of a weighted graph for arbitrary pattern sizes we arrive at the general formulation of the weighted vertex cover. We must assign an integer value to each vertex, so that for each hyperedge, given a hypergraph with weighted edges, the sum of the values assigned to each vertex is at least as large as the weight of the hyperedge [3].

2.7 Post-Hoc Optimization

Post-hoc optimization (PHO) was first introduced by Pommerening, Röger and Helmert in 2013 [15]. The general idea is to solve a linear program constructed from multiple PDB heuristics by minimizing the sum of the costs of moving tiles.
We start by defining a set of variables $\mathcal{X}$ that holds a variable $X_o \in \mathcal{X}$ for every operator $o \in \mathcal{O}$. Each variable represents the number of occurrences of operator o in a solution from the current state. Every solution to the puzzle must fulfill the constraints of the form shown in (2.4). As a result, we know that the sum of all variables must be a lower bound on the cost of an optimal solution.

$$\sum_{o \in \mathcal{O}} X_o \geq h_P(s) \tag{2.4}$$

From the admissibility of pattern database heuristics we can see that for any pattern P the pattern database heuristic is a lower bound on the sum of all variables induced by the pattern. Because the cost operators that do not affect the tiles in the pattern have a cost of zero, we can tighten the bound to the set of operators that affect the pattern.

$$\sum_{o \in \mathcal{O}_P} X_o \geq h_P(s) \tag{2.5}$$

We then construct a linear program (LP) from a set of patterns. Minimizing the sum of all variables X_o is equivalent to minimizing the length of our solution. Because we want minimize the number of moves in our solution, we write the objective function as the sum of all variables, and we use (2.5) for each pattern as our constraints.

$$\begin{array}{llll} \text{minimize} & & \sum_{o \in \mathcal{O}} X_o & \\ \text{subject to} & \sum_{o \in \mathcal{O}_P} X_o & \geq h_P(s) & \text{for all patterns } \; P \in \{P_1, ..., P_n\} \\ & X_o & \geq 0 & \text{for } \; o \in \mathcal{O} \end{array} \tag{2.6}$$

We call the function that maps a state to the solution of its linear program the *post hoc optimization heuristic*.

2.8 Offline Post-Hoc Optimization

Computing a linear program for every generated state is computationally expensive because it can mean computing thousands of LPs during search. *Offline post-hoc optimization* (OPHO) is a variant of post-hoc optimization originally described by Höft, Speck and Seipp in a paper where they explored ways of reusing previously computed results using a technique called sensitivity analysis [7]. The idea is to precompute a single linear program for each of a set number of sample states, allowing us to then use the results for the search by maximizing over the estimates. The main idea is that for an large number of sample states the OPHO heuristic approaches the PHO heursitic in terms of heuristic quality as it becomes more and more likely that at least one partition is at least close to optimal for that state.

The dual of a linear program is a second LP that is derived from the first (called the primal) such that variables in the primal become constraints in the dual and the constraints of the primal become the variables of the dual. Additionally the objective sense is inverted. The strong duality theorem states that any optimal solution to the primal has a solution to the dual which is equivalent [13]. If we rewrite our LP from (2.6) we get the dual:

$$\begin{array}{llll} \text{maximize} & \sum_{P_i \in \{P_1,\dots,P_n\}} Y_i h_P(s) & & \\ \text{subject to} & \sum_{i \text{ if } t \in P_i} Y_i & \leq 1 & \text{for every tile } \; t \in \{1,\dots,15\} \\ & Y_i & \geq 0 & \text{for } \; i \in \{1,\dots,n\} \end{array} \tag{2.7}$$

The dual solution is the set of weights Y that assign a weight between zero and one to each of the PDB heuristics. We call this a cost partition, as it partitions the cost of multiple PDB heuristics such that the weighted sum is admissible.

Because of the duality theorem we know that the weighted sum of the heuristics is exactly equal to the heuristic value computed by the primal. Therefore, the key idea is that if we precompute these weights for a number of sample states, we can then reuse the cost partitions during search. In other words the offline post-hoc optimization heuristic is the maximum value of the weighted sums of the weights Y with the PDB heuristics for a given state and where S is a set of sample states.

$$h^{OPHO}(s) = \max_{s' \in S} \sum_{P_i \in \{P_1,\dots,P_n\}} Y_i h_{P_i}(s) \tag{2.8}$$

3 Post-Hoc Optimization for the Sliding Tile Puzzle

The sliding tile puzzle has been a popular benchmark problem for several decades. The efforts to solve it more efficiently have inspired many developments in state space search guided by heuristics. While post-hoc optimization was formulated for arbitrary planning tasks, it is an extension of the ideas originally developed for the sliding tile puzzle. In the following sections we will discuss how we adapt post-hoc optimization back to the puzzle that inspired it.

3.1 Adapting Post-Hoc to the Domain

In its general form, post-hoc optimization uses one variable per operator. We can reduce the number of the variables needed by using the domain-specific properties of the STP domain. Consider the operators $X_{\langle 1,\mathrm{up}\rangle}$, $X_{\langle 1,\mathrm{down}\rangle}$ $X_{\langle 1,\mathrm{left}\rangle}$ and $X_{\langle 1,\mathrm{right}\rangle}$. Each of of these variables represents the number of times the first tile is moved in a particular direction. Whenever one of these variables appears in the linear program, all four of these variables appear together. This means we can combine all directions into a single variable X_t per tile, reducing the number of variables we need to account for in our calculations by a factor of four. This grouping of the variables is called a *relevant operator partition.* (3.1) shows the new form of the LP.

$$\begin{array}{llll} \text{minimize} & \sum_{t\in T} X_t & & \\ \text{subject to} & \sum_{t\in P} X_t & \geq h_P(s) & \text{for all patterns } \ P \in \{P_1, ..., P_n\} \\ & X_t & \geq 0 & \text{for } \ t \in T \end{array} \tag{3.1}$$

During search, each generated node requires us to solve exactly such a linear program. However, the only change in between each of the problems is on the right-hand side. Whenever a new state is expanded, the pattern database heuristic values are retrieved and replaced. Restarting the linear program solver from the previous problem's solution can help reduce the time it takes for the solver to evaluate the linear program. In our implementation this is handled by the CPLEX library.

3.2 Relationship to Existing Heuristics for the Puzzle

Consider the case where we are using post-hoc optimization with the set of all patterns of size one. In this scenario, the values that the LP assigns to each of the constraint variables will be exactly their Manhattan distance. This is because in the abstract state space of a pattern database with a pattern of size one, the tile can simply move to its goal position without any conflicts, which results in the PDB heuristic being the same heuristic as the Manhattan distance of that one tile. If this is the case, the constraints of the linear program will each be of the form $X_t \geq h_{\{t\}}(s)$ with a single variable per constraint. Therefore, each variable only appears once. The optimal solution to the LP will then be to assign exactly the value of the respective PDB heursitics to the variables. Finally, because the objective function is the sum of all variables the PHO heuristic will have the exact same value as the MD heuristic.

Furthermore, a pattern database with a pattern of two tiles exactly maps states to the pairwise distances of two tiles. Therefore, using post-hoc optimization with only patterns of size two means that every solution must assign values to these two such that their sum is at least their pairwise distance. This is a similarity of post-hoc optimization with the weighted vertex cover heuristic. The nodes in the MCG for the WVC are exactly the variables of PHO. The WVC heuristic requires the sum of the assigned values to be at least the value of the edge label while PHO also requires the sum to be at least as large as the value of the PDB heuristic on the right hand side of the constraints. However, there is an important difference between the two. The most commonly used form of the post-hoc optimization heuristic allows real values to be assigned to each of the LP variables, whereas the weighted vertex cover heuristic only permits integer values. Restricting the values to integers effectively makes the problem harder to solve if P $\neq$ NP, as it is in NP while PHO is solvable in polynomial time [3]. However, the quality of the heuristic can be slightly reduced in some cases. This is because in linear programs the optimal solution can also be a real value that is lower than the rounded up integer solution. Note that post-hoc optimization does not explicitly exclude a restriction to integer values, so WVC heuristic is exactly equivalent to the PHO as an integer program version.

3.3 Pattern collections

The performance of post-hoc optimization for a given state of the sliding tile puzzle depends on the collection of patterns used, making it crucial to identify pattern collections that perform well for arbitrary instances of the puzzle. However, testing every possible collection of patterns is infeasible due to the number of possibilities ($2^{2^{15}}$) and limited computational resources, as will be outlined in Section 4.1. To calculate the post-hoc optimization heuristic for a given state, we must select a collection of pattern databases and solve a linear program with constraints corresponding to these databases. The quality of these constraints directly impacts the heuristic's effectiveness. Therefore, our primary contribution in this thesis is to present guiding intuitions for choosing high-quality pattern collections, despite the vast number of possible combinations that make exhaustive testing impractical.

In a first direction, when comparing the performance of different collections of pattern

size	mem. p. PDB	ratio	req. total	num. patterns
1	0.22 KB	≈ 14	3.30 KB	15
2	3.13 KB	≈ 13	319.26 KB	102
3	40.67 KB	≈ 12	18.50 MB	455
4	488.65 KB	≈ 11	667.00 MB	1365
5	5.54 MB	≈ 10	1.62 GB	3003
6	55.37 MB	≈ 9	290.00 GB	5005
7	0.48 GB	≈ 8	3.08 TB	6435
8	3.86 GB	≈ 7	24.83 TB	6435
9	27.06 GB	≈ 6	135.43 TB	5005

Table 3.1: Columm one shows the required memory for a PDB of the given size. Column two shows how many of that size of PDB require the same amount of memory as one of the size that is one larger. The third column indicates how much memory is required to store all PDBs of that size. The last column shows the number of patterns that exist of the given size.

databases we must consider their sizes. Consider the case of two collections A and B containing an equal number of patterns, but the sum of the sizes of the respective pattern database files is larger for A than B. The pattern databases for A requires more memory and more computational work to generate than those for collection B. Furthermore, as we described previously, the larger PDBs have a larger abstract state space, that potentially holds more information relevant to the heuristic. Therefore, if we run the same algorithm once with collection A and once with collection B, and we were to measure a shorter run time for A than for B, that does not mean that A is strictly better than B. For reference, in Table 3.3 we tabulated the sizes up to size nine along with the size ratio between each size and the next, and the number of patterns that exist for that size. Note that the required memory of a pattern database only depends on the size of its pattern.

This observation leads us to our first intuition. With some prior indication through exploratory testing of our solver, we came to the hypothesis that a smaller set of large overlapping PDBs might yield a better performance than a large set of small PDBs. Natuarally, the sets must be chosen such that the sum of the required memory is equal as we described above. The reason we believe this might be the case is that using fewer large PDBs reduces the number of constraints for which the LP must be solved.

The second intuition we formulated was that pattern *connectedness* would have a correlation with the performance of PHO. We introduce here a measure for connectedness for which this hypothesis can be tested. We call the connectedness of a pattern P the sum of orthogonally adjacent pairs in that pattern.

$$|\{(t_1, t_2) \mid t_1, t_2 \in P_i \text{ where } t_1, t_2 \text{ are orthogonally adjacent and } t_1 \neq t_2\}| \qquad (3.2)$$

Intuitively, we expect patterns where the majority of tiles are placed next to each other in their respective goal positions to perform better than patterns that consist of many small non-adjacent groups of tiles. The motivation behind this is that to get any tile into its target position the tiles around it are the most likely to have to be moved out of the way or swapped places with that tile.

To further examine these intuitions we designed experiments that isolate these factors as well as possible which we describe in Chapter 4.

3.4 Optimizing Offline Post-Hoc Optimization

Offline post-hoc optimization is a slightly different approach from the regular post-hoc optimization heuristic. While PHO computes a linear program for every generated node, offline post-hoc optimization calculates sets of weights upfront for a series of sample states.
We include this variant of post-hoc optimization in this thesis on the one hand because it allows us to observe the weights that are given to different patterns, shedding some light on their individual relevance and on the other because we would like to test whether we are able to find configurations that yield performance that is close to the performance of regular post-hoc optimization with shorter run times. The fundamental trade-off lies between accuracy and computation time. Using more sample states improves the heuristic, but also makes the heuristic more expensive to compute. This is because for each additional set of weights an additional weighted sum as can be seen in (2.8) must be evaluated.
In this thesis we choose the set of sample states uniformly from the solvable component of the full state space of the puzzle. We do this because the solver should be general in the sense that it should work well for arbitrary states. Therefore, choosing states like this is the most representative method for the expected input.
One decisive advantage of calculating cost partitions in advance is that we can optimize the collection of weights over which we maximize during the search. In this thesis we explore this in two main ways. The first is to reduce the number of heuristics in the sum. If a particular PDB heuristic is always given a weight that approaches zero, we can simply remove the pattern entirely without significantly sacrificing heuristic quality. For PDB heuristics that have a weight of exactly zero for every sample state doing this does not impact the heuristic at all, so cutting these does not even have any trade off. We will see in Chapter 4 that this can drastically reduce the number of heuristics that need to be included. The second optimization that we can perform is to reduce the number of cost partitions that need to be compared. Trivially, we can remove any duplicate cost partitions, where each of the weights are the same. Furthermore, we can check if any of the partitions yield the same heuristic value as the partition calculated for that state for many of the other sample states. If so, we can remove those other partitions. This reduces the number of weighted sums that we need to calculate to find the maximum heuristic out of the set of weight vectors during the search.

4 Experiments

In this section we present the experiments conducted to evaluate our hypotheses. We attempt to find pattern collections for which the performance of post hoc optimization is satisfactory. We also determine the overhead solving a linear program for every visited state creates and quantify the quality of the heuristic for different collections and variations.

4.1 Methodology

We will briefly describe the setup we used to run our experiments. We programmed a custom implementation of a sliding tile puzzle solver in the C++ programming language on top of the existing HOG2 research code by Sturtevant [17]. HOG2 provides a sliding tile puzzle state space implementation and an interface for generating pattern databases. It also implements the IDA* search algorithm which performs the core search. For each expanded node, a linear program's bounds are updated using values retrieved from pattern database files generated in a precomputation step. Solving the linear program is handled by IBM's CPLEX solver version 22.1.1 using the C-callable interface [8].

Experiments use the Python package Lab [16] to run various configurations of the solver on a series of benchmark sliding tile puzzle instances. In this thesis we use two sets of benchmarks. The first is the set of 100 Korf instances [4], named after Richard Korf, who used them in his research. The Korf instances include states with very long optimal solutions. This can cause problems when running a very large amount of runs because experiments would need to run for too long to get results in a reasonable time. Therefore we constructed a second set of 100 instaces where solution lengths are limited to between 12 and 25 moves, which we will refer to as the Heuser instances analogously and for simplicity. We used a random-walk from the goal state with a set walk length of 25 and removed any states where the optimal solution was under 12 moves. We did this until we had exactly 100 instances.

The metrics we use to determine the performance of a run are the following:

- **System total run time:** We measure the system time from the moment the search algorithm starts to the termination of said search. Lower run times imply a more efficient execution of the search.

- **Number of expanded nodes:** We use the number of expanded nodes as a metric for the heuristic quality. If the solver expands fewer nodes there is less ambiguity about which move should be made next from a current state.
- **Number of generated nodes:** Analogously to the number of expanded nodes, the number of generated nodes provide a similar metric that is usually about three to four times the number of expansions, as there are always between two and four possible directions the blank tile can go for any given state.
- **Memory usage:** The memory usage of the solver on a given algorithm is an indicator of its resource efficiency. Runs can have very low run times but high memory requirements and might therefore still not be preferable.
- **Generated nodes per time:** By measuring the number of generated nodes against the run time we can get a sense of the average computational work required per expansion.

4.2 Pattern Sizes

In a first experiment, we want to test our hypothesis holds, namely that it is preferable to use a small collection of large pattern databases over a large collection of small patterns.
To ensure that the amount of memory is roughly equal for all collections we need to construct them such that the sum of the required memory the of PDBs in each collection is equal. We can ensure this by constructing collections using a number of PDBs of a given size. We can then build a collection using smaller PDBs of a given size with the same total memory by multiplying that number by ratio between the memory requirements of the respective PDBs sizes. We can see from Table 3.3 that for small PDB sizes there are not enough such patterns to satisfy this constraint. Consider the case of ten patterns of size three, which is near the minimum number of such PDBs for which collections are consistently sampled such that each tile is included in at least one pattern. To reach the same amount of memory usage we would need $10 * 13 = 130$ PDBs of size two which exceeds the number of unique patterns of that size. If, conversely, we choose very large PDBs we run into memory constraints in terms of the storage needed to store all PDBs of the given size. Consequently, we perform the experiment with collections of the sizes four to six.
In Table 4.1 we present the results of running this experiment with 200 uniformly sampled pattern collections per PDB size. Each of the collections were tested on the Korf instances as well as the Heuser instances set. We can see that the average number of expansions is the lowest for the 550 patterns of size four. This shows that using many small PDBs can work well with PHO, and does not directly confirm our hypothesis. On the other hand, we can clearly see that the average number of generated nodes per second and the average run time are significantly better for the large PDBs. We believe that this is a result of the LP taking longer to solve as the number of constraints are increased. This means, that if we want to get the best heuristic out of a limited amount of memory, it can be worthwhile to use many small PDBs. The 100 Heuser instances are used in addition to the Korf instances because the have the property that every solution is between lengths 12 and 25. When

collections	expansions	run time	gen. per second	memory
550 patterns of size 4	3319451	33.21 sec	3162.59	331.7 MB
50 patterns of size 5	98868551	270.65 sec	11356.11	333.2 MB
5 patterns of size 6	17171635	22.04 sec	23941.32	372.1 MB
550 patterns of size 4	720	665.05 ms	3630.4	331.7 MB
50 patterns of size 5	723	483.43 ms	8061.5	339.4 MB
5 patterns of size 6	767	258.74 ms	9687.7	382.2 MB

Table 4.1: The results of PDB size comparison experiment on the Korf (top) and Heuser (bottom) instances. The columns show the average number of expansions, the average run time instance, the average number of generated nodes per second and the average memory utilization.

the solution lengths are short, the effect of the difference in heuristic quality is less strong, therefore, we can see more clearly for this set of instances how the run time is impacted by the speed at which nodes are processed by the solver. Additionally we can see, that for the Korf instances the five patterns of size six produced a lower heuristic quality than the 550 of size four, but still significantly outperformed the 50 PDBs of size five. We were not able to find an explanation to why this effect was not present for the Heuser instances.
In summary, we found that if we are solely interested in the heuristic quality, it is generally preferable to use many smaller PDBs. However, the solver needs more time to find a solution to the linear program, which means that for the domain of the 15-puzzle, using a small set of large PDBs finds solutions with a shorter average run time.

4.3 Pattern Connectedness

To verify our hypothesis that pattern connectedness correlates with a collection's performance, we investigated the effect of pattern connectedness on the solver's performance by designing sets of pattern collections that were identical in both the number of patterns and their sizes. We show four such collections with varying levels of connectedness in Figure 4.1. We found that indeed the collection (a) with the most connected patterns outperformed the others on average with a shorter run time and less expansions for every instance out of the 100 Korf instances used in the experiment. The collections with lower scores in connectedness generally performed worse. However, the number of expansions was lower than expected for the collection (c). It achieved better results than collection (b) despite a lower connectedness score. We attribute this to the two-by-two chequered patterns in the collection, whose connected two-by-two squares have the advantage of better incorporating information on local conflicts. We come to this conclusion because the collection (d) replaces exactly these patterns with even less connected pairs and yields the poorest performance of these collections.
Furthermore, after having explored connectedness on a curated set of pattern collections with four large patterns we would like to verify if our hypothesis holds for a wider selection of collections. To do this we calculated the connectedness for each possible pattern of a given size and sorted the patterns into bins according to their connectedness. We repeatedly randomly sampled 15 patterns from the same bin to build series of collections with similar connectedness scores. We can see from Figure 4.2 that the distribution of connectedness is

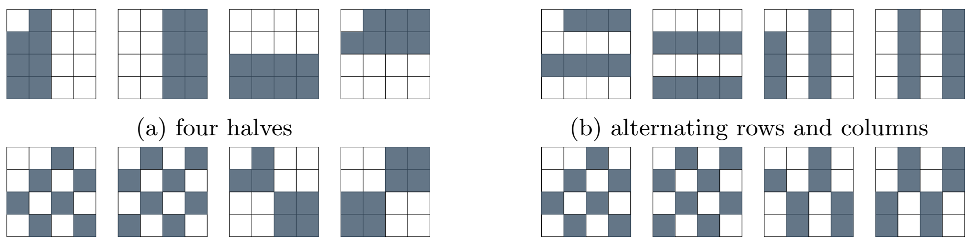
collection	expansions	run time	gen. p. sec.	memory	connectedness
a	947769	1.21 sec	26081.86	13.22 GB	36
b	22855157	24.31 sec	28390.91	13.22 GB	22
c	10602617	11.47 sec	29116.58	13.22 GB	14
d	24372382	25.70 sec	28612.02	12.71 GB	8

Table 4.2: Results of running four specific pattern collections from Figure 4.1 composed of two patterns of size eight and two of size seven on the set of 100 Korf instances. Column one shows the total number of expansions for all instances. Column two shows the average run time per instance. Column three shows the number of generated nodes per second. Column four shows the average peek memory utilization per instance. Column five shows the connectedness score.

roughly Gaussian. This means, that if we were to individually use each level of connectedness as a bin, we would have the problem, that the bins for the most and least connected patterns would be so small that the same patterns would be selected every time. To combat this, we made the bins two times as large, grouping two levels of connectedness together. As a consequence, the average connectedness is not the same for every collection out of a given bin, which is why we provide the average connectedness of the patterns in the collections for each connectedness level. Finally, we compared the average performance of each of these collections with each other.

Table 4.3 shows that the collections containing patterns with high connectedness caused fewer expansions, while also generating fewer nodes per second. We can see that our hypothesis, namely that more connected patterns are preferable, holds with regards to heuristic quality. However, we observe that the number of generated nodes per second are inversely correlated with the connectedness. This likely means that the LP solver is taking longer to solve LPs where the patterns are more connected.

In summary, connectedness is a good indicator for the heuristic quality that patterns provide. Conversely, the number of generated nodes per second is generally lower for collections with connected patterns.

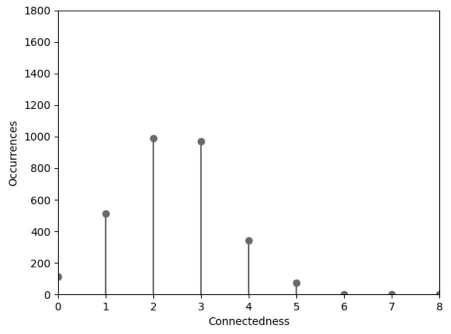

(a) four halves (b) alternating rows and columns

(c) chequered single and double (d) chequered variant

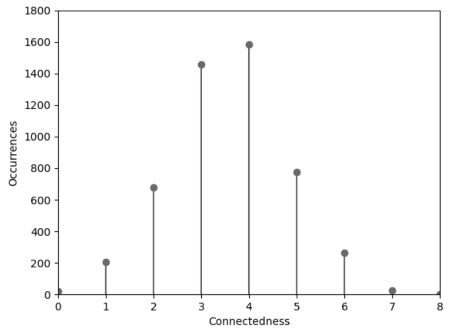
Figure 4.1: Four pattern collections with varying levels of connectedness with 8-8-7-7 pattern schemes.

4.4 Building Weight Vectors for Offline Post-Hoc Optimization

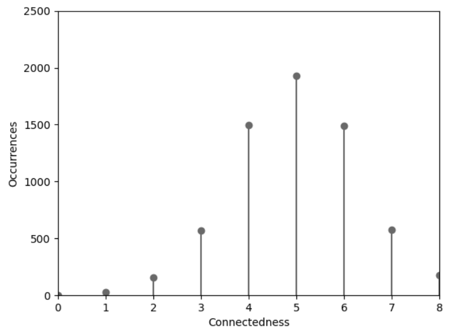
Offline post-hoc optimization can be broken up into two steps, a precomputation step and a search step. The precomputation step generates the weights for each of the sample states while the search step performs the actual search using the maximization over the weighted



(a) Size five



(b) Size six



(c) Size seven

Figure 4.2: The distribution of connectedness among all patterns for the pattern sizes five (a), six (b), and seven (c). It can be observed that the distribution is roughly Gaussian

connectedness	expansions	run time	memory	gen. p. second
.87	132290	.27 sec	332.85 MB	9752.91
2.66	115211	2.49 sec	332.85 MB	4413.49
4.33	91008	.96 sec	332.85 MB	3516.44
6.08	77541	2.32 sec	332.85 MB	1991.32

Table 4.3: Results of running 200 generated collections for each of four levels of connectedness on the set of 100 Korf instances. The columns show the average connectedness of patterns in the collections, the average number of expansions, average number of nodes generated, average run time and average number of expansions per second.

sums with the precomputed weights as the heuristic. In this section we will discuss the precomputation step. The results of the search step will be discussed in the next section.

In a first direction we explored the weight vectors that the precomputation produces, and try to observe the effect of the optimizations we described in Section 3.4. Our implementation calculates a weight vector for every sample state. In a first step, all patterns, that never recieve a non-zero weight are removed. Next, all weight vectors that are duplicates of other weight vectors are removed. Finally, we remove any weight vectors for which there is already another weight vector whose weighted sum with the PDB heuristics for that sample state yield the same heuristic value.

In initial test, we ran our implementation on each set of all patterns of a given size, up to size five, for exactly 100 sample states, without any of the optimizations. We observed

the calculated weights manually. We found that on average each weight vector only had a small number of non-zero weights. For size four the there were on average 8.56 out of 1820 non-zero weights per vector and 7.4 out of 3003 non-zero weights for size five. This had the effect, after applying the optimizations the first optimization, namely removing PDBs that never have a weight of more than zero, reduced the number of PDBs drasticly. For example, for the patterns of size five, the number of PDBs was reduced from 3003 to 421 from just this optimization. For these experiments the other optimizations, did not remove any weight vectors. In later experiments we refer to this collection for size five as 'OPHO-5'. To get an intuition for the patterns that remained we rendered Figure 4.3, which shows each of these patterns visually.
For large PDBs, we cannot generate all possible PDBs because of the run time and memory constraints. Therefore we are more limited in what we can try. As we observed in Section 4.3, patterns with a high connectedness score produce a better heuristic than those with low scores. That implies, that if we were to generate weights for the handmade patterns in Figure 4.1 from our previous experiments, the most connected PDBs ought to get the highest weights out of all of them. This was indeed the case. The only patterns that receive non-zero weights at all, when we run the experiment on 100 sample states are the patterns shown in Figure 4.4. Every weight vector had exactly two entries with a weight of one while the others were zero. If we regard the two patterns on the same line as an additive pair, either the first pair or second pair patterns had a weight of one in 84 of the 100 vectors while the last pair received a weight of one in 14 vectors, and the remaining patterns account for the remaining 2. In a later experiment we refer to this set of weight vectors as 'OPHO-4x8-4x7'.
For PDBs of size six, we were not able to compute weights for all 5005 patterns, because our attempt to do so ran for multiple days without terminating. Therefore, using what we know from Section 4.4, we decided to test the offline post-hoc optimization on the set of the 25 PDBs of size six with a connectedess score of seven, which is the maximum possible for this size, together with the 262 patterns with a connectedness score of six. For this collection of 287 patterns we then generated the weights. We noticed that for this collection 45 percent of weight vectors used a combination of 6 PDBs each with a weight of $0.\overline{33}$. We performed this once for 100 and once for 1000 sample states. In later experiments we will refer to these sets of weights as 'OPHO-6*'. The '*' indicates that we are only using the PDBs of size six where the connectedness is at least a value of six.

4.5 Solving the Sliding Tile Puzzle Offline

In Section 4.2 we identified the number of generated nodes per second as a major constraint for using large collections of small PDBs compared to small collections of large PDBs for regular post-hoc optimization. We also formulated that we believe that offline post-hoc optimization could be a good way to get around this. We believe this because OPHO lets us trim down the selection of PDBs in the collection, and also do not need to solve any linear programms during search. Instead, we calculate the maximum of the weighted sums of each PDB heuristic with the weight vectors we produced during the precomputation step. In this section we try to determine whether this approach can be a good option.

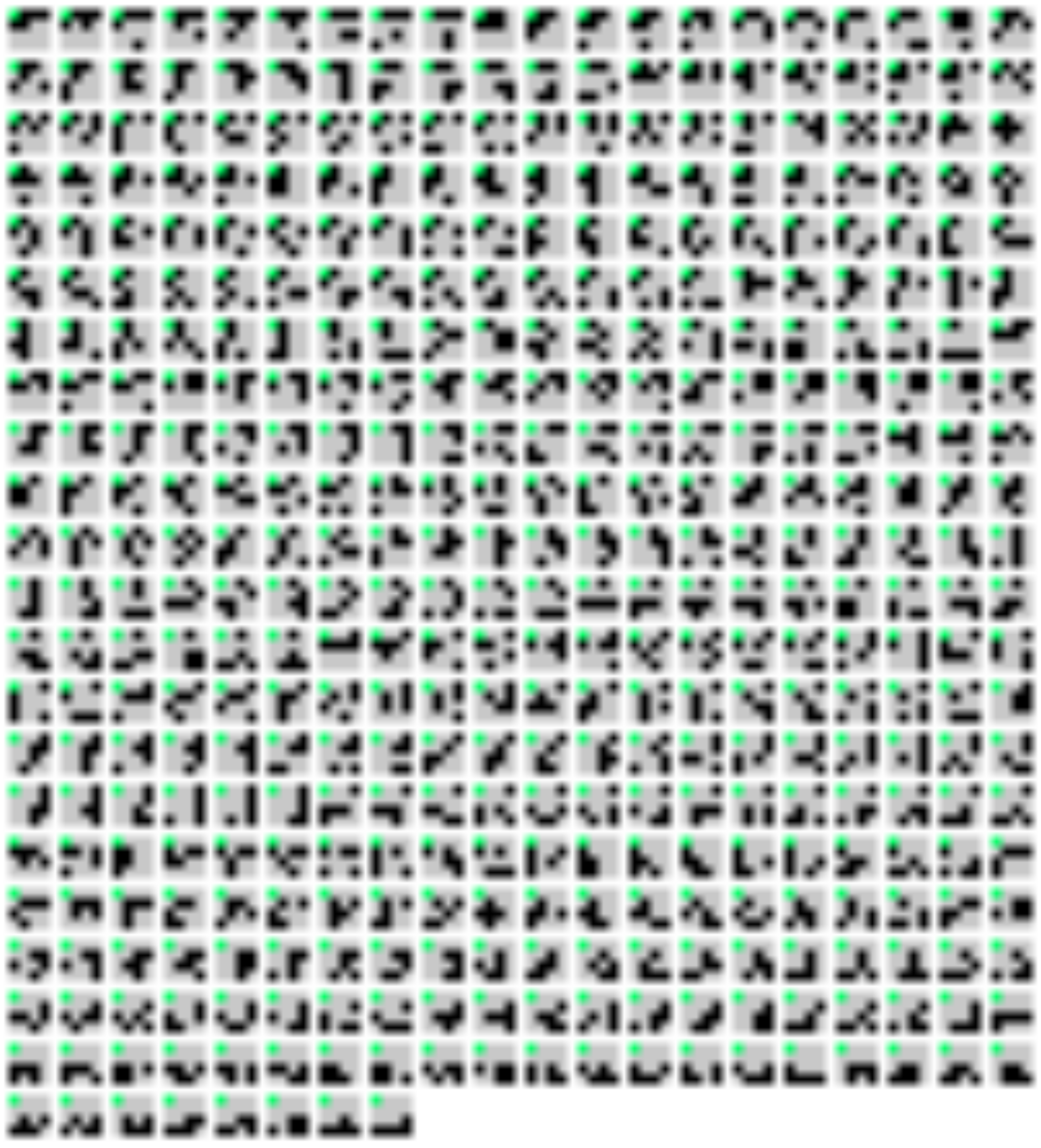


Figure 4.3: Image representing the patterns that had non-zero weights in the set of weight vectors 'OPHO-5', which was computed for the set of all patterns of size five for 100 uniformly sampled sliding tile puzzle states.

To compare whether offline post-hoc optimization can outperform regular post-hoc optimization, we run the collections we described in Section 4.4 with both the offline and regular post-hoc optimization solvers. For offline post-hoc optimization we used exactly the weight vector sets 'OPHO-6*', 'OPHO-5' and 'OPHO-4x8-4x7' that we described in Section 4.4. For regular post-hoc optimization we used the same same PDBs as for OPHO. This means, for example, 'PHO-6*' and a sample size of 100 means that we used regular post-hoc optimization with the set of all patterns that received a non-zero weight in the weight vector set 'OPHO-6*'. This keeps the memory utilization and the maximum possible heuristic quality constant between OPHO and PHO. The reason why we want this is that we primarily want to observe the number of generated nodes per second and the average run time in this

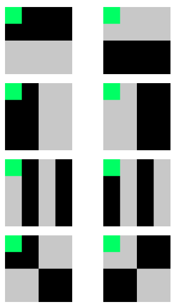


Figure 4.4: The patterns shown in this figure are the patterns that received non-zero weights in the set of weight vectors 'OPHO-4x8-4x7', which was computed on the set of fourteen patterns shown in Figure 4.1 for 100 sample states. Each of these patterns received a weight of exactly one.

algorithm	samples	PDBs	expansions	run time	memory	gen. p. sec.
OPHO-6*	100	178	11260811	249.38 sec	10.12 GB	1415.45
PHO-6*	100	178	10362045	291.24 sec	10.12 GB	1245.23
OPHO-6*	1000	259	10149347	516.87 sec	10.12 GB	655.70
PHO-6*	1000	259	9924651	603.77 sec	10.12 GB	612.35
OPHO-5	100	428	10623867	8354.13 sec	2.35 GB	3959.19
PHO-5	100	428	9824505	11092.18 sec	2.35 GB	3162.64
OPHO-4x8-4x7	100	8	829121	2.12 sec	22.34 GB	12474.14
PHO-4x8-4x7	100	8	829121	1.05 sec	22.34 GB	25926.64

Table 4.4: Shows the results of running the algorithms described in Section 4.4 on the 100 Korf instances. The third column indicates how many PDBs were used by the given algorithm. We show the number of expansions, the average run time per instance, the memeroy utilization and the number of generated nodes per second for each of the runs.

experiment.

We can see from Table 4.4, that for larger numbers of PDBs, offline post-hoc optimization is able to generate slightly more nodes per second than regular pot-hoc optimization. Conversely, for small numbers of PDBs such as for 'OPHO-4x8-4x7' regular post-hoc optimization is faster. We can also see that the heuristic quality for offline post-hoc optimization is slightly reduced in comparison with regular post-hoc optimization, but even for only 100 sample states OPHO gets very close to PHO heuristic. Additionally, we can observe that for 'OPHO-6*', the number of generated nodes per second drops when we use a higher number of samples. We would need to do additional experiments to more rigorously confirm these effects.

4.6 Performance comparison with PA

Many state-of-the-art sliding tile puzzle solvers use plain additive pattern databases to achieve short run times. Incorporating our knowledge of choosing pattern collections for post-hoc optimization we can now compare the performance of the best plain additive pattern database heuristics with the best we were able to find for the post-hoc optimization heuristic.

To get an estimate of the overhead of the individual solvers of our implementation we ran

algorithm	expansions	average run time	memory	gen. p. sec.
PA-8-7	3744197	.04 sec	5.66 GB	2418129.31
PHO-8-7	3744197	3.60 sec	5.66 GB	32932.30
OPHO-8-7	3744197	.23 sec	5.66 GB	515462.21
PA-6-6-3	59738614	.42 sec	.21 GB	4348799.60
PHO-6-6-3	59738614	3.81 sec	.21 GB	35932.30
OPHO-6-6-3	59738614	.75 sec	.21 GB	463559.95

Table 4.5: Comparison between PA, PHO and OPHO with identical pattern sets on the 100 Korf instances. The columns show the number of expansions, average run time per instance, memory utilization, and the generated nodes per second.

algorithm	expansions	average run time	memory	gen. p. sec.
PA-8-7	3744197	.04 sec	5.66 GB	2418129.31
PHO-8-7	3744197	3.60 sec	5.66 GB	32932.30
PHO-9x7	1282083367	1533.88 sec	5.11 GB	26337.46
PHO-81x6	698209996	2153.28 sec	5.60 GB	1013895.30
PHO-405x5	9825454	11002.28 sec	2.35 GB	3134.14

Table 4.6: Table showing the results of the PA comparison experiment on the Korf instances. We show the total number of expansions, the average run time per instance, the memory required and the number of generated nodes per second.

the three algorithms PA, PHO and OPHO with the exact same patterns. When comparing with PA, we are limited to collections of patterns that are additive. Naturally, if we do not use overlapping patterns we do not get any benefit from using post-hoc optimization over PA, but it is the only way we can directly compare the difference in run time for the algorithms with the same heuristic. The results can be seen in Table 4.5.

In a final experiment, we tried to find collections that use the same amount of memory as the best performing PA collection 'PA-8-7'. Our approach was to pick unique patterns in order of connectedness for smaller PDB sizes until the sum of their memory equaled the memory required for the plain additive PDBs. For the lowest connectedness score, we chose the remaining number by hand while making sure that each tile was included in a similar number of patterns. For the patterns of size five this would have resulted in roughly 800 PDBs, for which the overhead was too great for the runs to finish in a reasonable time for PHO. Therefore, we decided to include a run using only half of the PDBs of size five instead. We present the results in Table 4.6. Surprisingly, the results for this experiment show that the quality of the heuristic was best for the plain additive collection. This is something that we are not fully able to explain and is something we would like explore further. Still, for the overlapping collections using smaller PDBs caused fewer expansions than the larger PDBs. Note that we did not perform this experiment for offline post-hoc optimization because the number of PDBs for OPHO depends on the number of samples making it very difficult to specify the exact number of PDBs we want to use. However, the runs for size six in Table 4.4 use only slightly more memory and can serve as a reference.

5 Conclusion

Testing every possible combination of pattern databases is infeasible. We have aimed to identify configurations that maximize the performance of post-hoc optimization. In doing so, we have confirmed several intuitions that guide our approach.

Firstly, we have demonstrated for the PDB sizes four to six, given constant memory, post-hoc optimization causes the least amount of expansions when using a many smaller pattern databases, when compared to few large PDBs. Conversely, the number of PDBs in the collection is proportional to the time needed to solve a single linear program during search. Additionally, we found that pattern connectedness correlates with higher heuristic quality but also increases the computational intensity of solving linear programs, thereby slowing down the algorithm. These opposing effects make connectedness a less useful indicator of runtime efficiency, but a valuable tool for improving heuristic quality.

Moreover, our experiments with offline post-hoc optimization revealed that only a small number of patterns are needed to produce an optimal solution for randomly sampled states. We also showed that offline post-hoc optimization can be a good option if we want to approximate the performance of a large set small PDBs without solving a linear program at every generated node.

Finally, we attempted to achieve better a better heuristic quality using post-hoc optimization than the best plain additive pattern database heuristic using the same memory constraints. Our results in this experiment were not able to achieve this, however, requiring further research.

In conclusion, there are several opportunities for further exploration of this topic. We do not yet claim to have found an optimal way of building pattern collections for post-hoc optimization. One question that came out of our research was why pattern connectedness and PDB sizes seem to be partially inversely correlated with number of generated nodes that our implementation of PHO was able to achieve.

Bibliography

[1] Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Advances in Artifical Intelligence: 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 402–416, Berlin, Heidelberg, 1996. Springer.

[2] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[3] Ariel Felner. Early work on optimization-based heuristics for the sliding tile puzzle. In *Planning, Search, and Optimization - Papers Presented at the 29th AAAI Conference on Artificial Intelligence*, AAAI Workshop - Technical Report, pages 32–38, Austin, Texas, 2015. AI Access Foundation.

[4] Ariel Felner, Richard E Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[5] Michael R Garey and David S Johnson. *Computers and intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[6] Othar Hansson, Andrew Mayer, and Moti Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, 1992.

[7] Paul Höft, David Speck, and Jendrik Seipp. Sensitivity analysis for saturated post-hoc optimization in classical planning. In *Proceedings of the European Conference on Artificial Intelligence*, pages 1044–1051. IOS Press, 2023.

[8] International Business Machines Corporation. *V22.1: User's Manual for CPLEX*, 2022. URL https://www.ibm.com/docs/en/icos/22.1.1?topic=optimizers-users-manual-cplex.

[9] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

[10] Damian Knuchel. Post-hoc optimization for the sliding tile puzzle. Bachelor's thesis, University of Basel, 2021.

[11] Richard E Korf. Finding optimal solutions to the Rubik's cube using pattern databases. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, pages 700–705, 1997.

[12] Richard E. Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1):9–22, 2002.

[13] J. Matousek and B. Gärtner. *Understanding and Using Linear Programming*. Springer, Berlin, Heidelberg, 2006.

[14] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. 1982.

[15] Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the most out of pattern databases for classical planning. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, page 2357–2364. AAAI Press, 2013.

[16] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://github.com/aibasel/lab/tree/main. Accessed 13.7.2014.

[17] Nathan Sturtevant. HOG2. https://github.com/nathansttt/hog2. Accessed 13.7.2024.