Post-hoc Optimization for the Sliding Tile Puzzle

Bachelor Thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Silvan Sievers

Damian Knuchel
damian.knuchel@unibas.ch
18-050-674

October 6th 2021

Acknowledgments

I wish to express my sincere thanks to my supervisor Dr. Silvan Sievers, for guiding me through the process of my first thesis. His support and feedback was always very helpful. I would also like to thank Prof. Dr. Malte Helmert for allowing me to do my thesis in the Artificial Intelligence Research Group. Furthermore I like to thank my family and friends for supporting and motivating me throughout the process.

Abstract

Pattern databases are one of the most powerful heuristics in classical planning. They evaluate the perfect cost for a simplified sub-problem. The post-hoc optimization heuristic is a technique on how to optimally combine a set of pattern databases.
In this thesis, we will adapt the post-hoc optimization heuristic for the sliding tile puzzle. The sliding tile puzzle serves as a benchmark to compare the post-hoc optimization heuristic to already established methods, which also deal with the combining of pattern databases. We will then show how the post-hoc optimization heuristic is an improvement over the already established methods.

Table of Contents

1 Introduction

Heuristic search algorithms are an efficient method for solving state space problems. The better the given heuristic, the better performs the search algorithm. The heuristic has to meet different expectations for the different heuristic search algorithms. In case of A* the heuristic must be *consistent* and *admissible*. For IDA* it must be at least *admissible*. A multitude of methods to calculate different admissible heuristics were developed over the years, each with its improvements. We are using the sliding tile puzzle as a benchmark to compare different heuristics. First, we will discuss several heuristics which are already adapted for the sliding tile puzzle. The simpler ones are the *Manhattan distance* [4] and *linear conflicts* [5]. Those can be derived directly from a given state. The difference being, that the Manhattan distance only considers each element of a state, a variable, by itself while the linear conflicts heuristics considers them in pairs. It gets interesting using pattern databases. Pattern databases include the perfect cost estimate for sub-problems [2][3]. The original approach is to generate multiple pattern databases, evaluate all of them for a state and choose the maximum value [2]. Korf and Felner [8] introduced a method which makes it possible to use the information of multiple pattern databases in one heuristic. By partitioning the variables into disjoint sets, each variable will only be considered once. Therefore it is possible to add up all the disjoint pattern database values of a given state and it still being admissible. This approach is called *statically-partitioned additive pattern databases*. The current state of the art, called *dynamically-partitioned additive pattern database heuristic* [4], uses a similar approach, but instead of using a static partitioning scheme of the variables, it uses any set of pattern databases. For each state, all possible additive subsets of the pattern databases are calculated. The subset with the largest sum has the best heuristics for the given state.

The next approach is the *post-hoc optimization heuristic* [10] which is currently only defined for abstract planning tasks. The post-hoc optimization heuristic uses a set of pattern databases and weights all the variables in a pattern for all given pattern databases using a linear program. The linear program must be reevaluated for every new state. The sum of the weighted variables results in an admissible heuristic. Our goal is to adapt the post-hoc optimization heuristic for the sliding tile puzzle and compare it to the previously mentioned heuristics. We also discuss possible optimizations for the post-hoc optimization heuristic regarding memory and time usage.

2
Planning

To get a conceptual background and to create a deeper understanding in this thesis, we will briefly explain relevant definitions.

2.1 Planning Task

To understand the working and terminology used further during this thesis, we will first have to talk about *SAS$^+$ planning tasks* introduced by Bäckström and Nebel [1]. A planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_*, c \rangle$, where $\mathcal{V}$ is a finite set of state variables v, each assigned to a domain $\mathcal{D}_v$. A *state* is a complete assignment of the state variables $v \in \mathcal{V}$. $\mathcal{O}$ is a set of operators o, where each operator contains an *effect* and a *precondition.* An effect describes the modification of a state and the precondition includes certain constraints. The operators' effect can only be applied if the precondition has been met, as result we get a new state, the *successor* state. A cost function $c : \mathcal{O} \to \mathbb{R}_0^+$ applies a cost on each operator. The beginning of a planning task is given by the *initial* state s_0. s_* is a sub-state, a partial assignment of the state variables, which includes the information needed for a state to be counted as a goal state.
The cost of a sequence of operators is the sum of the cost of each operator in the sequence. An operator sequence that leads to a goal is called a plan. The goal of a planning task is to find an optimal plan from the given initial state s_0 to the given goal s_*, where optimal means, that the cost must be minimal.

2.2 Heuristic search

There are many methods to optimally solve a planning task and heuristic search is one of them. A heuristic [9] is a function $h : s \to \mathbb{R}_0^+ \cup \{\infty\}$ where s is a state. The value serves as an estimate on how close a state is to the goal state. The smaller a heuristic is, the closer it thinks we are to a goal state. A heuristic which is exactly the cost of the optimal plan of given state is called the perfect heuristic. The search algorithm we will be using is *IDA**. IDA* stands for *Iterative-Deepening A** and as the name suggests, uses the A* algorithm. We are using the iterative-deepening approach of A* for its memory efficiency. To be optimal we have to use an admissible heuristic. A heuristic is admissible if the heuristic value for all possible states is less than or equal to the perfect heuristic value of the same state. This has the effect, that we never overestimate an operator.

2.3 Pattern Database

Instead of calculating the heuristic values on the fly, Culberson and Schaeffer [2] introduced pattern databases that can be used to preprocess all perfect cost values for a given subset of variables $v \in \mathcal{V}$ called a pattern. To evaluate a state only the cost of the variables included in the pattern is considered. This sub-state is also called an *abstract* state. We now calculate the perfect heuristic value, using a breath-first search algorithm and a given goal state, for all possible abstract states of a given pattern. Using a hash function, we can generate a unique hash for all abstract states of the given pattern. We then can use the hash to get the heuristic value and abstract state, or vice versa, use an abstract state to get the hash of a heuristic. A pattern database can now be reused for solving planning tasks, as long as the goal state stays the same. This is an important property of pattern databases, as for each variable added to a pattern the amount of abstract states increases exponentially and therefore also the computational effort. Being able to reuse them we can neglect the computational effort in further usages. The heuristic value of a pattern database is admissible, as it solves a subset of the problem optimally. This leads to a heuristic value that is less than or equal to the perfect heuristic. Using multiple pattern databases we can ensure admissibility by taking the maximum value of all pattern database heuristics.

2.3.1 Post-hoc Optimization Heuristic

The post-hoc optimization heuristic was introduced by Pommerening et al. [10]. A characteristic of the post-hoc optimization heuristic is the usage of multiple pattern databases. For each pattern database heuristic, we consider all operators where their cost is greater than zero, which means the operator is relevant to the heuristic. Using this information, we can construct a linear program with the variables X_o.

Definition 1 (X_o)**.** *X_o is a variable with $o \in \mathcal{O}$, where X_o represents the cost incurred by o in a fixed, but undefined optimal plan.*

Pommerening et al. [10] introduced a specific notation to specify the relevance of an operator.

Definition 2 (Relevant operator partition)**.** *Let Π be a planning task with cost function c. For all i,...,n, let $c_i c$ be a cost function and h_i be an admissible heuristic for Π_{c_i}. The relevant operator partition $\mathcal{O}/\sim$ for $h_1, ..., h_n$ is the partition of $\mathcal{O}$ induced by the equivalence relation $\sim$ with:*

$$o \sim o' \text{ iff } \{i|c_i > 0\} = \{i|c_i(o') > 0\}.$$

Using a linear program we can now evaluate the variables for a given state. The following definition is according to Definition 2 from Pommerening et al. [10].

Definition 3 (h^{PhO})**.** *Let Π be a planning task with cost function c. For all $i = 1, ..., n$ let $c_i \leq c$ be a cost function and h_i be an admissible heuristic for Π_{c_i}. Let s be a state of Π.*

The estimate of the post-hoc optimization heuristic $h^{PhO}(s)$ is the objective value of the linear program:

$$\text{Minimize} \sum_{[o]\in\mathcal{O}/\sim} X_{[o]} \text{ subject to} \tag{2.1}$$

$$\sum_{[o]\in\mathcal{O}/\sim:c_i(o)>0} X_{[o]} \geq h_i(s) \qquad \text{for all } i \in \{1, ..., n\} \tag{2.2}$$

$$X_{[o]} \geq 0 \qquad \text{for all } [o] \in \mathcal{O}/\sim . \tag{2.3}$$

Each linear program can also be rewritten into its dual form where each variable gets to be a constraint and each constraint will be a variable. Pommerening et al. [10] defined the dual program for the post-hoc optimization as follows.

Definition 4 (Dual program)**.**

$$\text{Maximize} \sum_{i=1}^{n} Y_i h_i(s) \text{ subject to} \tag{2.4}$$

$$\sum_{i\in\{1,...,n\}:c_i(o)>0} Y_i \leq 1 \qquad \text{for all } [o] \in \mathcal{O}/\sim \tag{2.5}$$

$$Y_i \geq 0 \qquad \text{for all } i \in \{1, ..., n\}. \tag{2.6}$$

The variables Y_i in the dual program each belong to a heuristic. Each constraint is given by an operator, where only the heuristic variables are considered, for which the given operator is relevant. Of course, the heuristic variable must be greater or equal to 0 as we cannot have a negative heuristic. Both, the primal and dual notation of the linear program, should lead to the same solution.

3 Sliding Tile Puzzle

The *Sliding Tile Puzzle*, is a combinatorial puzzle originally invented by Noyes Palmer Chapman in 1879 [12]. A well-known variant of the Sliding Tile Puzzle is the 15-Puzzle. Each puzzle is made up of 15 square tiles with incremental numbers from 1 to 15 on them. Those tiles are now placed in a 4 by 4 square frame, leaving one empty space. The empty space can now be used to push an adjacent tile into it. A tile cannot leave the 4 by 4 frame and only tiles adjacent to the empty space can be moved.

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

Figure 3.1: Goal configuration of the 15-Puzzle.

To solve the puzzle, we can now move the tiles with the given restrictions. Fig. 3.1 shows a possible goal configuration and the one we will be talking about during this thesis. Hereby the number zero represents the empty space.

3.1 Sliding Tile Puzzle in Classical Planning

The domains $\mathcal{D}$ of the sliding tile puzzle are given by a set of tiles $\mathcal{T}_0 = \{0, ..., (width * height - 1)\}$, where the number zero still represents the blank. Given a sliding tile puzzle we can construct a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_o, s_*, c \rangle$. $\mathcal{V}$ is a finite set of variables v, each representing a tile $t \in \mathcal{T}_0$ where $\mathcal{T}_0$ is the set of Tiles including the blank. An operator effect swaps the position of a tile and the blank. The precondition of the operator is that the to be swapped tile must be adjacent to the blank. The initial state s_0 can be any possible state of the sliding tile puzzle. The goal s_* is a full assignment of the state variables $v \in \mathcal{V}$ and is shown in Fig. 3.1. Because the sliding tile puzzle has uniform costs for all operators, the cost function can be simplified to $c : \mathcal{O} \to 1$. Therefore, it is also true that the operator sequence with the minimum number of operators is equal to the operator sequence with the minimum cost and thus also represents an optimal solution.

3.2 Heuristics of the Sliding Tile Puzzle

Having defined a planning task for the sliding tile puzzle, we have a look at different admissible heuristics adapted for it.

3.2.1 Manhattan Distance

The first and also easiest method we are looking at is the so-called *Manhattan distance*. To evaluate the Manhattan distance we sum up the displacement of all tiles to their goal position [4]. This algorithm assumes that only the currently considered tile is in place and the rest consists of blanks. Therefore the tile can move along the shortest path possible to its goal position.

Definition 5 (Manhattan distance)**.** *Let s be the current state and $t \in T$ a tile.*
$xloc_s(t)$ is the current x coordinate and $yloc_s(t)$ the current y coordinate of t in the state s. $xloc_{s_*}(t)$ the x coordinate and $yloc_{s_*}(t)$ the y coordinate of the goal position of t. Then the Manhattan distance can be calculated as followed:

$$MD(s) = \sum_{t \in T} \left| xloc_s(t) - xloc_{s_*}(t) \right| + \left| yloc_s(t) - yloc_{s_*}(t) \right| \tag{3.1}$$

3.2.2 Linear Conflicts

While the Manhattan distance only considers one tile at once, the linear conflict heuristic considers two. This will naturally lead to a better heuristic value, as we now gather more information about the current state. The definition of a linear conflict is according to Hansson et al. [5] which is already specified for the use with STPs. Two tiles $t_j, t_k \in T$ are in a linear conflict if t_j and t_k are in the same row or column, the goal position t_{j,s_*} of t_j and t_{k,s_*} of t_k are also in the same row or column, t_j is between t_k and t_{k,s_*} and t_{j,s_*} is on the same side of t_{k,s_*} as t_j.
Using the same terminology as in the definition for the Manhattan distance we can write it down as followed:

Definition 6 (Linear conflict)**.** *Let s be the current state and $t_j, t_k \in T$ where T is a set of tiles.*

Let $xloc_s(t)$ be the current x-coordinate and $yloc_s(t)$ the current y-coordinate of t in the state s, $xloc_{s_*}(t)$ the x-coordinate and $yloc_{s_*}(t)$ the y-coordinate of the goal position of t. The two tiles t_j and t_k are in linear conflict if one of the following is true.

1.

$$xloc_s(t_j) = xloc_s(t_k) = xloc_{s_*}(t_j) = xloc_{s_*}(t_k) \tag{3.2}$$

$$xloc_{s_*}(t_j) < xloc_s(t_k) < xloc_s(t_j) \text{ and } xloc_{s_*}(t_j) < xloc_{s_*}(t_k) \tag{3.3}$$

2.

$$yloc_s(t_j) = yloc_s(t_k) = yloc_{s_*}(t_j) = yloc_{s_*}(t_k) \tag{3.4}$$

$$yloc_{s_*}(t_j) < yloc_s(t_k) < yloc_s(t_j) \text{ and } yloc_{s_*}(t_j) < yloc_{s_*}(t_k) \tag{3.5}$$

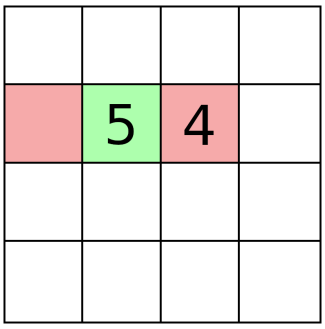

Figure 3.2: Linear Conflict between tile 4 and 5.

Example 1. *As seen in Fig. 3.2 tile 5 is between tile 4 and its goal position. The goal position of tile 5 is on the same side of tile 4s goal position as tile 4. Therefore all conditions are met and we have a linear conflict.*

Each linear conflict found has a total cost of two, plus it needs a heuristic value for each tile, which is provided by the Manhattan distance. The resulting heuristic function for the linear conflict heuristic is as followed.

Definition 7 (Linear conflict heuristic). *Let s be the current state and $t \in T$ a tile. Then the linear conflict heuristic is*

$$h^{LC}(s) = MD(s) + 2 * \sum_{\substack{t_j \in T \\ t_k \in T \setminus t_j}} LC(t_j, t_k) \tag{3.6}$$

where LC is 1 if there is a linear conflict and 0 if there is not.

3.2.3 Statically-partitioned Additive Pattern Database

Calculating multiple pattern databases, but only considering the maximum each time seems like a waste of information. Korf and Felner [8] introduced the *disjoint pattern databases,* but we will be using the term *statically-partitioned additive pattern databases* which Felner et al. [4] introduced. To ensure that a set of pattern databases is additive, each pattern of all pattern databases must be disjoint to each other, this means, no tile may be in more than one pattern. By considering each tile only ones, the sum of all pattern database heuristics of a given state is still admissible.

For the 15-Puzzle we will use the partitions used by Felner et al. [4]. The blank tile is part of all the patterns. If we would not consider the blank, all heuristics would be equal to the Manhattan distance. To keep the pattern databases additive, the cost of all operators using tiles that are not included in the pattern are equal to zero.

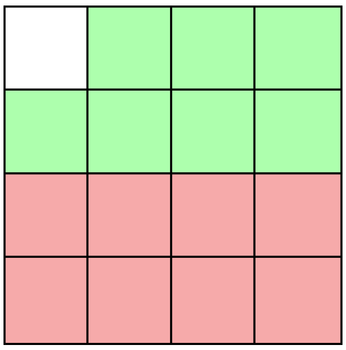
Figure 3.3: 7-8 partitioning

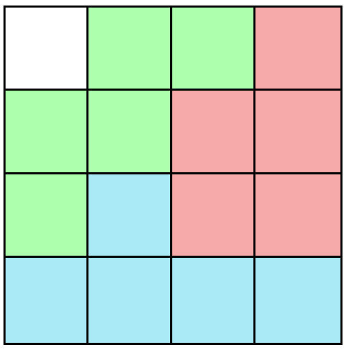
Figure 3.4: 5-5-5 partitioning

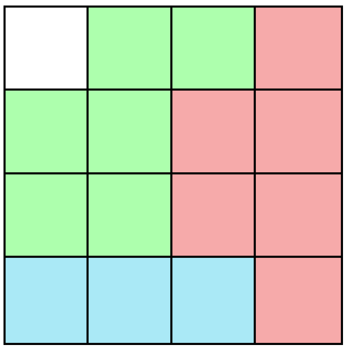
Figure 3.5: 6-6-3 partitioning

Each colour corresponds to a pattern. We can see that the partitioning in Fig. 3.3, Fig. 3.4 and Fig. 3.5 is disjoint and therefore suitable for statically-partitioned additive pattern databases.

3.2.4 Dynamically-partitioned Additive Pattern Database Heuristic

Instead of using only an additive set of pattern databases, Felner et al. [4] introduced a method which uses a non-additive set of pattern databases P^-. The goal is to find an additive subset $P^+ \subset P^-$ which maximizes the heuristic value for the current state. Finding the subset is the hard part. Felner et al. [4] used a method called *weighted vertex cover*. The set of pattern databases can be seen as a hyper-graph, where each variable is a vertex and each pattern database heuristic an edge connecting the variable-vertices included in the pattern. We now need the minimum sum of these edges, such that all vertices are only connected to one edge. This problem is proven to be NP-Complete and we will, later on, discuss a possible solution to it.

4 Adapting Post-hoc Optimization Heuristic for the Sliding Tile Puzzle

Having defined the planning task in the context of the sliding tile puzzle we have another look at the definition of the post-hoc optimization heuristic. We can simplify the notation specifically for the sliding tile puzzle through the recognition of interrelated variables. In the sliding tile puzzle, all operators which move a tile, move *only* this specific tile. This can be useful for finding all relevant operators for a heuristic. Knowing the pattern of a heuristic, we know the relevant tiles and therefore also the relevant operators. So the definition for relevant operators [Definition 2] can be reduced to: If the corresponding tile is in the pattern, the operator is relevant for the given heuristic. Therefore the partitioning $[o]$ of the operators $\mathcal{O}/\sim$ has one element per relevant tile.
We can now have another look at the primal and dual program of the post-hoc optimization heuristic.

$$\textit{Minimize} \sum_{[o]\in\mathcal{O}/\sim} X_{[o]} \text{ subject to} \tag{4.1}$$

$$\sum_{[o]\in\mathcal{O}/\sim:c_i(o)>0} X_{[o]} \geq h_i(s) \qquad \text{for all } i \in \{1,...,n\} \tag{4.2}$$

$$X_{[o]} \geq 0 \qquad \text{for all } [o] \in \mathcal{O}/\sim . \tag{4.3}$$

We now know that we have for each operator exactly one tile. For the primal program, this leads to exactly one operator variable X_o per tile. The constraints Eq. (4.2) are formed from the X_o matching the patterns of the respective heuristic.
This can also be adapted for the dual program.

$$\textit{Maximize} \sum_{i=1}^{n} Y_i h_i(s) \text{ subject to} \tag{4.4}$$

$$\sum_{i\in\{1,...,n\}:c_i(o)>0} Y_i \leq 1 \qquad \text{for all } [o] \in \mathcal{O}/\sim \tag{4.5}$$

$$Y_i \geq 0 \qquad \text{for all } i \in \{1,...,n\}. \tag{4.6}$$

To reiterate, each constraint corresponds to exactly one operator and each variable Y_i to a heuristic. For the sliding tile puzzle, this means that all patterns of the variables in a constraint contain the same constraint specific tile.

4.1 Optimizations for the Post-hoc Optimization Heuristic

Having to solve the linear program for every state we considered costs a lot of time. How can we reduce the computational effort? Our first approach is to use the *resolvability* of a linear program. After solving the linear program for the first time, the so-called *warm up* phase, we can update the variables and constraints boundaries. The changes to the linear program depend on the difference the current state has to the previously solved state. The linear program solver now will not have to solve the linear program completely for every iteration, but can check for the changes and only resolve the parts influenced by the changes.
Another optimization lays in the set of pattern databases. Instead of blindly using all the pattern databases provided, we can filter them according to their heuristics. We partition the pattern and then check if the additive pattern database heuristics of the partitions are the same as those of the complete pattern database. We do this for all abstract states. If there is one partitioning that has the same sum for each abstract state, we can ignore the pattern database, as it does not contain any information that cannot be obtained from the partitioned pattern databases.
For an additive partitioning of a pattern database to be exactly as efficient as the combined pattern database for all abstract states is pretty unlikely. But if we filter the pattern databases every time we are building the linear program, we must only check for the given state. It is way more likely for a partition to be as good as the complete pattern databases if we only consider a single state. Being able to remove a heuristic from the linear program would mean one less constraint in the primal program or one less variable in the dual program.

4.2 Using an Integer Program to Solve Weighted Vertex Cover

The dynamically-partitioned additive pattern database heuristic using weighted vertex cover is pretty similar to the post-hoc optimization heuristic, especially if we have a closer look at the dual program (Definition 4) as an integer program. But what is an integer program? Instead of solving the objective function by assigning real numbers to the variables, the integer program solves it using only integers. The integer program derived from Definition 4 corresponds one to one to the hypergraph described in Section 3.2.4. The vertices of the hypergraph as well as the constraints of the dual program correspond to the tiles of the sliding tile puzzle. The edges and the vertices connected to them are given by the pattern databases, if a vertex is in the pattern, the edge will connect to it. The weight of an edge is given by the heuristic. The same applies to the dual program, where each variable corresponds to a heuristic and is only added to the constraint if the corresponding tile is included in its pattern. Given that we are solving the dual program as an integer program, the variables can only be assigned 0 or 1 and only one variable can be 1 per constraint.

Because the heuristics are multiplied with their corresponding variable before being summed up, only the heuristics assigned 1 are considered. The others will be multiplied by 0 and therefore cancelled out. With only one heuristic assigned per constraint, and a constraint being corresponding to a tile, only one heuristic per tile can be chosen. All heuristics that have now been selected must therefore be disjoint from each other.
The only difference between the weighted vertex cover and the dual program of the post-hoc optimization heuristic is, that the linear program can assign values *between* 0 and 1, while the integer program can only assign 0 *or* 1 to a variable.

5 Experimental Evaluation

In the experiments, we test over 1000 randomly generated solvable initial states from the 15-puzzle. We will use the partitioning shown in Fig. 3.3, Fig. 3.4, Fig. 3.5 for the statically-partitioned additive pattern database heuristic. For the dynamically-partitioned additive pattern database heuristic and the post-hoc optimization heuristic, we will be using the same set of pattern databases. This set consists of all possible patterns up to a size of four variables. All experiments are carried out on the grid at the University of Basel using 2×10 Core Intel Xeon Silver 4114 2.2 GHz processors. Each instance being solved is limited to 30 minutes and can use at max 6354MB memory per CPU. The algorithms are implemented using a modified version of Nathan Sturtevants Hog2 [13] system written in C++. For solving the linear programs given by the post-hoc optimization heuristic and the weighted vertex cover will use the linear program solver provided by IBM, CPLEX version 12.9.0 [7]. We used the linear program solver interface provided by the Fast Downward planner [6] [11] to use CPLEX. Our complement is the addition of the post-hoc optimization heuristic. We will not be able to compare the values directly to Felner et al. [4] results, as we are not using the same planner, the same system nor the same states. We will compare the heuristic functions according to the initial heuristic value (the heuristic of the initial state), the number of expanded nodes , their total experiment coverage, the amount of generated nodes, the path length and the time needed to solve a problem. We will also have a look at the optimizations described in Section 4.1 and discuss how well they worked.

5.1 1000 Random 15-Puzzle Instances

We tested each algorithm on the same 1000 initial states for the 15-puzzle to get a comparison of the performance of each algorithm. The collected data is summarised in Table 5.1. To keep the tables clear, we will introduce some abbreviations. Instead of writing the full algorithm name, we will only use the defining word of the algorithm name. The statically-partitioned additive pattern database heuristic will be referred to as 'Static [partitioning], the dynamically-partitioned additive pattern database heuristic will be referred to as just 'Dynamic' and the post-hoc optimization heuristic as just 'Post-hoc [primal/dual].

Comparing the initial heuristic values and the expansions from the table it is visible that

Heuristic Function	coverage	expansions	generated	initial	path length	time
Manhattan distance	501	8906606.52	26680999.63	37201	19439	6.16
Linear conflicts	501	1201084.15	3581314.78	39285	19439	1.96
Static 5-5-5	501	65007.99	199955.80	42179	19439	0.12
Static 6-6-3	501	47161.61	146258.18	42857	19439	0.10
Static 7-8	501	4773.86	15168.42	45753	19439	0.01
Dynamic	381	4067.01	12901.91	44647	19439	504.95
Post-hoc Primal	473	4774.21	15189.47	44855	19439	85.67
Post-hoc Dual	474	4774.21	15189.47	44855	19439	90.62

Table 5.1: The coverage shows how many instances of the 1000 were solved given the previously describe restrictions of the experiment suite. Expansions denote the geometric mean of the expanded states. A lower value indicates a better-informed heuristic. Generated is the geometric mean of the generated states. Initial is the sum of the initial heuristic values, the higher the value, the better the initial values. Path length is the sum of all path lengths for all instances solved by every algorithm. The time shows the arithmetic mean on how long an algorithm took to solve a puzzle instance.

the Manhattan distance is performing the worst, second is the linear conflict heuristic with still a very high average on expanded states. The performance of the two heuristics is so poor that we will ignore them for the following tests. We still wanted to evaluate them to show how much pattern databases are compared to more basic heuristics. Next, we will have a look at the statically-partitioned additive pattern databases and how the different partitions performed. The 5-5-5 partitioning performed the worst of all three, then the 6-6-3 partitioning and last but not least the 7-8 partitioning. This can be explained by the sizes of the respective patterns. We can even see that the 7-8 partitioning performed better on average than the post-hoc optimization heuristic. This shows that the two pattern databases used by the 7-8 partitioning include more information than all the pattern databases up to the pattern size of 4 used by the post-hoc optimization heuristic. What stands out is the dynamically-partitioned additive pattern database heuristic, which given the expansions performed the best. Theoretically, it cannot be better than the post-hoc optimization heuristic. The integer program solved should only be able to evaluate a heuristic as good or worse than the linear program. One possibility for the lower expansions using the integer program over the linear program could be the tie-breaking situations. Having a look at the initial heuristic value of the dynamic algorithm we see that on average they are worse than the initial heuristic of the post-hoc algorithm. As expected, you can see that both, the primal and dual programs, performed equally well. As both heuristics perform equally well only one will be used for further evaluation. We decided to use the post-hoc dual as we are using the dual program as an integer program for the weighted vertex cover.

5.1.1 Initial Heuristic Value

We compare the initial heuristic values of the static 7-8 additive partitioning, the dynamically-partitioned additive pattern database heuristic and the post-hoc optimization heuristic using scatter plots. Each point in the graph is a different task. To reiterate, as all heuristics used are admissible, a higher initial value is better. The value in the axis description includes a value showing how well it performed compared to the other heuristic.

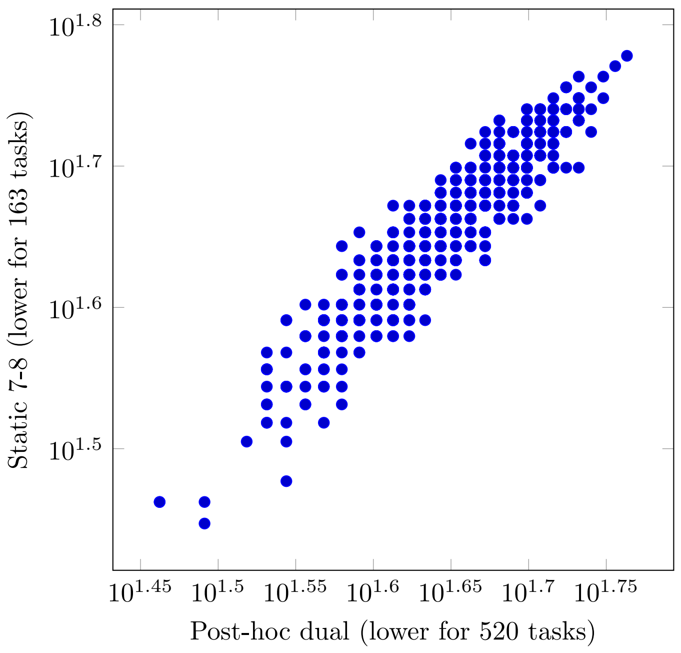

Figure 5.1: Initial heuristic comparison dual program vs static 7-8

As seen in Fig. 5.1, none of the two is strictly better than the other heuristic. But the post-hoc dual was worse than the static 7-8 partitioning in 231 cases while the static 7-8 partitioning was only worse in 78 cases, which can be seen on the axis description. This strengthens the assumption, that the 7-8 partitioning of the 15-puzzle includes more information on the planning task than the set of pattern databases used for the post-hoc optimization heuristic.

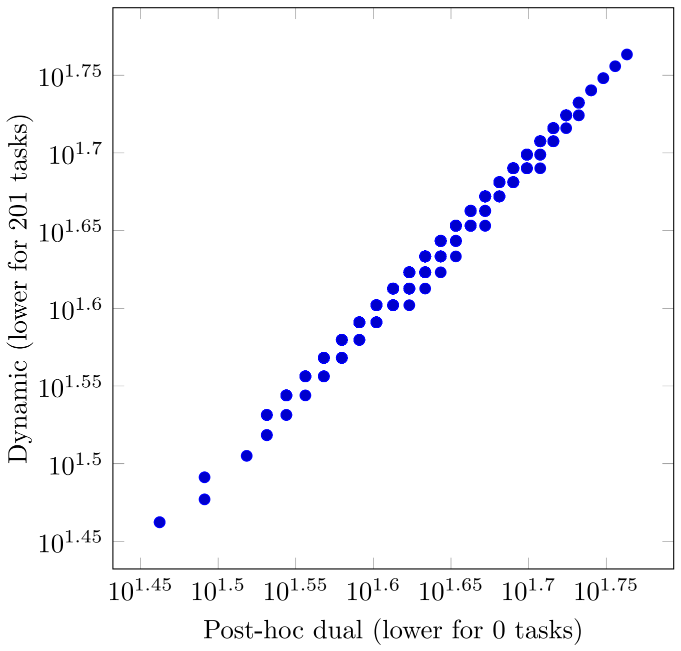

Figure 5.2: Initial heuristic comparison post-hoc dual vs dynamic

Looking at the comparison of the post-hoc optimization heuristic and the dynamically-partitioned additive pattern database heuristic seen in Fig. 5.2, as one would expect, the post-hoc optimization heuristic never performed worse than the dynamic heuristic. As previously mentioned, this is very easy to explain. If the weighted vertex cover would find the perfect heuristic using the integer program then the linear program would find it too, as both algorithms use the same objective function, variables and constraints. But the post-hoc optimization heuristic may find a heuristic value that the integer program can only approach, but not calculate exactly.

5.1.2 Expansion

We compare the expansions of the static 7-8 additive partitioning, weighted vertex cover and post-hoc optimization heuristic using scatter plots. Each point in the graph is a different task. The fewer expansions an algorithm has, the better it performed.

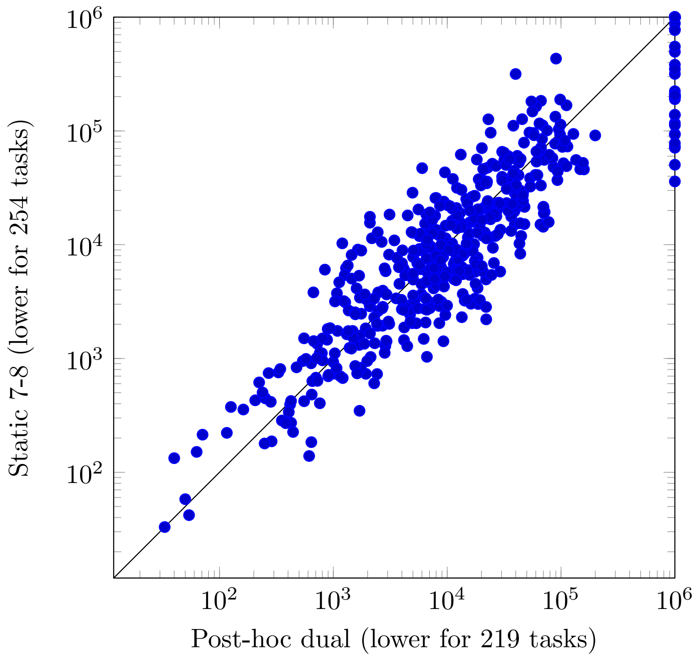

Figure 5.3: Expansion comparison dual program vs static 7-8

Not surprising is the fact, that the static 7-8 additive partitioning has on average a lower expansion for a task. We already saw that the initial heuristic of the 7-8 additive partitioning is on average better than the initial heuristic of the dual program. Interestingly, the dual program can nearly keep up with the 7-8 partitioning, as seen in Fig. 5.3, even though it uses pattern databases which by itself include way less information over the entire planning task.

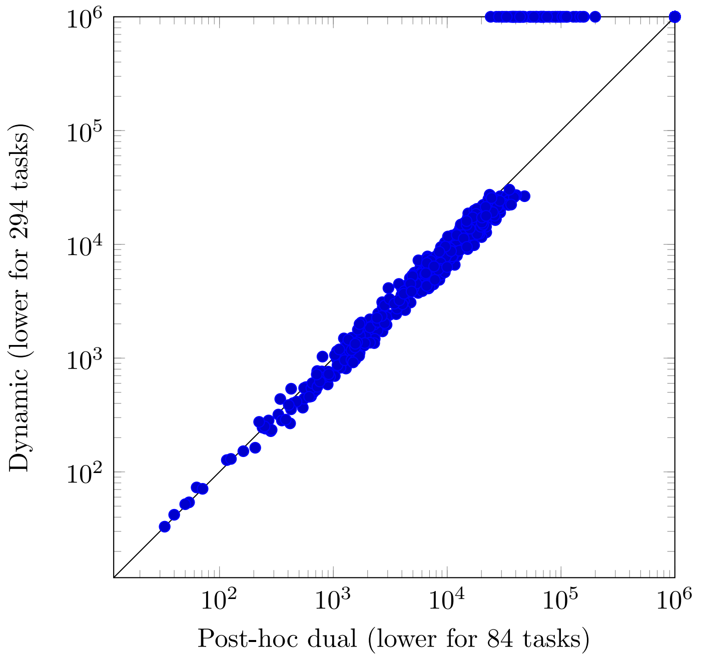

Figure 5.4: Expansion comparison post-hoc dual program vs dynamically-partitioning additive pattern database heuristic

The dynamically-partitioned additive pattern database heuristic has surprisingly on average fewer expansions than the post-hoc optimization heuristic. It performed better in ∼77% of the 381 instances solved with the dynamic approach. We have not found a clear explanation for this and could only guess.

5.2 Resolving the Linear Programs

We are testing if the resolvability of the linear and integer programs can be used to decrease the computation time needed for the post-hoc optimization and dynamically-partitioned additive pattern database heuristic. We are using the set of 1000 different states to compare the resolve approach with its corresponding non-resolving approach.

Heuristic Function	**Time**
Post-hoc Primal	48.58
Post-hoc Primal Resolved	48.80
Post-hoc Dual	48.43
Post-hoc Dual Resolved	48.51
Dynamic	262.07
Dynamic Resolved	262.07

Table 5.2: Average time for solving 1000 different sliding tile puzzle instances.

Comparing the times seen in Table 5.2 between the original linear/integer program and the ones using resolve we see, that there seems to be no improvement. We could not find a clear explanation for this behaviour.

5.3 Post-hoc: Filtering Non-Improving Heuristics

We are testing how many heuristics are deemed non-improving using 1000 different states to evaluate. To reiterate from Section 4.1, our first approach was to filter the pattern databases comparing all possible abstract state heuristics. We could not find any case where a partition was as good as the complete set. Therefore we went on with our next idea, to filter the pattern database for only the given state while generating the linear program.

Heuristic Function	**Total Constraints**
Post-hoc Primal	1940000
Post-hoc Primal Filtered	46608

The *post-hoc primal* total constraints are the unfiltered amount of constraints for the given 1000 states. *post-hoc primal filtered* shows the number of runs for the given 1000 states that have an improvement across their possible partitionings. It is visible, that for most heuristics an additive partition existed, whose heuristic was as good as the originals. On average, 97.6% of the constraints in a linear program are not necessary. This also means we do not have to load 97.6% of the pattern databases into the memory.
But there still is a problem. The time needed to solve a single sliding tile puzzle using the filtered approach has increased enormously. Evaluating all possible partitions for all pattern databases is very time-consuming by itself, but it has to be done every time we want to evaluate a heuristic. So far that we could not conduct the complete experiments in time.

6 Conclusion

In this thesis, we adapted the post-hoc optimization heuristic for the sliding tile puzzle. We described different methods on how to optimize the post-hoc optimization heuristic. We discussed a method on how to solve the weighted vertex cover problem using the post-hoc optimization heuristics linear program. We compared the post-hoc optimization heuristic with different admissible heuristics for the sliding tile puzzle and showed how the combination of pattern databases using the post-hoc optimisation heuristic improves on previously established heuristics.

6.1 Future Work

Our approach to resolving the linear program is very primitive. We just update the constraints and variables and hope for the best. It would be interesting to see if there is a possibility to decide when to resolve the linear program or to generate it from scratch.
The filtering of the pattern databases used for the post-hoc optimization during every iteration can discard a lot of pattern databases per state, but it is too expensive in time as we have to evaluate all pattern databases each time we solve a linear program. It would be exciting if we could introduce a more efficient method to filter the pattern databases. If we had more time we could have tried to filter the pattern databases again as a pre-processing step, but instead of only removing a pattern database if a partitioning exists which is equal for all abstract states, we could remove a pattern database if for all abstract states at least one partitioning exists which is equal to the complete pattern. If this would filter enough pattern databases, we could combine it with the currently used method to filter every time we solve a linear program and may be able to decrease memory and time needed to evaluate the post-hoc optimization heuristic.
The could not explain the results from Fig. 5.4. It would be exciting to have another look at both heuristics and the algorithms and see where and why exactly the dynamically-partitioned additive pattern database heuristic approach takes a better turn than the post-hoc optimization heuristic.

Bibliography

[1] Christer Bäckström and Bernhard Nebel. Complexity results for SAS$^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

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

[3] Stefan Edelkamp. Planning with pattern databases. In *Sixth European Conference on Planning*, pages 84–90, 2001.

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

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

[6] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[7] IBM. Cplex optimization studio, 2021. URL https://www.ibm.com/products/ilog-cplex-optimization-studio.

[8] Richard E Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial intelligence*, 134(1-2):9–22, 2002.

[9] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.

[10] Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the most out of pattern databases for classical planning. In *Twenty-Third International Joint Conference on Artificial Intelligence*, pages 1–8, 2013.

[11] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward lab, 2017. URL https://doi.org/10.5281/zenodo.790461.

[12] J. Slocum and D. Sonneveld. *The 15 Puzzle Book: How It Drove the World Crazy.* Slocum Puzzle Foundation, 2006. ISBN 9781890980153. URL https://books.google.ch/books?id=qWofPQAACAAJ.

[13] Nathan Sturtevant. Hog2 (hierarchical open graph 2), 2021. URL https://github.com/nathansttt/hog2.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Post-hoc Optimization for the Sliding Tile Puzzle

Name Assesor:	Malte Helmert
Name Student:	Damian Knuchel
Matriculation No.:	18-050-674

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Trimbach, 06.10.2021 Student: [illegible]

Will this work be published?

[✓ No]

Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of:

Place, Date: Trimbach, 06.10.2021 Student: [illegible]

Place, Date: Assessor:

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .