University Basel

Automatic Selection of Pattern Collections for Domain Independent Planning

Master Thesis

Sascha Scherrer
s.scherrer@unibas.ch

June 3, 2014

Examiner: Malte Helmert
Supervisors: Florian Pommerening, Martin Wehrle

Acknowledgments

I would like to thank Prof. Dr. Malte Helmert for providing the opportunity to write my master thesis in the area of artificial intelligence. I would also like to thank Prof. Dr. Malte Helmert, Dr. Martin Wehrle and Florian Pommerening for suggesting multiple possible topics for my thesis as well as helping me select this one. I am thankful for the support, suggestions and ideas provided by Dr. Martin Wehrle and Florian Pommerening during the last six months. Finally, I thank my friends, colleagues and my family for supporting me during this time.

Abstract

Heuristic search with admissible heuristics is the leading approach to cost-optimal, domain-independent planning. Pattern database heuristics—a type of abstraction heuristics—are state-of-the-art admissible heuristics. Two recent pattern database heuristics are the iPDB heuristic by Haslum et al. and the PhO heuristic by Pommerening et al..

The iPDB procedure performs a hill climbing search in the space of pattern collections and evaluates selected patterns using the canonical heuristic. We apply different techniques to the iPDB procedure, improving its hill climbing algorithm as well as the quality of the resulting heuristic. The second recent heuristic—the PhO heuristic—obtains strong heuristic values through linear programming. We present different techniques to influence and improve on the PhO heuristic.

We evaluate the modified iPDB and PhO heuristics on the IPC benchmark suite and show that these abstraction heuristics can compete with other state-of-the-art heuristics in cost-optimal, domain-independent planning.

Contents

1 Introduction

Domain-independent classical planning deals with finding a sequence of actions—called a plan—that lead from an initial state to a goal state. A common approach to classical planning is heuristic search in the state space. Heuristic search uses a search algorithm combined with a heuristic function to guide it. From the vast amount of different types of heuristics, abstraction heuristics are an increasingly popular choice. A common type of abstraction heuristics are heuristics based on pattern databases [Culberson and Schaeffer, 1998, Edelkamp, 2001]. Pattern databases (PDBs)—especially big ones—require a lot of memory to store which leads to the usual approach of selecting multiple smaller PDBs instead of a single big one. One of the main problems concerning multiple PDBs is that the quality of the resulting heuristic depends strongly on the selection of patterns. Another big problem is combining the information of multiple PDBs. Different ways of selecting patterns often require different ways of combining PDBs to get the most out of them.

Treating pattern selection as an optimization problem and solving it using an genetic algorithm [Edelkamp, 2006] was one of the first approaches to automatic pattern selection. The iPDB procedure [Haslum et al., 2007] uses a hill climbing search in the space of pattern collections to find a good pattern collection. It used the canonical heuristic [Haslum et al., 2007] to combine the PDBs generated from the pattern collection. The hill climbing search is guided by a measurement of the improvement that adding a new pattern to the collection would have.

The PhO heuristic takes a more straight-forward approach to pattern selection by selecting all patterns up to a given pattern size. The PhO heuristic achieves good heuristic values using the patterns to generate a set of constraints for a linear program (LP) and solving this LP.

We examine different aspects of the iPDB heuristic and the PhO heuristic and explore different techniques to increase their performance as well as their quality. In Section 2 we provide all necessary background information needed to understand the different approaches taken to modify the existing heuristics. In Section 3 and Section 4 we will describe the investigated changes and directly evaluate them experimentally. We will come to a conclusion and propose some future work in Section 5.

2 Background

In order to be able to discuss all ideas, changes and improvements that were studied some background knowledge is necessary.

2.1 Planning

A SAS$^+$ planning task as introduced by Bäckström and Nebel [1995] with action costs is a tuple $\Pi = (V, O, s_0, s_*, c)$ where V is a finite set of variables, each variable $v \in V$ having its own finite domain D_v. A partial state is a partial function over V assigning a value $s(v) \in D_v$ to some variables $\mathit{Vars}(s) \subseteq V$. A state is a partial state assigning a value to every variable. The state s_0 is called the *initial state*; the *goal* s_* is a partial state. Two partial states s and s' are consistent if there is no variable $v \in V$ for which both states are defined *and* have different values. A partial state s' that is defined on some variables $\mathit{Vars}(s') = P \subseteq V$ and is consistent with a state s can be written as a projection of s onto P: $s' = s_{|P}$.

The finite set O consists of all *operators*, each having a partial state pre_o as the *precondition* and a partial state eff_o as the *effect*. The cost function $c : O \to \mathbb{N}_0$ assigns each operator $o \in O$ its *cost*. An operator $o \in O$ is *applicable* in a state s if its precondition is consistent with s. Applying an operator $o \in O$ to a state s if o is applicable in s results in a state $s[o]$ with

$$s[o](v) = \begin{cases} \mathit{eff}_o(v) & \text{if } v \in \mathit{Vars}(\mathit{eff}_o) \\ s(v) & \text{otherwise} \end{cases}$$

The result of applying an operator which is not applicable in a state is undefined. The successors of a state are all states that can be reached by applying an operator to that state. Applying a sequence of operators $\pi = \langle o_1, o_2, ..., o_n \rangle$ is defined as:

$$s[\pi] = \begin{cases} s & \text{if } \pi = \langle\rangle \\ s[o_1][\langle o_2, ..., o_n \rangle] & \text{otherwise} \end{cases}$$

The result of applying a sequence of operators of which one operators is not applicable in the respective state is undefined. The finite set of states S consists of all states of Π. All states $S_* \subseteq S$ that are consistent with the (partial) goal state s_* are called *goal states*.

An s-plan for a state $s \in S$ is a sequence of operators π so that $s[\pi]$ is defined and

consistent with s_*. The cost of an operator sequence $\pi = \langle o_1, o_2, ..., o_n \rangle$ is defined as

$$c(\pi) = \sum_{i=1}^{n} c(o_i)$$

An optimal s-plan of $s \in S$ is an s-plan with minimal cost. A *plan* is an s_0-plan.

The *causal graph* of a planning task (V, O, s_0, s_*, c) is defined as a directed graph consisting of all variables in V as nodes. Its nodes are connected with *precondition arcs* from node u to node $v \neq u$ if there is an operator $o \in O$ with $u \in pre_o$ and $v \in eff_o$. Nodes are also connected with *co-effect arcs* from u to v and vice versa if there exists an operator $o \in O$ with $u, v \in eff_o$. Two variables u, v are *causally connected* if their corresponding nodes in the causal graph are connected either by a precondition arc or by a co-effect arc.

Finding an optimal plan for a planning task can be done by heuristic search in the state space.

2.2 Heuristic Search

A heuristic is a function $h : S \to \mathbb{N}_0 \cup \{\infty\}$ and is an estimator for the optimal plan cost in a state. The heuristic function which returns the optimal plan cost for every state is called h^*. A heuristic h is *admissible* if $h(s) \leq h^*(s)$ holds for every state. A heuristic is *consistent* if $h(s) \leq h(s[o]) + c(o)$ holds for every states s and every operator o applicable in s. The optimal heuristic h^* always returns the optimal plan cost and is in general not efficiently computable because SAS$^+$ is a generalization of STRIPS which is PSPACE-complete as shown by Bylander [1994].

Heuristic search is done by searching in the state space guided by a heuristic. Usually a best-first search algorithm is used which always expands the most promising states first based on some metric. A common example is the search algorithm A* [Hart et al., 1968] which expands the state with lowest $f(s) = g(s) + h(s)$ first. The so called f-value depends on the distance from the initial state $g(s)$ and a heuristic value $h(s)$. If the heuristic h is admissible then A* guarantees that an optimal plan will be found if one exists.

2.3 Pattern Databases

Pattern databases (PDBs) were first introduced by Culberson and Schaeffer [1998] as a domain specific heuristic for solving the 15- and 24-puzzle. PDBs have been

adapted to cost-optimal planning by Edelkamp [2001]. PDBs can also be applied to domain-independent planning:

A *pattern* is a subset of variables $P \subseteq V$ of a planning task $\Pi = (V, O, s_0, s_*, c)$. Projecting the planning task Π using the pattern P results in an abstracted planning task $\Pi^P = (P, O^P, s_0^P, s_*^P, c^P)$. The states of Π^P are only defined on variables in P and are therefore partial states in Π. The operators O^P are the operators O of Π with their preconditions and effects restricted to the variables in P. The cost of the optimal s-plan for every state s in Π^P gets computed and stored in a *pattern database*.

The heuristic value of a state s of Π is then computed by simply looking up the optimal plan cost for $s_{|P}$ in the pattern database, we write this as $h^P(s)$. A pattern database heuristic is always admissible due to the fact that every plan in Π is a plan in Π^P and therefore the cost of an optimal plan in Π^P must be lower than or equal to the cost of an optimal plan in Π. Pattern databases are independent of the initial state of a planning task and can therefore be reused if two tasks only differ in their initial state. As one can imagine, computing such a pattern database for big patterns can take quite some time and storage space. In fact, creating a pattern database for the full pattern—containing each variable in the planning task—requires computing the optimal plan cost for every state. A pattern database with the empty pattern has a single entry which has to be a goal in Π^P and is therefore equivalent to $h_0 : S \to \{0\}$. The step from using a single PDB to using multiple PDBs introduces two interesting problems.

2.4 Using Multiple Pattern Databases

In domain independent planning the time used to create PDBs is usually shared with the time used for searching for a plan. This time is often limited in some way (from seconds to minutes to hours or days) and has to be considered when creating PDBs. The patterns to use have to be selected automatically without human interaction and deeper knowledge of the problem at hand. In addition the time available for solving a problem in domain independent planning may make it difficult to create large pattern databases. Two algorithms that deal with the problems of PDBs in domain independent planning and automatic selection of patterns are the PhO (Posthoc Optimization) heuristic by Pommerening et al. [2013] and the iPDB heuristic by Haslum et al. [2007].

As with all admissible heuristics we can combine pattern databases by simply taking the maximum value of multiple pattern databases heuristics. The heuristic values of two PDBs can even be added if their patterns are additive. Two patterns P_1 and P_2

are *additive* if the set of operators that affect a variable in P_1 is disjoint from the set of operators that affect a variable in P_2. A set of patterns is additive if all patterns in the set are pairwise additive. An additive set of patterns A can be combined into a heuristic function:

$$h^A(s) = \sum_{P \in A} h^P(s)$$

There is always a unique, best way of combining the patterns of a pattern collection C into a collection S_C of all maximal (with respect to set inclusion) additive subsets of C. The heuristic function

$$h^C(s) = \max_{A \in S_C} \sum_{P \in A} h^P(s)$$

is called the *canonical heuristic* function for the pattern collection C.

2.5 iPDB

The iPDB procedure as introduced by Haslum et al. [2007] uses a hill climbing algorithm in the space of pattern collections to accumulate a set of patterns which are then combined and evaluated with the canonical heuristic.

2.5.1 Selecting Patterns

The initial pattern collection of the iPDB procedure consists of one pattern per goal variable containing that goal variable. A candidate pattern is a pattern that is the same as a pattern in the pattern collection except for one additional variable. This additional variable has to be causally connected to at least one variable already in the pattern. In every iteration one candidate pattern gets added to the pattern collection and the set of candidate patterns gets extended accordingly. The neighborhood of a pattern collection consists of all pattern collections that contain one candidate pattern in addition to all patterns in the current pattern collection.

The decision of which pattern to add is made depending on the so called counting approximation. Every pattern collection in the neighborhood gets evaluated on a set of samples using the canonical heuristic. The number of samples on which the canonical heuristic improves using a neighbor pattern collection is called the *improvement* of the respective candidate pattern. The candidate with the highest improvement is added to the set of patterns and the neighborhood gets extended accordingly. There are several limits necessary to ensure a successful execution of the algorithm as bigger patterns

are preferred over smaller patterns due to their higher heuristic values. To prevent the algorithm from selecting bigger and bigger patterns and using up all available memory a per-pattern limit as well as a total limit on the PDB size is used. To ensure termination within reasonable time the minimum value of improvement for the counting approximation is also limited. A high-level description of the iPDB pattern selection can be seen in Algorithm 1.

Algorithm 1 Base iPDB

1: $collection = init()$
2: $candidates = gen\_candidates(collection)$
3: $stop = false$
4: **repeat**
5: $samples = sample\_states()$
6: $\{best_c, best_{impr}\} = evaluate(samples, collection, candidates)$
7: $collection = collection \cup \{best_c\}$
8: $candidates = candidates \setminus \{best_c\}$
9: $candidates = candidates \cup gen\_candidates(best_c)$
10: **if** $best_{impr} < min_{impr}$ or $candidates = \{\}$ **then**
11: $stop = true$
12: **end if**
13: **until** $stop$

2.5.2 Sample States

Optimally, samples would be uniformly taken from the search space which is practically impossible without doing the search first. An approximation is achieved by using random walks with a random length chosen by a binomial distribution with a mean of the estimated solution length. The estimated solution length is computed by multiplying the heuristic value of the initial state by 2 to compensate for underestimation and dividing by the average operator cost.

2.5.3 Combining PDBs

The iPDB algorithm uses the canonical heuristic as described in Section 2.4 to evaluate its set of pattern databases.

2.6 PhO

The posthoc optimization heuristic by Pommerening et al. [2013] uses linear programs (LPs) and PDBs to compute the sum of incurred costs of all operators.

2.6.1 Selecting Patterns

The PhO heuristic uses a so called *systematic pattern selection* to selects all interesting patterns up to a given pattern size. A pattern is *interesting* if the subgraph of the causal graph induced by the pattern is weakly connected and contains a directed path via precondition arcs from each node to some goal variable node. Every pattern that is *not interesting* can be replace by one or multiple (additive) smaller patterns resulting in the same heuristic values [Pommerening et al., 2013].

2.6.2 Combining PDBs

For a pattern P we define the set of relevant operators $rel_P(O) \subseteq O$ as all operators that have effects which are defined on at least one variable in P. A pattern database with pattern P therefore only tracks costs that are incurred by operators in $rel_P(O)$ since all other operators do not change the abstract state. We call the summed up costs of an operator $o \in O$ in a fixed but unknown optimal plan X_o. The estimate of a PDB with pattern P can therefore never exceed the incurred costs of operators in $rel_P(O)$:

$$h^P(s) \leq \sum_{o \in rel_P(O)} X_o$$

This results is one inequality per PDB which can be used to build an LP for a state s and pattern collection C:

$$\begin{array}{lll} \textit{minimize:} & \sum_{o \in O} X_o & \\ \textit{subject to:} & \sum_{o \in rel_P(O)} X_o \geq h^P(s) & \textit{forall } P \in C \\ & X_o \geq 0 & \textit{forall } o \in O \end{array}$$

The objective value of the LP can be used as an estimate for the cost of the fixed but still unknown optimal plan. The PhO heuristic is admissible as proven by Pommerening et al. [2013].

3 Extensions of the iPDB Heuristic

In this section will discuss the changes investigated and applied to the iPDB heuristic. The evaluation will be directly after the description of each proposed technique due the number of changes investigated.

The International Planning Competition (1998–2011) tasks for optimal planning were used to benchmark performance. All experiments were done with a time limit of 30 minutes and a memory limit of 2 GB on machines with two 8-core Intel Xeon E5-2660 CPUs (one core per task). A^* was used as the search algorithm in all experiments. All implementations and changes were integrated into the Fast Downward planning system [Helmert, 2006]. Tables are shortened in way that rows which have the same value in all columns are aggregated together into a single row labeled *Others*.

We built our modifications into the implementation of the iPDB procedure by Pommerening et al. [2013] which is based on the implementation by Sievers et al. [2012]. The base implementation using a minimum improvement limit of 10 will be called h^{iPDB}.

3.1 Limiting Pattern Selection Time

The notion of improvement of a candidate pattern is a relative quality measure and therefore quite good for selecting the best pattern to add. When used as a stopping criterion the improvement does not offer much information about the absolute quality of the heuristic that can be obtained by the set of patterns and can therefore lead to premature stopping or prolonged continuation of the hill climbing search. In practice, the improvement may stay above the minimum improvement even after enough good patterns have been selected and remain above it until the whole available time is used up. An example of this behavior on a specific problem (airport-p18) can be seen in Figure 1. The graph shows that the optimal point to stop searching for patterns (in this specific instance) would be after six iterations but the improvement remains above 300 for many more iterations. To prevent this from happening some other stopping criteria have to be in place.

The simplest way of limiting long runtime is by introducing a time limit. The implementation this work is based on already contained a imprecise time limit which we replace with a more accurate one. Our time limit is checked while the candidates are evaluated and stops the evaluation process immediately once the time limit is exceeded. We add the best candidate evaluated in this iteration to the pattern collection and stop the pattern selection.

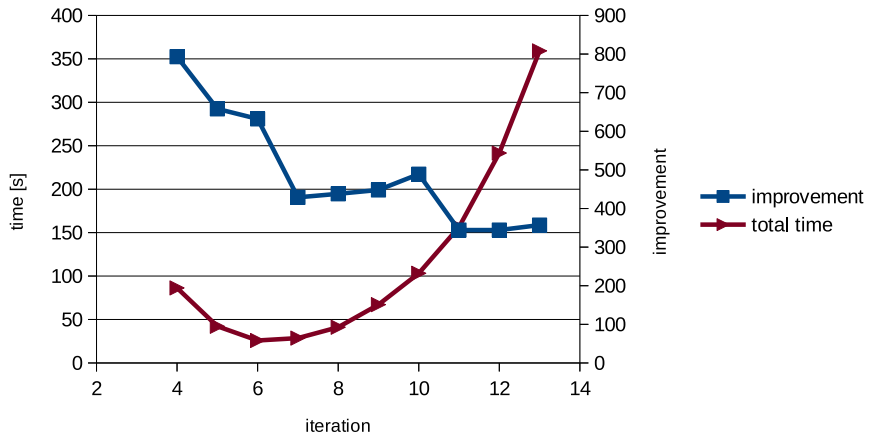

Figure 1: Best improvement per iteration and total runtime if stopped after n$^{\text{th}}$ iteration on *airport-p18*

The iPDB implementation with a time limit of s seconds will be referred to as $h^{iPDB}_{t=s}$ from now on. The results in Table 1 show that the existence of a time limit is more important than the actual value the time limit is set to. Three values from one third to two thirds of the total time have been tested with very similar results. All further versions will use a time limit of 900 seconds except when stated differently.

3.2 Limiting Minimum Improvement

The problem with a fixed limit on the minimum improvement is that having it set too low results in long runtime with little gain while having it set too high stops the algorithm too soon and increases the chance to get a weaker heuristic. We introduce a dynamic limit to counteract those problems and help find the balance between building a pattern collection and searching for an optimal plan. Our dynamic limit is based on the idea that the limit continually rises until it finally is higher than the current best improvement. The dynamic limit in iteration n with improvement of $improvement_i$ in iteration $i < n$ with a constant a is:

$$limit_n = a * \sum_{i=1..n-1} improvement_i$$

The results of the evaluation can be seen in Table 2 and show that our dynamic

	h^{iPDB}	$h^{iPDB}_{t=600}$	$h^{iPDB}_{t=900}$	$h^{iPDB}_{t=1200}$
airport (50)	28	**30**	**30**	**30**
parcprinter-08-strips (30)	15	19	**20**	**20**
parcprinter-opt11-strips (20)	11	15	**16**	**16**
pipesworld-tankage (50)	16	**17**	16	16
woodworking-opt08-strips (30)	10	**14**	**14**	**14**
woodworking-opt11-strips (20)	5	**9**	**9**	**9**
Others (1196)	579	579	579	579
Sum (1396)	664	683	**684**	**684**

Table 1: Time Limit Results

limit while solving more problems than the base version is inferior to a time limit. Our dynamic limit is more susceptible to stopping too soon due to low improvement than the base version caused by the monotonically increasing limit. While improving coverage slightly on a few domains (e.g. airport) compared to the time limit the small losses in many other domains result in an overall worse coverage. The iPDB implementation with a dynamic limit with value a will be referred to as $h^{iPDB}_{dyn=a}$ from now on. To be certain that this dynamic limit is worse than a time limit more experiments with lower values for a should be conducted. A decline in coverage for values of a close to zero is expected as the dynamic limit goes towards zero.

3.3 Increasing the Neighborhood

The downside of using a hill climbing algorithm to select good patterns is the fact that hill climbing in general only finds local optima. Figure 2 shows the improvement values of patterns added in each iteration on a specific problem (pipesworld-notankage-p09). As can be seen, the improvement drops down to a very low value before going up in the next iteration. Depending on the problem and the limit, the algorithm might stop at the first low value and miss all useful patterns that would come later.

We introduce a counter measure by extending the neighborhood to find better patterns that otherwise would not be found. This has been implemented by adding not only patterns that have one variable more than the currently selected ones but patterns that have up to n more variables. A n-variable extension of P is a pattern $E = P \cup \{v_1, v_2, ..., v_n\}$ with v_1 causally connected to $u \in P$ and v_i causally connected to v_{i+1}.

	$h^{iPDB}_{t=900}$	$h^{iPDB}_{dyn=0.0125}$	$h^{iPDB}_{dyn=0.025}$	$h^{iPDB}_{dyn=0.05}$	$h^{iPDB}_{dyn=0.1}$	$h^{iPDB}_{dyn=0.2}$
airport (50)	30	30	30	31	**34**	25
depot (22)	**8**	7	**8**	**8**	**8**	**8**
driverlog (20)	**13**	**13**	**13**	12	12	12
elevators-opt08-strips (30)	**20**	19	19	19	18	17
elevators-opt11-strips (20)	**16**	15	15	15	15	14
freecell (80)	**20**	**20**	**20**	19	19	19
logistics00 (28)	**21**	**21**	20	20	20	20
miconic (150)	**55**	53	53	53	52	50
mystery (30)	**16**	15	15	15	15	15
nomystery-opt11-strips (20)	**16**	15	15	15	14	14
parcprinter-08-strips (30)	**20**	19	19	19	17	16
parcprinter-opt11-strips (20)	**16**	15	15	15	13	12
pegsol-08-strips (30)	**30**	**30**	29	29	27	27
pegsol-opt11-strips (20)	**20**	**20**	19	19	17	17
pipesworld-notankage (50)	**17**	16	16	16	16	16
pipesworld-tankage (50)	16	16	16	16	**17**	14
sokoban-opt08-strips (30)	**29**	**29**	**29**	**29**	28	28
trucks-strips (30)	**8**	**8**	**8**	**8**	**8**	6
visitall-opt11-strips (20)	16	**17**	**17**	**17**	16	16
woodworking-opt08-strips (30)	**14**	**14**	**14**	**14**	**14**	12
woodworking-opt11-strips (20)	**9**	**9**	**9**	**9**	**9**	7
zenotravel (20)	**11**	**11**	**11**	**11**	10	10
Others (596)	263	263	263	263	263	263
Sum (1396)	**684**	675	673	672	662	638

Table 2: Dynamic Limit Results

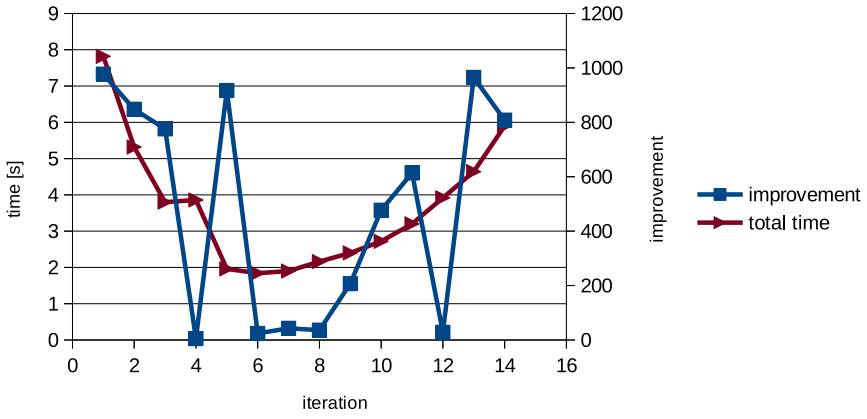

Figure 2: Best improvement per iteration and total runtime if stopped after n$^{\text{th}}$ iteration on *pipesworld-notankage-p09*

Due to how the evaluation step and computation of the improvement works, all PDBs have to be kept in memory the whole time. Depending on the problem and how many variables are added to generate new patterns there might be a lot of memory occupied, sometimes even enough to reach the memory limit of 2 GB. To prevent a crash in these cases we added an estimation for the memory usage of all PDBs together and use it to limit the generation of new PDBs during iterations. When this limit (set to 1.6 GB of 2 GB) is reached, no new PDBs are allowed to be generated but the selection process continues. The already created PDBs get evaluated and the best gets added to the selections until another limit stops the algorithm. The version of the algorithm which extends patterns by n new variables will be called h^{iPDB}_{nvar}. A high-level description of the iPDB pattern selection with increased neighborhood can be seen in Algorithm 2.

The result of adding multiple variables at once combined with the memory limit can be seen in Table 3. Note that the configuration with $n = 1$ differs from h^{iPDB} only in the memory limit. Pattern extensions with multiple variables seem to increase coverage by quite a bit. Extensions by more variables generate many more candidate patterns which rapidly fill the available memory and also increase the time needed to evaluate all candidates. These are probably the main reasons why h^{iPDB}_{2var} has the highest coverage. As Table 4 shows, when more variables are used to extend patterns,

	h^{iPDB}_{1var}	h^{iPDB}_{2var}	h^{iPDB}_{3var}	h^{iPDB}_{4var}
airport (50)	**30**	**30**	26	16
depot (22)	8	**11**	10	9
driverlog (20)	13	**14**	13	**14**
elevators-opt08-strips (30)	20	**22**	**22**	21
elevators-opt11-strips (20)	16	**18**	**18**	17
floortile-opt11-strips (20)	2	2	**6**	5
grid (5)	**3**	**3**	**3**	2
mystery (30)	**16**	**16**	**16**	15
nomystery-opt11-strips (20)	16	**18**	**18**	**18**
parcprinter-08-strips (30)	20	22	**23**	**23**
parcprinter-opt11-strips (20)	16	**17**	**17**	**17**
pegsol-08-strips (30)	**30**	29	27	27
pegsol-opt11-strips (20)	**20**	19	17	17
Pipesworld-notankage (50)	17	**19**	17	17
pipesworld-tankage (50)	**16**	**16**	**16**	10
psr-small (50)	49	**50**	**50**	**50**
tidybot-opt11-strips (20)	**14**	**14**	**14**	4
tpp (30)	6	6	**8**	**8**
transport-opt08-strips (30)	11	**14**	13	13
transport-opt11-strips (20)	6	**10**	9	9
trucks-strips (30)	8	8	**9**	**9**
visitall-opt11-strips (20)	16	**17**	**17**	**17**
woodworking-opt08-strips (30)	**14**	**14**	13	11
woodworking-opt11-strips (20)	**9**	**9**	8	6
Others (729)	308	308	308	308
Sum (1396)	684	**706**	698	663

Table 3: Increased Neighborhood Results

Algorithm 2 iPDB with increased neighborhood

```
1: collection = init()
2: candidates = gen_candidates(collection)
3: stop = false
4: repeat
5:    samples = sample_states()
6:    {best_c, best_impr} = evaluate(samples, collection, candidates)
7:    collection = collection ∪ {best_c}
8:    candidates = candidates \ {best_c}
9:    if not memory_limit_reached() then
10:      candidates = candidates ∪ gen_candidates(best_c, n)
11:   end if
12:   if best_impr < min_impr or candidates = {} or time_limit_reached() then
13:      stop = true
14:   end if
15: until stop
```

h^{iPDB}_{1var}	h^{iPDB}_{2var}	h^{iPDB}_{3var}	h^{iPDB}_{4var}
1360	1361	1345	1249

Table 4: Number of instances where iPDB pattern selection finished

fewer instances actually complete the pattern selection procedure. This is mainly due to completely filling the available memory and crashing despite there being a memory limit. So far, no solution to fix this problem has been found.

3.4 Ignoring Candidate Pattern

We tested a reduction of the search space by removing all patterns from the candidate set that seem useless. This was done by removing all patterns with an improvement lower than the minimum improvement limit. We remove patterns that might be useful later on if the removed pattern can be used additively with a not yet added pattern. This version of the procedure will be called h^{iPDB}_{ign} and is outlined in Algorithm 3.

The results of this approach can be seen in Table 5 and show almost no change in coverage. When looking at the number of instances which finished pattern selection as shown in Table 6 we see that there seem to be more memory issues than without ignoring patterns. This is possibly due memory fragmentation and/or memory leaks. Never the less, the almost equal coverage show that there is at least some kind of

	h^{iPDB}_{2var}	$h^{iPDB}_{ign,1var}$	$h^{iPDB}_{ign,2var}$	$h^{iPDB}_{ign,3var}$
airport (50)	**30**	**30**	24	14
depot (22)	**11**	7	10	8
driverlog (20)	**14**	13	**14**	13
elevators-opt08-strips (30)	**22**	19	**22**	**22**
elevators-opt11-strips (20)	**18**	15	**18**	**18**
floortile-opt11-strips (20)	2	2	2	**6**
freecell (80)	**20**	19	**20**	**20**
logistics00 (28)	**21**	20	20	20
miconic (150)	**55**	**55**	54	54
nomystery-opt11-strips (20)	**18**	16	**18**	**18**
parcprinter-08-strips (30)	22	23	**29**	**29**
parcprinter-opt11-strips (20)	17	18	**19**	**19**
pegsol-08-strips (30)	29	**30**	29	29
pegsol-opt11-strips (20)	19	**20**	19	19
pipesworld-notankage (50)	**19**	17	16	10
pipesworld-tankage (50)	**16**	**16**	14	8
psr-small (50)	**50**	49	**50**	**50**
sokoban-opt08-strips (30)	**29**	**29**	**29**	27
tpp (30)	6	6	6	**8**
transport-opt08-strips (30)	**14**	11	**14**	13
transport-opt11-strips (20)	**10**	6	**10**	9
trucks-strips (30)	8	8	8	**9**
visitall-opt11-strips (20)	**17**	16	**17**	**17**
woodworking-opt08-strips (30)	14	**16**	**16**	13
woodworking-opt11-strips (20)	9	**11**	**11**	8
Others (496)	216	216	216	216
Sum (1396)	**706**	688	705	677

Table 5: Ignoring Candidates Results

Algorithm 3 iPDB with candidate ignoring

1: $collection = init()$
2: $candidates = gen\_candidates(collection)$
3: $stop = false$
4: **repeat**
5: $samples = sample\_states()$
6: $\{best_c, best_{impr}\} = evaluate(samples, collection, candidates)$
7: $collection = collection \cup \{best_c\}$
8: $candidates = candidates \setminus \{best_c\}$
9: **for all** *candidate* in *candidates* with $improvement(candidate) < min_{impr}$ **do**
10: $candidates = candidates \setminus \{candidate\}$
11: **end for**
12: **if** not $memory\_limit\_reached()$ **then**
13: $candidates = candidates \cup gen\_candidates(best_c, n)$
14: **end if**
15: **if** $best_{impr} < min_{impr}$ or $candidates = \{\}$ or $time\_limit\_reached()$ **then**
16: $stop = true$
17: **end if**
18: **until** *stop*

$h^{iPDB}_{ign,1var}$	$h^{iPDB}_{ign,2var}$	$h^{iPDB}_{ign,3var}$
1363	1308	1224

Table 6: Number of instances where iPDB pattern selection finished

potential in discarding "useless" candidates assuming one can find some better way of doing it.

3.5 Sampling using FF Heuristic

The sampling algorithm was examined and a slight change was tested. To see how a different sampling method behaves combined with iPDB a more target oriented approach was taken. Instead of sampling uniformly by doing random walks, the FF heuristic [Hoffmann and Nebel, 2001] is used to guide the first walk and add all evaluated states on the path to the sample set. This first walk is stopped once it reaches a goal or a dead-end and sampling is continued as described in Section 2.5.2 using random walks.

The results of the changed sampling can be seen in Table 7 and Table 8. This method of sampling states decreased the quality of the heuristic quite a bit as can be seen by

	h^{iPDB}_{FF}	h^{iPDB}_{2var}
airport (50)	**37**	30
depot (22)	10	**11**
driverlog (20)	13	**14**
elevators-opt08-strips (30)	21	**22**
elevators-opt11-strips (20)	17	**18**
logistics00 (28)	20	**21**
mystery (30)	15	**16**
nomystery-opt11-strips (20)	**19**	18
pipesworld-notankage (50)	18	**19**
pipesworld-tankage (50)	**17**	16
transport-opt11-strips (20)	9	**10**
trucks-strips (30)	**10**	8
visitall-opt11-strips (20)	16	**17**
woodworking-opt08-strips (30)	12	**14**
woodworking-opt11-strips (20)	7	**9**
zenotravel (20)	10	**11**
Others (936)	452	452
Sum (1396)	703	**706**

Table 7: Sampling Results

the higher amount of expanded states. Although this sampling method is worse than the default one, it caused strange artifacts by not reaching the memory limit unlike the compared version of iPDB. For example, all 7 tasks in the airport domain that have been solved with the modified sampling have not been solved by the compared version due to reaching the memory limit during pattern selection and never starting the search.

These results should be taken with a grain of salt as multiple effects interact with each other (mainly crashes due to reaching the memory limit and non-uniform sampling). The results in Table 8 are also aggregated using geometric mean and do not necessarily reflect the single values ideally.

3.6 Iteration Duration Estimation

The possible positive effect of skipping iterations that are expected to violate the time limit has been considered. Using the knowledge gained from Section 3.1 we assume that saving some small amount of time has no relevant impact on the performance of

	h^{iPDB}_{FF}	h^{iPDB}_{2var}
airport (30)	**0.10**	**0.10**
barman-opt11-strips (4)	5539215.39	**2879968.23**
blocks (28)	857.59	**665.75**
depot (10)	45867.83	**13404.80**
driverlog (13)	3290.75	**273.02**
elevators-opt08-strips (21)	16483.35	**12193.38**
elevators-opt11-strips (17)	26332.80	**21713.80**
floortile-opt11-strips (2)	116800.62	**111605.44**
freecell (20)	**317.29**	355.73
grid (3)	**90.56**	**90.56**
gripper (7)	58403.75	**58199.55**
logistics00 (20)	35.88	**17.55**
logistics98 (5)	**1838.69**	10375.12
miconic (55)	7037.87	**3365.79**
mprime (23)	**87.32**	116.91
mystery (16)	85.22	**51.15**
nomystery-opt11-strips (18)	**2310.32**	2317.53
openstacks-opt08-strips (19)	**11223.60**	**11223.60**
openstacks-opt11-strips (14)	**38595.00**	**38595.00**
openstacks-strips (7)	**6.73**	7.31
parcprinter-08-strips (22)	**0.10**	**0.10**
parcprinter-opt11-strips (17)	**0.10**	**0.10**
parking-opt11-strips (5)	**149905.12**	**149905.12**
pathways-noneg (4)	**2353.62**	**2353.62**
pegsol-08-strips (29)	1881.28	**1872.51**
pegsol-opt11-strips (19)	**24082.45**	24338.06
pipesworld-notankage (18)	3808.83	**2093.30**
pipesworld-tankage (16)	1014.31	**887.78**
psr-small (50)	1.25	**0.52**
rovers (7)	2637.28	**380.81**
satellite (6)	8421.72	**7709.15**
scanalyzer-08-strips (13)	185.51	**108.67**
scanalyzer-opt11-strips (10)	974.23	**885.02**
sokoban-opt08-strips (29)	47434.23	**31875.09**
sokoban-opt11-strips (20)	38697.34	**26029.07**
tidybot-opt11-strips (14)	17552.72	**16212.23**
tpp (6)	460.35	**79.65**
transport-opt08-strips (14)	2292.03	**662.92**
transport-opt11-strips (9)	134667.07	**43232.39**
trucks-strips (8)	**1017.33**	12214.37
visitall-opt11-strips (16)	1.06	**0.18**
woodworking-opt08-strips (12)	359.74	**6.91**
woodworking-opt11-strips (7)	854.39	**30.40**
zenotravel (10)	196.88	**48.70**
Geometric mean (693)	1117.15	**675.03**

Table 8: Sampling Results: Expansions without last f-layer
smaller values are better

$\Delta t > 0$	$\lvert\Delta t\rvert \leq 1$	$mean(\Delta t)$	$\min(\Delta t)$	$\max(\Delta t)$
26.6%	69.35%	-1.39s	-345.43s	615.85s

Table 9: Time Estimation Evaluation, $\Delta t = estimate - real$

the algorithm. We implemented iteration duration estimation as linear extrapolation from previous iterations. Estimates were compared to the real time in each iteration and aggregated into Table 9. To evaluate the impact of such a duration estimation a more exact estimation would be needed, possibly by using more input data like the number of candidate patterns.

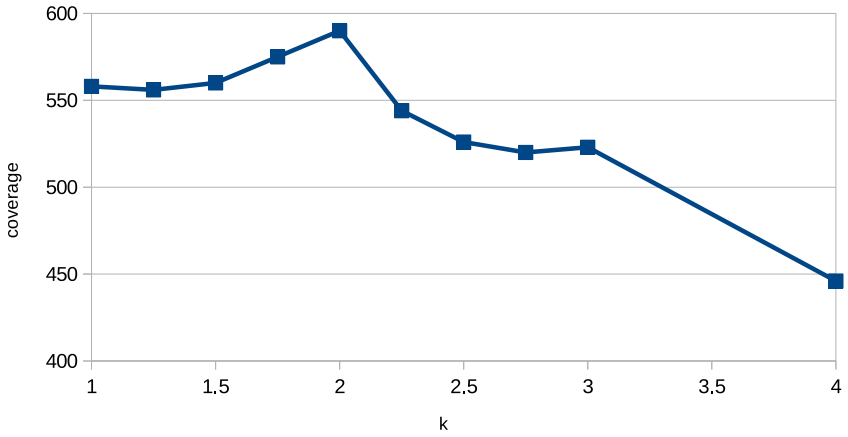

Figure 3: Coverage for different k values

4 Extensions of the PhO Heuristic

The same setup as described in Section 3 was used.

The PhO procedure that has been used and modified is the version implemented by Pommerening et al. [2013]. The base implementation will be called h^{PhO}.

4.1 Partial Systematic Sizes

The systematic pattern selection selects all interesting patterns up to a given pattern size k. The first thing investigated was how the systematic pattern selection can be changed to allow for non integer values for k. This was mostly done the get a deeper understanding of how the amount and selection of patterns interact with the evaluation using LPs. For a non integer value k, all patterns with size up to and including $\lfloor k \rfloor$ are generated as before. In addition, of all patterns with size $\lfloor k \rfloor + 1$ a percentage of $k - \lfloor k \rfloor$ are selected (by generation order). The PhO heuristic that selects patterns up to a pattern size K will be called h^{PhO}_K.

The evaluation as seen in Table 10 shows that such partial layers tend do worse (indicated by the fact that $k = \{1, 2, 3\}$ are local optima as seen in Figure 3). This is probably caused by groups of inequalities in the LP which only yield good heuristic values if the whole group is present.

	h_1^{PhO}	$h_{1.25}^{PhO}$	$h_{1.5}^{PhO}$	$h_{1.75}^{PhO}$	h_2^{PhO}	$h_{2.25}^{PhO}$	$h_{2.5}^{PhO}$	$h_{2.75}^{PhO}$	h_3^{PhO}	h_4^{PhO}
airport (50)	22	**24**	23	23	23	14	13	13	12	7
barman-opt11-strips (20)	**4**	**4**	**4**	**4**	**4**	**4**	0	0	0	0
blocks (35)	**28**	27	26	26	26	23	20	18	18	17
depot (22)	**7**	**7**	**7**	**7**	**7**	4	2	2	2	2
driverlog (20)	11	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
elevators-opt08-strips (30)	11	12	15	16	17	17	17	17	**18**	**18**
elevators-opt11-strips (20)	9	10	13	13	14	14	14	14	14	**15**
floortile-opt11-strips (20)	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	0
freecell (80)	**14**	**14**	**14**	**14**	**14**	8	8	8	7	6
grid (5)	1	1	1	1	**2**	1	**2**	**2**	**2**	1
gripper (20)	**7**	**7**	**7**	**7**	**7**	6	6	6	6	5
logistics00 (28)	16	16	17	**20**	**20**	**20**	**20**	**20**	**20**	**20**
logistics98 (35)	4	4	4	**5**	**5**	**5**	**5**	**5**	**5**	**5**
mprime (35)	**20**	**20**	19	19	19	17	17	17	17	13
mystery (30)	**14**	13	13	13	13	13	13	12	11	9
nomystery-opt11-strips (20)	10	13	15	**16**	**16**	15	15	15	15	15
openstacks-opt08-strips (30)	**19**	17	17	17	17	16	15	15	14	10
openstacks-opt11-strips (20)	**14**	12	12	12	12	11	10	10	9	5
openstacks-strips (30)	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	5
parcprinter-08-strips (30)	15	15	15	15	16	16	16	16	23	**27**
parcprinter-opt11-strips (20)	11	11	11	11	12	12	12	12	18	**20**
parking-opt11-strips (20)	**3**	1	1	1	1	0	0	0	0	0
pegsol-08-strips (30)	**27**	**27**	**27**	**27**	**27**	26	26	26	26	18
pegsol-opt11-strips (20)	**17**	**17**	**17**	**17**	**17**	16	16	16	16	5
pipesworld-notankage (50)	14	14	14	**15**	14	11	9	8	9	3
pipesworld-tankage (50)	**10**	7	7	7	6	5	4	4	4	3
psr-small (50)	**49**	**49**	**49**	**49**	**49**	48	48	48	47	46
rovers (40)	6	6	6	6	6	**7**	**7**	**7**	**7**	**7**
satellite (36)	**6**	**6**	**6**	**6**	**6**	**6**	**6**	5	5	4
sokoban-opt08-strips (30)	23	22	21	24	**27**	19	17	17	14	5
sokoban-opt11-strips (20)	19	19	18	**20**	**20**	16	14	14	11	3
tidybot-opt11-strips (20)	7	7	7	9	10	**11**	**11**	**11**	**11**	3
tpp (30)	6	6	6	6	6	6	6	6	6	**8**
transport-opt08-strips (30)	**11**	**11**	10	10	10	**11**	**11**	**11**	**11**	**11**
transport-opt11-strips (20)	**6**	**6**	5	5	5	**6**	**6**	**6**	**6**	**6**
trucks-strips (30)	5	5	5	**6**	**6**	**6**	**6**	**6**	**6**	**6**
visitall-opt11-strips (20)	**16**	**16**	**16**	**16**	**16**	**16**	**16**	15	15	12
woodworking-opt08-strips (30)	9	10	11	11	**15**	14	14	14	14	12
woodworking-opt11-strips (20)	4	5	6	6	**10**	9	9	9	9	8
Others (250)	74	74	74	74	74	74	74	74	74	74
Sum (1396)	558	556	560	575	**590**	544	526	520	523	446

Table 10: Partial Layers Coverage

granularity	global	per-domain	per-problem
coverage	590	633	641

Table 11: k-Selection Granularity

4.2 Dynamically Chosen Systematic Size

Although the overall coverage is the highest for $k = 2$, the optimal value for k can be chosen on different granularities:

1. Global, the best overall value ($k = 2$) is chosen
2. Per-domain, the best value for each domain is chosen
3. Per-problem, the best value for every single problem is chosen

These three ways to choose the optimal value for k have been evaluated. It is important to note here that these three ways each need a different amount of prior knowledge: no prior knowledge for the first, domain knowledge for the second and problem knowledge for the third (so options 2. and 3. are more of theoretical interest). The results in Table 11 show that choosing a value for k at a finer granularity increases the coverage. The higher increase from choosing globally to choosing per-domain than from choosing per-domain to choosing per-problem is expected because problems get exponentially more difficult. Although expected, the difference of 43 additional problems solved to only 8 additional problems solved seems relevant. This indicates that the best value of k depends more on the domain of the problem than on the problem itself.

4.3 Limiting Number & Total Size of PDBs

The PhO heuristic returns good heuristic values but at the cost of slow evaluations. State evaluations become slower with more constraints in the LP which directly depends on the number of patterns in the pattern collection. To somewhat limit the time needed for each evaluation the number of patterns as well as the total size of all PDBs together can be limited. Different values for both limits were tested on a subset of domains. The domains were chosen by selecting domains that have different values for the optimal systematic size k as seen in Table 12.

Different combination of limits for number of PDBs (100, 400, 700 and 1000) and for the total size (sum of number of entries) of PDBs (10'000, 100'000, 200'000 and

Domain	Optimal k
airport	1-2
parcprinter-08-strips	3-4
pipesworld-tankage	1
sokoban-opt08-strips	2
tpp	4+
transport-opt08-strips	1-4
woodworking-opt08-strips	2-3

Table 12: Selected Domains

	total size limit				
#PDB limit	10'000	100'000	200'000	500'000	no limit
100	94	95	94	95	95
400	102	102	102	102	102
700	102	98	98	96	96
1'000	101	94	93	92	90
no limit	80	80	80	80	80

Table 13: Coverage of Different Limit Combinations

	h^{PhO}_{2}	$h^{PhO}_{2,e=0.01}$	$h^{PhO}_{2,e=0.005}$	$h^{PhO}_{2,e=0.001}$
airport (50)	23	22	23	**24**
grid (5)	**2**	**2**	**2**	1
mprime (35)	**19**	**19**	**19**	18
parking-opt11-strips (20)	1	1	1	**3**
pipesworld-tankage (50)	**7**	6	**7**	**7**
tidybot-opt11-strips (20)	**10**	**10**	**10**	8
woodworking-opt11-strips (20)	**10**	**10**	**10**	9
Others (1196)	519	519	519	519
Sum (1396)	**591**	589	**591**	589

Table 14: Limiting Evaluation Time Coverage

500'000) have been tested as shown in Table 13. Limiting the total size of all PDBs influences the coverage only slightly while limiting the number of PDBs influences the coverage quite a bit. This is mainly because the number of PDBs is the same as the amount of constraints in the LP while a limit on the total size only indirectly (over the number of PDBs) influences the number of constraints. A limit on the number of PDBs of around 400 PDBs performed the best on the tested domains.

4.4 Limiting Evaluation Time

A similiar approach to limiting the number and the total size of PDBs is limitting the actual time it takes to evaluate states. This is done by computing the heuristic on a set of sample states (sampled the same way as described in Section 2.5.2). The time needed for evaluating the sample states gets measured and if it exceeds the limit no more patterns are allowed to be added to the pattern collection. Only a small limit on the evaluation time influences the coverage of the PhO heuristic as shown in Table 14. Further values for the limit should be tested to gain more knowledge on the effects of limiting evaluation time. We expect that a limit on the number of PDBs and a limit on the evaluation time have similar effects and that possibly only one should be used.

4.5 Pruning Unused Constraints

Limiting the number of PDBs on generation has the wanted effect of reducing the pattern collection but does this independently of the patterns. A different approach is to generate more patterns and then later remove part of them depending on some

criterion. When solving an LP with more constraints than variables some constraints are usually not used, meaning they could be removed for this specific LP without changing the solution. By evaluating the PhO heuristic on a set of sample states and looking at the LP it is easy to extract which constraints were used in each solution. With this information the constraints which were used the least often can be removed. We present a pruning technique that removes all constraints that have not been used at all on a set of sample states. The number of samples indirectly influences how many constraints get removed due to fewer samples giving fewer chances for the constraints to be used. The PhO heuristic with systematic size K and pruning using number of samples S will be called $h^{PhO}_{K,sS}$.

We performed experiments for systematic sizes $k = \{2, 3\}$ and different numbers of sample states $samples = \{1, 5, 10, 50, 100, 200\}$ as seen in Table 15. The results show that the coverage is the highest with a very small number of samples. Such a small number of samples leads to the removal of many constraints but might do so in an arbitrary way. The increase in coverage looks promising, especially the increase in coverage for systematic size 3.

	h^{PhO}_{2}	$h^{PhO}_{2,s1}$	$h^{PhO}_{2,s10}$	$h^{PhO}_{2,s50}$	h^{PhO}_{3}	$h^{PhO}_{3,s1}$	$h^{PhO}_{3,s5}$	$h^{PhO}_{3,s10}$	$h^{PhO}_{3,s50}$	$h^{PhO}_{3,s100}$
airport (50)	23	**31**	**31**	29	12	15	14	14	12	12
barman-opt11-strips (20)	**4**	**4**	**4**	**4**	0	**4**	2	0	0	0
blocks (35)	26	**28**	**28**	27	18	26	26	25	22	22
depot (22)	**7**	**7**	**7**	**7**	2	4	3	3	2	2
elevators-opt08-strips (30)	17	12	16	16	**18**	13	16	17	**18**	**18**
elevators-opt11-strips (20)	**14**	10	13	13	**14**	10	13	**14**	**14**	**14**
freecell (80)	**14**	**14**	**14**	**14**	7	12	11	10	7	7
grid (5)	**2**	1	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
gripper (20)	**7**	**7**	**7**	**7**	6	6	6	6	6	6
miconic (150)	50	50	50	50	50	**51**	**51**	**51**	**51**	**51**
mprime (35)	**19**	**19**	**19**	**19**	17	17	17	17	17	17
mystery (30)	**13**	**13**	**13**	**13**	12	12	12	12	11	12
nomystery-opt11-strips (20)	**16**	12	15	**16**	15	11	14	14	15	15
openstacks-opt08-strips (30)	**17**	**17**	**17**	**17**	14	14	14	14	14	14
openstacks-opt11-strips (20)	**12**	**12**	**12**	**12**	9	9	9	9	9	9
parcprinter-08-strips (30)	16	15	16	16	**23**	19	**23**	**23**	**23**	**23**
parcprinter-opt11-strips (20)	12	11	12	12	**18**	15	**18**	**18**	**18**	**18**
parking-opt11-strips (20)	**1**	**1**	**1**	**1**	0	0	0	0	0	0
pegsol-08-strips (30)	**27**	**27**	**27**	**27**	26	26	26	26	26	26
pegsol-opt11-strips (20)	**17**	**17**	**17**	**17**	16	16	16	16	16	16
pipesworld-notankage (50)	**14**	**14**	**14**	**14**	9	12	10	10	9	9
pipesworld-tankage (50)	6	**8**	**8**	7	4	5	5	4	4	4
psr-small (50)	**49**	**49**	**49**	**49**	47	48	48	47	47	47
rovers (40)	6	6	6	6	**7**	**7**	**7**	**7**	**7**	**7**
satellite (36)	**6**	**6**	**6**	**6**	5	5	5	5	5	5
scanalyzer-08-strips (30)	7	**8**	7	7	7	7	7	7	7	7
scanalyzer-opt11-strips (20)	4	**5**	4	4	4	4	4	4	4	4
sokoban-opt08-strips (30)	27	**28**	**28**	**28**	14	24	24	24	24	24
sokoban-opt11-strips (20)	**20**	**20**	**20**	**20**	11	19	19	19	18	18
tidybot-opt11-strips (20)	10	10	10	10	11	**12**	**12**	**12**	**12**	**12**
transport-opt08-strips (30)	10	**11**	10	10	**11**	**11**	**11**	**11**	**11**	**11**
transport-opt11-strips (20)	5	6	5	5	6	6	6	**7**	**7**	**7**
visitall-opt11-strips (20)	**16**	**16**	**16**	**16**	15	**16**	**16**	**16**	**16**	**16**
woodworking-opt08-strips (30)	**15**	11	14	**15**	14	13	14	14	14	14
woodworking-opt11-strips (20)	**10**	6	9	**10**	9	8	9	9	9	9
Others (243)	71	71	71	71	71	71	71	71	71	71
Sum (1396)	590	583	**598**	597	524	550	561	558	548	549

Table 15: Pruning Constraints Coverage

5 Conclusion and Future Work

Comparing the two investigated abstraction heuristics with another state-of-the-art heuristic—the LM-Cut heuristic [Helmert and Domshlak, 2009]—in Table 16 we can see that both abstractions heuristics are viable and can compete with the LM-Cut heuristic. The results are slightly distorted due to predominant miconic domain with its 150 task (compared to the average of 31 tasks per domain). An aggregation over all domains except miconic has been added for easier comparison.

5.1 Conclusion

Many techniques have been tested on the iPDB heuristic and the PhO heuristic giving new knowledge about their inner working. Especially the extension of candidate patterns by multiple variables in iPDB and the pruning of constraints in PhO have been found to increase the performance of the heuristics. We hope that the insights gained as a result of this thesis can be used to further improve on the iPDB and PhO heuristics.

5.2 Future Work

Some possible future improvements to the two heuristics have already been hinted at in the evaluation. Extending candidate patterns in iPDB with multiple variables helps coping with local optima but further techniques that are known to work with hill climbing algorithms may prove to be applicable. One such techniques is stochastic hill climbing which would choose patterns with a probability proportional to their improvement. Improving the used-memory estimation might also improve the heuristic as a crash means instant failure while the problem might have been solved if it were not for the crash. A possible way to reduce memory usage—that could be investigated—is to use dominance pruning once the memory limit is reached and then remove all candidate patterns that are no longer in the (now smaller) neighborhood of the selected pattern set.

A method to extract information about the domain of a problem could increase the quality of the PhO heuristic by quite a bit and could be further investigated. We believe that constraint pruning can be refined by changing the pruning criterion for better results.

	h^{iPDB}_{2var}	$h^{LM\text{-}Cut}$	$h^{PhO}_{2,s10}$
airport (50)	30	28	**31**
depot (22)	**11**	7	7
driverlog (20)	**14**	13	12
elevators-opt08-strips (30)	**22**	**22**	16
elevators-opt11-strips (20)	**18**	**18**	13
floortile-opt11-strips (20)	2	**7**	2
freecell (80)	**20**	15	14
grid (5)	**3**	2	2
logistics00 (28)	**21**	20	20
logistics98 (35)	5	**6**	5
miconic (150)	55	**141**	50
mprime (35)	**23**	22	19
mystery (30)	16	**17**	13
nomystery-opt11-strips (20)	**18**	14	15
openstacks-opt08-strips (30)	**19**	**19**	17
openstacks-opt11-strips (20)	**14**	**14**	12
parcprinter-08-strips (30)	**22**	18	16
parcprinter-opt11-strips (20)	**17**	13	12
parking-opt11-strips (20)	**5**	3	1
pathways-noneg (30)	4	**5**	4
pegsol-08-strips (30)	**29**	27	27
pegsol-opt11-strips (20)	**19**	17	17
pipesworld-notankage (50)	**19**	17	14
pipesworld-tankage (50)	**16**	12	8
psr-small (50)	**50**	49	49
rovers (40)	**7**	**7**	6
satellite (36)	6	**7**	6
scanalyzer-08-strips (30)	13	**15**	7
scanalyzer-opt11-strips (20)	10	**12**	4
sokoban-opt08-strips (30)	29	**30**	28
tidybot-opt11-strips (20)	**14**	**14**	10
transport-opt08-strips (30)	**14**	11	10
transport-opt11-strips (20)	**10**	6	5
trucks-strips (30)	8	**10**	6
visitall-opt11-strips (20)	**17**	11	16
woodworking-opt08-strips (30)	14	**17**	14
woodworking-opt11-strips (20)	9	**12**	9
zenotravel (20)	11	**13**	9
Sum (1396)	706	**763**	598
Sum without miconic (1246)	**651**	622	548

Table 16: Heuristic Comparison

References

[Bylander, 1994] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

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

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

[Edelkamp, 2001] Stefan Edelkamp. Planning with pattern databases. In *Pre-proceedings of the Sixth European Conference on Planning*, pages 13–24, 2001.

[Edelkamp, 2006] Stefan Edelkamp. Automated creation of pattern database search heuristics. In Stefan Edelkamp and Alessio Lomuscio, editors, *Proceedings of the Fourth Workshop on Model Checking and Artificial Intelligence*, pages 36–51, 2006.

[Hart et al., 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, pages 100–107, 1968.

[Haslum et al., 2007] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In Robert C. Holte and Adele How, editors, *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1007–1012, 2007.

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

[Helmert and Domshlak, 2009] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 162–169, 2009.

[Hoffmann and Nebel, 2001] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Pommerening et al., 2013] Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the most out of pattern databases for classical planning. In Francesca Rossi, editor, *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 2357–2364, 2013.

[Sievers et al., 2012] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. Efficient implementation of pattern database heuristics for classical planning. In Daniel Borrajo, Ariel Felner, Richard Korf, Maxim Likhachev, Carlos Linares Lopez, Wheeler Ruml, and Nathan Sturtevant, editors, *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, pages 105–111, 2012.