Efficient Evaluation of Large Abstractions for Decoupled Search: Merge-and-Shrink and Symbolic Pattern Databases

Daniel Gnad[1], **Silvan Sievers**[2], **Álvaro Torralba**[3]

[1] Linköping University, Sweden
[2] University of Basel, Switzerland
[3] Aalborg University, Denmark
daniel.gnad@liu.se, silvan.sievers@unibas.ch, alto@cs.aau.dk

Abstract

Abstraction heuristics are a state-of-the-art technique to solve classical planning problems optimally. A common approach is to precompute many small abstractions and combine them admissibly using cost partitioning. Recent work has shown that this approach does not work out well when using such heuristics for decoupled state space search, where search nodes represent potentially large sets of states. This is due to the fact that admissibly combining the estimates of several heuristics without sacrificing accuracy is **NP**-hard for decoupled states. In this work we propose to use a single large abstraction instead. We focus on merge-and-shrink and symbolic pattern database heuristics, which are designed to produce such abstractions. For these heuristics, we prove that the evaluation of decoupled states is **NP**-hard in general, but we also identify conditions under which it is polynomial. We introduce algorithms for both the general and the polynomial case. Our experimental evaluation shows that single large abstraction heuristics lead to strong performance when the heuristic evaluation is polynomial.

Introduction

Classical planning is the problem of finding a sequence of deterministic actions leading from a given initial state of the world to a goal (e.g., Ghallab, Nau, and Traverso 2004). In this paper, we are concerned with *optimally* solving classical planning tasks. The dominant approach of the recent years is the A* algorithm (Hart, Nilsson, and Raphael 1968) in conjunction with *admissible heuristics* (Pearl 1984). There are several ways to represent the state space: for example, *explicit* state spaces keep individual states, while *symbolic* state spaces use decision diagrams to represent sets of states (e.g., Torralba et al. 2017). *Decoupled search* (Gnad and Hoffmann 2018) is another alternative based on *factoring* the variables of a planning task into center and leaf variables, similar to other factored planning approaches (Amir and Engelhardt 2003; Brafman and Domshlak 2006; Fabre et al. 2010). The main difference is that this is done so that variables of different leaf factors are conditionally independent. The search is then restricted to branch over global actions only and each search node consists of a center state and a set of leaf states that is associated with an additional cost.

For explicit search, state-of-the-art heuristics are based on *abstractions* combined with *saturated cost partitioning* (e.g., Seipp and Helmert 2018; Seipp, Keller, and Helmert 2020; Seipp 2021). For decoupled search, any kind of explicit state heuristic can in principle also be used to evaluate decoupled states using a compilation of the task based on the decoupled state to be evaluated. This has the insurmountable drawback for abstraction heuristics that they would need to be entirely recomputed for each state evaluation rather than precomputing them once as usually. Recently, Sievers, Gnad, and Torralba (2022) introduced the alternative of enumerating all potentially exponentially many explicit *member states* represented by the decoupled state. However, this infeasible in practice, too. They further showed that single *pattern databases* (PDBs) (Culberson and Schaeffer 1998; Edelkamp 2001) can be evaluated in polynomial time in their size. However, admissibly combining a collection of PDBs without losing information compared to explicitly enumerating all member states is an **NP**-hard problem.

In this paper, to avoid the problem of admissibly combining heuristics, we consider large single abstraction heuristics. In particular, *symbolic* PDBs (Edelkamp 2002; Kissmann and Edelkamp 2011; Torralba, Linares López, and Borrajo 2018) allow representing much larger patterns than explicit ones, coming close to the state of the art in explicit search (Franco and Torralba 2019). Furthermore, *merge-and-shrink* heuristics (e.g., Helmert et al. 2014; Sievers and Helmert 2021) are the most general type of abstractions and typically reflect all variables of a task to some degree. We show that the compact representation of the data structures underlying symbolic PDBs and merge-and-shrink heuristics, namely *algebraic decision diagrams* (ADDs) (Bahar et al. 1997) and *factored mappings* (FMs) (Helmert, Röger, and Sievers 2015), comes with the price that evaluating a decoupled state with these data structures is an **NP**-hard problem. Fortunately, we can show that when restricting the variable order underlying the ADD or the merge strategy underlying the FM to be *compliant* with the factoring used for decoupled search, heuristic evaluation is polynomial in the size of the ADD or FM. Our experimental evaluation shows that restricting the heuristics to be compliant with the factoring has no negative impact on heuristic quality, and that decoupled search with both compliant symbolic PDBs and merge-and-shrink heuristics outperforms its explicit search counterpart.

We have some full proofs, code, and experimental data in an online appendix (Gnad, Sievers, and Torralba 2023).

Background

Classical Planning

A *variable space* is a finite set $\mathcal{V}$ of variables v, each with a finite domain $\mathcal{D}(v)$. A *partial state* s assigns all variables in $vars(s) \subseteq \mathcal{V}$ to a value $s[v] \in \mathcal{D}(v)$. We also view s as a set of *facts* $v \mapsto s[v]$ for all $v \in vars(s)$. If $vars(s) = \mathcal{V}$, s is called a *state*. For subset $\mathcal{V}' \subseteq \mathcal{V}$, we write $s[\mathcal{V}']$ to denote the restriction of s to $vars(s) \cap \mathcal{V}'$. We write $S(\mathcal{V})$ for the set of states (assignments) defined over $\mathcal{V}$.

We consider planning tasks in the SAS$^+$ formalism, where a task is defined as $\Pi = \langle \mathcal{V}, \mathcal{A}, I, G \rangle$. $\mathcal{V}$ is a variable space. $\mathcal{A}$ is a finite set of *actions* $a = \langle \mathsf{pre}(a), \mathsf{eff}(a), c(a) \rangle$, where $\mathsf{pre}(a)$ and $\mathsf{eff}(a)$ are partial states called *precondition* and *effect* of a, respectively, and $c(a) \in \mathbb{R}_0^+$ is the *cost* of a. I is the *initial state* and G is a partial state called the *goal*.

A *transition system* is defined as $\Theta = \langle S, L, c, T, s_I, S_G \rangle$. S is a finite set of states. L is a finite set of *labels* and $c : L \mapsto \mathbb{R}_0^+$ a *label cost function*. $T \subseteq S \times L \times S$ is a *transition relation*. $s_I \in S$ is the initial state and $S_G \subseteq S$ is the set of goal states. A plan π for Θ is a sequence of labels leading from s_I to some goal state. The cost of π, $c(\pi)$, is the sum of the label costs.

A planning task Π as defined above induces a transition system $\Theta(\Pi) = \langle S, L, c, T, s^I, S^G \rangle$ as follows: $S = S(\mathcal{V})$, $L = \mathcal{A}$; $T = \{\langle s, \ell, t \rangle \mid s, t \in S, \ell \in L, \mathsf{pre}(\ell) \subseteq s, t = s[\![\ell]\!]\}$: action ℓ is *applicable* in s if its precondition is satisfied in s and its application leads to *successor state* t which assigns all variables according to the effect of ℓ and leaves all other variables unchanged; $s^I = I$; $S^G = \{s \in S \mid G \subseteq s\}$. A plan for Π is a plan for $\Theta(\Pi)$. It is *optimal* if its cost is minimal among all plans. Optimally solving a task means to find an optimal plan or to show that no plan exists.

A *heuristic* for transition system Θ with states S is a function $h_\Theta : S \to \mathbb{R}_0^+$ that estimates the true cost of reaching a goal state from a given state s, denoted as the perfect heuristic $h^*_\Theta(s)$. h_Θ is *admissible* if $h_\Theta(s) \leq h^*(s)$ for all $s \in S$. We often drop Θ and write h if Θ is clear from context.

Merge-and-Shrink

The merge-and-shrink framework is based on *transformations* of transition systems to compute *abstractions* of a given transition system. To represent state mappings of these transformations, merge-and-shrink employs *factored mappings* (FMs) (Sievers and Helmert 2021). FMs over a variable space $\mathcal{V}$ are inductively defined as follows. An FM σ has an associated finite non-empty value set $vals(\sigma)$ and an associated table σ^{tab}. An *atomic* FM σ has an associated variable $v \in \mathcal{V}$ and its table is a function $\sigma^{\text{tab}} : \mathcal{D}(v) \mapsto vals(\sigma)$. A *merge* FM σ has left and right component FMs σ_L and σ_R and its table is a function $\sigma^{\text{tab}} : vals(\sigma_L) \times vals(\sigma_R) \mapsto vals(\sigma)$. FMs σ *represent the function* $[\![\sigma]\!] : S(\mathcal{V}) \mapsto vals(\sigma)$. Let $s \in S(\mathcal{V})$. For an atomic FM σ with associated variable v, $[\![\sigma]\!](s) = \sigma^{\text{tab}}(s[v])$. For a merge FM σ, $[\![\sigma]\!](s) = \sigma^{\text{tab}}([\![\sigma_L]\!](s), [\![\sigma_R]\!](s))$.

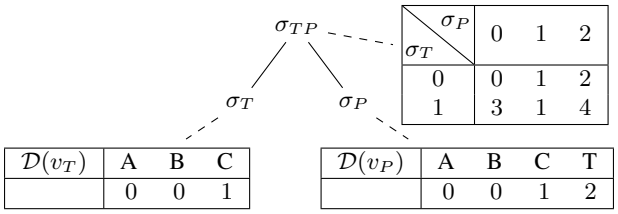

Figure 1: FM σ_{TP} over variable space $\{T, P\}$.

Algorithm 1: Evaluating FMs on explicit states.

```
   Input: FM σ, state s
   Output: [[σ]](s)
1  def FM-eval(σ, s):
2      return FM-traverse(σ, s)
3  def FM-traverse(σ, s):
4      if vars(σ) = {v} then                    // atomic FM
5          return σ^tab(s[v])
6      else
7          i_L ← FM-traverse(σ_L, s)
8          i_R ← FM-traverse(σ_R, s)
9          return σ^tab(i_L, i_R)
```

FMs can be understood as binary trees, with atomic FMs corresponding to leaf nodes and merge FMs to inner nodes. In this context, by *descendants* of σ, we denote all component FMs in the subtree rooted by σ (including itself), and we define $vars(\sigma)$ as the set of associated variables of all atomic descendants of σ. Consider a simple example with $\mathcal{V} = \{T, P\}$ encoding the position of a truck and a package in a logistics task with three locations A, B, C, i.e., $\mathcal{D}(T) = \{\text{A}, \text{B}, \text{C}\}$ and $\mathcal{D}(P) = \{\text{A}, \text{B}, \text{C}, \text{T}\}$. The root FM σ_{TP} is a merge FM with left and right component FMs σ_T and σ_P, which are atomic FMs with associated variable P and T, respectively. Figure 1 visualizes the underlying tree structure of σ_{TP}, labeling each node with the FM it corresponds to and connecting it to its table with a dashed line.

We define the *size* of an FM σ, $\|\sigma\|$, and of its table σ^{tab}, $\|\sigma^{\text{tab}}\|$, inductively as follows: if σ is atomic with associated variable v, $\|\sigma\| = \|\sigma^{\text{tab}}\| = |\mathcal{D}(v)|$. If it is a merge, then $\|\sigma^{\text{tab}}\| = |vals(\sigma_L)| \cdot |vals(\sigma_R)|$ and $\|\sigma\| = \|\sigma^{\text{tab}}\| + \|\sigma_L\| + \|\sigma_R\|$. In our example $\|\sigma_{TP}\| = 2 \cdot 3 + 3 + 4 = 13$.

Plugging state $s = \{v_T \mapsto \text{A}, v_P \mapsto T\}$ into the inductive definition of the computation of an FM, we get $[\![\sigma_{TP}]\!](s) = \sigma^{\text{tab}}_{TP}([\![\sigma_T]\!](s), [\![\sigma_P]\!](s)) = \sigma^{\text{tab}}_{TP}(\sigma^{\text{tab}}_T(s[v_T]), \sigma^{\text{tab}}_P(s[v_P])) = \sigma^{\text{tab}}_{TP}(\sigma^{\text{tab}}_T(\text{A}), \sigma^{\text{tab}}_P(\text{T})) = \sigma^{\text{tab}}_{TP}(0, 2) = 2$. Algorithm 1 shows an equivalent recursive procedure for evaluating a state with an FM: the call `FM-eval`(σ_{TP}, s) recurses into evaluating the left component σ_T in line 7, which returns $\sigma^{\text{tab}}_T(s[v_T]) = \text{A}$ in line 5 because σ_T is atomic. After analogously computing $\sigma^{\text{tab}}_P(s[v_P]) = \text{T}$ from the recursive call in line 8, the end result $\sigma^{\text{tab}}_{TP}(\text{A}, \text{T})$ is returned via lines 9 and 2.

For the purpose of computing abstractions of a task Π, merge-and-shrink maintains a *factored transition system* (FTS) F, which is a set of transition systems Θ_i (called

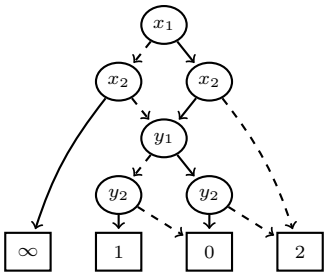

Figure 2: ADD representing a heuristic function.

factors) sharing the same labels, and a set Σ of FMs σ_i mapping the states of $\Theta(\Pi)$ to abstract states of Θ_i. Initially, Initially, F contains the *atomic factors* corresponding to the variables of Π and Σ contains atomic FMs for these atomic factors. Both F and Σ are repeatedly transformed using the four standard transformations. Merging replaces two factors of F by their product. Shrinking applies an abstraction to some factor of F. Label reduction applies an abstraction to the common label set of F. Pruning removes some states and their transitions from a factor of F. At any point, each factor Θ_i of F together with its corresponding FM σ_i of Σ induces a heuristic for $\Theta(\Pi)$: $h_i^{M\&S}(s) = h^*_{\Theta_i}([\![\sigma_i]\!](s))$. The overall merge-and-shrink heuristic is defined as $h^{M\&S}(s) = \max_i h_i^{M\&S}(s)$.

A *merge strategy* determines the order in which the atomic factors of Π (and thus the atomic FMs) are merged over the course of the algorithm. It is called *linear* if it merges the atomic factors one by one, leading to a degenerate chain-like tree. Otherwise, merge strategies are called *non-linear*.

Symbolic Pattern Databases

Pattern databases (PDBs) are abstraction heuristics with a simple abstraction mapping: the projection onto a subset of variables called the pattern. Symbolic PDBs precompute the distances by performing a symbolic backward search (McMillan 1993; Edelkamp 2002), using reduced ordered *binary decision diagrams* (BDDs) (Bryant 1986) to compactly represent sets of states. This allows for patterns containing many (or even all) variables (Kissmann and Edelkamp 2011; Torralba, Linares López, and Borrajo 2018). To keep the computation tractable, the search can be stopped anytime, using the resulting perimeter around the goal as a heuristic (Anderson, Holte, and Schaeffer 2007).

The resulting heuristic can be represented as an *algebraic decision diagram* (ADD) (Bahar et al. 1997). ADDs are a variant of BDDs where there may be more than two terminal nodes, each corresponding to a possible heuristic value. Figure 2 shows an example of an ADD representing a heuristic h for a task with $\mathcal{V} = \{x_1, x_2, y_1, y_2\}$. Note that in ADDs all variables are binary, so they operate on a different set of variables $\mathcal{V}_{\mathsf{ADD}}$. To deal with this, one can simply encode each $v \in \mathcal{V}$ using $\log_2(\mathcal{D}(v))$ binary variables. We leave this conversion implicit, and assume WLOG that $\mathcal{V} = \mathcal{V}_{\mathsf{ADD}}$.

ADDs have internal and terminal nodes. Each terminal node is associated with a numeric constant, which in our case represents a possible heuristic value ($\{0, 1, 2, \infty\}$ in our example). Each internal node n is characterized by a variable $n.v$ and two children: the 0-child (represented with a dashed edge) and the 1-child (represented with a solid edge). Given a state s, one can compute $h(s)$ by a single traversal of the ADD, following the path from the root node to a terminal node. At each internal node n, we follow the solid edge if $s[n.v] = 1$ and the dashed edge when $s[n.v] = 0$.

ADDs are always *reduced* and *ordered*. In reduced ADDs, two equivalent functions are always represented by the same node and no internal node has the same child in both edges. Ordered ADDs have a fixed variable order such that for any two nodes, if n' is a child of n then $n.v < n'.v$, according to some total order on $\mathcal{V}_{\mathsf{ADD}}$. The size of an ADD, $|A|$, is simply its number of nodes.

Decoupled Search

Decoupled state space search is a technique that decomposes a given planning task Π by partitioning its variables (Gnad and Hoffmann 2018). Concretely, the variables are partitioned into subsets that form a *factoring* $\mathcal{F} = \langle C, \mathcal{L}\rangle$, where C is the (possibly empty) *center factor* of $\mathcal{F}$ and $\mathcal{L}$ is its set of non-empty leaf factors, such that $C \cup \bigcup_{L\in\mathcal{L}} L = \mathcal{V}$. This definition captures the recently introduced notion of *generalized factorings* (Gnad, Torralba, and Fišer 2022). For a variable $v \in \mathcal{V}$, by $\mathcal{F}(v)$ we denote the factor in which v is contained. Complete assignments to C, respectively an $L \in \mathcal{L}$, are called *center states* respectively *leaf states*. $S^{\mathcal{L}}$ is the set of all leaf states and that of a leaf L is denoted S^L.

A factoring $\mathcal{F}$ induces a partitioning of the actions into *global actions* $\mathcal{A}^G$ and *leaf actions* $\mathcal{A}^{\mathcal{L}}$. For every $L \in \mathcal{L}$, the leaf actions $\mathcal{A}^L$ of L are those actions that have effects only on L and are preconditioned by variables in $C \cup L$, formally $\mathcal{A}^L := \{a \in \mathcal{A} \mid vars(\mathsf{eff}(a)) \subseteq L, vars(\mathsf{pre}(a)) \subseteq C \cup L\}$. We define $\mathcal{A}^{\mathcal{L}} := \bigcup_{L\in\mathcal{L}} \mathcal{A}^L$ and $\mathcal{A}^G := \mathcal{A} \backslash \mathcal{A}^{\mathcal{L}}$. A sequence of global actions π^G applicable in I in the projection of Π onto C is called a *global path*, a sequence of leaf actions π^L applicable in I in the projection onto L is called a *leaf path*.

In decoupled search, search nodes correspond to sets of states, called decoupled states. A *decoupled state* $s^{\mathcal{F}}$ is a pair $\langle s^C(s^{\mathcal{F}}), \mathsf{prices}(s^{\mathcal{F}})\rangle$ of a center state $s^C(s^{\mathcal{F}})$ and a *pricing function* $\mathsf{prices}(s^{\mathcal{F}})$, where $\mathsf{prices}(s^{\mathcal{F}}) : S^{\mathcal{L}} \mapsto \mathbb{R}^{0+} \cup \infty$ assigns a finite non-negative *price* to every leaf state reached in $s^{\mathcal{F}}$ and ∞ to unreached leaf states. We define the *size* of a decoupled state as $|s^{\mathcal{F}}| := |\{s^L \in S^{\mathcal{L}} \mid \mathsf{prices}(s^{\mathcal{F}})[s^L] < \infty\}|$. A set of facts p is reached in a decoupled state $s^{\mathcal{F}}$, denoted $s^{\mathcal{F}} \models p$, if $p[C] \subseteq s^C(s^{\mathcal{F}})$ and for all $L \in \mathcal{L}$ there exists a leaf state $s^L \supseteq p[L]$ such that $\mathsf{prices}(s^{\mathcal{F}})[s^L] < \infty$.

Every decoupled state $s^{\mathcal{F}}$ represents a set of explicit states, its *member states*. These are exactly the states s where $s^{\mathcal{F}} \models s$. The set of all member states of $s^{\mathcal{F}}$ is denoted $[s^{\mathcal{F}}]$. Every member state of $s^{\mathcal{F}}$ is associated with a price, which is the sum of the leaf state prices it consists of, formally $\mathsf{price}(s^{\mathcal{F}}, s) := \sum_{L\in\mathcal{L}} \mathsf{prices}(s^{\mathcal{F}})[s[L]]$.

We denote the global path on which a decoupled state $s^{\mathcal{F}}$ is reached during search by $\pi^G(s^{\mathcal{F}})$. Along every search path, the pricing function keeps track of the minimum cost of reaching leaf states along $\pi^G(s^{\mathcal{F}})$ via *compliant* leaf paths from the initial state. A leaf path is compliant with $\pi^G(s^{\mathcal{F}})$

if it can be embedded into $\pi^G(s^\mathcal{F})$ and the resulting action sequence is applicable in I in the projection of Π onto $C \cup L$.

Decoupled heuristics estimate the remaining cost to reach a goal state for decoupled states $s^\mathcal{F}$, by approximating the minimum such cost of any member state in $[s^\mathcal{F}]$. Because, unlike explicit-state search, the cost of reaching $s^\mathcal{F}$ only takes into account the cost of the global actions, it is important to consider the pricing function in the heuristic, too. We denote decoupled heuristics derived from an explicit-state heuristic h by $h_\mathcal{F}$. Recently, Sievers, Gnad, and Torralba (2022) defined the so-called *explicit decoupled heuristic* $h_{\mathcal{F},\text{ex}}$, which evaluates a heuristic h separately for all member states of $s^\mathcal{F}$ and serves as our baseline; formally $h_{\mathcal{F},\text{ex}}(s^\mathcal{F}) := \min_{s \in [s^\mathcal{F}]} \mathsf{price}(s^\mathcal{F}, s) + h(s)$.

Complexity of Evaluating Large Abstractions

Recent work has proven that PDB heuristics can be evaluated for decoupled states in polynomial time, i. e., linear in the size of the abstract state space (Sievers, Gnad, and Torralba 2022). The abstractions we consider in this work, in contrast, do not store a value for every abstract state but keep a compact data structure instead. In the case of merge-and-shrink heuristics, these take the form of factored mappings (FM), whereas symbolic PDB heuristics are stored as algebraic decision diagrams (ADD).

In this section, we show that evaluating a decoupled heuristic exactly on these data structures is **NP**-complete. The proof works by a reduction from 3SAT, encoding a planning task Π, factoring $\mathcal{F}$, and a compactly represented abstraction heuristic h such that evaluating h on a specific decoupled state $s^\mathcal{F}$ yields ∞ iff the corresponding 3-CNF formula is unsatisfiable. We next provide the task definition, then show **NP**-completeness when an ADD is used to represent the heuristic. From this, hardness for FMs follows from a known connection between ADDs and FMs.

The task encoding is the same as the one presented in the proof of Theorem 1 by Sievers, Gnad, and Torralba (2022). Given a 3-CNF formula ϕ with propositions X and clauses $\{C_1, \dots, C_m\}$, we construct a planning task Π with variables $\mathcal{V} = \{v_g\} \cup \{v_x \mid x \in X\} \cup \{v_x^{L_i}, v_y^{L_i}, v_z^{L_i}, v^{L_i} \mid x, y, z \in C_i\}$ with domains $\mathcal{D}(v) = \{u, 0, 1\}$ (unassigned, false, true) for $v \in \{v_w^{L_i} \mid w \in \{x, z, y\}, 1 \leq i \leq m\}$ and $\mathcal{D}(v) = \{i, g\}$ (initial, goal) for $v \in \{v_g\} \cup \{v_x \mid x \in X\} \cup \{v^{L_i} \mid 1 \leq i \leq m\}$. Initially, all variables have value u, respectively i. The goal requires all variables except the $v_w^{L_i}$ ones to have value g.

For every clause C_i, there is a leaf factor $L_i = \{v_x^{L_i}, v_y^{L_i}, v_z^{L_i}, v^{L_i}\}$ that contains a variable $v_w^{L_i}$ for every propositional variable w in C_i and v^{L_i} for book-keeping. The center factor is $C = \{v_g\} \cup \{v_x \mid x \in X\}$.

In every leaf, there exist actions a_i^L that set v^{L_i} from i to g and set the propositional variables to any assignment that satisfies C_i (7 actions per clause C_i). Moreover, there exist $2|X| + 1$ global actions, two per $x \in X$ with precondition $\mathsf{pre}(a_x^0) = \{v_x = i\} \cup \{v_x^{L_i} = 0 \mid \forall L_i : x \in C_i\}$ and effect $\mathsf{eff}(a_x^0) = \{v_x = g\}$, analogously for a_x^1, and a last action a_g that sets v_g from i to g with additional preconditions $\{v^{L_i} = g \mid 1 \leq i \leq m\}$. All actions have cost 0.

In decoupled search, the initial state $I^\mathcal{F}$ has a price of 0 for all leaf states, i. e., the initial leaf states $\{v^{L_i} = i, v_x^{L_i} = u, v_y^{L_i} = u, v_z^{L_i} = u\}$ and all satisfying assignments for x, y, z in clause C_i with the fact $v^{L_i} = g$.

Consider the decoupled state $s^\mathcal{F}$ that results from applying a_g to $I^\mathcal{F}$. In that state, all leaf states where $v^{L_i} = i$ are no longer reached due to the precondition $v^{L_i} = g$, so in all member states of $s^\mathcal{F}$ all variables encoding propositions are assigned a truth value. Note that $s^\mathcal{F}$ is solvable iff the 3-CNF is satisfiable, as the a_x^0, a_x^1 actions ensure that the goal values g can only be reached if all variables $v_x^{L_i}, v_x^{L_j}, \dots$ that encode the assignment to x have the same value.

Our proof works by showing that we can compactly represent a heuristic that detects $s^\mathcal{F}$ as a dead-end using an ADD.

Theorem 1. *Let A be an ADD that represents a PDB heuristic h, $s^\mathcal{F}$ a decoupled state, and $B \in \mathbb{R}^+ \cup \{\infty\}$ a bound. It is **NP**-complete to decide if $h_\mathcal{F}(s^\mathcal{F}) < B$.*

Proof sketch. For membership, one can guess a state $s \in [s^\mathcal{F}]$ and check if $h(s) < B$ in polynomial time.

For hardness, consider the planning task encoding 3SAT described above. We construct A using a variable order that keeps next to each other all variables encoding the same proposition $X = \{x, y, z, \dots\}$ in different clauses: $\langle v_x, v_x^{L_1}, \dots, v_x^{L_k}, v_y, v_y^{L_o}, \dots, v_y^{L_p}, v_z, v_z^{L_q}, \dots, v_z^{L_r}, \dots \rangle$.

In A we encode the function that assigns ∞ to all states where for some proposition x, $v_x = b$ and there are two clauses for which x has different values (i.e. $v_x^{L_i} \neq v_x^{L_j}$). All other states are assigned 0. We can bound the size of A by $2|X||C|$. Basically, any cluster of variables $v_x^{L_1}, \dots, v_x^{L_k}, v_x$ for each $x \in X$ is expressed as an ADD that uses $2k \leq 2|C|$ internal nodes, as at every of the k layers, we only need two nodes to distinguish the case where all previous variables were 0 or 1. Note that no node needs to represent the case where two variables differ because in that case, we go directly to the terminal node ∞. An extra node is needed to check the value of v_x, which is compensated by the fact that the first layer only has a single node. The clusters are independent, and therefore the corresponding ADDs are just stacked on top of each other (Edelkamp and Kissmann 2008). Figure 2 shows an example where $h_{\mathsf{ADD}}(s^\mathcal{F}) = 0$ iff $(x_1 = x_2) \wedge (y_1 = y_2)$.

Hence, all involved components, the planning task, factoring, and ADD are polynomially bounded by the size of the 3-CNF formula. The claim follows since the state $s^\mathcal{F}$ that results from applying a_g in $I^\mathcal{F}$ is detected as a dead-end by $h_\mathcal{F}$ iff the formula is satisfiable. □

For FMs, we remark that every ADD can be transformed into an FM in polynomial time (Edelkamp, Kissmann, and Torralba 2012; Torralba 2015).

Corollary 2. *Let σ be an FM that represents a merge-and-shrink heuristic h and $s^\mathcal{F}$ a decoupled state. It is **NP**-complete to decide if $h_\mathcal{F}(s^\mathcal{F}) < B$.*

Merge-and-Shrink

Factored mappings (FMs), like decoupled states, are data structures that compactly represent large sets of states. As

we have seen in the previous section, combining them arbitrarily renders the evaluation of a decoupled state with an FM an **NP**-complete problem. The reason is that we need to "unpack" the decoupled state and evaluate all member states with the FM, thus losing the compactness of the data structures. To avoid this problem, we need to ensure that decoupled states and FMs are "compatible". We next introduce the notion of compliance of FMs and factorings and then provide a dynamic-programming algorithm that evaluates FMs for decoupled states. The algorithm works for general FMs, but has polynomial runtime only for FMs that are compliant with the factoring, where it caches intermediate results.

We assume that the root FM σ_h has merged all variables, i. e., $vars(\sigma_h) = \mathcal{V}$. This is not a restriction, as in cases where this does not hold, i. e., $vars(\sigma_h) = V \subset \mathcal{V}$, we can simply redefine the factoring $\mathcal{F} = \langle C, \mathcal{L} \rangle$ for σ_h such that $\mathcal{F}^{\sigma_h} := \langle C \cap V, \{L \cap V \mid L \in \mathcal{L}, L \cap V \neq \emptyset\} \rangle$.[1]

We start by defining the leaves covered by an FM:

Definition 3 (Covered Leaves by FMs, Nestedness)**.** *Let $\mathcal{F}$ be a factoring. The set of leaf factors* covered *by an FM σ is defined as $cov(\sigma) := \{L \in \mathcal{L} \mid L \cap vars(\sigma) \neq \emptyset\}$. The set of leaf factors* fully covered *by σ is defined as $fc(\sigma) := \{L \in \mathcal{L} \mid L \subseteq vars(\sigma)\}$. The set of leaf factors* partially covered *by an FM σ is defined as $pc(\sigma) := cov(\sigma) \setminus fc(\sigma)$. The set of leaf factors* exactly covered *by an FM σ is defined as $ec(\sigma) := fc(\sigma) \setminus (fc(\sigma_L) \cup fc(\sigma_R))$ if σ is a merge FM, and $ec(\sigma) := fc(\sigma)$ otherwise. The* nestedness *of a merge FM σ is defined as $\mathcal{N}(\sigma) := max(\mathcal{N}(\sigma_L), \mathcal{N}(\sigma_R), |pc(\sigma_L) \cup pc(\sigma_R)|)$, and $\mathcal{N}(\sigma) := 0$ for atomic FMs σ.*

In words, a leaf L is covered by an FM σ if both share some variables; L is fully covered by σ if all its variables are contained in σ; L is partially covered if a strict subset of its variables is contained in σ. The latter implies that the handling of L cannot yet be finished in σ, but only in one of its ancestors. Finally, every leaf L is exactly covered in exactly one FM σ, namely the lowest FM in the tree that covers all variables of L. Based on this, we define the nestedness of an FM σ as the number of leaves whose handling has not finished in the left and right component FMs. We also call the leaves in $pc(\sigma_L) \cup pc(\sigma_R)$ *open* in σ. As we will see below, in the worst case our algorithm has to evaluate all combinations of leaf states belonging to open leaves in σ. The maximum number of these leaves is exactly $\mathcal{N}(\sigma_h)$.

Based on these notions, we can restrict the structure of FMs to ensure *compliance* with a factoring.

Definition 4 (Compliant FMs)**.** *An FM σ is* compliant *with a factoring $\mathcal{F}$ if $\mathcal{N}(\sigma) \leq 1$. An FM σ is* strongly compliant *with a factoring $\mathcal{F}$ if for all $L \in \mathcal{L}$ there exists a descendant σ' of σ such that $cov(\sigma') = ec(\sigma') = \{L\}$.*

In a compliant FM, merging variables of different leaves cannot be interleaved, so at most one leaf can be open at the same time, although fully covered leaves can be interleaved. A strongly compliant FM must fully merge all variables in each leaf before merging variables from another leaf. Center variables can always be merged at any point. Note that, for every factoring where leaves contain a single variable, every FM is strongly compliant. We further remark that strong compliance implies compliance, so it is a special case.

Proposition 5. *Let $\mathcal{F}$ be a factoring and σ an FM. If σ is strongly compliant with $\mathcal{F}$, then σ is compliant with $\mathcal{F}$.*

We now have all ingredients to present our algorithm for evaluating a decoupled state $s^{\mathcal{F}}$ with an FM σ_h. Similarly to how Algorithm 1 evaluates explicit states, the decoupled variant recursively traverses the FM. However, as a decoupled state represents a set of explicit states, the algorithm needs to compute a set of values V_{σ_h} rather than a single one. Here, V_{σ_h} is the subset of values $vals(\sigma_h)$ that the member states s of $s^{\mathcal{F}}$ are mapped to by σ_h, i.e., $V_{\sigma_h} = \{\sigma_h^{\text{tab}}(s) \mid s \in [s^{\mathcal{F}}]\}$. In fact, the algorithm not only computes the set of values V_{σ_h}, but a price mapping P_{σ_h} which maps each $v \in V_{\sigma_h}$ to the minimum price p of any member state s mapped to v.

Algorithm 2 shows the function `FM-eval` for evaluating decoupled state $s^{\mathcal{F}}$ with FM σ_h. It initializes partial state s to the center state of $s^{\mathcal{F}}$ (line 2), which will be augmented by different combinations of reached leaf states of $s^{\mathcal{F}}$ during the traversal. `FM-eval` then computes the price mapping P_{σ_h} explained above, by calling `handle-FM` (line 3), explained below. The returned heuristic is computed as the minimum over the sum of v (which in σ_h is the heuristic value) and the minimum price p corresponding to v (line 4).

Next, we provide an overview of an execution trace of the algorithm. Function `handle-FM` (ignore the cache for the moment) deals with the given FM σ in the sense that it computes the price mapping P_σ for σ. It does so using two auxiliary functions: if σ exactly covers at least one leaf (line 10), that means that we can compute all combinations of reached leaf states of $s^{\mathcal{F}}$ for those leaves $\mathcal{L}_{ec}$. This is done recursively by the function `enum-ec-leaves`, which proceeds to calling `FM-traverse-dec` for each such combination. If σ does not exactly cover any leaf, the algorithm directly proceeds with calling `FM-traverse-dec` (line 12). Finally, `FM-traverse-dec` is the decoupled counterpart of `FM-traverse` of Algorithm 1 and as such terminates recursion if σ is atomic and otherwise recursively calls `handle-FM` for the component FMs of σ.

The function `enum-ec-leaves` takes a parameter $\mathcal{L}_{ec}$, initially set to the exactly covered leaves of the given FM σ. Each recursive call deals with one leaf L of $\mathcal{L}_{ec}$ by iterating over its reached leaf states in $s^{\mathcal{F}}$ (lines 19 and 20), extending partial state s accordingly and entering recursion (line 21). Thereby it computes all combinations of reached leaf states across $\mathcal{L}_{ec}$. For each reached leaf state, the price mapping P_σ, initialized to infinity for all values of the FM (line 18), is updated by minimizing, for each value, the sum of the price of the reached leaf state s^L plus the price p computed recursively for the remaining leaves in $\mathcal{L}_{ec}$ (lines 22 and 23). This ensures that the returned price mapping P_σ maps value $v \in vals(\sigma)$ to the minimum price of any member state $s' \in [s^{\mathcal{F}}]$ that is compatible with s and that is mapped to v by σ^{tab}. Note that multiple member states can be mapped to the same value due to shrinking, which is why the algorithm minimizes the price over these in line 23. Recursion ends

[1] We further assume that every variable $v \in \mathcal{V}$ is associated to exactly one atomic descendant of σ_h.

Algorithm 2: Evaluating FMs on decoupled states.

Input: FM σ_h, decoupled state $s^{\mathcal{F}}$
Output: $[\![\sigma_h]\!](s^{\mathcal{F}})$

```
1  def FM-eval(σ_h, s^F):
2      s ← s^C(s^F)          // s is a partial state
3      P_{σ_h} ← handle-FM(σ_h, s^F, s)
4      return min_{v↦p∈P_{σ_h}} v + p   // heuristic + price
5  def handle-FM(σ, s^F, s):
6      V ← vars(σ) ∩ ⋃_{L∈pc(σ)} L      // cache vars
7      if s[V] ∈ cache_σ then
8          return cache_σ[s[V]]
9      if ec(σ) ≠ ∅ then
10         P_σ ← enum-ec-leaves(σ, s^F, s, ec(σ))
11     else
12         P_σ ← FM-traverse-dec(σ, s^F, s)
13     cache_σ[s[V]] ← P_σ
14     return P_σ
15 def enum-ec-leaves(σ, s^F, s, L_ec):
16     if L_ec = ∅ then
           // all v ∈ ⋃_{L∈ec(σ)} L are defined in s
17         return FM-traverse-dec(σ, s^F, s)
18     P_σ ← {v ↦ ∞ | v ∈ vals(σ)}      // init. map
19     let L ∈ L_ec; L'_ec ← L_ec \ {L}
20     for s^L ∈ S^L : prices(s^F)[s^L] < ∞ do
21         P'_σ ← enum-ec-leaves(σ, s^F, s ∪ s^L, L'_ec)
22         for v ↦ p ∈ P'_σ do
23             P_σ[v] ← min(P_σ[v], p + prices(s^F)[s^L])
24     return P_σ
25 def FM-traverse-dec(σ, s^F, s):
26     P_σ ← {v ↦ ∞ | v ∈ vals(σ)}      // init. map
27     if vars(σ) = {v} then            // atomic FM
28         P_σ[σ^tab(s[v])] ← 0
29     else                             // merge FM
30         P_{σ_L} ← handle-FM(σ_L, s^F, s)
31         P_{σ_R} ← handle-FM(σ_R, s^F, s)
32         forall v^L ↦ p^L ∈ P_{σ_L} do
33             forall v^R ↦ p^R ∈ P_{σ_R} do
34                 v = σ^tab(v^L, v^R)
35                 P_σ[v] ← min(P_σ[v], p^L + p^R)
36     return P_σ
```

if $\mathcal{L}_{ec}$ is empty, in which case algorithm execution continues with `FM-traverse-dec` (line 17), with partial state s defined on the variables of all exactly covered leaves of σ.

The function `FM-traverse-dec` traverses the FMs similarly to the explicit-state case. If σ is atomic, it looks up the value of the variable v in s and maps $\sigma^{\text{tab}}(s[v])$ to price 0 in line 28 (recall that the leaf state prices are taken into account by `enum-ec-leaves`). For merge FMs, `FM-traverse-dec` enters recursion to handle the two components σ_L and σ_R (lines 30 and 31). After returning, the function multiplies out the computed values v^L and v^R, looks up the corresponding value v in σ^{tab}, and stores in P_σ the minimal sum of the prices p^L and p^R (lines 32 to 35).

We next explain the functionality of the cache. The cache is needed to ensure polynomial-time computability for compliant FMs, but is not required for strongly-compliant FMs. There is a separate cache for every FM σ, all of which are being reset for every new decoupled state $s^{\mathcal{F}}$.

The cache for an FM σ is indexed by a set of facts over the variables V (line 6). These are exactly the variables whose corresponding atomic FMs are "below" σ, but which have been set in a merge FM "above" σ. These variables have a fixed assignment until `handle-FM` returns from σ, but influence the price mapping P_σ computed for σ. Thus, if P_σ has been computed for a specific $s[V]$, there is no point in recomputing that value again in a different call to σ for the same fact set $s[V]$. To understand why the cache is necessary at all, observe that the function `handle-FM` is called at least once for every descendant of σ_h. It is called exactly once for σ_h itself, but up to $|[s^{\mathcal{F}}]|$ times for every other FM. The number of calls is exactly $|[s^{\mathcal{F}}]|$ if all leaves are exactly covered in σ_h, i. e., $ec(\sigma_h) = \mathcal{L}$. This is because `enum-ec-leaves` will recursively enumerate all member states of $s^{\mathcal{F}}$, call `FM-traverse-dec` for each of them, which then calls `handle-FM` on all FMs. With the cache, the value is only computed once for every $s[V]$ and looked up in the cache for subsequent calls to `handle-FM` on σ, which can save exponentially many recomputations.

For compliant σ_h, $\mathcal{N}(\sigma) \leq 1$ for all descendants σ and thus, if $ec(\sigma) = \{L\}$ then $pc(\sigma) = \emptyset$ because $pc(\sigma_L) \cup pc(\sigma_R) = pc(\sigma) \cup ec(\sigma)$. Hence, there are no cache variables in σ, so its price mapping is computed only once and cached for subsequent calls. With compliant σ_h, for every leaf $L \in \mathcal{L}$ there exists a descendant σ of σ_h such that the price mapping P_σ computed in σ for the decoupled state $s^{\mathcal{F}}$ does only depend on the reached leaf states of L in $s^{\mathcal{F}}$. So there is no need to evaluate any FM with different combinations of reached leaf states across leaves, and the cache ensures that P_σ is only computed once, even though `handle-FM` might be called often on σ.

Proposition 6. *Algorithm 2 runs in time* $O(\|\sigma_h\| \cdot |s^{\mathcal{F}}|^{\mathcal{N}(\sigma_h)})$.

Proof sketch. The number of recursive calls to the `handle-FM` function for each FM node σ is bounded by the number of entries in the cache, which is bounded by $O(|s^{\mathcal{F}}|^{|pc(\sigma)|})$. Each call has a number of recursive calls that end up in `FM-traverse-dec`. The number of such calls is bounded by $O(|s^{\mathcal{F}}|^{|ec(\sigma)|})$. Therefore, the total number of recursive calls for each σ is bounded by $O(|s^{\mathcal{F}}|^{|ec(\sigma)|+|pc(\sigma)|}) = O(|s^{\mathcal{F}}|^{\mathcal{N}(\sigma_h)})$, as $pc(\sigma_L) \cup pc(\sigma_R) = pc(\sigma) \cup ec(\sigma)$.

The amount of work within each of those calls (e.g. lines 22–23 and 32–35), is bounded by the number of entries in σ. $\|\sigma_h\|$ is the total number of entries in all tables, so it also accounts for the fact that every evaluation of σ_h has to visit every descendant of σ_h. □

This shows that the runtime is exponential in the nesting in the general case, traversing the entire FM tree with every member state in the worst case. With compliant σ_h

($\mathcal{N}(\sigma_h) \leq 1$), the runtime is linear in the size of σ_h and the evaluated decoupled state $s^{\mathcal{F}}$. For strongly-compliant FMs (e.g., factorings with singleton leaf factors $L = \{v\}$), no cache is needed to achieve polynomial runtime. This is the case because every FM where $|ec(\sigma)| > 1$ is visited exactly once by the algorithm (as $ec(\sigma') = \emptyset$ for all its ancestors).

Symbolic Pattern Databases

Symbolic Pattern Databases precompute the heuristic stored in the form of a decision diagram. Here, we assume that the heuristic has already been encoded as an ADD, h_{ADD}. We start by defining compliant variable orderings. Then, we will show that this is a sufficient condition for evaluating the heuristic for a decoupled state in polynomial time.

Definition 7 (Compliant ADDs)**.** *A variable ordering is* compliant *with a factoring $\mathcal{F}$ if:*

$$\forall_{v<v'<v''}(\mathcal{F}(v) = \mathcal{F}(v'') \neq C) \implies \mathcal{F}(v') \in \{\mathcal{F}(v), C\}$$

In compliant ADDs, all variables of a leaf are placed next to each other, possibly interleaved by center variables. This naturally induces an order on the leaves. We now introduce two approaches to evaluate ADD heuristics.

Evaluation via ADD Operations

Our first approach leverages the ability of applying operations to ADDs for which existing libraries have highly optimized implementations (Somenzi 1997). As the heuristic is already encoded as an ADD h_{ADD}, we encode the pricing function of $s^{\mathcal{F}}$ as an ADD p_{ADD}, too. Then, $h_{\mathsf{ADD}} + p_{\mathsf{ADD}}$ simply returns an ADD that represents the mapping from explicit states to their price plus heuristic value. We minimize this sum by returning the lowest value of any terminal node.

It is well-known that the sum operation on ADDs runs in time $O(|p_{\mathsf{ADD}}||h_{\mathsf{ADD}}|)$ if both ADDs are represented under the same variable ordering. Thus, the procedure runs in polynomial time if and only if the size of p_{ADD}, when using the same variable ordering as h_{ADD}, is polynomial.

As expected from Theorem 1, for non-compliant variable orderings we have no guarantees on the size of p_{ADD}. If the variable ordering is compliant, however, we can bound p_{ADD} by the number of leaf states $S^{\mathcal{L}}$ and the number of possible different price sums for any combination of leaf states across leaves. Formally, these combinations are given by:

$$Prices(s^{\mathcal{F}}) = \{\sum_{L \in \mathcal{L}'} \mathsf{prices}(s^{\mathcal{F}})[s^L] \mid s^L \in S^L, \mathcal{L}' \subseteq \mathcal{L}\}$$

Hence, the ADD size for pricing functions is polynomial:

Proposition 8. *The ADD encoding p_{ADD} of the pricing function of $s^{\mathcal{F}}$ using a compliant variable ordering has size bounded by $O(|\mathcal{V}_{\mathsf{ADD}}||S^{\mathcal{L}}||Prices(s^{\mathcal{F}})|)$.*

Proof sketch. Ignoring the center variables, p_{ADD} is obtained from summing up the pricing function of each leaf, whose size is bounded by the number of leaf states. Note that the prices of different leaves are independent, and all variables of a leaf are contiguous in the variable ordering. So, after processing a leaf, the variable assignments are irrelevant (like in the example in Figure 2, where no information of x is relevant in the subtrees of y or z) and the only information passed to the next leaf is the price. Therefore, we only need to copy the same subtree once per possible price of the leaf states above in the variable ordering. Finally, the center variables have a fixed value, so they can increase the number of nodes at most by a factor of $|\mathcal{V}_{\mathsf{ADD}}|$. □

Note that $|Prices(s^{\mathcal{F}})|$ is polynomially bounded on unit-cost tasks, but not when the prices of leaf states are arbitrary. Overall, this is a viable method whenever the variable ordering of h_{ADD} is compliant with the variable ordering for domains where leaf states do not have very diverse prices.

Still, one can do better by exploiting the structure of the pricing function in a more direct way. One option is to use decision diagrams that guarantee a polynomial size for any pricing function (e.g. EVMDDs, Ciardo and Siminiceanu 2011). Instead, we next propose an algorithm that uses the pricing function directly, avoiding the need for compiling it into a decision-diagram representation.

Evaluation via ADD Traversal

Algorithm 3 performs the lookup by traversing a compliant ADD. The function `ADD-trav-dec`$(n, s_p, \mathcal{L}_{rel})$ computes $\min_{s \in [s^{\mathcal{F}}], s_p \subseteq s} \sum_{L \in \mathcal{L}_{rel}} \mathsf{prices}(s^{\mathcal{F}})[s[L]] + h_n(s)$, where h_n is the heuristic represented by node n of the ADD, s_p is a partial state specified for center variables and leaves whose leaf states are enumerated by an ancestor node, and $\mathcal{L}_{rel}$ are the relevant leaves whose prices should be added. The result is the desired value when called from line 3 with the root node of h_{ADD}, taking into account all leaves ($\mathcal{L}_{rel} = \mathcal{L}$), and not constraining any leaf variable ($s_p = s^C(s^{\mathcal{F}})$).

Lines 4-11 basically correspond to the lookup of an explicit state s_p, as described in the background. However, s_p is initially only defined over center variables. Therefore, the ADD traversal may reach a node n whose variable $n.v$ has undefined value in s_p, i.e., $n.v$ belongs to a leaf $L = \mathcal{F}(n.v)$ which has not appeared in the path from the root node to n. In that case (line 15), the algorithm enumerates all reached $s^L \in S^L$, extending s_p and doing a separate traversal for each to compute the minimal price plus heuristic value.

We avoid the enumeration of all member states by dynamic programming, storing a value $\mathsf{cache}[n] = \min_{s \in [s^{\mathcal{F}}]} \sum_{L \in \mathcal{L}_{\geq}(n)} \mathsf{prices}(s^{\mathcal{F}})[s[L]] + h_n(s)$, where $\mathcal{L}_{\geq}(n)$ corresponds to all leaves below $n.v$ in the variable ordering: $\mathcal{L}_{\geq}(n) := \{L \in \mathcal{L} \mid \exists v \in L \text{ s.t. } v \geq n.v\}$. This value can be cached because it is independent of s_p and $\mathcal{L}_{rel}$. As the variable ordering is compliant, the subtree beneath n is independent of all variables above $n.v$ and the value of h_n only depends on variables from $C \cup \mathcal{L}_{\geq}(n)$. Thus, the prices of leaf states from other relevant leaves $\mathcal{L}_{rel} \setminus \mathcal{L}_{\geq}(n)$ can be added separately in line 19.

The value of $\mathsf{cache}[n]$ is independent of s_p because s_p has always value $s^C(s^{\mathcal{F}})$ on center variables, and whenever $\mathsf{cache}[n]$ is used s_p has the same value (undefined) on all variables from leaves $\mathcal{L}_{\geq}(n)$. This is so because, in compliant variable orderings, it is always the case that $v < n.v \implies \mathcal{F}(v) \notin \mathcal{L}_{\geq}(n) \setminus \{\mathcal{F}(n.v)\}$ for all variables v and nodes n, so in all calls to `ADD-trav-dec` s_p is undefined on $\mathcal{L}_{\geq}(n) \setminus \{\mathcal{F}(n.v)\}$. Finally, $\mathsf{cache}[n]$ is only used

Algorithm 3: Evaluating ADDs on decoupled states.

Input: Decoupled state $s^{\mathcal{F}}$, ADD h_{ADD} compliant with $\mathcal{F}$
Output: $h_{\mathsf{ADD}^{\mathcal{F}}}(s^{\mathcal{F}})$

1 **def** `ADD-eval(`$h_{\mathsf{ADD}}, s^{\mathcal{F}}$`)`**:**
2 $s_p \leftarrow s^C(s^{\mathcal{F}})$ `// `s_p` is a partial state`
3 **return** `ADD-trav-dec(`h_{ADD}.root-node, s_p, $\mathcal{L}$`)`
4 **def** `ADD-trav-dec(`n, s_p, $\mathcal{L}_{rel}$`)`**:**
5 **if** is_terminal(n) **then**
6 cache[n] $\leftarrow n.value$
7 **else if** $s_p[n.v]$ *is defined* **then**
8 **if** $s_p[n.v] = 0$ **then**
9 **return** `ADD-trav-dec(`n.0-child, s_p, $\mathcal{L}_{rel}$`)`
10 **else**
11 **return** `ADD-trav-dec(`n.1-child, s_p, $\mathcal{L}_{rel}$`)`
12 **else if** n *is not in* cache **then**
13 $L \leftarrow \mathcal{F}(n.v)$
14 cache[n] $\leftarrow \infty$
15 **for** $s^L \in S^L$: prices($s^{\mathcal{F}}$)[s^L] $< \infty$ **do**
16 $s_p[v] \leftarrow s^L[v]$ for all $v \in L$
17 $h \leftarrow$`ADD-trav-dec(`n, s_p, $\mathcal{L}_{\geq}(n) \setminus \{L\}$`)`
18 cache[n] $\leftarrow \min($cache[n], prices($s^{\mathcal{F}}$)[s^L] $+ h)$
19 **return** cache[n] $+ \sum_{L \in \mathcal{L}_{rel} \setminus \mathcal{L}_{\geq}(n)} \min_{s^L \in S^L}$ prices($s^{\mathcal{F}}$)[s^L]

whenever s_p is undefined for $\mathcal{F}(n.v)$ as otherwise $s_p[n.v]$ is defined in line 7.

Proposition 9. *Algorithm 3 runs in time* $O(|h_{\mathsf{ADD}}||\mathcal{V}_{\mathsf{ADD}}||S^{\mathcal{L}}||\mathcal{L}|)$.

Proof sketch. The main source of calls to the `ADD-trav-dec` function is the loop in line 15. However, as the results are stored in a cache, the number of such calls is bounded by $|h_{\mathsf{ADD}}| \cdot \max_{L \in \mathcal{L}} |S^L|$. All other sources of calls to the `ADD-trav-dec` function have at most a single recursive call. Therefore, this can increase the total number of calls at most by a factor of $|\mathcal{V}_{\mathsf{ADD}}|$, the height of the tree. Each call of the `ADD-trav-dec` function takes time $O(|\mathcal{L}|)$ due to line 19, and the loop in line 15 is already accounted for when considering the number of recursive calls. □

The algorithm could be adapted to support non-compliant orders simply by changing the key of the cache to be dependent on the value of s on leaves in $\mathcal{L}_{\geq}(n)$. Of course, the running time would not be polynomially-bounded anymore, as there are exponentially many possible such assignments.

Experiments

We evaluate our algorithms on the optimal suite of the Autoscale benchmark set (Torralba, Seipp, and Sievers 2021). Our implementation is based on the decoupled search planner by Gnad and Hoffmann (2018), which builds on the Fast Downward framework (Helmert 2006). We conducted experiments on a cluster of Intel Xeon Gold 6130 CPUs using Downward Lab (Seipp et al. 2017), with runtime and memory limits of 30min and 3.5GiB per run.

As a factoring strategy for decoupled search, we pick the best configuration reported by Gnad, Torralba, and Fišer (2022), called bM80s. For merge-and-shrink (M&S) heuristics, we report results for the sbMIASM merge strategy (Sievers, Wehrle, and Helmert 2016), using bisimulation shrinking (Nissim, Hoffmann, and Helmert 2011), and exact label reduction (Sievers, Wehrle, and Helmert 2014). We limit the size of FM tables to 50.000 entries. We experimented with other strategies, but found this to perform best in terms of total coverage, and show the same trends. For a better analysis of the differences between compliant and non-compliant FMs, we use a random linear merge strategy in one of our plots, because sbMIASM often produces strongly compliant FMs. For symbolic pattern-database heuristics (sPDB), abstractions are computed with Gamer PDBs (Kissmann and Edelkamp 2011), using the symbolic search enhancements by Torralba et al. (2017) and the CUDD 3.0.0 library (Somenzi 1997). We also experimented with a full perimeter (Kissmann and Edelkamp 2011), but found Gamer PDBs to perform better and show the same picture with respect to how evaluating decoupled states affects the performance. We use a runtime limit of 15min to compute heuristics.

For both types of heuristics, we use two variants to compute the abstraction: a general computation that does not take into account the factoring (**g**), and a compliant variant (**c**) that implies polynomial-time computability, and allows us to use the ADD traversal for sPDBs. For **c**, we restrict the merge strategy to first fully merge each leaf factor L before merging any other variable into an intermediate factor that partially includes L. For sPDBs, we force all variables in a leaf to be contiguous in the variable order. In both cases, this is slightly more restrictive than the definition of strongly compliant FMs and compliant variable orders, since center variables are never interleaved with leaf variables. However, the changes had little effect on the final heuristic, so we did not further investigate different compliant strategies.

In Table 1, we show coverage results (number of solved instances) for all algorithms, distinguishing between explicit-state (Expl.) and decoupled search. Best coverage in a domain is highlighted in bold. First, observe that indeed there is almost no difference between the general and compliant data structures in explicit-state search, so the restriction does not have a large impact on heuristic quality. This is confirmed when comparing the sizes of the search spaces, which we cannot show due to space limitations.

We consider two evaluation algorithms of decoupled states for M&S: enumeration of the all member states (Enum.), which computes the explicit decoupled heuristic, and the dynamic programming approach from Algorithm 2 (DP). The explicit-state enumeration generally performs quite bad and is not competitive with explicit-state search. There is also only little difference between the **g** and **c** variants, confirming that the heuristic quality is very similar. The dynamic programming algorithm clearly benefits from enforcing FMs to be strongly compliant. On many domains, though, we do not see a major difference in terms of coverage, because sbMIASM, by chance, produces many (strongly) compliant FMs. We further discuss this below.

	Merge-and-Shrink						Symbolic PDBs						
	Ex. S.		Dec. Search				Ex. S.		Decoupled Search				
			Enum.		DP				Enum.		AOps		DP
Domain	**g**	**c**	**g**	**c**	**g**	**c**	**g**	**c**	**g**	**c**	**g**	**c**	**c**
Agri 2	0	0	0	0	0	0	0	0	0	0	0	0	0
Child 30	**9**	**9**	**9**	**9**	**9**	**9**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
Data 30	**10**	**10**	9	9	9	9	**13**	**13**	**13**	**13**	**13**	**13**	**13**
Depo 30	**13**	**13**	**13**	**13**	**13**	**13**	**12**	**12**	**12**	**12**	**12**	11	**12**
Drive 30	**12**	**12**	5	5	10	10	**6**	**6**	5	5	5	5	5
Elev 30	8	8	**10**	**10**	9	9	9	9	**10**	**10**	9	9	9
Floor 30	**6**	**6**	**6**	**6**	**6**	**6**	**7**	**7**	5	5	5	5	5
Grid 30	**14**	**14**	**14**	**14**	**14**	**14**	**27**	**27**	26	25	25	**27**	26
Hiki 15	**7**	**7**	5	5	5	5	**6**	**6**	**6**	**6**	**6**	**6**	**6**
Logi 30	**14**	**14**	13	13	**14**	**14**	**15**	**15**	13	13	14	14	14
Mico 30	**5**	3	**5**	**5**	**5**	**5**	**9**	8	7	7	**9**	8	8
NoM 30	11	11	7	7	**24**	**24**	7	7	7	7	15	15	**16**
Open 30	**6**	**6**	4	4	4	4	**6**	**6**	4	4	4	4	4
Orga 15	**2**	**2**	**2**	**2**	1	**2**	**5**	**5**	3	4	4	4	4
ParcP 27	**20**	**20**	7	7	12	**20**	9	9	7	7	6	8	**27**
Path 30	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	6	6	6	6	6
Rove 30	2	2	4	**4**	**4**	**4**	**13**	12	**13**	12	**13**	12	12
Sate 30	**21**	**21**	14	14	20	20	18	**19**	12	12	18	**19**	**19**
Scan 2	0	0	0	0	0	0	0	0	0	0	0	0	0
Tidy 22	**6**	**6**	**6**	**6**	**6**	**6**	**7**	**7**	6	6	6	6	6
TPP 30	4	4	5	4	**17**	**17**	2	3	5	5	7	14	**27**
Tran 30	**14**	**14**	11	11	11	11	**16**	**16**	15	15	15	15	**16**
Wood 28	4	5	5	5	5	**7**	8	7	7	7	**9**	8	**9**
Zeno 30	14	14	13	13	**16**	**16**	11	10	12	11	15	15	**16**
$\sum$ 621	211	210	176	175	223	**234**	226	224	205	203	227	235	**271**

Table 1: The table shows coverage (number of solved instances) on our benchmark set. Best coverage is highlighted in bold face, separately for M&S and sPDBs.

For sPDBs we consider the evaluation based on ADD operations (ADDOps, or AOps in short), and the dynamic-programming variant from Algorithm 3 (DP). The picture is similar as for M&S: there is little difference when enforcing compliant orders (**g** vs. **c**) in the explicit search and state-enumeration configurations, but compliant orders are beneficial for ADDOps. The DP algorithm leads to a significant gain in coverage, showing the advantage of the specialized algorithm.

We further illustrate the advantage of the optimized algorithms in the runtime plots in Figure 3. The plots show per-instance data comparing only the search time, so excluding the construction of the heuristic. The plot in the top compares the ADDOps algorithm using general (x-axis) and compliant variable orders (y-axis), showing a significant speed-up for compliant orders. For M&S with sbMIASM (middle plot), the two variants behave almost identically. Looking at the FMs produced by the strategy shows that most have very low nestedness and many of them are actually strongly compliant. Therefore, we show the same evaluation for a random linear merge strategy (bottom plot), which at least sometimes produces FMs with higher nestedness. This is indicated by the different point shapes in that plot. We observe that for instances in which the FMs are not strongly compliant anyway, forcing them to be so indeed leads to a speed-up of up to several orders of magnitude.

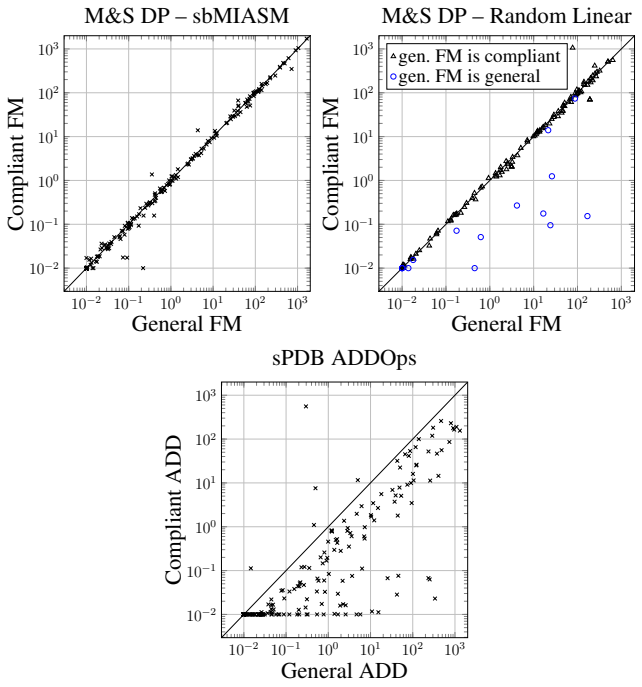

Figure 3: Per-instance search-time plot for M&S with sbMIASM (top left) and random linear merging (top right), and ADDOps (bottom), comparing general to compliant FMs, respectively ADDs.

We confirmed that this is actually due to an optimized computation because the number of decoupled state evaluations varies only very little.

Conclusion

In this paper, we introduced merge-and-shrink heuristics and symbolic pattern databases for decoupled search. We showed that evaluating these heuristics on decoupled states is **NP**-hard in general, i.e., when the compact representations of the heuristics and decoupled states do not comply. However, when restricting the FMs underlying merge-and-shrink heuristics or the ADDs underlying symbolic PDBs so that they are compliant with the factoring used for decoupled search, heuristic evaluation takes polynomial time. Our experiments demonstrate that these compliant heuristics solve significantly more tasks than their explicit counterparts.

In future work, we want to investigate how multiple abstraction heuristics can be combined efficiently in a better way, possibly using cost partitioning. Another interesting direction is the combination of merge-and-shrink heuristics tailored to detecting unsolvability (Hoffmann, Kissmann, and Torralba 2014) with decoupled search to prove planning tasks unsolvable. Finally, we want to investigate how state compression methods beyond those studied in this work relate and can be made compatible to each other.

Acknowledgements

Daniel Gnad and Silvan Sievers were partially supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation programme under grant agreement no. 952215. Daniel Gnad was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Silvan Sievers has received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639). The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreements no. 2022-06725 and no. 2018-05973.

References

Amir, E.; and Engelhardt, B. 2003. Factored Planning. In *Proc. IJCAI 2003*, 929–935.

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial Pattern Databases. In *Proc. SARA 2007*, 20–34.

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1997. Algebraic Decision Diagrams and Their Applications. *Formal Methods in System Design*, 10(2–3): 171–206.

Brafman, R. I.; and Domshlak, C. 2006. Factored Planning: How, When and When Not. In *Proc. AAAI 2006*, 809–814.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8): 677–691.

Ciardo, G.; and Siminiceanu, R. 2011. Using Edge-Valued Decision Diagrams for Symbolic Generation of Shortest Paths. In *Proc. FMCAD 2002*, 256–273.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proc. ECP 2001*, 84–90.

Edelkamp, S. 2002. Symbolic Pattern Databases in Heuristic Search Planning. In *Proc. AIPS 2002*, 274–283.

Edelkamp, S.; and Kissmann, P. 2008. Limits and Possibilities of BDDs in State Space Search. In *Proc. AAAI 2008*, 1452–1453.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012. Symbolic A^* Search with Pattern Databases and the Merge-and-Shrink Abstraction. In *Proc. ECAI 2012*, 306–311.

Fabre, E.; Jezequel, L.; Haslum, P.; and Thiébaux, S. 2010. Cost-Optimal Factored Planning: Promises and Pitfalls. In *Proc. ICAPS 2010*, 65–72.

Franco, S.; and Torralba, Á. 2019. Interleaving Search and Heuristic Improvement. In *Proc. SoCS 2019*, 130–134.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gnad, D.; and Hoffmann, J. 2018. Star-Topology Decoupled State Space Search. *AIJ*, 257: 24–60.

Gnad, D.; Sievers, S.; and Torralba, A. 2023. Supplementary Material, Code, and Experimental Data for the ICAPS 2023 paper "Efficient Evaluation of Large Abstractions for Decoupled Search: Merge-and-Shrink and Symbolic Pattern Databases". https://doi.org/10.5281/zenodo.7741805.

Gnad, D.; Torralba, Á.; and Fišer, D. 2022. Beyond Stars - Generalized Topologies for Decoupled Search. In *Proc. ICAPS 2022*, 110–118.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *JACM*, 61(3): 16:1–63.

Helmert, M.; Röger, G.; and Sievers, S. 2015. On the Expressive Power of Non-Linear Merge-and-Shrink Representations. In *Proc. ICAPS 2015*, 106–114.

Hoffmann, J.; Kissmann, P.; and Torralba, Á. 2014. "Distance"? Who Cares? Tailoring Merge-and-Shrink Heuristics to Detect Unsolvability. In *Proc. ECAI 2014*, 441–446.

Kissmann, P.; and Edelkamp, S. 2011. Improving Cost-Optimal Domain-Independent Symbolic Planning. In *Proc. AAAI 2011*, 992–997.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning. In *Proc. IJCAI 2011*, 1983–1990.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Seipp, J. 2021. Online Saturated Cost Partitioning for Classical Planning. In *Proc. ICAPS 2021*, 317–321.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *JAIR*, 62: 535–577.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *JAIR*, 67: 129–167.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Sievers, S.; Gnad, D.; and Torralba, Á. 2022. Additive Pattern Databases for Decoupled Search. In *Proc. SoCS 2022*, 180–189.

Sievers, S.; and Helmert, M. 2021. Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems. *JAIR*, 71: 781–883.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized Label Reduction for Merge-and-Shrink Heuristics. In *Proc. AAAI 2014*, 2358–2366.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An Analysis of Merge Strategies for Merge-and-Shrink Heuristics. In *Proc. ICAPS 2016*, 294–298.

Somenzi, F. 1997. CUDD: CU decision diagram package. Technical report, University of Colorado at Boulder.

Torralba, Á. 2015. *Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning*. Ph.D. thesis, Universidad Carlos III de Madrid.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient Symbolic Search for Cost-optimal Planning. *AIJ*, 242: 52–79.

Torralba, Á.; Linares López, C.; and Borrajo, D. 2018. Symbolic perimeter abstraction heuristics for cost-optimal planning. *AIJ*, 259: 1–31.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In *Proc. ICAPS 2021*, 376–384.