Downward Path Preserving State Space Abstractions

Sandra Zilles ZILLES@CS.UALBERTA.CA
Marcel A. Ball MARCEL@CS.UALBERTA.CA
Robert C. Holte HOLTE@CS.UALBERTA.CA
University of Alberta, Department of Computing Science
Edmonton, Alberta, Canada T6G 2H8

Abstract

Abstraction is a popular technique for speeding up planning and search. A problem that often arises in using abstraction is the generation of abstract states, called spurious states, from which the goal state is reachable in the abstract space but for which there is no corresponding state in the original space from which the goal state can be reached. The experiments in this paper demonstrate that this problem may arise even when standard abstraction methods are applied to benchmark planning problem domains: spurious states cause the pattern databases representing the heuristics to be excessively large and slow down planning and search by reducing the heuristic values. Known automated techniques to get rid of a large portion of spurious states turn out to avoid the memory problem, while at the same time not avoiding the problem of bad heuristic quality.

The main contribution of this paper is theoretical. We formally define a characteristic property—the downward path preserving property (DPP)—that guarantees an abstraction will not contain spurious states. How this property can be achieved is studied both for techniques focussing on automated domain-independent abstraction and for techniques focussing on domain-specific abstraction.

We analyze the computational complexity of *(i)* testing the downward path preserving property for a given state space and abstraction and of *(ii)* determining whether this property is achievable at all for a given state space. Strong hardness results show a close connection between these decision problems and the plan existence problem in typical planning settings including SAS$^+$ and propositional STRIPS.

On the positive side, we identify formal conditions under which finding downward path preserving abstractions is provably tractable and show that some popular heuristic search and planning domains have an encoding that matches these conditions. This includes a new encoding of the Blocks World domain, for which DPP abstractions can be easily defined.

1. Introduction

Abstraction is a technique for speeding up planning and search that has been successfully applied in various planning domains. The idea is to build an "abstract" version of a given state space that contains many fewer states so that planning or search in the abstract space is very rapid. This can be exploited in two ways: *(i)* by constructing a plan between two states in the original space by refining abstract plans found between the corresponding abstract states or *(ii)* by defining a heuristic function to guide planning and search by using actual distances in the abstract space as estimates of distances in the original space.

Domain-independent approaches focus on automated abstraction techniques that can be applied to basically any state space representable in standard planning representation languages such as SAS$^+$ or STRIPS. *Domain-specific* approaches typically handcraft abstractions for given state spaces by exploiting symmetries or any other knowledge of the given problem domain.

In principle, an abstraction can be any kind of mapping of the original state space to a smaller state space, as long as it does not remove edges between states.[1] However, this paper addresses two types of abstraction: projection and domain abstraction. In both cases, we assume that states in the original state space are represented as attribute-value pairs over a fixed set of attributes (variables) that range over certain value sets. The two types of abstractions we consider have the following characteristics.

- When defining a *projection*, abstraction means to ignore some of the attributes (*i.e.*, to ignore some of the components of the vector, and thus formally to project to a lower-dimensional space).

- Alternatively, for *domain abstraction*, abstraction means to reduce the value sets of attributes by identifying certain values with each other.

A potential problem with abstraction—in either setting—is that it can introduce what we call "spurious states"; their deleterious effects and ways to avoid them are the focus of this paper. Given a goal state g in the original state space, a spurious state is an abstract state that has a path to the abstraction of g but is not the abstract image of any original state having a path to g. Spurious states cause several difficulties. First, an abstract plan containing spurious states will, by definition, be unrefinable, and if the length of the abstract plan is being used as a heuristic it will often be overly optimistic because of "short-cuts" created in the abstract space by spurious states. Unrefinable abstract plans and low heuristic values can drastically slow down planning and search. Secondly, if abstract distances are being stored in memory, as is done with pattern databases (PDBs), introduced by Culberson and Schaeffer (1996, 1998), spurious states will increase the memory requirements, sometimes dramatically.

The existence and damaging effects of spurious states are illustrated in a small experiment in Section 2, which applies a standard abstraction method to a variety of benchmark planning problem domains. We also show that obvious approaches to filtering out the spurious states do not entirely succeed.

This motivates the main contribution of this paper, which is a formal analysis of the problem of avoiding spurious states in abstraction.

Our results are meaningful for both domain-independent and domain-dependent approaches. To this end we first of all introduce a common language for representing planning domains as well as search domains. In this formal framework we introduce what we call the "Downward Path Preserving" (DPP) property. This characteristic (*i.e.*, necessary and sufficient) property specifies state space abstractions that do not contain spurious states at all.

Intractability results. One of our main contributions is a computational complexity analysis of two closely related problems, namely (A) to determine whether or not a given abstraction has the DPP property, and (B) to determine whether or not a given state space possesses a DPP abstraction at all.

1. Formal definitions of abstractions in general and specific types of abstractions will be given in Section 3.2.

Our complexity analyses show that both problems are, in general, hard to solve and that efficient general algorithms to produce DPP abstractions are thus unlikely to exist.

This does not mean that it is hopeless to try to automatically minimize the number of spurious states when abstracting a given state space. It just means that it is hard to avoid spurious states completely. However, our experimental result in Section 2 illustrate that potentially the spurious states that have the most damaging effect on the heuristic values are exactly those that are hard to filter out with existing automated methods. In particular, when using abstractions to define high-quality heuristics, it is not enough to avoid most of the spurious states; ideally one would have to avoid them all.

Tractability results. The negative results on the computational complexity are mitigated by tractability results for special cases—our second main contribution. We identify simple formal conditions on state spaces that allow for easily constructing DPP abstractions. Some typical problem domains turn out to be representable in a way that these conditions apply. However, some of them also have natural encodings that do not match the formal conditions of our theorems, and that, moreover, in fact do not allow for DPP abstractions at all. This shows that the way a problem domain is encoded is crucial for the design of DPP abstractions. Given two equivalent representations, one might immediately yield DPP abstractions, whereas the other might prevent DPP abstractions.

Our formal conditions allowing for a construction of DPP abstractions can be seen as design guidelines for representing state spaces.

Among the examples that illustrate the effect of varying encodings of problem domains, we show that the Blocks World benchmark domain possesses a simple encoding, which, unlike all previous encodings of this domain, allows for immediate DPP abstractions. By "immediate" we here mean that these abstractions can be defined without applying filtering techniques, for which it is hard to guarantee that spurious states are avoided. Such filtering techniques, as used for instance by Haslum, Bonet, and Geffner (2005), typically constrain the abstraction by not generating abstract states that violate the mutual exclusion of automatically detected pairs of mutually exclusive atoms, *cf.* (Bonet & Geffner, 2001).

2. DPP abstractions in practice

We ran experiments on a modification of the Blocks World domain with a fixed number of named table positions, each of which can hold one stack of blocks. There are 4 operators to move a block: *(i)* from on top of one block to on top of another, *(ii)* from on top of a block to a specific position on the table, *(iii)* from a table position to on top of a block, and *(iv)* from one table position to another table position. Our results are based on a domain with 7 blocks and 4 table positions, which has 604,800 states in the original state space that are reachable from the start state. Because of the symmetry of the domain, the start state is also reachable from any of these 604,800 states. In particular, if one of these is chosen as a goal state, then the same 604,800 states are reachable from the goal state as well.

We used a standard method for abstracting planning domains, as introduced by Edelkamp (2001), and defined the abstract space by inducing the abstract operators, abstracting the goal state (all of the blocks stacked in order on the fourth table position with block 1 at the top), and then storing all abstract states that can reach the image of the goal state in a PDB. Our abstraction kept the values of the variables representing the state of the four table positions, as well as what is on top of blocks 5, 6 and 7; it ignored what is on top of blocks 1, 2, 3 and 4.

The 604,800 states that the goal state was originally reachable from were mapped to 89,400 states; however, there were actually 1,310,720 abstract states stored in the PDB—more than twice as many as in the original state space.

2.1 Cause and consequence of violating the DPP property

The reason we have more abstract states in the PDB than our mapping suggests is that in the abstract space, due to ignoring certain variables, there are abstract states s_{ab} for which the following holds.

- The abstract goal state g_{ab} is reachable from s_{ab}.
- The original goal state g is not reachable from any state s in the original space that is abstracted to s_{ab}.

This means the abstraction introduces paths into the abstract space for which no corresponding paths exist in the original space. In other words, the abstraction is not downward path preserving (DPP). States like s_{ab} that are "newly" connected to the abstract goal are called "spurious states". The occurrence of such states is known to the heuristic search and planning community, and reported for instance by Holte and Hernádvölgyi (2004) and by Haslum, Botea, Helmert, Bonet, and Koenig (2007). Techniques like constrained abstraction, as proposed by Haslum *et al.* (2005), have a positive effect in that they can reduce the number of spurious states drastically. However, there are no practical methods known so far to avoid spurious states completely; neither is there any formal research on the complexity of the problem of avoiding them.

The following example from our experiments illustrates how spurious states can be generated.

> *The predicates encoding what is on the four table positions* `P1` *through* `P4`*, as well as what is on top of blocks* `B5`*,* `B6` *and* `B7` *are kept; those encoding what is on blocks* `B1` *through* `B4` *are discarded.*
>
> *Consider the following abstract state α:*
>
> ```
> (clear P1), (clear P2), (clear P3),
> (on B7 P4), (on B6 B7), (on B5 B6), (on B4 B5)
> ```
>
> *Next consider the action* `(MoveBlock B4 B3 B5)`
>
> *Preconditions:* `(clear B4), (clear B3), (on B4 B5)`
> *Effects:* `(on B4 B3), (clear B5), not(on B4 B5), not(clear B3)`
>
> *This action abstracts to* `(MoveBlock B4 B3 B5)`
>
> *Preconditions:* `(on B4 B5)`
> *Effects:* `(clear B5), not(on B4 B5)`

> *The abstract version of this operator would apply to the abstract state α above resulting in the following abstract state β:*
>
> ```
> (clear P1), (clear P2), (clear P3),
> (on B7 P4), (on B6 B7), (on B5 B6), (clear B5)
> ```
>
> *No state in the pre-image of α can reach a state that maps to β, since the blocks* `B1`, `B2`, `B3`, *and* `B4` *cannot have any position (they can't be on* `P1`, `P2`, `P3`, `B5` *as those are all known to be clear).*

One of the negative effects of abstractions that are not DPP is the (excessive) additional memory needed to store the abstract space, if PDBs are being used.

A second and perhaps worse consequence is that a violation of the DPP property may slow down planning and search methods that use the abstractions. For example, heuristic planning and search methods use abstraction to define heuristics—if the exact distances in the abstract space are known, they can be used as heuristic values for distances between pre-image states in the original state space (Pearl, 1984; Prieditis, 1993; McDermott, 1999; Haslum & Geffner, 2000; Bonet & Geffner, 2001). Here non-DPP abstractions will produce overly optimistic heuristic values if the spurious states create a shortcut in the abstract space.

An initial idea is to filter out all spurious states before actually computing the PDB heuristic and thus to get a DPP abstraction. In our example here, where the original state space contains only about 600,000 states, we could do this enumeratively: we exhaustively enumerated the original state space to get a list of all spurious states. In general, of course, such a method is impractical. In our experiments we approached the problem of removing spurious states in steps.

1. We removed spurious states that violate simple pairwise mutual exclusions (i.e., states in which two binary variables are true although this never happens for original states from which the goal state is reachable); note that in general there are no methods known for finding *all* such mutual exclusive variables, but we removed those we could detect using a standard method (Bonet & Geffner, 2001).

2. We hand-crafted additional rules for filtering states.

3. We did a complete exploration of the original state space to remove the last spurious states.

Table 1 and Figure 1 show how the number of abstract states, the average heuristic value of the 89,400 non-spurious abstract states, and the distribution of their heuristic values differ between

- the PDB built without any filtering of spurious states ("No Filtering"),

- the PDB after elimination of spurious states violating automatically detected pairwise mutex conditions ("Pairwise Filtering"),

- the PDB after additional elimination of spurious states violating manually encoded triple mutex conditions[2] ("Special Filtering"), and
- the PDB after elimination of *all* spurious states by hand ("Complete Filtering").

Filtering Mode	# Abstract States	Average H.
No Filtering	1,310,720	7.10012
Pairwise Filtering	90,941	7.10012
Special Filtering	90,000	7.16217
Complete Filtering	89,400	7.21264

Table 1: Number of abstract states and average heuristic value of a PDB for the 7-Blocks World with 4 named positions. We show the average heuristic value only for the 89,400 non-spurious states (the abstractions of the original states from which the goal is reachable).

Table 1 shows that the problem of additional memory requirements caused by spurious states is basically avoided by automatically constructed "pairwise mutex filters"; they reduce the number of abstract states from 1,310,720 to 90,941, which is close to the actual 89,400 non-spurious abstract states. The handcrafted special mutex filters remove another 941 spurious states, resulting in an abstract space of 90,000 states with only 600 remaining spurious states.

However, after applying the pairwise mutex filters, the average heuristic value of the 89,400 "true" abstract states does not change at all when compared to the unfiltered PDB—in both cases it is about 7.10. After applying the handcrafted mutex filters, this value rises to 7.16; the average heuristic value in the completely filtered abstract space is 7.21.

It might seem that the distribution of heuristic values is only slightly different. However, this small change in the distribution has a significant effect on the speed of search. Using the unfiltered PDB, IDA* expands 361,861 nodes (on average, over 100 randomly generated start states with an average solution length of 10.95), whereas with the filtered PDB it expands only 287,954 nodes, a reduction of 26%, see Table 2.

PDB Filtering Mode	Avg. Nodes Expanded
Pairwise Filtering	361,861
Special Filtering	319,325
Complete Filtering	287,954

Table 2: Number of nodes expanded by IDA* with different PDB heuristics as in Table 1.

Consequently, simple constrained abstraction techniques, which would result in the 90,941 abstract states in this example, can avoid the problem of memory requirements

2. Such conditions are, for instance, "if block b_1 is on block b_2 and block b_2 is on block b_3, then block b_3 cannot be on block b_1".

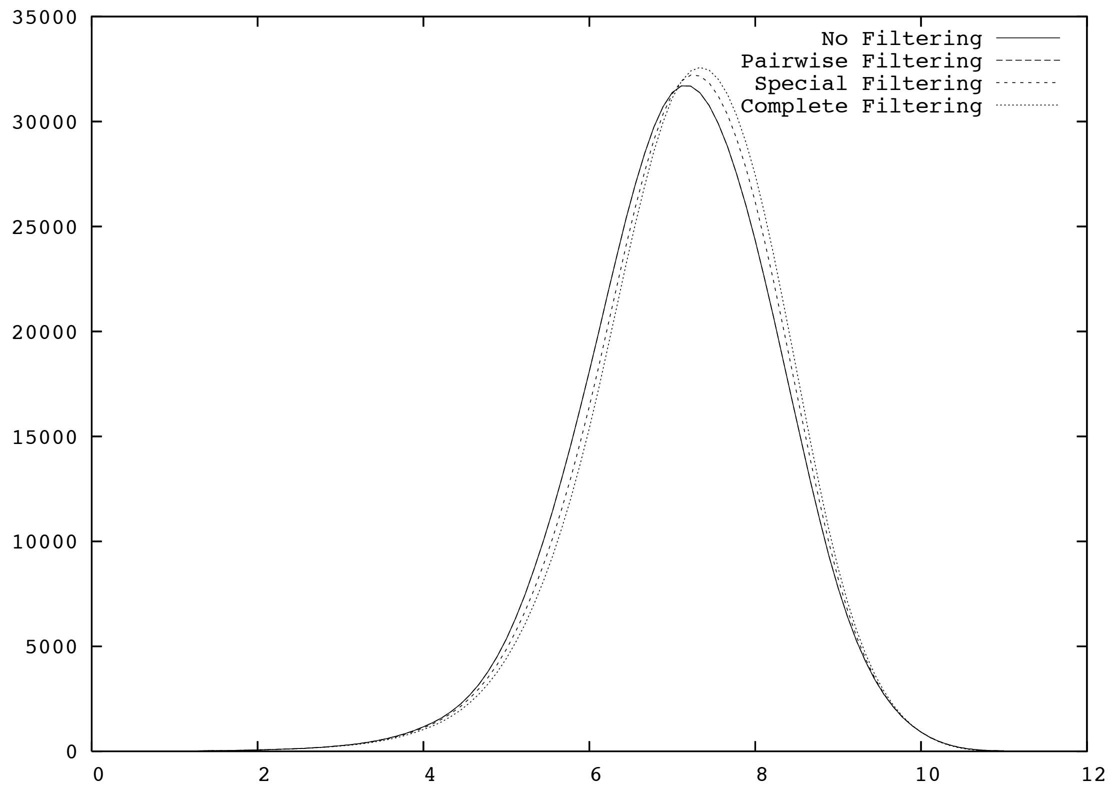

Figure 1: Plot of the heuristic distributions as in Table 1. The horizontal axis represents heuristic values from 0 to 12. The vertical axis represents the number of abstract states (out of 89,400). The plot for "Pairwise Filtering" coincides with the plot for "No Filtering".

caused by spurious states, while at the same time not avoiding the problem of bad heuristic quality.

Our observations suggest that the very few spurious states that remain after applying standard filtering techniques might be exactly those that have a negative effect on the quality of the heuristic. In other words, standard techniques here help to get rid of only those spurious states that do *not* influence the heuristic negatively. Thus, for the design of high-quality heuristics, it is not enough to try to avoid *most* spurious states—good quality heuristics can not be guaranteed if not *all* spurious states are avoided.

We did similar experiments for other standard PDDL planning domains—Blocks World in several variations, Depots, Satellites, and Rovers—and PDDL implementations of the 15-Puzzle and 4-peg Towers-of-Hanoi, *cf.* (Ball, 2009). In all cases the abstractions produced were not DPP. In some cases, pairwise mutex filters got rid of all spurious states, but in other cases, *e.g.*, Depots, they did not. Since those experiments were on a larger scale, we do not know whether the phenomenon that only the remaining spurious states actually had an effect on the quality of the heuristics repeats here.

2.2 Achieving the DPP property

Motivated by these experimental results we study ways to deal with the problem of finding DPP abstractions in general.

Ways to automatically construct DPP abstractions in general could be

- Designing an abstraction and an algorithm that automatically detects whether or not an abstract state is the image of a state from which the goal state is reachable in the original state space. Such an algorithm might have to check reachability (i.e., plan existence) in the original space—usually an intractable problem. This is discussed formally below in the decision problem REACHABLE.

- Automatically generating abstractions and designing an algorithm that determines for each of them whether or not it satisfies the DPP property. Such an algorithm would have to solve the IS-DPP problem studied below.

- Designing an algorithm that determines for any given state space whether or not a DPP abstraction exists at all (in order to support from-scratch construction of DPP abstractions). This algorithm would solve the problem we call EXIST-DPP below.

These questions are the subject of the analysis in Section 4.1.

A way to construct DPP abstractions for specific problem domains could be

- Finding an encoding of the state space that satisfies conditions that are known to yield DPP abstractions with a particular abstraction method.

This is the subject of the analysis in Section 4.2.

3. Formal model of downward path preserving state space abstractions

In this section we will formally define state spaces, abstractions, and the DPP property. This will give us a unified representation language for planning-style problem domains and search-style problem domains, so that we will be able to formally analyze DPP and non-DPP projections and domain abstractions.

The representation language will be used for studying the problem of automatically generating abstractions (Section 4.1) and of appropriately representing state spaces in order to avoid non-DPP abstractions (Section 4.2).

The following simple notations will be used for this purpose.

If A is a finite set then $|A|$ denotes the cardinality of A. The symbol $\emptyset$ is used for denoting the empty set.

If A, B are any two sets and $\psi : A \to B$ a mapping then $\psi(A)$ denotes the set $\{\psi(a) \mid a \in A\} \subseteq B$.

Let Σ, Γ be finite alphabets, $n \in \mathbb{N}$, and $\psi : \Sigma^n \to \Gamma^n$. ψ is called a string homomorphism if there is a mapping $\psi_0 : \Sigma \to \Gamma$ such that $\psi(\sigma_1, \dots, \sigma_n) = (\psi_0(\sigma_1), \dots, \psi_0(\sigma_n))$ for all $\sigma_1, \dots, \sigma_n \in \Sigma$.

3.1 State space representation

Usually a state space is defined as a weighted directed graph, but since none of the issues discussed in this paper are affected by edge weights we ignore them in our definitions.

Definition 1 *A state space is a triple $\mathcal{S} = (\Sigma, n, \Pi)$ where Σ is a finite alphabet, $n \in \mathbb{N}$, and $\Pi \subseteq \Sigma^n \times \Sigma^n$. Every $s \in \Sigma^n$ is called a state and every pair $(s, s') \in \Pi$ is called an edge from state s to state s'.*

Definition 2 *Let $\mathcal{S} = (\Sigma, n, \Pi)$ be a state space, $s, s' \in \Sigma^n$ states. s' is reachable from s (in $\mathcal{S}$) if there is a sequence $\pi = (s_1, s_2, \dots, s_z) \in (\Sigma^n)^+$ such that $s_1 = s$, $s_z = s'$, and $(s_i, s_{i+1}) \in \Pi$ for all $i \in \{1, \dots, z-1\}$. Define*

$$\Delta(s, \mathcal{S}) = \{s' \in \Sigma^n \mid s' \textit{ is reachable from } s \textit{ in } \mathcal{S}\}.$$

Note that we assume that the set of all states in a state space $\mathcal{S}$ is always the full set of n-tuples over Σ. In contrast to that it is also common to define a state space via a seed state $s^* \in \Sigma^n$ and all states reachable from that state (edges and thus reachability might be defined in the form of operators). In such a definition, the set of states in $\mathcal{S}$ would only be one connected component of Σ^n. For technical and practical reasons these different perspectives on state spaces do not play a role in this paper and so for simplicity we always assume the set of states to be equal to Σ^n. It is not uncommon either to have different alphabets for each component of a state instead of just one alphabet Σ for all n components. Again for simplicity we choose only one alphabet Σ without loss of generality of our results.

Our concern is with the complexity of certain decision problems on state spaces. Clearly, since the size of a problem instance is a critical issue in complexity analysis, we need to distinguish between different more or less compact ways of representing state spaces.

An *explicit* way to represent a state space would be to write down Σ, n, and the set Π of all edges explicitly. This is in general not a compact representation and thus usually impractical. Except for cases in which explicit representation helps us to underline some of our hardness results below, we instead follow the PSVN notation introduced by Hernádvölgyi and Holte (1999); we call this an *implicit* representation of a state space. Here we define edges via parameterized operators, so that one operator can represent a large set of edges.

For instance, consider $\Sigma = \{a, b, c\}$ and $n = 4$. Let x_1 and x_2 be variable symbols. The operator

$$\langle x_1, c, x_1, x_2\rangle \to \langle b, x_1, x_2, x_2\rangle$$

means that every state s which has a 'c' in component 2, identical entries σ in components 1 and 3, and any value σ' in component 4 has an outgoing edge to the state that has a 'b' in component 1, σ in component 2, and σ' in components 3 and 4. For example, $(\langle a, c, a, a\rangle, \langle b, a, a, a\rangle)$ and $(\langle b, c, b, a\rangle, \langle b, b, a, a\rangle)$ are edges induced by this operator, whereas neither $(\langle a, a, a, a\rangle, \langle b, a, a, a\rangle)$ nor $(\langle a, c, a, a\rangle, \langle b, b, a, a\rangle)$ are.

Definition 3 *Let $\mathcal{S} = (\Sigma, n, \Pi)$ be a state space and $X = \{x_i \mid i \in \mathbb{N}\}$ a set of variables, $X \cap \Sigma = \emptyset$. Any n-tuple $p = \langle y_1, \ldots, y_n\rangle \in (\Sigma \cup X)^n$ is called a state pattern.*

Let $p = \langle y_1, \ldots, y_n\rangle$ and $p' = \langle y'_1, \ldots, y'_n\rangle$ be state patterns. $p \to p'$ is called an operator if, for all $i \in \{1, \ldots, n\}$, $y'_i \in \Sigma$ or $y'_i = y_j$ for some $j \in \{1, \ldots, n\}$. A state $s = \langle \sigma_1, \ldots, \sigma_n\rangle \in \Sigma^n$ matches p if for all $i \in \{1, \ldots, n\}$ the following conditions hold.

1. *$y_i = \sigma_i$ or $y_i \in X$,*
2. *if $y_i = y_j$ for some $j \in \{1, \ldots, n\}$ then $\sigma_i = \sigma_j$.*

An edge (s, s') matches the operator $o = p \to p'$ if s matches p, s' matches p', and for all $i, j \in \{1, \ldots, n\}$

$$y'_i = y_j \Rightarrow s'_i = s_j\,.$$

(Σ, n, O) is a representation for $\mathcal{S}$ if O is a set of operators such that $(s, s') \in \Pi$ if and only if there is an $o \in O$ such that (s, s') matches o.

Some remarks on this representation are necessary:

1. We assume an efficient representation of operators such that the size of a representation of each $o \in O$ shrinks proportionally with the number of variables affected by o, such as a set-theoretic representation comparable to SAS$^+$ (Bäckström, 1992) or propositional STRIPS (Fikes & Nilsson, 1971). We use a different notation in this paper just for the sake of readability; however given that each variable occurs in at least one operator (which we assume since otherwise some variables would be irrelevant) this representation is asymptotically of the same order.

2. Initial states and goal states are not part of our definition of state spaces. They come into play as soon as reachability (*i.e.*, plan existence) problems and search or planning problems are considered. We consider the state space as an environment in which new planning problems can be defined by choosing new start and goal states; this is the same as the separation of problem domain definition and problem definition in PDDL, *cf.* (McDermott, 2000).

3. Our implicit (PSVN) representation is expressive enough to model state spaces defined in propositional STRIPS or SAS$^+$ notation. In particular, any propositional STRIPS or SAS$^+$ plan existence problem can be transformed into a similar one in our notation in time and space polynomial in the size of the input.

4. Explicit representation as opposed to implicit (PSVN) representation is in general impractical; we define it in order to emphasize the complexity of one of the decision problems studied below.

3.2 State space abstractions

An abstraction of a state space $\mathcal{S}$ is a state space $\mathcal{S}_\psi$ with a mapping ψ from the states in $\mathcal{S}$ to the states in $\mathcal{S}_\psi$. The edges are considered to be induced by ψ.

Definition 4 *Let $\mathcal{S} = (\Sigma, n, \Pi)$ be a state space, $\Gamma \subseteq \Sigma$, and $m \in \mathbb{N}$. Any mapping $\psi : \Sigma^n \to \Gamma^m$ induces a state space $\mathcal{S}_\psi = (\Gamma, m, \Pi_\psi)$ over Γ^m where, for all $t, t' \in \Gamma^m$ we have $(t, t') \in \Pi_\psi$ if and only if there are states $s, s' \in \Sigma^n$ such that $\psi(s) = t$, $\psi(s') = t'$, and $(s, s') \in \Pi$. $\mathcal{S}_\psi$ is called an abstraction of $\mathcal{S}$, ψ is called an abstraction mapping.*

Thus ψ is a graph homomorphism between the graphs (Σ^n, Π) and (Γ^m, Π_ψ), where without loss of generality and just for simplicity we assume $\Gamma \subseteq \Sigma$. Note that our definition precludes the abstract space containing edges that are not induced by ψ (i.e., Π_ψ is the minimal set of edges for which ψ is a graph homomorphism between (Σ^n, Π) and (Γ^m, Π_ψ)).[3] So all hardness results that we prove below for this special case also hold in the general case where Π_ψ is only a subset of the edges in the abstract state space.

We consider two types of abstraction.

Domain abstraction. Here $m = n$ but $|\Gamma| < |\Sigma|$. A domain abstraction[4] mapping is a surjective string homomorphism $\psi : \Sigma^n \to \Gamma^n$ with $\psi(\psi(\sigma)) = \psi(\sigma)$ for all $\sigma \in \Sigma$. The trivial case is $|\Gamma| = 1$.

Projection. Here $\Gamma = \Sigma$ but $m < n$. A projection mapping ψ is defined via a subset $M = \{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}$ of cardinality m such that $\psi(\sigma_1, \ldots, \sigma_n) = (\sigma_{i_1}, \ldots, \sigma_{i_m})$ for all $\sigma_1, \ldots, \sigma_n \in \Sigma$. The trivial case is $M = \emptyset$.

Note that, if $\mathcal{S} = (\Sigma, n, \Pi)$ is a state space and ψ an abstraction mapping then

$$\psi(\Delta(s, \mathcal{S})) \subseteq \Delta(\psi(s), \mathcal{S}_\psi)$$

holds for every $s \in \mathcal{S}$ by definition. Interestingly, the opposite inclusion does not necessarily hold—and this is exactly what causes the problems observed in our experiments. For any given original state s^* (not necessarily start state or goal state), a spurious state is an abstract state $t \in \Delta(\psi(s^*), \mathcal{S}_\psi) \setminus \psi(\Delta(s^*, \mathcal{S}))$, *i.e.*, an abstract state that is reachable from the abstract image of s^* but has no pre-image in the original state that is reachable from s^*.

The role s^* plays in this consideration would be the role of the start state when doing forward search; in this case one would want to avoid spurious states reachable from $\psi(s^*)$ in the abstract space. When doing retrograde search (*e.g.*, to build a PDB), one should think

3. This coincides with a previously studied notion of homomorphism for abstraction mappings (Helmert, Haslum, & Hoffmann, 2007).

4. Here the term "domain" refers to the domain Σ of the variables used for state space encodings, while often we use the term "domain" to refer to the general problem domain, *e.g.*, the Blocks World domain with k blocks. It will always be clear from the context which of these two concepts is meant.

of $\mathcal{S}$ as being the transpose of the original state space, *i.e.*, the state space after inverting all edges, and of s^* as being the goal state. In this case one would want to avoid spurious states reachable from the abstract goal $\psi(s^*)$.

Consequently, for any fixed start state s^* (or a goal state s^* in the transpose of the state space), an abstraction ψ avoiding unwanted spurious states would fulfill $\Delta(\psi(s^*), \mathcal{S}_\psi) \subseteq \psi(\Delta(s^*, \mathcal{S}))$. Thinking in terms of a problem domain without fixing start or goal in advance, this definition can be generalized to a criterion concerning all possible states s^*. This motivates our definition of downward path preserving abstractions.

Definition 5 *Let $\mathcal{S} = (\Sigma, n, \Pi)$ be a state space and ψ an abstraction mapping. ψ is called a downward path preserving (DPP) abstraction of $\mathcal{S}$ if*

$$\psi(\Delta(s, \mathcal{S})) = \Delta(\psi(s), \mathcal{S}_\psi)$$

for all $s \in \Sigma^n$. For any particular state $s^* \in \Sigma^n$ the abstraction ψ is DPP for s^* and $\mathcal{S}$ if $\psi(\Delta(s^*, \mathcal{S})) = \Delta(\psi(s^*), \mathcal{S}_\psi)$.

Figure 2 illustrates the difference between DPP and non-DPP abstractions, both for the case of domain abstraction and for the case of projection. Note that the initial state space chosen here consists of three components that are not connected to each other (plus a large set of isolated states not shown)—this situation is possible given our definition of state space. The dotted boxes in the abstractions indicate which of the original states are no longer distinguishable from certain other states after abstraction.

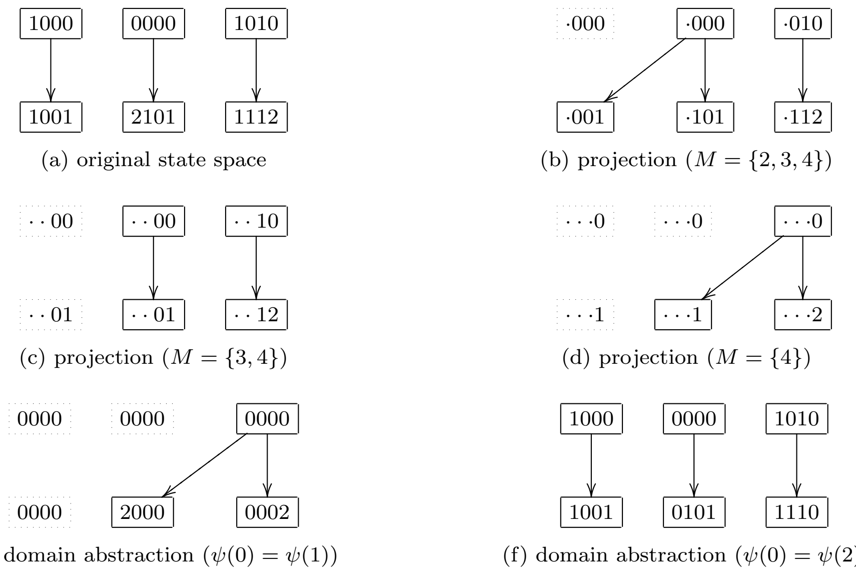
1000
0000
1010
1001
2101
1112
(a) original state space
·000
·000
·010
·001
·101
·112
(b) projection ($M = \{2,3,4\}$)
··00
··00
··10
··01
··01
··12
(c) projection ($M = \{3,4\}$)
···0
···0
···0
···1
···1
···2
(d) projection ($M = \{4\}$)
0000
0000
0000
0000
2000
0002
(e) domain abstraction ($\psi(0) = \psi(1)$)
1000
0000
1010
1001
0101
1110
(f) domain abstraction ($\psi(0) = \psi(2)$)

Figure 2: A state space and abstractions. Projections (b) and (d) are not DPP, projection (c) is DPP. Domain abstraction (e) is not DPP, domain abstraction (f) is DPP but it does not reduce the number of states.

4. DPP abstractions in theory

In this section we discuss the complexity of the problem of generating DPP abstractions formally. Subsection 4.1 addresses this problem in several very general versions, which turn out to be intractable. Subsection 4.2 then discusses easily testable conditions under which a given state space allows for very natural DPP abstractions, both for the case of projection and for the case of domain abstraction. These conditions show that the concrete representation of a state space matters for the design of good abstractions, an issue that is further discussed in Subsection 4.3.

4.1 Computational complexity of finding DPP abstractions

As explained above (Section 2.2), there are three decision problems of immediate relevance for the automatic construction of DPP abstractions. Their computational complexity is the topic of this section.

Throughout this section we assume that abstraction mappings can be computed in polynomial time, so that they are in fact never the bottleneck when we obtain hardness results. Moreover, all results presented here transfer to SAS$^+$ notation and STRIPS notation. The greater expressiveness of the notation we use is not exploited in the proofs anywhere.

We analyze the following decision problems.

REACHABLE. Given a state space $\mathcal{S}$ (in explicit or implicit representation) and two corresponding states s and s', decide whether or not $s' \in \Delta(s, \mathcal{S})$.

IS-DPP. Given a state space $\mathcal{S}$ (in explicit or implicit representation) and an abstraction mapping ψ (of type domain abstraction or projection), decide whether or not ψ is a DPP abstraction of $\mathcal{S}$.

EXIST-DPP. Given a type of abstraction (either domain abstraction or projection), a state space $\mathcal{S}$ over Σ^n (in explicit or implicit representation), a subset $\Gamma \subseteq \Sigma$ and a number m (where either $n = m$ (domain abstraction) or $\Sigma = \Gamma$ (projection)), decide whether or not there is a non-trivial DPP abstraction mapping ψ of $\mathcal{S}$, of the given type, inducing a state space $\mathcal{S}_\psi$ over Γ^m.

In what follows we summarize our results on the computational complexity of the three decision problems defined above. The first result is just a review.

Theorem 1 1. *REACHABLE is PSPACE-hard for the case of implicit state space representation.*

2. *REACHABLE is in P for the case of explicit state space representation.*

3. *REACHABLE is in P for the case of either type of state space representation if the dimension n is fixed a priori.*

Proof. The first statement follows from the fact that reachability for SAS$^+$—a subproblem of ours—is already *PSPACE*-complete (Bäckström, 1992). Note here that transforming SAS$^+$ problems into our notation requires only polynomial space, we omit the details. The second statement is a trivial and well-known fact—reachability can be tested in time linear in the number of edges. Assertion 3 finally follows from the fact that for fixed n the number of states in the state space is polynomial in the size of the alphabet Σ, so an exhaustive check is tractable. □

The hardness result in the first statement of this theorem turns out to be very useful for the analysis of the IS-DPP problem, due to the following reducibility result.

Theorem 2 *1. REACHABLE is polynomially reducible to IS-DPP for the case of domain abstraction.*

2. REACHABLE is polynomially reducible to IS-DPP for the case of projection.

The proof is given in the Appendix below. This reducibility result then is the main ingredient in the proof of the following complexity properties of the IS-DPP problem.

Corollary 3 *1. IS-DPP is PSPACE-hard for the case of implicit state space representation and either type of abstraction.*

2. IS-DPP is in P for the case of explicit state space representation and either type of abstraction.

3. IS-DPP is in P for the case of either type of state space representation and either type of abstraction if the dimension n is fixed a priori*.*

Proof. (1). This follows from Theorem 2 and Theorem 1.1.

(2). For each state that has an out-going edge (and is thus explicitly given) one can list all reachable states in both spaces and thus compare $\psi(\Delta(s, \mathcal{S}))$ to $\Delta(\psi(s), \mathcal{S}_\psi)$ for every $s \in \mathcal{S}$, where $\mathcal{S}$ and ψ form the input instance.

(3). This follows from the fact that for fixed n the number of states in the original space is polynomial in the size of Σ and the number of states in the abstract space is polynomial in the size of Γ (where $\Sigma = \Gamma$ in case of projection). For each of the polynomially many states one can exhaustively check the polynomially many reachable states in both spaces and thus compare $\psi(\Delta(s, \mathcal{S}))$ to $\Delta(\psi(s), \mathcal{S}_\psi)$ for every $s \in \mathcal{S}$, where $\mathcal{S}$ and ψ are the input instance. □

At this point one might ask why we consider explicit state space representation at all—it is not compact and it thus yields trivial tractability results that are of no practical value—for both REACHABLE and IS-DPP.

The reason to nevertheless include explicit representation is in order to illustrate the hardness of EXIST-DPP. The next theorem shows that even for explicit representation and for fixed state space dimension the existence of a DPP domain abstraction can presumably not be decided in polynomial time, see Assertion 3.

Theorem 4 *1. EXIST-DPP is PSPACE-hard for the case of projection and implicit state space representation.*

2. EXIST-DPP is in P for the case of projection and either type of state space representation if the dimension n is fixed a priori*.*

3. *EXIST-DPP is NP-hard for the case of domain abstraction and either type of state space representation even if the dimension n is fixed with $n > 2$* a priori.

The proof is given in the Appendix below.

The reader might be concerned that the *NP*-hardness results here are due to the fact that we ask for general DPP abstractions instead of just for abstractions that are DPP for a specific state s^* and $\mathcal{S}$ in the sense of Definition 5. However, all *NP*-hardness results proven here hold even in the case that s^* is part of the problem instance and we only ask for DPP abstractions with respect to s^* and $\mathcal{S}$. We denote these variants of the decision problems by adding a subscript s^*. A proof of the following corollary is given in the Appendix.

Corollary 5 1. *IS-DPP$_{s^*}$ is PSPACE-hard for the case of implicit state space representation and either type of abstraction.*

2. *EXIST-DPP$_{s^*}$ is PSPACE-hard for the case of projection and implicit state space representation.*

3. *EXIST-DPP$_{s^*}$ is NP-hard for the case of domain abstraction and either type of state space representation even if the dimension n is fixed with $n > 2$* a priori.

4.2 Sufficient conditions for the existence of DPP abstractions

The intractability results shown above are important to notice. However, more interesting from a practical point of view are easily testable criteria that help us to design DPP projections and/or domain abstractions for given state spaces (in case such abstractions exist at all).

This section provides such conditions for the case of projection and for the case of domain abstraction, each with examples of standard problem domains that meet these easily testable criteria.

Moreover, we show that standard planning domains like the Blocks World, for which the usual encoding does not meet these criteria, can be very naturally encoded in a way that our criteria become applicable and DPP abstractions can be defined very easily.

The problem of finding an encoding of a problem domain, such that our easily testable criteria apply, is hard nevertheless. But the criteria provided here can be used as design principles for encoding state spaces in domain-specific planning.

4.2.1 Projection

An example of a standard problem domain in which certain types of projections obviously yield DPP abstractions is Rubik's Cube (Korf, 1997). In standard encodings of this problem domain either one of the following projections is DPP.

- Ignore *all* variables that encode information about *corner cubies.*
- Ignore *all* variables that encode information about *edge cubies.*

The intuitive reason is that for every operator, even though it affects both corner cubies and edge cubies, the effects on corner cubies only depend on the preconditions on corner cubies (and the effects of edge cubies only depend on preconditions on edge cubies). Moreover, no operator checks for the exact values of variables—it is just a permutation of some corner cubie variables and a permutation of some edge cubie variables.

Hence, if a projection ignores the whole set of corner cubies (or the whole set of edge cubies), it cannot cause spurious states and thus has to be DPP.

In contrast, if one ignores only part of the variables encoding corner cubies (or only part of the variables encoding edge cubies), there is a high risk of getting non-DPP abstractions.

Formally, this is an example of an easily testable general criterion that guarantees the existence of DPP projections, and, more than that, even tells us how to achieve DPP projections.

We use the following definition for the formal statement of our result.

Definition 6 *Let (Σ, n, O) be a representation for a state space. Let $o = \langle y_1, \ldots, y_n \rangle \rightarrow \langle y'_1, \ldots, y'_n \rangle$ be an operator in O and $I \subseteq \{1, \ldots, n\}$ a set of indices. We say that o is closed on I if the following three conditions are satisfied.*

1. *$y_i \notin \Sigma$ for all $i \in I$.*

2. *$y_i \neq y_j$ for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$.*

3. *$\{y'_i \mid i \in I\} \subseteq \{y_i \mid i \in I\}$.*

Closedness of an operator on a set I of indices means that the operator's preconditions have only pairwise distinct variables, no constants in the state pattern components indexed by I, and the operator's effects on the components in I is to just shuffle them (duplication of variables in the effects is allowed).

For example, in the Rubik's Cube domain, every operator is closed on the set of indices of variables encoding the corner cubies. That means the operator's effects on the corner cubie variables are given by shuffling and/or copying (here even permuting) the corner cubie variables in the operator's preconditions (the latter not containing constants or duplicates of variable names).

The same holds for the set of indices of variables encoding the edge cubies.

Our formal criterion allowing for DPP abstractions can then be stated as follows.

Theorem 6 *Let (Σ, n, O) be a representation for a state space $\mathcal{S}$. Suppose there is a disjoint partitioning*

$$I_1 \cup \ldots \cup I_z = \{1, \ldots, n\},$$

such that o is closed on I_j for every $o \in O$ and every $j \in \{1, \ldots, z\}$.

Let $j \in \{1, \ldots, z\}$ and $I_j = \{j_1, \ldots, j_l\}$.

Then the mapping ψ with

$$\psi(\sigma_1, \ldots, \sigma_n) = (\sigma_{j_1}, \ldots, \sigma_{j_l})$$

defines a DPP abstraction of $\mathcal{S}$.

Proof. According to Definition 5, we have to prove that

$$\psi(\Delta(s, \mathcal{S})) = \Delta(\psi(s), \mathcal{S}_\psi)$$

holds for all $s \in \Sigma^n$.

Since $\psi(\Delta(s, \mathcal{S})) \subseteq \Delta(\psi(s), \mathcal{S}_\psi)$ trivially holds for all $s \in \Sigma^n$, we only need to show that $\Delta(\psi(s), \mathcal{S}_\psi) \subseteq \psi(\Delta(s, \mathcal{S}))$ for all $s \in \Sigma^n$.

This inclusion relation is implied by the following fact, which we are going to prove next.

Fact. For every state $s \in \Sigma^n$ and for every two abstract states $t, t' \in \psi(\Sigma^n)$ with $\psi(s) = t$, and every operator $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$ the following holds. If (t, t') matches $\langle y_{j_1}, \ldots, y_{j_l} \rangle \to \langle y'_{j_1}, \ldots, y'_{j_l} \rangle$, then there is a state $s' \in \Sigma^n$ with

(i) $\psi(s') = t'$, *and*

(ii) (s, s') *matches* $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle$.

To prove this fact, let $s = \langle \sigma_1, \ldots, \sigma_n \rangle \in \Sigma^n$. Let $t \in \Sigma^l$ with $\psi(s) = t$, i.e., $t = \langle \sigma_{j_1}, \ldots, \sigma_{j_l} \rangle$. Assume $t' \in \Sigma^l$ is an abstract state and $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots y'_n \rangle \in O$ is an operator such that (t, t') matches the abstract version $\langle y_{j_1}, \ldots, y_{j_l} \rangle \to \langle y'_{j_1}, \ldots, y'_{j_l} \rangle$ of o.

Because of the closedness properties 1 and 2, together with the fact that $t = \langle \sigma_{j_1}, \ldots, \sigma_{j_l} \rangle$ matches $\langle y_{j_1}, \ldots, y_{j_l} \rangle$, we know that s matches $\langle y_1, \ldots, y_n \rangle$.

Since s matches the left hand side $\langle y_1, \ldots, y_n \rangle$ of o, there is a (unique) state $s' = \langle \sigma'_1, \ldots, \sigma'_n \rangle \in \Sigma^n$ such that (s, s') matches o. In particular, s' matches $\langle y'_1, \ldots, y'_n \rangle$. This implies that $\psi(s') = \langle \sigma'_{j_1}, \ldots, \sigma'_{j_l} \rangle$ matches $\langle y'_{j_1}, \ldots, y'_{j_l} \rangle$. Note that t' matches $\langle y'_{j_1}, \ldots, y'_{j_l} \rangle$ as well.

Finally, to prove that $t' = \psi(s')$ note that $\{y'_{j_1}, \ldots, y'_{j_l}\} \subseteq \{y_{j_1}, \ldots, y_{j_l}\}$ because of the closedness property 3. We show that $\sigma'_{j_i} = \tau'_i$ for all $i \in \{1, \ldots, l\}$, where $t' = \langle \tau'_1, \ldots, \tau'_l \rangle$. So let $i \in \{1, \ldots, l\}$. Because of the closedness property 3, $y'_{j_i} = y_{j_k}$ for some $k \in \{1, \ldots, l\}$. Thus $\sigma'_{j_i} = \sigma_{j_k} = \tau'_i$, because (s, s') matches o and (t, t') matches the abstract version of o. Hence $t' = \psi(s')$.

This proves the fact and thus the theorem. □

For illustration, consider again Rubik's Cube. Every variable in the standard representation of this problem domain (Korf, 1997) encodes either a property of a corner cubie or a property of an edge cubie. Thus we have a disjoint partitioning of our state components into two sets (one for corner cubie variables and one for edge cubie variables). Since every operator is both closed on the set of indices of corner cubie variables and closed on the set of indices of edge cubie variables, we can project out either all corner cubie variables or all edge cubie variables without violating the DPP property.

Note that the criterion in Theorem 6 can be easily tested. The Rubik's Cube example shows that it can be applied to quite "natural" problem domains.

4.2.2 Domain abstraction

A typical example in which commonly used domain abstractions usually never violate the DPP property is the Sliding Tile puzzle, *cf.* (Slocum & Sonneveld, 2006).

This puzzle consists of $n^2 - 1$ numbered tiles that can be moved in an $n \times n$ grid. Every state is characterized by the grid positions of the tiles numbered $1, \ldots, n^2 - 1$ and the remaining "blank" position, *i.e.*, the only grid position that does not contain a tile. Every operator moves a tile i adjacent to the blank position into the blank position, at the same time making the previous position of tile i blank. See Figure 3 for illustration of the case $n = 3$, *i.e.*, the 8-tile Sliding Tile puzzle (8-puzzle for short).

1	4	2
3	7	5
B	6	8

B	1	2
3	4	5
6	7	8

Figure 3: The 8-puzzle—scrambled state and typical goal state. B represents the blank.

The commonly used domain abstractions here identify some of the names of the tiles. However, they always preserve the information about the unique position of the blank. This is quite intuitive, since the blank obviously plays a special role. It can swap positions with neighbouring tiles. Since the names of the tiles are irrelevant for whether or not they can be swapped with the blank, it seems very natural to define abstractions by ignoring the names of tiles while keeping track of the unique blank position.

In contrast, it is not hard to see that abstractions which identify one or more of the actual tiles with the blank and thus introduce several blank positions at the same time, in general are non-DPP.

To see how the case of the Sliding Tile puzzle can be generalized to a criterion for DPP domain abstractions, let us have a look at the operators used for representing the corresponding problem domains.

For instance, for the 8-puzzle, one typically uses 9 variables. For each of the 9 "physical positions", there is exactly one variable. Its value is B if the position is blank and its value is the name of the tile in that position (ranging from 1 to 8) otherwise. Operators then always look as follows.

$$\langle x_1, x_2, B, x_4, x_5, x_6, x_7, x_8, x_9 \rangle \to \langle x_1, x_2, x_6, x_4, x_5, B, x_7, x_8, x_9 \rangle$$

This operator says that the tile in the sixth position can be moved upwards to the third position if the latter is blank—independent of the name of the tile in the sixth position.

It is easy to see that for such a representation of the 8-puzzle (or any version of the Sliding Tile puzzle in general), the only constant value ever occurring in the preconditions of an operator is B. No operator is conditional on the name of any tile and hence the names of tiles can be ignored without violating the DPP property.

Imagine that a version of the Sliding Tile puzzle had only operators that are conditional on the position of the blank and the position of the tile numbered 8. Then still every domain abstraction that identifies some of the names ranging from 1 to 7 with each other (but not with B or 8) would be DPP.

The following theorem generalizes these observations. Again it requires the introduction of a property of operators.

Definition 7 *Let (Σ, n, O) be a representation for a state space. Let $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle$ be an operator in O and $\Sigma_0 \subseteq \Sigma$. We say that o is independent on Σ_0 if the following two conditions are satisfied.*

1. $y_i \notin \Sigma_0$ *for all* $i \in \{1, \ldots, n\}$.

2. $y_i \neq y_j$ *for all* $i, j \in \{1, \ldots, n\}$ *with* $i \neq j$.

For instance, in the standard representation of the Sliding Tile puzzle, every operator is independent on the set of the names of tiles (but not on the set $\{B\}$).

Theorem 7 *Let* (Σ, n, O) *be a representation for a state space* $\mathcal{S}$*. Suppose there is a disjoint partitioning*

$$\Sigma_0 \cup \Sigma_1 = \Sigma\,,$$

such that every $o \in O$ *is independent on* Σ_0*.*

Then every domain abstraction mapping ψ *with*

$$\psi(\Sigma_0) \subseteq \Sigma_0 \text{ and } \psi(\sigma) = \sigma \text{ for every } \sigma \in \Sigma_1$$

defines a DPP abstraction of $\mathcal{S}$*.*

Proof. As in the proof of Theorem 6, we only need to show that $\Delta(\psi(s), \mathcal{S}_\psi) \subseteq \psi(\Delta(s, \mathcal{S}))$ for all $s \in \Sigma^n$. This inclusion relation is implied by the following fact, which we are going to prove next. We define $\psi(y) = y$ if y is a variable.

Fact. For every state $s \in \Sigma^n$ *and for every two abstract states* $t, t' \in \psi(\Sigma^n)$ *with* $\psi(s) = t$*, and every operator* $o = \langle y_1, \ldots, y_n \rangle \rightarrow \langle y'_1, \ldots, y'_n \rangle \in O$ *the following holds. If* (t, t') *matches* $\langle \psi(y_1), \ldots, \psi(y_n) \rangle \rightarrow \langle \psi(y'_1), \ldots, \psi(y'_n) \rangle$*, then there is a state* $s' \in \Sigma^n$ *with*

(i) $\psi(s') = t'$*, and*

(ii) (s, s') *matches* $o = \langle y_1, \ldots, y_n \rangle \rightarrow \langle y'_1, \ldots, y'_n \rangle$*.*

To prove this fact, let $s = \langle \sigma_1, \ldots, \sigma_n \rangle \in \Sigma^n$. Let $t \in \Sigma^n$ with $\psi(s) = t$, *i.e.*, $t = \langle \psi(\sigma_1), \ldots, \psi(\sigma_n) \rangle$. Assume $t' \in \psi(\Sigma^n)$ is an abstract state and $o = \langle y_1, \ldots, y_n \rangle \rightarrow \langle y'_1, \ldots, y'_n \rangle \in O$ is an operator such that (t, t') matches the abstract version

$$\langle \psi(y_1), \ldots, \psi(y_n) \rangle \rightarrow \langle \psi(y'_1), \ldots, \psi(y'_n) \rangle$$

of o.

Since $t = \langle \psi(\sigma_1), \ldots, \psi(\sigma_n) \rangle$ matches $\langle \psi(y_1), \ldots, \psi(y_n) \rangle$, we can conclude that s also matches $\langle y_1, \ldots, y_n \rangle$. If this was not the case, then, because of the independence properties, s and t would have to disagree in some component i such that

$$(\sigma_i \in \Sigma_1 \text{ or } \psi(\sigma_i) \in \Sigma_1) \text{ and } \sigma_i \neq \psi(\sigma_i)\,,$$

since no values in Σ_0 occur in the left hand side of o. But this would mean that $\psi(\Sigma_0) \not\subseteq \Sigma_0$ or $\psi(\sigma) \neq \sigma$ for some $\sigma \in \Sigma_1$, which would contradict the choice of ψ.

Since s matches the left hand side $\langle y_1, \ldots, y_n \rangle$ of o, there is a (unique) state $s' = \langle \sigma'_1, \ldots, \sigma'_n \rangle \in \Sigma^n$ such that (s, s') matches o. In particular, s' matches $\langle y'_1, \ldots, y'_n \rangle$.

This implies that $\psi(s') = \langle \psi(\sigma'_1), \ldots, \psi(\sigma'_n) \rangle$ matches $\langle \psi(y'_1), \ldots, \psi(y'_n) \rangle$. Note that t' also matches $\langle \psi(y'_1), \ldots, \psi(y'_n) \rangle$.

Finally, to prove that $t' = \psi(s')$ we show that $\psi(\sigma'_i) = \tau'_i$ for every $i \in \{1, \ldots, n\}$, where $t' = \langle \tau'_1, \ldots, \tau'_n \rangle$. So let $i \in \{1, \ldots, n\}$.

- If $\psi(y'_i) \in \Sigma$, then $\sigma'_i = \tau'_i = \psi(y'_i)$ since both $\psi(s')$ and t' match $\langle \psi(y'_1), \ldots, \psi(y'_n) \rangle$. Thus $\psi(\sigma'_i) = \psi(\psi(\sigma'_i)) = \psi(y'_i) = \tau'_i$, since ψ is a domain abstraction.
- If $\psi(y'_i)$ is a variable, then $\psi(y'_i) = y'_i = y_j$ for some $j \in \{1, \ldots, n\}$ by the PSVN definition of operators (Definition 3). Then $\sigma'_i = \sigma_j$, since (s, s') matches o. Since (t, t') matches the abstract version of o, we have $\tau'_i = \psi(\sigma_j)$ and thus $\tau'_i = \psi(\sigma'_i)$.

Consequently, $t' = \psi(s')$.

This proves the fact and thus the theorem. □

For the standard representation of the 8-puzzle, this theorem would apply to $\Sigma_0 = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $\Sigma_1 = \{B\}$. The theorem then says that every domain abstraction that maps the set of tile names to a smaller set of tile names (*i.e.*, identifies some tile names) and leaves the blank symbol untouched will be DPP.

4.3 Encoding matters

As Theorems 6 and 7 show, certain properties of state spaces immediately allow for DPP abstractions. To be precise, these properties concern not the problem domain as such but its encoding as a state space. That means that, for a single planning or search domain, there might be two isomorphic state spaces describing the problem domain such that one of them has DPP projections (or DPP domain abstractions) while the other does not.

A straightforward question is whether every planning or search domain can be encoded in a way that DPP abstractions can easily be designed. Moreover, even if this is possible for a certain problem domain, it is far from trivial to find such encodings, in particular to find them automatically. Nevertheless there is a rich history of research on automatic problem reformulation that addresses this issue (for example (Amarel, 1968; Benjamin, 1989; Korf, 1980; Lowry, 1988; Van Baalen, 1992)).

However, two examples of planning domains will illustrate that, if one deviates from standard STRIPS encodings, quite intuitive encodings that match Theorem 7 can be found. The consequence is that a careful design of problem domain encodings may make the design of DPP abstractions and thus hopefully also planning and search easier.

The two problem domains used for illustration are *(i)* the Blocks World with table positions, which turned out to be very problematic in terms of non-DPP abstractions in our experiments described in Section 2, and *(ii)* the standard Blocks World domain.

4.3.1 Blocks World with table positions

The standard STRIPS encoding of the Blocks World with table positions, *cf.* Section 2, does not match the provably sufficient conditions allowing for DPP abstractions as given in Theorems 6 and 7. Domain abstraction in STRIPS notation does not make any sense since all variables are binary-valued. Projection in the standard STRIPS encoding of the Blocks World with table positions in fact experimentally seems to always introduce spurious states. So far there is no known tractable way to automatically get DPP abstractions in this problem domain—at least for the standard encoding.

However, using the intuitive knowledge that for none of the actions are the names of blocks essential (comparable to the names of tiles in the Sliding Tile puzzle), one can encode the Blocks World with table positions in a very natural way such that Theorem 7 becomes applicable, *i.e.*, DPP domain abstractions can be easily defined.

The encoding for k blocks and $q < k$ table positions looks as follows.

$\mathcal{S} = (\Sigma, n, \Pi)$ where

- $\Sigma = \{b_1, \ldots, b_k\} \cup \{0, 1, \ldots, k\}$,
- $n = q(k+1)$,
- Π is induced by a set O of operators (described below).

Intuitively, a state representation has q sections of $k+1$ variables each. The p^{th} section represents the stack of blocks in table position p, where zeros in the right part of the section represent "no block" when the stack in position p is not of full height k. The first variable in a section has a value in $\{0, 1, \ldots, k\}$ indicating the number of blocks stacked up in this table position. The remaining k variables in this section all have values in $\{b_1, \ldots, b_k\} \cup \{0\}$ representing the names of blocks in the order they are stacked in on table position p. For instance, if on table position p there are three blocks stacked, namely block b_2 on the table, block b_1 on top of b_2, and block b_4 on top of b_1, then the p^{th} section of variables would be

$$3, b_2, b_1, b_4, 0, 0, \ldots 0$$

with a total of $k - 3$ zeros.

The set O has a total of $\binom{q}{2}(k-1)^2$ operators, namely $\binom{q}{2}$ for every pair of numbers (h_i, h_j) with $0 < h_i \leq k$ and $0 \leq h_j < k$. For every pair of table positions p_i and p_j, where h_i is the number of blocks stacked on position p_i and h_j is the number of blocks stacked on position p_j, there is an operator moving the topmost block on the (non-empty) stack in position p_i to the top of the (non-full) stack in position p_j.

For instance, for $h_i = 3$ and $h_j = 1$, O contains one of the following operators, depending on whether $i < j$ or $j < i$.

$$\begin{aligned}
&\langle\ \underbrace{h_1, x_{11}, \ldots, x_{1k}}_{\text{section } 1}, \ldots, \underbrace{3, x_{i1}, x_{i2}, x_{i3}, 0, \ldots, 0}_{\text{section } i}, \ldots, \underbrace{1, x_{j1}, 0, \ldots, 0}_{\text{section } j}, \ldots, \underbrace{h_q, x_{q1}, \ldots, x_{qk}}_{\text{section } q}\ \rangle \\
\rightarrow\ &\langle\ \underbrace{h_1, x_{11}, \ldots, x_{1k}}_{\text{section } 1}, \ldots, \underbrace{2, x_{i1}, x_{i2}, 0, \ldots, 0}_{\text{section } i}, \ldots, \underbrace{2, x_{j1}, x_{i3}, 0, \ldots, 0}_{\text{section } j}, \ldots, \underbrace{h_q, x_{q1}, \ldots, x_{qk}}_{\text{section } q}\ \rangle\,,
\end{aligned}$$

$$\begin{aligned}
&\langle\ \underbrace{h_1, x_{11}, \ldots, x_{1k}}_{\text{section } 1}, \ldots, \underbrace{1, x_{j1}, 0, \ldots, 0}_{\text{section } j}, \ldots, \underbrace{3, x_{i1}, x_{i2}, x_{i3}, 0, \ldots, 0}_{\text{section } i}, \ldots, \underbrace{h_q, x_{q1}, \ldots, x_{qk}}_{\text{section } q}\ \rangle \\
\rightarrow\ &\langle\ \underbrace{h_1, x_{11}, \ldots, x_{1k}}_{\text{section } 1}, \ldots, \underbrace{2, x_{j1}, x_{i3}, 0, \ldots, 0}_{\text{section } j}, \ldots, \underbrace{2, x_{i1}, x_{i2}, 0, \ldots, 0}_{\text{section } i}, \ldots, \underbrace{h_q, x_{q1}, \ldots, x_{qk}}_{\text{section } q}\ \rangle
\end{aligned}$$

Now obviously the names of blocks do not occur in the preconditions of any operator; moreover, no variables occur twice in the preconditions of any operator. Hence Theorem 7 is applicable where

$$\Sigma_0 = \{b_1, \ldots, b_k\} \text{ and } \Sigma_1 = \{0, 1, \ldots, k\}\,.$$

Consequently, every domain abstraction identifying the names of (some of) the blocks is DPP when this encoding is used.

Note that the size of the encoding is polynomial in the number of blocks and table positions.

4.3.2 Blocks World

A slight modification yields an encoding for the standard Blocks World (without table positions) matching Theorem 7.

We treat this planning domain with k blocks like the Blocks World with k blocks and k table positions. The only difference is that states in which the same stacks of blocks occur (just with swapped table positions) have to be identified as equal here. For instance, for $k = 4$, the state

$$\langle \underbrace{1, b_3, 0, 0, 0,}_{\text{section 1}} \underbrace{3, b_2, b_1, b_4, 0,}_{\text{section 2}} \underbrace{0, 0, 0, 0, 0,}_{\text{section 3}} \underbrace{0, 0, 0, 0, 0}_{\text{section 4}} \rangle$$

is equivalent to the state

$$\langle \underbrace{0, 0, 0, 0, 0,}_{\text{section 1}} \underbrace{3, b_2, b_1, b_4, 0,}_{\text{section 2}} \underbrace{0, 0, 0, 0, 0,}_{\text{section 3}} \underbrace{1, b_3, 0, 0, 0}_{\text{section 4}} \rangle$$

This is achieved by introducing operator costs. We adopt all operators used in the encoding of the Blocks World with table positions above, each at a cost of 1, and introduce $\binom{k}{2}$ new operators

$$\begin{aligned} & \langle \underbrace{h_1, x_{11}, \ldots, x_{1k}}_{\text{section 1}}, \ldots, \underbrace{h_i, x_{i1}, \ldots, x_{ik}}_{\text{section } i}, \ldots, \underbrace{h_j, x_{j1}, \ldots, x_{jk}}_{\text{section } j}, \ldots, \underbrace{h_k, x_{k1}, \ldots, x_{kk}}_{\text{section } k} \rangle \\ \to \quad & \langle \underbrace{h_1, x_{11}, \ldots, x_{1k}}_{\text{section 1}}, \ldots, \underbrace{h_j, x_{j1}, \ldots, x_{jk}}_{\text{section } i}, \ldots, \underbrace{h_i, x_{i1}, \ldots, x_{ik}}_{\text{section } j}, \ldots, \underbrace{h_k, x_{k1}, \ldots, x_{kk}}_{\text{section } k} \rangle , \end{aligned}$$

plus their $\binom{k}{2}$ inverses (for every pair (i, j)). Each of these new operators is assigned a cost of 0.

This encoding of the Blocks World is of polynomial size in the number of blocks, and Theorem 7 applies with

$$\Sigma_0 = \{b_1, \ldots, b_k\} \text{ and } \Sigma_1 = \{0, 1, \ldots, k\} .$$

Consequently, every domain abstraction ignoring the names of (some of) the blocks is DPP, when this encoding is used.

5. Related work

In the heuristic search literature, the problem of violating the DPP property has been addressed by Holte and Hernádvölgyi (2004) using the term "non-surjectivity". Holte and Hernádvölgyi analyze different structural properties of state spaces and abstractions that are likely to cause violation of the DPP property. (Haslum et al., 2005) report the problem of heuristic values being too small because of spurious states and experimentally show that enforcing mutex constraints, as was discussed in Section 2, substantially speeds up A*. Most recently Haslum et al. (2007)—dealing with the problem of how to define "good" abstractions—report that PDB heuristics often turn out to be overly optimistic due to violation of exactly what we call the DPP property.

Thus, the literature on heuristic planning and search has recognized the importance of the problem of spurious states and has also recognized that the mutex method can be used to partially solve it, but it has not given a general characterization nor studied the complexity of detecting or avoiding the problem. This paper is to our knowledge the first one to systematically study the occurrence and the consequences of non-DPP abstractions in general and to treat this problem from a complexity-theoretic point of view.

However, a property closely related to DPP—the downward refinement property (DRP)—was developed in the literature on refinement-style planning (Bacchus & Yang, 1991, 1994). The concern in that work was to identify a property that ensures that solution paths (*i.e.*, plans) in an abstract space are guaranteed to be "monotonically refinable" into a solution in the original state space. DRP requires both a start state and a goal state to be given, not just one of the two, and has a narrower focus than DPP's concern about all reachable states—there can be states that violate DPP but are not on any solution path and therefore irrelevant for DRP. In this regard, DRP is a less stringent requirement than DPP. But from the following point of view DRP is more demanding than DPP and actually entails DPP. Any abstract state a that is on any solution path is obviously reachable from the abstract start state. DPP requires there to exist a path from the start state to some state in the pre-image of a; DRP requires this too, but also requires that this path have certain additional properties.

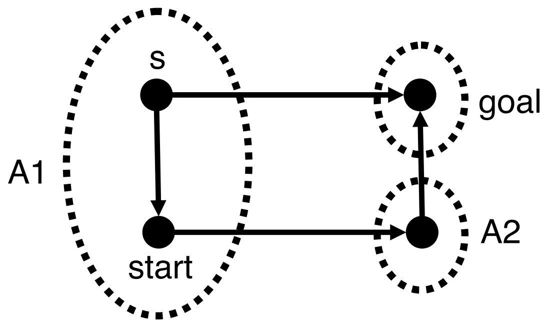

Figure 4: A state space and abstraction that is DPP but not DRP.

Figure 4 shows a state space (the dark circles are its states) and an abstraction (dotted ovals) that is DPP but not DRP. There is an abstract solution that uses one operator to go directly from the abstract start state (A1) to the abstract goal state. However, in the original state space this operator, when applied to the actual start state, produces a state that is in abstract state A2, not the abstract goal state. The refinement of the abstract plan is therefore not "monotonic" as DRP requires.

Just as we did for DPP in Section 4.2, some studies of DRP look for easily tested properties of a state space representation that guarantee DRP. The original DRP paper (Bacchus & Yang, 1991) identified two. The first is "complete independence", which looks to partition the operators according to the variables they test or change. This condition would also guarantee that certain projections satisfy the DPP property. However, it requires that the given operators can be classified into mutually completely independent sets, such that every problem instance can be solved by sequentially solving (in arbitrary order) for the variables affected by operators in one of these sets. Practical problems are not expected to fulfil this condition.

The second is called "necessary connectivity". A variation on this criterion called the "safe abstraction criterion" is described in (Haslum, 2007), which also points out, as we have done in Subsection 4.2, that the exact details of how a state space is represented can affect key properties of the abstractions that standard abstraction techniques produce. The verification of this criterion in general is intractable, so in its full form it does not qualify as an easily testable property.

6. Conclusions

We addressed the problem of state space abstractions with "spurious states"— a problem that has been mentioned in the planning and search literature but has so far never been analyzed systematically.

Experimentally we demonstrated that this problem occurs in standard abstraction methods applied to standard planning domains and that it has a damaging effect on search and planning efficiency. In particular, we showed that automated techniques, even if they help to get rid of almost all the spurious states, can leave the poor quality of the corresponding heuristic unaffected.

This suggests that the spurious states that have the most damaging effect on the heuristic values are exactly those that are hard to filter out with standard (automated) methods. In particular, when using abstractions to define high-quality heuristics, it is not enough to avoid most of the spurious states; ideally one would have to avoid them all.

Motivated by this observation, we studied the problem of avoiding spurious states on a formal level, our main contributions being as follows.

- We defined a formal criterion, called the downward path-preserving (DPP) property, that is necessary and sufficient for abstractions not to contain spurious states.

- We showed that for both standard types of abstraction (projection and domain abstraction) it is in general hard to decide whether a given abstraction of a state space is DPP, or whether a given state space possesses a DPP abstraction at all.

- We provided formal criteria on state space encodings under which the definition of DPP abstractions is straightforward, thus allowing for simple abstraction methods that are guaranteed not to produce any spurious states at all. We showed that some standard heuristic search domains meet those criteria, and the resulting suggested abstractions are intuitive.

- We illustrated the effect that problem domain encodings can have on the ease with which DPP abstractions can be defined. The standard Blocks World benchmark domain turned out to allow for straightforward and intuitive DPP abstractions, under the condition that one deviates from the standard encodings of this domain.

Our positive results in Theorems 6 and 7 yield very promising criteria for the design of problem domain encodings, yet they have strong limitations. Not only is it unlikely that every typical benchmark domain can be encoded in a way that our theorems apply. Even in case we knew about a given state space that it could be re-encoded to match our sufficient conditions, how would we obtain the appropriate encoding? This problem in general seems at least as hard as finding DPP abstractions using the given encoding.

Nevertheless, our results show that, when trying to model a problem domain as a state space, for a particular planning application, it is worth considering whether the state space can be encoded in a way that DPP abstractions can be obtained easily, *e.g.*, by meeting the sufficient conditions provided by our theorems.

Appendix A. Proofs

Proof of Theorem 2.1 We consider implicit state space representation only; the explicit case works analogously.

A reduction mapping from REACHABLE instances to IS-DPP instances for domain abstraction is defined as follows.
Input. $\mathcal{S} = (\Sigma, n, O)$, $s = \langle s_1, \ldots, s_n \rangle \in \Sigma^n$, $s' = \langle s'_1, \ldots, s'_n \rangle \in \Sigma^n$.
Output. $(\hat{\mathcal{S}}, \psi)$ with the following properties.

- $\hat{\mathcal{S}} = (\Sigma \cup \{a, b\}, n+1, \hat{O})$ where a and b are two distinct symbols not contained in Σ and $\hat{O}$ contains all operators

$$\langle o^l_1, \ldots, o^l_n, a \rangle \rightarrow \langle o^r_1, \ldots, o^r_n, a \rangle$$

for $\langle o^l_1, \ldots, o^l_n \rangle \rightarrow \langle o^r_1, \ldots, o^r_n \rangle \in O$ and additionally the two operators

$$\begin{aligned} \hat{o}^s &= \langle s_1, \ldots, s_n, b \rangle \rightarrow \langle s_1, \ldots, s_n, a \rangle \,, \\ \hat{o}^{s'} &= \langle s_1, \ldots, s_n, b \rangle \rightarrow \langle s'_1, \ldots, s'_n, a \rangle \,. \end{aligned}$$

- ψ is the mapping that maps both a and b to a and leaves all characters in Σ unchanged.

This is obviously a polynomial mapping from REACHABLE instances to IS-DPP instances.

It remains to show that it maps positive instances to positive instances and negative ones to negative ones; we do that informally. For that purpose note that the subspace of states in $\hat{\mathcal{S}}$ that have the value a in their last variable form a "copy" of the state space $\mathcal{S}$. The only additional edges in $\hat{\mathcal{S}}$ go *(i)* from $\langle s_1, \ldots, s_n, b \rangle$ to $\langle s_1, \ldots, s_n, a \rangle$ and *(ii)* from $\langle s_1, \ldots, s_n, b \rangle$ to $\langle s'_1, \ldots, s'_n, a \rangle$ (as given by $\hat{o}^s$ and $\hat{o}^{s'}$). So mapping a and b to a yields a state space in which the only edges are "copies" of those in $\mathcal{S}$—with just one exception—an edge from the "copy" of s to the "copy" of s'.

Positive instances of REACHABLE are mapped to positive instances of IS-DPP. If $s' \in \Delta(s, \mathcal{S})$ this exceptional edge does not yield any new pairs of reachable states, hence the abstraction ψ of $\hat{\mathcal{S}}$ has the DPP property.

Negative instances of REACHABLE are mapped to negative instances of IS-DPP. If $s' \notin \Delta(s, \mathcal{S})$ the exceptional edge introduces a connection (path) from the "copy" of s to the "copy" of s'. Consequently, the state $\langle s'_1, \ldots, s'_n, a \rangle$

- belongs to the set $\Delta(\psi(\langle s_1, \ldots, s_n, a \rangle), \hat{\mathcal{S}}_\psi)$ of all states reachable from the abstract version of $\langle s_1, \ldots, s_n, a \rangle$ in the abstract space $\hat{\mathcal{S}}_\psi$, but
- does not belong to the set $\psi(\Delta(\langle s_1, \ldots, s_n, a \rangle, \hat{\mathcal{S}}))$ of all abstract images of states reachable from $\langle s_1, \ldots, s_n, a \rangle$ in the original space $\hat{\mathcal{S}}$.

Hence the abstraction ψ of $\hat{\mathcal{S}}$ is not DPP for $\langle s_1, \ldots, s_n, a \rangle$, in the sense of Definition 5.

This completes the proof. □

Proof of Theorem 2.2 We consider implicit state space representation only; the explicit case works analogously.

The proof proceeds similarly to that of Theorem 2.1. The only difference is that now a and b are two different symbols *belonging to* Σ and the projection is defined to ignore the last one of $n + 1$ given variables.

Note here that REACHABLE is still *NP*-hard when restricted to instances where the alphabet over which the state space is defined has at least 2 symbols. The reduction mapping is then defined by the following input/output behaviour.

Input. $\mathcal{S} = (\Sigma, n, O)$, $s = \langle s_1, \ldots, s_n \rangle \in \Sigma^n$, $s' = \langle s'_1, \ldots, s'_n \rangle \in \Sigma^n$, where a and b are two distinct symbols contained in Σ.

Output. $(\hat{\mathcal{S}}, \psi)$ with the following properties.

- $\hat{\mathcal{S}} = (\Sigma, n + 1, \hat{O})$ and $\hat{O}$ contains all operators
$$\langle o^l_1, \ldots, o^l_n, a \rangle \to \langle o^r_1, \ldots, o^r_n, a \rangle$$
for $\langle o^l_1, \ldots, o^l_n \rangle \to \langle o^r_1, \ldots, o^r_n \rangle \in O$ and additionally the two operators
$$\begin{aligned} \hat{o}^s &= \langle s_1, \ldots, s_n, b \rangle \to \langle s_1, \ldots, s_n, a \rangle \,, \\ \hat{o}^{s'} &= \langle s_1, \ldots, s_n, b \rangle \to \langle s'_1, \ldots, s'_n, a \rangle \,. \end{aligned}$$
- ψ is the projection defined via the subset $M = \{1, \ldots, n\}$ by $\psi(\sigma_1, \ldots, \sigma_{n+1}) = (\sigma_1, \ldots, \sigma_n)$ for all $\sigma_1, \ldots, \sigma_{n+1} \in \Sigma$.

This is obviously a polynomial mapping from REACHABLE instances to IS-DPP instances.

It remains to show that it maps positive instances to positive instances and negative ones to negative ones. This is done in the same way as for the domain abstraction case. Intuitively, projecting out the last variable here has the same effect as identifying a with b in the domain abstraction above.

This completes the proof. □

Proof of Theorem 4.1 Note that the proof of Theorem 2 actually shows that the problem IS-DPP-n of deciding, given a state space (Σ, n, O), whether or not projecting out only the n^{th} variable generates a DPP abstraction, is already *PSPACE*-hard. We reduce IS-DPP-n in polynomial space to EXIST-DPP for the projection case as follows.

Input. $\mathcal{S} = (\Sigma, n, O)$.

Output. $(\hat{\mathcal{S}}, \Sigma \cup \{a, b\}, m)$, where $m = n - 1$ and $\hat{\mathcal{S}} = (\Sigma \cup \{a, b\}, n, \hat{O})$. Here a and b are two distinct symbols not contained in Σ. $\hat{O}$ is a set of operators that contains O and additionally, for every $j \in \{1, \ldots, n-1\}$, a new operator

$$\langle \sigma, \ldots, \sigma, a, \sigma, \ldots, \sigma \rangle \to \langle b, \ldots, b \rangle$$

where $\sigma \in \Sigma$ is a fixed symbol and a appears in the j^{th} position on the left hand side. This is a positive problem instance of EXIST-DPP if and only if there exists a DPP projection of $\hat{\mathcal{S}}$ on an abstract space defined over $m = n - 1$ variables.

Note that *(i)* a never appears in the n^{th} position on the left hand side of any operator, and *(ii)* neither a nor b ever appear on the right hand side of any operator in O.

It is not hard to prove that projections ignoring any of the first $n - 1$ variables will always be non-DPP with respect to $\hat{\mathcal{S}}$. In particular, if there is a DPP projection of $\hat{\mathcal{S}}$ that ignores just one variable then this one variable must be the n^{th} variable.

With this property and the construction of $\hat{\mathcal{S}}$ we can show that $\mathcal{S}$ is a positive instance of IS-DPP-n iff $(\hat{\mathcal{S}}, n - 1)$ is a positive instance of EXIST-DPP, which then immediately implies Theorem 4.1.

Positive instances of IS-DPP-n are mapped to positive instances of EXIST-DPP. Let $\mathcal{S} = (\Sigma, n, O)$ be a positive instance of IS-DPP-n. Then projecting out only the n^{th} variable in $\mathcal{S}$ results in a DPP abstraction. Since the additional operators introduced in $\hat{O}$ all require the symbol a (which is not contained in Σ) in one of the first $n - 1$ variables in the precondition, projecting out only the n^{th} variable in $\hat{\mathcal{S}} = (\Sigma \cup \{a, b\}, n, \hat{O})$ is also a DPP abstraction. Hence $\hat{\mathcal{S}} = (\Sigma \cup \{a, b\}, n, \hat{O})$ is a positive instance of EXIST-DPP.

Negative instances of IS-DPP-n are mapped to negative instances of EXIST-DPP. Let $\mathcal{S} = (\Sigma, n, O)$ be a negative instance of IS-DPP-n. Then projecting out only the n^{th} variable in $\mathcal{S}$ results in a non-DPP abstraction. Obviously projecting out only the n^{th} variable in $\hat{\mathcal{S}}$ then also causes a non-DPP abstraction. Moreover, by the remark above, every projection of $\hat{\mathcal{S}}$ that ignores any of the first $n - 1$ variables must be non-DPP as well. Hence $\hat{\mathcal{S}} = (\Sigma \cup \{a, b\}, n, \hat{O})$ has no DPP projection, *i.e.*, it is a negative instance of EXIST-DPP (in the case of projection). □

Proof of Theorem 4.2 In general, if the states have n components, there are $2^n - 2$ possible sets of components that could be chosen for a non-trivial projection. If n is fixed there is a constant number of possible non-trivial projections. For each of them, IS-DPP can be tested in polynomial time in the size of Σ, Π, and ψ, *cf.* Corollary 3. □

In order to prove Theorem 4.3, we need some additional definitions and propositions. These concern decision problems known to be *NP*-hard, as well as parts of the corresponding proofs, since we will need to exploit the constructions therein.

The following definition specifies two decision problems related to EXIST-DPP.

Definition 8 (Schaefer, 1978; Shimozono & Miyano, 1995) *1. The decision problem NOT-ALL-EQUAL-3SAT is defined as follows. Given a formula H in propositional logic, H in CNF, such that each clause in H contains exactly 3 literals, decide whether or not there is an assignment ρ that satisfies H, such that every clause in H contains at least one literal that is not satisfied by ρ.*[5]

2. The decision problem SHI-MIY95 is defined as follows. Given two finite alphabets Σ, Γ with $|\Sigma| > |\Gamma|$, given two sets $A_1, A_2 \subseteq \Sigma^*$ with $A_1 \cap A_2 = \emptyset$, decide whether or not there is a homomorphism $\varphi : \Sigma^* \to \Gamma^*$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$.[6]

Remark: Furthermore notice in the definition of SHI-MIY95 that we can assume $\Gamma \subseteq \Sigma$ for every two sets Σ and Γ that are part of an instance of SHI-MIY95, without changing the complexity of SHI-MIY95 or any other relevant properties of the problem.

Schaefer (1978) showed the following lemma.

Lemma 8 (Schaefer, 1978; Garey & Johnson, 1979) *NOT-ALL-EQUAL-3SAT is NP-hard.*

This was exploited by Shimozono and Miyano to show that the problem SHI-MIY95 is *NP*-hard, too.

Lemma 9 (Shimozono & Miyano, 1995) *SHI-MIY95 is NP-hard.*

The reduction used by Shimozono and Miyano to prove Lemma 9 is essential for our proof of Theorem 4.3; hence we give their proof here in some detail.

Proof of Lemma 9. The proof is done by polynomial reduction from NOT-ALL-EQUAL-3SAT. This is sufficient because of Lemma 8.

The required reduction mapping instances of NOT-ALL-EQUAL-3SAT to SHI-MIY95 is defined as follows.

Input. A formula H in propositional logic, given as a CNF

$$H = c_1 \wedge \ldots \wedge c_\imath$$

where, for $i \in \{1, \ldots, \imath\}$,

$$c_i = (l_{i1} \vee l_{i2} \vee l_{i3})$$

is a clause of three literals l_{i1}, l_{i2}, l_{i3} over the variables $x_1, \ldots, x_\jmath$.

Output. $(\Sigma, \Gamma, A_1, A_2)$ with

- $\Sigma = \{\mathtt{t}, \mathtt{f}\} \cup \{x_j, \overline{x_j} \mid 1 \leq j \leq \jmath\}$,
- $\Gamma = \{0, 1\}$,

5. Note that for any positive instance of NOT-ALL-EQUAL-3SAT any witnessing assignment ρ satisfies at least one literal per clause in H and dissatisfies at least one literal per clause in H.
6. This problem has a learning theoretic motivation. Imagine A_1 is a set of positive training data, A_2 a set of negative training data disjoint with A_1, both represented over Σ. Can you then "index" the letters in Σ using the smaller alphabet Γ and still have disjoint training sets after rewriting all data using Γ?

- $A_1 = \{\mathtt{tft}\} \cup \{x_j\overline{x_j}x_j \mid 1 \le j \le \jmath\} \cup \{l_{i1}l_{i2}l_{i3} \mid 1 \le i \le \imath\}$,
- $A_2 = \{\mathtt{ttt},\mathtt{fff}\}$.

This is obviously a polynomial mapping from NOT-ALL-EQUAL-3SAT instances to SHI-MIY95 instances. It remains to show that it maps positive instances to positive instances and negative ones to negative ones.[7]

Positive instances of NOT-ALL-EQUAL-3SAT are mapped to positive instances of SHI-MIY95. Let H be a positive instance of NOT-ALL-EQUAL-3SAT. Let $\rho = (r_1, \dots, r_\jmath) \in \{0,1\}^\jmath$ be an assignment of the variables $x_1, \dots, x_\jmath$, such that H is satisfied by ρ with at least one satisfied and at least one dissatisfied literal per clause. Let $\varphi(\mathtt{t}) = 1,\ \varphi(\mathtt{f}) = 0$, and $\varphi(x_j) = r_j,\ \varphi(\overline{x_j}) = \overline{r_i}$ for $1 \le j \le \jmath$, where

$$\overline{r_j} = \begin{cases} 0\,, & \text{if } r_j = 1\,, \\ 1\,, & \text{if } r_j = 0\,. \end{cases}$$

Now it is not hard to prove that φ is a homomorphism from Σ^* to Γ^* with $\varphi(A_1) \cup \varphi(A_2) = \emptyset$.

Negative instances of NOT-ALL-EQUAL-3SAT are mapped to negative instances of SHI-MIY95. The details of this part of the proof are omitted. □

The following variations of SHI-MIY95 will turn out to be useful for our proof of Theorem 4.3 (and for the proof of Corollary 5.3).

Definition 9 1. *The decision problem ALPH-INDEX is defined as follows. Given two finite alphabets Σ, Γ with $|\Sigma| > |\Gamma|$, given $\boldsymbol{n \in \mathbb{N}, n > 2}$, given $\boldsymbol{A_1, A_2 \subseteq \Sigma^n}$ with $A_1 \cap A_2 = \emptyset$, decide whether or not there is a **surjective** homomorphism $\varphi : \boldsymbol{\Sigma^n \longrightarrow \Gamma^n}$ with*

$$\varphi(A_1) \cap \varphi(A_2) = \emptyset \;\; \textit{or} \;\; \boldsymbol{\varphi(A_1) = \varphi(A_2)}\,.$$

2. *The decision problem ALPH-INDEX′ is defined as follows. Given two finite alphabets Σ, Γ with $|\Sigma| > |\Gamma|$, $\boldsymbol{n \in \mathbb{N}, n > 2}$, given $\boldsymbol{A_1, A_2 \subseteq \Sigma^n}$ with $A_1 \cap A_2 = \emptyset$, decide whether or not there is a **surjective** homomorphism $\varphi : \boldsymbol{\Sigma^n \longrightarrow \Gamma^n}$ with*

$$\varphi(A_1) \cap \varphi(A_2) = \emptyset \;\; \textit{or} \;\; \boldsymbol{\varphi(A_1) \subseteq \varphi(A_2)}\,.$$

Remark: Notice in the definition of both ALPH-INDEX and ALPH-INDEX′ that we can assume $\Gamma \subseteq \Sigma$ for every two sets Σ and Γ that are part of a problem instance, without changing the complexity of the problem or any other relevant properties of the problem.

We use the proof of Lemma 9 to show the following main lemma for the proof of Theorem 4.3 (and for the proof of Corollary 5.3).

Lemma 10 1. *ALPH-INDEX is NP-hard, even for fixed $n > 2$.*

2. *ALPH-INDEX′ is NP-hard, even for fixed $n > 2$.*

Proof. **ad 1.** The essence of the proof is the observation (from the proof of Lemma 9) that every problem instance $(\Sigma, \Gamma, A_1, A_2)$ of SHI-MIY95 that ever appears as the image of the reduction from NOT-ALL-EQUAL-3SAT fulfills the following three properties:

7. We omit the details that are not relevant for our proof of Theorem 4.3.

(a) all strings in the sets A_1 and A_2 always have the same length n ($n > 2$);

(b) if $(\Sigma, \Gamma, A_1, A_2)$ is a positive instance of SHI-MIY95 then every homomorphism φ witnessing this must be surjective;

(c) if $\varphi : \Sigma^* \to \Gamma^*$ is *any* surjective function then $\varphi(A_1) \neq \varphi(A_2)$.

Based on this observation, we prove Assertion 1 by showing that

- If $(\Sigma, \Gamma, A_1, A_2)$ is a positive instance of SHI-MIY95 then we can assume without loss of generality that there is a surjective homomorphism $\varphi : \Sigma^n \to \Gamma^n$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) = \varphi(A_2)$.
- If there is a surjective homomorphism $\varphi : \Sigma^n \to \Gamma^n$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) = \varphi(A_2)$ then we can without loss of generality assume that $(\Sigma, \Gamma, A_1, A_2)$ is a positive instance of SHI-MIY95.

This immediately implies Assertion 1 via the proof of Lemma 9.

First, let $(\Sigma, \Gamma, A_1, A_2)$ be a positive instance of SHI-MIY95. Then there is a homomorphism $\varphi : \Sigma^* \to \Gamma^*$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$. By Properties (a) and (b) above, we can assume without loss of generality that $A_1, A_2 \subseteq \Sigma^n$ for some fixed $n > 2$, and that φ is surjective. Hence there is a surjective homomorphism $\varphi : \Sigma^n \to \Gamma^n$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$. Because of Property (c), we can assume without loss of generality that there is a surjective homomorphism $\varphi : \Sigma^n \to \Gamma^n$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) = \varphi(A_2)$.[8]

Second, let $\varphi : \Sigma^n \to \Gamma^n$ be a surjective homomorphism with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) = \varphi(A_2)$. φ is also a surjective homomorphism mapping Σ^* onto Γ^*. Property (c) then allows us to assume $\varphi(A_1) \neq \varphi(A_2)$, and hence $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ by our premise. Thus $(\Sigma, \Gamma, A_1, A_2)$ is a positive instance of SHI-MIY95.

ad 2. Switching from ALPH-INDEX to ALPH-INDEX′, the problem remains *NP*-hard, since in the proof of Assertion 1 we can replace Property (c) of the constructed instances by

(c) if $\varphi : \Sigma^* \to \Gamma^*$ is *any* surjective function then $\varphi(A_1) \not\subseteq \varphi(A_2)$.

Hence, in the reduction given by Shimozono and Miyano (see the proof of Lemma 9), a constructed instance $(\Sigma, \Gamma, A_1, A_2)$ is a positive instance of SHI-MIY95 iff there is a surjective homomorphism $\varphi : \Sigma^n \to \Gamma^n$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) \subseteq \varphi(A_2)$. Assertion (2) follows. □

This now prepares us for the proof of Theorem 4.3.

Proof of Theorem 4.3 We prove this by showing that ALPH-INDEX (for fixed $n > 2$) is polynomially reducible to EXIST-DPP (with the same value of n) for the case of either type of state space representation and domain abstraction.

The reduction is straightforward. The desired polynomial-time reduction function is defined to have the following input/output behaviour.

8. Note that the or-clause here is a part of the statement that will never be fulfilled, due to Property (c). Hence it does not do any harm to add it. For the proof of Theorem 4.3 though it is essential that this clause is contained in the definition of ALPH-INDEX.

Input. Two finite alphabets Σ, Γ with $|\Sigma| > |\Gamma|$, $n \in \mathbb{N}$, $n > 2$, and two sets $A_1, A_2 \subseteq \Sigma^n$ with $A_1 \cap A_2 = \emptyset$, $A_i = \{s^i_1, \ldots, s^i_{k_i}\}$ for $i \in \{1, 2\}$.

Again we may assume without loss of generality that $\Gamma \subseteq \Sigma$.

Output. $\mathcal{S} = (\Sigma, n, \Pi)$ and Γ, where Π is represented explicitly by

$$\begin{aligned}\Pi \;=\; & \{(s^1_{k_1}, s^1_1)\} \cup \{(s^1_j, s^1_{j+1}) \mid 1 \le j < k_1\} \cup \\ & \{(s^2_{k_2}, s^2_1)\} \cup \{(s^2_j, s^2_{j+1}) \mid 1 \le j < k_2\}\end{aligned}$$

Here the states are represented by strings of fixed length n; the value of a state variable is just a single character. Note that in $\mathcal{S}$, every state corresponding to a string in A_1 is reachable from any other state corresponding to a string in A_1 but not from any state corresponding to a string in A_2. The same holds with A_1 and A_2 swapped.

It is not hard to verify that the input is a positive instance of ALPH-INDEX iff the output is a positive instance of EXIST-DPP for domain abstraction:

First, assume the input is a positive instance of ALPH-INDEX. Then there is a surjective homomorphism $\varphi : \Sigma^n \to \Gamma^n$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) = \varphi(A_2)$. Use φ as a domain abstraction on the output instance $\mathcal{S}$. If $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ then the abstraction induced by φ identifies states corresponding to strings in A_1 only with states corresponding to strings in A_1 (and analogously for A_2). Therefore no spurious states are introduced by the abstraction. If $\varphi(A_1) = \varphi(A_2)$ then φ induces the trivial DPP domain abstraction. The surjectivity of φ guarantees that the domain abstraction is non-trivial if Γ is non-trivial. Consequently, $\mathcal{S}$ has a non-trivial DPP domain abstraction and thus the output is a positive instance for EXIST-DPP.

Second, assume the output is a positive instance for EXIST-DPP for the case of domain abstraction. Then there exists a surjective string homomorphism $\varphi : \Sigma^n \to \Gamma^n$ that induces a DPP domain abstraction for $\mathcal{S}$. Assume $\varphi(A_1) \cap \varphi(A_2) \neq \emptyset$ and $\varphi(A_1) \neq \varphi(A_2)$, where we again identify states with strings. Without loss of generality say $\varphi(A_1) \setminus \varphi(A_2) \neq \emptyset$. Let $t \in \varphi(A_1) \cap \varphi(A_2)$ and $\varphi(s') \in \varphi(A_1) \setminus \varphi(A_2)$. Let $s \in A_2$ with $\varphi(s) = t$. This implies that $\varphi(s')$ is reachable from $t = \varphi(s)$ in the abstract space induced by φ, although no pre-image of $\varphi(s')$ is reachable from s in $\mathcal{S}$. Hence φ does not induce a DPP abstraction for $\mathcal{S}$—a contradiction. Therefore $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) = \varphi(A_2)$. Since $\varphi : \Sigma^n \to \Gamma^n$ is surjective, this implies that the input instance is a positive instance of ALPH-INDEX. □

Proof of Corollary 5 **ad 1.** This follows from the proof of Theorem 2; we only have to add a suitable state s^* to the instance of IS-DPP constructed therein. Obviously, the state $\langle s_1, \ldots, s_n, a\rangle$ in the proof of Theorem 2 can take the role of s^* in the desired instance of IS-DPP$_{s^*}$. □

ad 2. This follows from the proof of Theorem 4.1 by adding a suitable state s^* (here $s^* = \langle \sigma, \ldots, \sigma\rangle$) to the instance of EXIST-DPP constructed there. □

ad 3. To prove this we polynomially reduce the problem ALPH-INDEX′ (for fixed $n > 2$) EXIST-DPP$_{s^*}$ (with the same value of n) for the case of either type of state space representation and domain abstraction.

The proof then is in analogy with the proof of Theorem 4.3, where in the reduction the state s^* is chosen arbitrarily from the set $\{s^2_1, \ldots, s^2_{k_2}\}$. □

References

Amarel, S. (1968). On representations of problems of reasoning about actions. In Michie, D. (Ed.), *Machine Intelligence*, Vol. 3, pp. 131–171.

Bacchus, F., & Yang, Q. (1991). The downward refinement property. In *IJCAI 1991, Proceedings of the 12th International Conference on Artificial Intelligence, Sydney, Australia, August 24-30, 1991*, pp. 286–293. Morgan Kaufmann.

Bacchus, F., & Yang, Q. (1994). Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, *71*(1), 43–100.

Bäckström, C. (1992). Equivalence and tractability results for SAS$^+$ planning. In *Proceedings of the 3rd International Conference on Principles on Knowledge Representation and Reasoning*, pp. 126–137.

Ball, M. A. (2009). Compression of pattern databases using algebraic decision diagrams. Master's thesis, University of Alberta.

Benjamin, D. P. (1989). *Change of Representation and Inductive Bias.* Kluwer Academic Publishers, Norwell, MA, USA.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1-2), 5–33.

Culberson, J., & Schaeffer, J. (1996). Searching with pattern databases. In *AI 1996, Proceedings of the 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, Toronto, Ontario, Canada, May 21-24, 1996*, Vol. 1081 of *Lecture Notes in Computer Science*, pp. 402–416. Springer.

Culberson, J., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Edelkamp, S. (2001). Planning with pattern databases. In *Proceedings of the European Conference on Planning*, pp. 13–24.

Fikes, R., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3/4), 189–208.

Haslum, P. (2007). Reducing accidental complexity in planning problems. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pp. 1898–1903.

Haslum, P., Bonet, B., & Geffner, H. (2005). New admissible heuristics for domain-independent planning. In *Proceedings of the 20th AAAI Conference on Artificial Intelligence*, pp. 1163–1168.

Haslum, P., Botea, A., Helmert, M., Bonet, B., & Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pp. 1007–1012.

Haslum, P., & Geffner, H. (2000). Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems*, pp. 140–149.

Helmert, M., Haslum, P., & Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, pp. 176–183.

Hernádvölgyi, I., & Holte, R. (1999). PSVN: A vector representation for production systems. Tech. rep. TR-99-04, University of Ottawa Computer Science.

Holte, R., & Hernádvölgyi, I. (2004). Steps towards the automatic creation of search heuristics. Tech. rep. TR04-02, Department of Computing Science, University of Alberta.

Korf, R. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI/IAAI*, pp. 700–705.

Korf, R. E. (1980). Towards a model of representation changes. *Artificial Intelligence*, *14*(1), 41–78.

Lowry, M. R. (1988). Invariant logic: A calculus for problem reformulation. In *AAAI*, pp. 14–18.

McDermott, D. (1999). Using regression-match graphs to control search in planning. *Artificial Intelligence*, *109*(1-2), 111–159.

McDermott, D. (2000). The 1998 AI planning systems competition. *AI Magazine*, *2*(2), 35–55.

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

Prieditis, A. (1993). Machine discovery of effective admissible heuristics. *Machine Learning*, *12*, 117–141.

Schaefer, T. (1978). The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing, 1-3 May 1978, San Diego, California, USA*, pp. 216–226. ACM.

Slocum, J., & Sonneveld, D. (2006). *The 15 Puzzle*. Slocum Puzzle Foundation.

Van Baalen, J. (1992). Automated design of specialized representations. *Artificial Intellgence*, *54*(1), 121–198.