Perfect Domain-Dependent Pattern Database Heuristics

Abstract

Pattern database (PDB) heuristics are an important component of many of today’s strongest algorithms for optimal classical planning. Especially when combining a collection of PDB heuristics via cost partitioning, the resulting estimates become very accurate. Corrêa and Pommerening (2019) showed empirically that small patterns are enough to obtain perfect heuristics for many common planning domains. However, their analysis is post-hoc and does not show how to actually obtain the patterns. We address this gap in a case study of two domains, Gripper and Logistics, and introduce domain-dependent algorithms that obtain perfect pattern collections, that is, sets of patterns for which an optimal cost partitioning results in the true solution cost. Compared to domain-independent approaches that systematically generate patterns of increasing size until reaching the perfect heuristic, our algorithms yield fewer patterns and smaller induced projections.

Introduction

Over five decades after its inception, A∗ search (Hart, Nilsson, and Raphael 1968) remains the go-to approach for solving optimal classical planning tasks. A∗ is a graph search algorithm that uses a heuristic function (Pearl 1984) to estimate goal distances. If the heuristic is admissible, i.e., it never overestimates the true goal distance of a state, the solution found by A∗ is guaranteed to be optimal.

One of the strongest sources of heuristics for A∗ are abstractions (e.g., Seipp and Helmert 2018; Sievers and Helmert 2021). An abstraction groups several concrete states into a single abstract state to obtain a smaller state space for which all goal distances can be precomputed. Since abstraction preserves paths, all abstract goal distances are lower bounds on the true distances and can therefore serve as admissible estimates in an A∗ search.

One of the simplest forms of abstraction is projection, where the input task is projected onto a subset of its variables, called a pattern. We call the resulting heuristic a pattern database (PDB) heuristic. These rather simple heuristics can be turned into strong estimators by combining a collection of them into a single heuristic. One way to do this while preserving admissibility is by adding estimates from projections that affect disjoint sets of variables. (Korf and Felner 2002; Edelkamp 2006; Haslum et al. 2007).

The preferable option though, both theoretically and experimentally, is to compute a cost partitioning over the projections (Katz and Domshlak 2010; Seipp, Keller, and Helmert 2017). A cost partitioning divides the original action costs among the projections so that the sum of distributed costs does not exceed the original costs for each action (Pommerening et al. 2015).

A cost partitioning over a set of projections can be interpreted as a potential heuristic (Pommerening et al. 2015; Seipp, Pommerening, and Helmert 2015; Seipp et al. 2016; Pommerening, Helmert, and Bonet 2017) and Corrêa and Pommerening (2019) showed empirically that small patterns suffice to encode perfect potential heuristics for many planning tasks from the International Planning Competition (IPC). Their results imply that there are also small patterns that are sufficient for an optimal cost partitioning to yield a perfect estimate. However, their analysis only applies to the individual tasks that are part of the IPC suite and small enough to make their linear program formulation feasible. Furthermore, the results are obtained in a post-hoc manner, after solving the task: their approach is unable to create new heuristics. Therefore, two main questions remain unsolved and we address them in this work: 1) are small patterns enough for whole domains of tasks?, and 2) can we find algorithms that compute small pattern collections that yield perfect estimates? In this paper, we show that the answer to both questions is affirmative for the two well-known IPC domains Gripper and Logistics.

Background

The notation and definitions in this section are based on the work by Seipp, Keller, and Helmert (2020).

Classical Planning

We consider planning tasks in the SAS+ representation (Bäckström and Nebel 1995; Helmert 2009), defined as a tuple $\Pi = (\mathcal{V}, \mathcal{O}, s_0, s^*)$ where $\mathcal{V}$ is a finite set of variables $v \in \mathcal{V}$ with a finite domain $\text{dom}(v)$. A state is a function that assigns each variable $v$ to a value in its domain $\text{dom}(v)$. States can be full or partial assignments. We sometimes treat states as sets of variable-value pairs. We use $S(\Pi)$ to denote the set of states in $\Pi$.

Each operator $o$ in the finite set of operators $\mathcal{O}$ has a precondition $\text{pre}(o)$, an effect $\text{eff}(o)$, and a cost $\text{cost}(o)$. Both
pre(o) and eff(o) are partial states and cost : O → ℝ is a function mapping operators to real-valued costs. The input tasks we consider assign unit cost to all operators. An operator o ∈ O is applicable in state s if pre(o) ⊆ s. Applying o in s yields the state s′ = eff(o)[s] if v is mentioned in eff(o) and s′ = s otherwise. The full state s₀ is the initial state and the partial state s* is the goal description.

Planning tasks are closely tied to the notion of transition systems. We define a transition system T as a directed labeled graph (S, L, T, s₀, s*), with nodes S, labels L, labeled edges T, initial state s₀ and goal states s*. A SAS⁺ planning task Π = (V, O, s₀, s*) induces the transition system (S(Π), O, {(s, o, s′) | s ∈ S(Π), o ∈ O, pre(o) ⊆ s, s₀, {s ∈ S(Π) | s* ⊆ s}}).  

A heuristic for a transition system T is a function h:S→ℝ∪{−∞,∞}. The perfect heuristic h* returns the true goal distance for solvable states and ∞ otherwise. A heuristic h is admissible if h(s) ≤ h*(s) for all states s ∈ T. The causal graph G of a planning task Π = (V, O, s₀, s*) is a directed graph with vertices V and two types of edges: precondition-effect edges from variable vᵢ to variable vⱼ iff there is an operator o ∈ O where vᵢ occurs in the precondition and vⱼ occurs in the effect, and undirected effect-effect edges between variable pairs vᵢ and vⱼ for which there is an operator that affects both variables.

Abstractions and Cost Partitioning
Abstractions are a general way to obtain heuristics from a transition system. An abstraction T′ is a simplification of a transition system T produced by an abstraction function α : S’→S”. If the abstract state space defined by α is small enough, h* can be precomputed with an exhaustive search and used as a heuristic for the original task. Since abstraction preserves paths, abstraction heuristics are admissible. Projections are a special type of abstractions, obtained by ignoring all but a subset of variables in the task (Culberson and Schaeffer 1998; Edelkamp 2001). We call such a subset a pattern and multiple patterns a pattern collection.

A cost partitioning (Katz and Domshlak 2010) for a transition system T is a tuple of operator cost functions (cost₁, ..., costₙ) that partition the original cost function cost so that ∑ₙ₌₁ costₙ(o) ≤ cost(o) for all operators o ∈ O. Originally, each cost function was limited to assigning non-negative costs, but Pommerening et al. (2015) generalized the concept to arbitrary real values. Computing n heuristics under a cost partitioning with n cost functions guarantees that the sum of heuristic estimates remains admissible.

We call a pattern collection C for task Π perfect if there is a cost partitioning C that yields the perfect solution cost for Π. A pattern collection is perfect for a planning domain if it is perfect for all tasks in the domain.

Systematic Pattern Generation
As a baseline for our domain-dependent algorithms that compute perfect pattern collections, we use two algorithms that systematically generate patterns up to a given size k for task (V, O, s₀, s*) with n variables. Both will eventually yield a perfect pattern collection once k becomes large enough, often earlier than when k = n. The first, naïve, generates all ∑ₖ₌₁ |V| patterns (Felner, Korf, and Hanan 2004). The second, interesting, generates the set of patterns up to size k that are interesting for general cost partitioning (Pommerening, Röger, and Helmert 2013; Pommerening et al. 2021). A pattern P is interesting if it cannot be partitioned into a pair of smaller patterns without reducing the maximum heuristic value obtainable from P via cost partitioning.

To simplify notation, we define the Cartesian product of disjoint sets to produce a set of sets instead of tuples: A × B = \{\{a, b\} | a ∈ A, b ∈ B\}.

Now we can start our investigation into devising algorithms that compute perfect heuristic estimates based on pattern database heuristics. For this analysis, we use two common IPC domains, Gripper and Logistics.

Gripper
In the Gripper domain (McDermott 2000), a robot with two grippers (left, right) has to move an even number of balls from one room (A) to another (B). The robot is initially in room A.

Represented in the SAS⁺ formalism, a Gripper task Π = (V, O, s₀, s*) has variables V = \{r\} ∪ G ⊆ B that encode the position of the robot (r), the content of the two grippers (G = \{gᵢ, gᵢ’\}) and the position of each of the n balls (B = \{b₁, ..., bₙ\}). The variable domains are:

• dom(r) = \{A, B\}
• dom(gᵢ) = dom(gᵢ’) = B ∪ \{free\}
• dom(bᵢ) = \{A, B, gᵢ\} for all balls bᵢ ∈ B

The set O consists of operators for moving between the two rooms (move AB and move BA), and picking up or dropping ball bᵢ in room x with gripper y (pixy, dixy). All operators have unit cost.

The initial state s₀ = \{r ↔ A, gᵢ ↔ free, gᵢ’ ↔ free\} ∪ \{bᵢ ↔ A | bᵢ ∈ B\} and goal state s* = \{bᵢ ↔ B | bᵢ ∈ B\} are the same for all instances.

We define the following pattern collection for Gripper, consisting of n + 2 patterns, and show that it is perfect:

C_G = \{\{gᵢ, r\}, \{gᵢ’, r\}\} ∪ \{\{bᵢ\} | bᵢ ∈ B\}

Theorem 1. Pattern collection C_G is perfect for Gripper.

Proof. A Gripper task with n balls can be solved optimally in 3n − 1 steps (Helmer and Mattmüller 2008) by picking and dropping each ball (2n steps), moving n/2 pairs of balls to B (n/2 steps) and moving back to A, except for the last pair (n/2 − 1 steps). To show that there is a cost partitioning over C_G which yields a heuristic value of 3n − 1 for the initial state of all Gripper tasks, we provide a suitable cost partitioning in Table 1 and a visual proof in Figures 1, 2, and 3, depicting the graph structure, transition costs, and heuristic values for the three pattern types in C_G. This cost partitioning yields the perfect estimate of 3n − 1 for the initial state since each of the n patterns \{bᵢ\} contributes 3 cost units and \{gᵢ, r\} contributes −1 cost unit.

□
Having defined a perfect pattern collection for Gripper, we now compare it to the systematic pattern generators in terms of the number of patterns and the total number of abstract states in the induced projections that are needed to obtain a perfect estimate.

The interesting pattern collection for Gripper is
\[
\left\{\{r\}, \{g_{r}\}\right\} \cup \left\{\{b\} \mid b \in B\right\} \cup \\
\left\{\{r, g_{r}\}, \{r, g_{r}\}\right\} \cup (G \times B).
\]

The comparison between \(C_{G}\) and interesting in Table 2 shows that our pattern collection has the same asymptotic pattern size, but induces projections whose size is linear in the number of balls instead of quadratic.

Let \(h^{C_{G}}\) be the heuristic that computes an optimal cost partitioning over \(C_{G}\) for a given state. Then \(h^{C_{G}}\) yields the optimal heuristic value for every Gripper state where there is an even number of balls (including 0) in both rooms, which is the case for all states in optimal solutions for Gripper. Therefore, \(h^{C_{G}}\) reflects the known result that finding optimal plans for Gripper is possible in polynomial time (Helmert 2008).

**Logistics**

Logistics (McDermott 2000) is a transportation domain in which \(m\) packages \((P = \{p_1, \ldots, p_m\})\) have to be transported from a start location to a goal location using \(n\) trucks \((T = \{t_1, \ldots, t_n\})\) and one or two airplanes \((A = \{a_1, (a_2)\})\). Each task has \(n\) cities \((c_{i}t_{j}g_{k} = \{f_i, s_i\})\) and each city consists of an airport \(f_i\) and a suburb \(s_i\). There is exactly

---

Table 1: Optimal cost partitioning for \(C_{G}\).

<table>
<thead>
<tr>
<th>Operator</th>
<th>{(g_r, r}}</th>
<th>{(g_r, r}}</th>
<th>{(b_1}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>move (A B)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>move (B A)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pick (b_i) (A right)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pick (b_i) (B right)</td>
<td>0</td>
<td>2</td>
<td>-3</td>
</tr>
<tr>
<td>pick (b_i) (A left)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pick (b_i) (B left)</td>
<td>2</td>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>drop (b_i) (A right)</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>drop (b_i) (B right)</td>
<td>0</td>
<td>-2</td>
<td>3</td>
</tr>
<tr>
<td>drop (b_i) (A left)</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>drop (b_i) (B left)</td>
<td>-2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2: Number of patterns and projection sizes for systematically generated pattern collections (up to pattern size 2) and the \(C_G\) pattern collection.

<table>
<thead>
<tr>
<th>#patterns</th>
<th>#states</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>(\frac{1}{2}n^2 + \frac{7}{2}n + 6)</td>
</tr>
<tr>
<td>interesting</td>
<td>(4n + 5)</td>
</tr>
<tr>
<td>(C_G)</td>
<td>(n + 2)</td>
</tr>
</tbody>
</table>
The initial and goal location of a package when considering
The initial and goal city of a package:

\( p \in \mathcal{P} \)

\( \text{is-relevant}(p) \) iff \( s_0[p] \neq s^*[p] \)

The initial and goal city of a package:

\( \text{init-city}(p_j) = \text{city}_y \) if \( s_0[p_j] = \text{city}_y \)

\( \text{goal-city}(p_j) = \text{city}_y \) if \( s^*[p_j] = \text{city}_y \)

The initial and goal location of a package when considering only the truck subtask \( t \):

\( \text{init-for-city}(p_j, i) = \left\{ \begin{array}{ll} s_0[p_j] & \text{if } \text{init-city}(p_j) = \text{city}_y \\ f_i & \text{if } \text{goal-city}(p_j) = \text{city}_y \\ \text{none} & \text{otherwise} \end{array} \right. \)

\( \text{goal-for-city}(p_j, i) = \left\{ \begin{array}{ll} s^*[p_j] & \text{if } \text{init-city}(p_j) = \text{city}_y \\ f_i & \text{if } \text{goal-city}(p_j) = \text{city}_y \\ \text{none} & \text{otherwise} \end{array} \right. \)

The number of drives a package \( p \) induces for the subtask that considers only truck \( t \):

\( \text{drives}(t_i, p_j) = \left\{ \begin{array}{ll} 0 & \text{if } \text{init-city}(p_j) \neq \text{city}(t_i) \land \text{goal-city}(p_j) \neq \text{city}(t_i) \\ 0 & \text{if } \text{init-for-city}(p_j, i) = \text{goal-for-city}(p_j, i) \\ 1 & \text{if } \text{init-for-city}(p_j, i) = s_0[t_i] \\ 2 & \text{otherwise} \end{array} \right. \)

Logistics tasks can naturally be divided into two kinds of subtasks defined by the two types of vehicles (Helmert 2008; Paul et al. 2017): for each city there is one truck subtask, and there is another subtask that considers all airplanes and airports. Therefore each subproblem is a planning task with only one type of vehicle connected to the other subproblems by the airports. As shown by Paul et al. (2017) this gives a factored planning decomposition with precedence constraints on the connecting factor: the packages.

Algorithm 1: Perfect pattern selection for Logistics.

**Input:** SAS\(^+\) Logistics task

**Output:** perfect pattern collection \( C_L \)

1: \( C_L = \emptyset \)
2: for all \( t \in T \) do
3: \( \gamma^* = \arg \max_p \text{drives}(p, t_i) \)
4: \( \gamma_L += \{ t_i, \gamma^* \} \)
5: \( \text{flights} = \{ \} \)
6: for all \( p \in \mathcal{P} \) do
7: if \( \{ \text{init-city}(p), \text{goal-city}(p) \} \notin \text{flights} \) then
8: \( \text{flights} += \{ \text{init-city}(p), \text{goal-city}(p) \} \)
9: \( C_L += \{ \gamma^* \} \)
10: \( C_L += \{ \gamma^* \} \) \quad \triangleright \text{Only if } |A| = 2.
11: for all \( p \in \mathcal{P} \) do
12: if is-relevant(\( p \)) and \( p \) is not in any pattern in \( C_L \) then
13: \( C_L += \{ p \} \)

Lemma 1. A locally optimally solved city\( y \) imposes precedence constraints of the form: incoming \( \prec \) outgoing if \( s_0[t_i] = f_i \), where incoming are all packages that have to be delivered to city\( y \) with truck \( t_i \) (goal-city(\( p \)) = city\( y \) \& init-city(\( p \)) \neq city\( y \) \& drives(\( t_i, p \)) \geq 0) and outgoing are all packages that have to be delivered to other cities from city\( y \) with truck \( t_i \) (init-city(\( p \)) = city\( y \) \& goal-city(\( p \)) \neq city\( y \) \& drives(\( t_i, p \)) \geq 0).

A set of precedence constraints results in a conflict iff the graph formed by interpreting each constraint as a directed edge has a cycle. Therefore, we can safely combine local solutions from the subproblems if the resulting graph is acyclic. We call such Logistics tasks decoupled.

Definition 1. A Logistics task is decoupled if the precedence constraints of the subproblems form an acyclic graph.

In the following, we show that each of these subproblems can be individually captured perfectly with a cost partitioning over the pattern collection \( C_L \) defined by Algorithm 1 using the cost partitioning from Table 3.

Lemma 2. The \( C_L \) pattern collection is perfect for all truck subtasks.

Proof. The task for a truck problem in isolation is to bring all packages from init-for-city to goal-for-city. As reflected by the function drives, a package can only require zero, one, or two movements from a truck. For this proof, we first show that selecting the package with the highest number of drives always includes all drives for all other packages. For the packages with zero drives this is trivial and a package with two drives covers both other cases because there exist only two locations. Lastly, a package with one drive will always cover all other packages with one drive, because if there was a package requiring an additional movement of the truck, it would require two drives. This means that choosing the pattern with maximum drives for the pattern \( \{ t, p \} \) will always contribute enough cost units for all drives necessary.
Figure 4: Abstract transition system for the pattern \( \{ t, p \} \). Each state shows its variable assignment and heuristic values from the cost partitioning in Table 3. Transitions are labeled with operator abbreviations and costs. Inferable operator arguments, operators that induce only self-loop transitions of zero cost, and all states outside of the truck’s city are omitted. Since we allow for different initial states, we do not mark them here.

Table 3: Cost partitioning for \( C_L \) from Algorithm 1. Each cell shows which pattern gets the full cost of which operator.

![Abstract transition system for the pattern \( \{ a_1, p \} \) for two cities and two airplanes. See Figure 4 for a general explanation.](image)

![Abstract transition system for the pattern \( \{ a, p \} \) for one airplane and two airports. See Figure 4 for a general explanation. All truck-related states are omitted.](image)

![Abstract transition system for the pattern \( \{ p \} \) for two cities and two airplanes. See Figure 4 for a general explanation.](image)
in the truck’s city. Figures 7 and 4 show that the patterns \( \{t, p\} \) and \( \{p\} \) always cover the pick-up and drop-off costs for their packages. Therefore, all necessary actions (pick-up, drives, drop-off) are covered, showing that \( C_L \) is perfect for all truck subtasks.

**Lemma 3.** The \( C_L \) pattern collection is perfect for all airplane subtasks.

**Proof.** We split the proof into two cases: one airplane and two airplanes. With one airplane the corresponding abstraction in Figure 6 shows that pairing a package with the airplane results in a solution estimate that is perfect for that package. If a package is not paired with an airplane Figure 7 shows that its solution estimates only misses the transportation with the airplane. Pairing all unique destinations \( \{init-city, goal-city\} \) ensures that each airplane movement with a package is accounted for.

For the case of two airplanes, the corresponding abstractions are shown in Figure 5. For a single delivery, the pairing with \( a_1 \) will result in 2 cost units, and the pairing with \( a_2 \) in 1 unit. This again covers all 3 units necessary for delivering a package. Single packages and airplane distribution work the same as before making the heuristic cover all necessary actions.

Additional flights before pick-up are also covered in both cases, as the fly operators always get their full costs in the cost partitioning. Therefore all necessary actions are covered in both cases, showing that \( C_L \) is perfect for all airplane subtasks.

Finally, the combination of Lemmas 2 and 3 yields the desired optimality result for decoupled Logistics tasks.

**Theorem 2.** The \( C_L \) pattern collection is perfect for decoupled Logistics tasks.

**Proof.** As each subtask is perfectly accounted for, the combination of all patterns yields a perfect estimate, since for decoupled tasks we can combine locally optimal solutions to obtain a globally optimal solution.

The **interesting** systematic pattern generation method yields the following pattern collection for Logistics tasks:

\[
\{\{t\} | t \in T\} \cup \{\{a\} | a \in A\} \cup \{\{p\} | p \in P\} \cup (T \cup A) \times P
\]

As Table 4 shows, the \( C_L \) pattern collection again compares favorably against the systematic approaches. The number of patterns is quadratic in \( m, n \), and \( f \) for the naive approach and \( O(mn) \) for interesting, while for our pattern collection it is \( O(mnf) \). Regarding the total projection sizes, the asymptotic complexity is \( O(mnf^2) \) for both the interesting systematic and our approach.

**Related Work**

Our work is closely related to the analysis by Helmert and Mattmüller (2008), who determined the accuracy of several heuristics for classical planning in multiple domains, including Gripper and Logistics. They showed that the heuristic accuracy of single patterns can be arbitrarily bad in both domains. Furthermore, they analyzed **additive** pattern databases, a restricted form of cost partitioning. Two patterns are additive if no operator affects any variable from both patterns. The authors showed that for additive pattern databases heuristics the asymptotic heuristic accuracy relative to the true solution cost is \( \frac{2}{3} \) for Gripper and \( \frac{1}{2} \) for Logistics. By removing the additivity criterion for pattern collections and by using cost partitioning to compute estimates, we are able to raise the heuristic accuracy to 1. However, for Logistics a direct comparison is difficult because Helmert and Mattmüller studied a more general version of Logistics, where there can be more than two airplanes. For such tasks, even all patterns of size up to 2 are no longer perfect for airplane subtasks.

**Conclusions and Future Work**

We showed how to find pattern collections that allow obtaining perfect distance estimates for Gripper and Logistics. In both domains, our algorithms need fewer patterns and smaller projections to obtain perfect estimates than approaches that systematically generate larger patterns.

An obvious extension of our work will be to use the computed pattern collections in an A* search and compare them to domain-specific solvers. Our long-term goal is to learn pattern selection algorithms that approximate perfect pattern collections.

**References**


Table 4: Comparison of the systematic pattern generation methods (with patterns up to size 2) to $C_L$. Here, $n$ is the number of cities, $f$ is the number of airplanes, and $m$ is the number of packages. For $C_L$, we assume an initial state that incurs the most expensive solution.

<table>
<thead>
<tr>
<th>Method</th>
<th>#patterns</th>
<th>#states</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>$\frac{1}{2}((n + f + m)^2 + n + f + m)$</td>
<td>$\frac{9m^2n^2}{2} + 3m^2nf + \frac{m^2f^2}{2} + 3mn^2f + \frac{3mn^2}{2} + mnf^2 - mnf + 3mn - \frac{mf^2}{2} + m + n^2f^2 + \frac{3nf^2}{2} + 2n^2 + nf$</td>
</tr>
<tr>
<td>interesting</td>
<td>$m(n + f + 1) + n + f$</td>
<td>$2n + 3mn + mf + nf + 6mn^2 + mnf^2 + 3mn^2f + 2mnf$</td>
</tr>
<tr>
<td>$C_L$</td>
<td>$\min(n, m) + \min(n(n - 1), m)f + \max(0, m - \max(\min(n, m), n(n - 1)))$</td>
<td>$(3n + f) \max(0, m - \min(n(n - 1), \min(n, m))) + n(3n^2 + p) \min(m, n(n - 1)) + 2(3n + p) \min(m, n)$</td>
</tr>
</tbody>
</table>


