The computational complexity of avoiding spurious states in state space abstraction

Sandra Zilles[*,a,1], Robert C. Holte[b]

[a] *University of Regina, Department of Computer Science, Regina, Saskatchewan, Canada S4S 0A2*
[b] *University of Alberta, Department of Computing Science Edmonton, Alberta, Canada T6G 2H8*

Abstract

Abstraction is a powerful technique for speeding up planning and search. A problem that can arise 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. Spurious states can be harmful, in practice, because they can create artificial shortcuts in the abstract space that slow down planning and search, and they can greatly increase the memory needed to store heuristic information derived from the abstract space (*e.g.*, pattern databases).

This paper analyzes the computational complexity of creating abstractions that do not contain spurious states. We define a property—the downward path preserving property (DPP)—that formally captures the notion that an abstraction does not result in spurious states. We then 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. The strong hardness results shown carry over to typical description languages for planning problems, including SAS$^+$ and propositional STRIPS. On the positive side, we identify and illustrate formal conditions under which finding downward path

[*]Corresponding author

Email addresses: zilles@cs.uregina.ca (Sandra Zilles), holte@cs.ualberta.ca (Robert C. Holte)

[1]Most of this work was carried out while Sandra Zilles was at the University of Alberta.

preserving abstractions is provably tractable.

Key words: abstraction, heuristic search, planning

---

1. Introduction

Abstraction is a technique for speeding up planning and search that has been successfully applied in various 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 connecting two states in the original space by *refining* an abstract plan connecting the corresponding abstract states [Kno94, HMZM96, Sac74], or *(ii)* by defining a heuristic function to guide planning and serach by using actual distances in the abstract space as estimates of distances in the original space [Pea84, Pri93, McD99, HG00, BG01].

A potential problem with abstraction is that it can introduce what we call "spurious states". 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 shortcuts 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 [CS96, CS98], spurious states will increase the memory requirements and PDB construction time, sometimes dramatically.

The following example of four states $s_1, \ldots, s_4$ illustrates how spurious states can arise. Assume, as shown in Figure 1(a), that s_2 is reachable from s_1 and s_4 is reachable from s_3 but neither s_3 nor s_4 is reachable from either s_1 or s_2. If an abstraction ψ identifies s_2 and s_3 with each other, but maps s_1 and s_4 to two separate abstract states, then the abstract state $\psi(s_1)$ is spurious with respect to the goal state s_4. As shown in Figure 1(b), $\psi(s_4)$ is reachable from $\psi(s_1)$ but $\psi(s_1)$ has no pre-image in the original space from which s_4 can be reached.

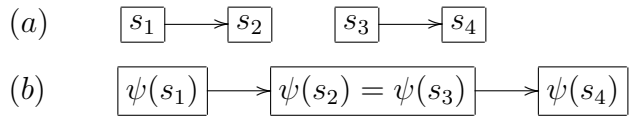


Figure 1: (a) Original state space and (b) abstract space with spurious state $\psi(s_1)$.

The occurrence of such states is known to the heuristic search and planning community, and reported for instance by Holte and Hernádvölgyi [HH04] and by Haslum *et al.* [HBH$^+$07]. Techniques like constrained abstraction, as proposed in a different paper by Haslum *et al.* [HBG05], can sometimes reduce the number of spurious states. 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.

This paper formally analyzes the problem of avoiding spurious states in abstraction. To this end we first introduce a common language for representing planning and search domains. In this formal framework we then introduce the "Downward Path Preserving" (DPP) property to refer to state space abstractions that do not contain any spurious states.

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.[2] However, this paper addresses two specific types of abstraction: projection [Ede01] and domain abstraction [HH00]. 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 are defined informally as follows.

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

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

[2]Formal definitions of abstractions in general and specific types of abstractions will be given in Section 3.2.

Intractability results. Our main contribution 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.

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.

Tractability results. The negative results on the computational complexity are mitigated by tractability results for special cases—our second contribution. We identify simple formal conditions on state spaces that allow DPP abstractions to be easily constructed.

Some problem domains turn out to be representable in a way that satisfies these conditions even though other natural ways of representing them do not match the formal conditions of our theorems, and, in fact, do not allow for DPP abstractions at all. This is illustrated with a variant of the Blocks World planning problem. This shows that the way a problem domain is encoded is crucial for the design of DPP abstractions. Given two equivalent encodings, 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 practical design guidelines for representing state spaces.

2. Motivating Example

In this section we illustrate that spurious states are of practical concern—they can arise when standard abstraction methods are applied to typical planning problems and are not entirely eliminated by standard "mutex" methods. In this example, the problem domain is the Blocks World with table positions, a variation of the standard Blocks World domain in which there are a fixed number of named (*i.e.,* distinguishable) table positions, each of which can hold one stack of blocks. There are four 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. We consider the version of this domain with seven blocks and four table positions, which has 604,800 states that are reachable from any given state.

The standard STRIPS encoding of this domain has predicates encoding what is on each of the four table positions `P1` through `P4`, as well as

what is on top of each of the blocks B1 through B7. A typical action is Move-B4-from-B5-to-B3, which moves B4 from being on top of B5 to being on top of B3. It is represented as follows:

Preconditions: clear-B4=true, clear-B3=true, on-B4-B5=true
Effects: on-B4-B3=true, clear-B5=true, on-B4-B5=false, clear-B3=false

Projection is the standard method for abstracting propositional STRIPS encodings. If, in this example, we project out the predicates indicating what is on top of blocks B1 through B4, the 604,800 reachable states in the original state space get mapped to just 89,400 abstract states, but they are intermingled with 1,221,320 spurious states, resulting in an abstract space containing a total of 1,310,720 reachable abstract states—more than twice as many states as there are in the original state space.

To illustrate how spurious states get created in this example, consider the abstract state α in which the following predicates are true and all others are false:

> clear-P1, clear-P2, clear-P3, on-B7-P4, on-B6-B7, on-B5-B6, on-B4-B5

This abstract state is not spurious: any normal Blocks World state in which blocks B1 through B3 are sitting at the top of the stack in position P4, in any order, would map to this state when the predicates indicating what is on top of blocks B1 through B4 are projected out. This projection also converts Move-B4-from-B5-to-B3 to the following abstract operator:

> **Preconditions:** on-B4-B5=true
> **Effects:** clear-B5=true, on-B4-B5=false

Although Move-B4-from-B5-to-B3 is not applicable to any normal Blocks World state that maps to α, this abstraction of Move-B4-from-B5-to-B3 can be applied to α. Doing so produces the abstract state in which the following predicates are true and all others are false:

> clear-P1, clear-P2, clear-P3, on-B7-P4, on-B6-B7, on-B5-B6, clear-B5

This abstract state does not correspond to any reachable state in the normal Blocks World since the blocks B1, B2, B3, and B4 must be floating in the air (they can't be on P1, P2, P3, or B5 (the top block in the stack at P4) because those are all known to be clear).

Filtering Method	# Abstract States	Average h
No Filtering	1,310,720	7.10012
Automatic "Mutex" Filtering	90,941	7.10012
Handcrafted "Mutex" Filtering	90,000	7.16217
Complete Filtering	89,400	7.21264

Table 1: Number of abstract states and average heuristic value for the abstraction described in the text. We show the average heuristic value only for the 89,400 non-spurious abstract states (the abstractions of the reachable states in the original state space).

Table 1 shows how the number of abstract states, and the average heuristic value of the 89,400 non-spurious abstract states, are affected by various methods for reducing the number of spurious states. The first row gives the data when no spurious states are filtered out, and the last row shows the data when all the spurious states are filtered out. Complete elimination of spurious states reduces the number of abstract states by more than an order of magnitude and increases the average heuristic value of the 89,400 non-spurious states from 7.10012 to 7.21264.

The second row of Table 1 shows the effects of a standard "mutex" method for detecting (and therefore eliminating) spurious states [BG01]. This method works by automatically computing pairwise mutual exclusions, *i*.e., combinations of value assignments to two of the state variables that cannot possibly occur in a state from which the goal state can be reached. Note that in general there are no efficient methods known for finding *all* such mutual exclusions. The standard mutex method eliminates the vast majority of the spurious states: only 1,541 spurious states remain after it is applied. Note, however, that the average heuristic value of the 89,400 non-spurious abstract states has not changed at all—the spurious states eliminated by the standard mutex method were not the ones creating shortcuts in the abstract space.

The third row of Table 1 shows that 941 of the remaining spurious states are eliminated by the use of additional mutex filters that we handcrafted specially for this domain encoding. The heuristic value of the 89,400 non-spurious abstract states is reduced, but is still slightly larger than the heuristic value when all spurious states are filtered out. This small difference in the heuristic value has a significant effect on the speed of search. When all the spurious states are filtered out, IDA* expands only 287,954 nodes (on

average, over 100 randomly generated start states with an average solution length of 10.95), whereas with the spurious states remaining after the mutex methods were applied, it expands 319,325 nodes (see Table 2).

This example demonstrates that the standard abstraction methods applied to typical problems can introduce a large number of spurious states, and that even if standard automatic methods can eliminate most of the spurious states those that remain can degrade search performance.

Filtering Method	Avg. Nodes Expanded
Automatic "Mutex" Filtering	361,861
Handcrafted "Mutex" Filtering	319,325
Complete Filtering	287,954

Table 2: Number of nodes expanded by IDA* with different filtering methods.

3. Formal model of downward path preserving state space abstractions

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

The following notation will be used for this purpose.

If A is a finite set then $|A|$ denotes the cardinality of A. The symbol $\emptyset$ denotes 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 $\psi(\sigma_1, \ldots, \sigma_n) = (\psi_0(\sigma_1), \ldots, \psi_0(\sigma_n))$ for all $\sigma_1, \ldots, \sigma_n \in \Sigma$. We can then identify ψ with ψ_0. For convenience, slightly abusing notation, we will use the same function symbol (ψ) for both the mapping from Σ^n to Γ^n and for the mapping from Σ to Γ.

3.1. State space representation

Usually a state space represents 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* $\pi = (s_1, s_2, \ldots, s_z) \in (\Sigma^n)^+$ *such that* $s_1 = s$*,* $s_z = s'$*, and* $(s_i, s_{i+1}) \in \Pi$ *for all* $i \in \{1, \ldots, z-1\}$*. Define*

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

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 also not uncommon 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 [HH99, HH00]; 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

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

means that every state s that 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 $j \in \{1, \ldots, n\}$. p is called the precondition of the operator, p' is called the postcondition. 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') (with $s = \langle \sigma_1, \ldots, \sigma_n\rangle \in \Sigma^n$ and $s' = \langle \sigma'_1, \ldots, \sigma'_n\rangle \in \Sigma^n$) matches $p \to p'$ if s matches p, s' matches p', and for all $i, j \in \{1, \ldots, n\}$

$$y'_i = y_j \Rightarrow \sigma'_i = \sigma_j\,.$$

(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. Our implicit (PSVN) representation is expressive enough to model state spaces defined in propositional STRIPS [FN71] or SAS$^+$ [Bäc92] notation. In particular, any propositional STRIPS or SAS$^+$ domain encoding or plan existence problem can be transformed into a similar one in our notation in time and space polynomial in the size of the input.
2. Although it is not relevant for our analysis, we note that PSVN can be used to more compactly represent certain state spaces than SAS$^+$ or propositional STRIPS. For instance, in order to represent the PSVN operator $\langle x_1, \ldots, x_n\rangle \to \langle x_2, \ldots, x_n, x_1\rangle$ in SAS$^+$, a separate operator for every possible combination of values of the n state variables would be required.

3. 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.* [McD00].

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] All the 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 Γ is a proper subset of Σ. 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 $|\Gamma| = 1$.

[3] This coincides with a previously studied notion of homomorphism for abstraction mappings [HHH07].

[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.

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 Section 2. For any given original state s^* (not necessarily the start state or goal state), a spurious state is an abstract state that is reachable from the abstract image of s^* but has no pre-image in the original state space that is reachable from s^*.

Definition 5. *Let $\mathcal{S} = (\Sigma, n, \Pi)$ be a state space, $\Gamma \subseteq \Sigma$, and $m \in \mathbb{N}$. Let $\psi : \Sigma^n \to \Gamma^m$ be an abstraction mapping and let $s^* \in \Sigma^n$, $t \in \Gamma^m$. t is called spurious with respect to s^* if $t \in \Delta(\psi(s^*), \mathcal{S}_\psi) \setminus \psi(\Delta(s^*, \mathcal{S}))$.*

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 regression search backwards from the goal state (*e.g.*, to build a pattern database), one should think 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 $\psi(s^*)$ in the transpose of the given state space. This is the perspective taken in Sections 1 and 2.

It is important to distinguish between spurious states, as we have defined them here, and the "spurious" states that are often generated during regression search [BG01]. The latter are defined as states that are reachable from the goal (in the transpose of the state space) but from which the start state cannot be reached. A more apt name for them would be "deadend" states. They have no intrinsic relation to spurious states, in our sense, since they are defined in terms of two states within a single state space, whereas spurious states, in our sense, are defined in terms of one state (s^*) and two states spaces (the original and the abstract space). Throughout this paper we use the term "spurious" to refer only to spurious states as we have defined them.

For any fixed start state s^* (or goal state s^* in the transpose of the state space), an abstraction ψ avoiding 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 6. *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*

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

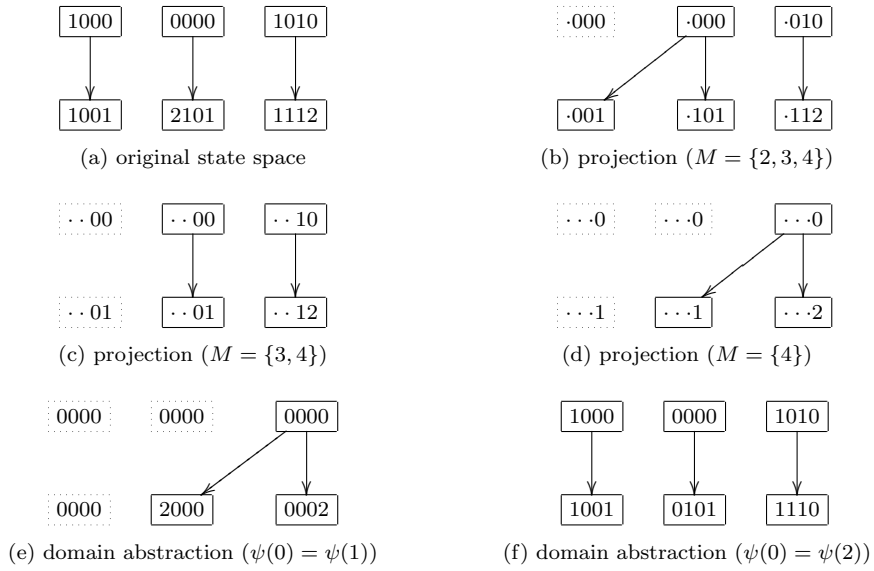
for all $s \in \Sigma^n$. For any particular state $s^* \in \Sigma^n$ the abstraction ψ is DPP for s^* and $\mathcal{S}$ if $\psi(\Delta(s^*, \mathcal{S})) = \Delta(\psi(s^*), \mathcal{S}_\psi)$.

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.

4. Computational complexity of finding DPP abstractions

In this section we formally analyze the complexity of the problem of generating DPP abstractions. This problem is addressed in several very general versions, which turn out to be intractable. Later, Section 5 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 state space encoding matters for the design of DPP abstractions, an issue that is further discussed in Section 6.

There are three decision problems of immediate relevance for the automatic construction of DPP abstractions. Their computational complexity is the topic of this section.



(a) original state space (b) projection ($M = \{2, 3, 4\}$)

(c) projection ($M = \{3, 4\}$) (d) projection ($M = \{4\}$)

(e) domain abstraction ($\psi(0) = \psi(1)$) (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 in the connected component shown.

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 propositional STRIPS notation.

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 give results on the computational complexity of these three decision problems. Proofs for all theorems in this section are given in the Appendix.

The first result is just a review and mainly due to [Bäc92].

Theorem 1. 1. *REACHABLE is PSPACE-complete 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.*

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 preceding reducibility results are the main ingredients in the proofs of the following complexity properties of the IS-DPP problem.

Corollary 3. 1. *IS-DPP is PSPACE-complete 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.

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-complete 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-complete for the case of domain abstraction and either type of state space representation if the dimension n is fixed with* $n > 2$ a priori.

The reader might be concerned that the 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 6. However, all 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^*.

Corollary 5. 1. *IS-DPP$_{s^*}$ is PSPACE-complete for the case of implicit state space representation and either type of abstraction.*
2. *EXIST-DPP$_{s^*}$ is PSPACE-complete for the case of projection and implicit state space representation.*
3. *EXIST-DPP$_{s^*}$ is NP-complete for the case of domain abstraction and either type of state space representation if the dimension n is fixed with* $n > 2$ a priori.

5. Sufficient conditions for the existence of DPP abstractions

The preceding intractability results show that the general problem of avoiding abstractions with spurious states is hard, but this does not preclude there being special circumstances in which the DPP property can be easily tested and perhaps even guaranteed.

This section provides easily testable criteria that help us to design DPP projections and domain abstractions for given state spaces (in case such abstractions exist at all), and gives examples of encodings of typical problem domains that meet these easily testable criteria. By "encoding of a problem domain" we mean the choice of a state space (Σ, n, Π) for the problem domain.

Later on, we give an example of a typical planning problem for which the usual encoding does not meet these criteria, but which 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 state space encodings of problem domains.

For the ease of notation, we extend the definition of domain abstraction and projection mappings ψ to state patterns as follows. Let Σ be a finite alphabet, $\Gamma \subseteq \Sigma$, $X = \{x_i \mid i \in \mathbb{N}\}$ a set of variables disjoint with Σ, $n, m \in \mathbb{N}$ with $m \leq n$. If $\psi : \Sigma^n \to \Gamma^n$ is a domain abstraction mapping, then we set $\psi(x_i) = x_i$ for all $i \in \mathbb{N}$ and define $\psi(\langle y_1, \ldots, y_n \rangle) = \langle \psi(y_1), \ldots, \psi(y_n) \rangle$. If $\psi : \Sigma^n \to \Sigma^m$ is a projection mapping with $\psi(\langle \sigma_1, \ldots, \sigma_n \rangle) = \langle \sigma_{i_1}, \ldots, \sigma_{i_m} \rangle$, then we define $\psi(\langle y_1, \ldots, y_n \rangle) = \langle y_{i_1}, \ldots, y_{i_m} \rangle$. For $t, t' \in \Gamma^m$, we define the meaning of "(t, t') matches $\psi(\langle y_1, \ldots, y_n \rangle) \to \psi(\langle y'_1, \ldots, y'_n \rangle)$" by analogy with Definition 3. Note that for projections $\psi(\langle y_1, \ldots, y_n \rangle) \to \psi(\langle y'_1, \ldots, y'_n \rangle)$ is not necessarily an operator in the sense of Definition 3, because there might be an $i \in \{i_1, \ldots, i_m\}$ for which y'_i is a variable but $y'_i \notin \{y_{i_1}, \ldots, y_{i_m}\}$.

We observe the following fact.

Lemma 6. *Let $\mathcal{S} = (\Sigma, n, O)$ be a state space representation, ψ any projection or domain abstraction mapping defined over Σ^n, and $t, t' \in \psi(\Sigma^n)$. If $(t, t') \in \Pi_\psi$ then there is some $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$ such that (t, t') matches $\psi(\langle y_1, \ldots, y_n \rangle) \to \psi(\langle y'_1, \ldots, y'_n \rangle)$.*

Proof. Since $(t, t') \in \Pi_\psi$ and Π_ψ is minimal, there is some $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$ and some $s = \langle \sigma_1, \ldots, \sigma_n \rangle$, $s' = \langle \sigma'_1, \ldots, \sigma'_n \rangle \in \Sigma^n$ such that $\psi(s) = t$, $\psi(s') = t'$, and (s, s') matches o. $\psi(s) = t$ and $\psi(s') = t'$ implies that t matches $\psi(\langle y_1, \ldots, y_n \rangle)$ and t' matches $\psi(\langle y'_1, \ldots, y'_n \rangle)$. Since (s, s') matches o, we obtain $[y'_i = y_j \Rightarrow \sigma'_i = \sigma_j]$ for all $i, j \in \{1, \ldots, n\}$, and therefore, in particular, (t, t') matches $\psi(\langle y_1, \ldots, y_n \rangle) \to \psi(\langle y'_1, \ldots, y'_n \rangle)$. □

This allows us to formally define a property that is sufficient for an abstraction to be DPP. Later on, we will show how to use this property as a design principle for creating abstractions in the special cases of projection and domain abstraction.

Definition 7. *Let $\mathcal{S} = (\Sigma, n, O)$ be a state space representation, ψ any projection or domain abstraction mapping defined over Σ^n, and $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$. ψ is precondition-preserving for o if $\psi(\langle y_1, \ldots, y_n \rangle) \to$*

$\psi(\langle y'_1, \ldots, y'_n \rangle)$ is an operator and for all $s \in \Sigma^n$ and all $t \in \psi(\Sigma^n)$ the following condition holds.

$$\textit{If } \psi(s) = t \textit{ and } t \textit{ matches } \psi(\langle y_1, \ldots, y_n \rangle) \textit{ then } s \textit{ matches } \langle y_1, \ldots, y_n \rangle.$$

ψ is precondition-preserving if ψ is precondition-preserving for all $o \in O$.

The abstraction in Section 2 is an example of an abstraction that is not precondition-preserving. When the preconditions of `Move-B4-from-B5-to-B3` are abstracted, they are matched by the abstract state α but there are states the abstraction maps to α that do not satisfy the preconditions of `Move-B4-from-B5-to-B3`.

The following theorem states that preserving preconditions of operators is sufficient for creating DPP abstractions.

Theorem 7. *Let $\mathcal{S} = (\Sigma, n, O)$ be a state space representation and ψ a projection or domain abstraction mapping defined over Σ^n. If ψ is precondition-preserving then ψ is DPP.*

Proof. Let $\mathcal{S}_\psi = (\Gamma, m, \Pi_\psi)$. According to Definition 6, 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 follows inductively from the next fact, which we are going to prove below.

Fact. For every state $s \in \Sigma^n$ and for every two abstract states $t, t' \in \psi(\Sigma^n)$ with $\psi(s) = t$, the following holds. If $(t, t') \in \Pi_\psi$, then there is a state $s' \in \Sigma^n$ and an operator $o \in O$ with

(i) $\psi(s') = t'$, and
(ii) (s, s') matches o.

To prove this fact, let $s = \langle \sigma_1, \ldots, \sigma_n \rangle \in \Sigma^n$. Let $t = \langle \tau_1, \ldots, \tau_m \rangle \in \Gamma^m$ with $\psi(s) = t$. Suppose $t' = \langle \tau'_1, \ldots, \tau'_m \rangle \in \Gamma^m$ is an abstract state and $(t, t') \in \Pi_\psi$. By Lemma 6 there is some operator $o = \langle y_1, \ldots, y_n \rangle \rightarrow \langle y'_1, \ldots, y'_n \rangle \in O$ such that (t, t') matches $\psi(\langle y_1, \ldots, y_n \rangle) \rightarrow \psi(\langle y'_1, \ldots, y'_n \rangle)$.

Because ψ is precondition-preserving, together with the fact that t matches $\psi(\langle y_1, \ldots, y_n \rangle)$, we know that s matches $\langle y_1, \ldots, y_n \rangle$ and $\psi(\langle y_1, \ldots, y_n \rangle) \rightarrow \psi(\langle y'_1, \ldots, y'_n \rangle)$ is an operator.

Since s matches the precondition $\langle y_1, \ldots, y_n \rangle$ 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 $\langle y'_1, \ldots, y'_n \rangle$ and $[y'_i = y_j \Rightarrow \sigma'_i = \sigma_j]$ for all $i, j \in \{1, \ldots, n\}$. This implies that $\psi(s')$ matches $\psi(\langle y'_1, \ldots, y'_n \rangle)$. Note that t' matches $\psi(\langle y'_1, \ldots, y'_n \rangle)$ as well.

Finally, we prove $t' = \psi(s')$. Let $\langle z_1, \ldots, z_m \rangle = \psi(\langle y_1, \ldots, y_n \rangle)$ and $\langle z'_1, \ldots, z'_m \rangle = \psi(\langle y'_1, \ldots, y'_n \rangle)$. Let $\psi(s') = \langle \xi_1, \ldots, \xi_m \rangle$ and let $i \in \{1, \ldots, m\}$. If z'_i is a constant, because both t' and $\psi(s')$ match $\psi(\langle y'_1, \ldots, y'_n \rangle)$, it immediately follows that $\tau'_i = \xi_i$ since both will be equal to z'_i. Now consider the case when z'_i is a variable. Because $\psi(\langle y_1, \ldots, y_n \rangle) \rightarrow \psi(\langle y'_1, \ldots, y'_n \rangle)$ is an operator, z'_i must occur in the pattern $\psi(\langle y_1, \ldots, y_n \rangle) = \langle z_1, \ldots, z_m \rangle$ and thus also in the pattern $\langle y_1, \ldots, y_n \rangle$. Thus there is a $k \in \{1, \ldots, n\}$ such that $z'_i = y_k$ and ψ maps the component σ_k to τ'_i.[5] Similarly, there is a $\hat{k} \in \{1, \ldots, n\}$ such that $z'_i = y_{\hat{k}}$ and ψ maps the component $\sigma_{\hat{k}}$ to ξ_i.[6] Since $y_k = z'_i = y_{\hat{k}}$ and s matches $\langle y_1, \ldots, y_n \rangle$, we obtain $\sigma_k = \sigma_{\hat{k}}$ and therefore $\tau'_i = \xi_i$. Hence we see that $\tau'_i = \xi_i$ whether z'_i is a constant or a variable. Since the choice of i was arbitrary, we have $t' = \psi(s')$.

This proves the fact and thus the theorem. □

With these general definitions and results in place, it is now just a matter of working out what it means for each specific type of abstraction to be precondition-preserving.

5.1. Projection

An example of a typical problem domain in which certain types of projections in typical encodings obviously yield DPP abstractions is Rubik's Cube [Kor97]. In standard encodings of this problem domain either one of the following projections is DPP.

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

[5] This means $\psi(\sigma_k) = \tau'_i$ in the case of domain abstraction and $\sigma_k = \tau'_i$ in the case of projection.

[6] This means $\psi(\sigma_{\hat{k}}) = \xi_i$ in the case of domain abstraction and $\sigma_{\hat{k}} = \xi_i$ in the case of projection.

- 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 some of the variables encoding the corner cubies (or only some of the variables encoding the 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.

Theorem 8. *Let (Σ, n, O) be a representation for a state space. Suppose there is an $M \subseteq \{1, \ldots, n\}$ such that, for all $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$, the following three conditions are satisfied for all $i, j \in \{1, \ldots, n\}$.*

1. *If $i \in M$ and $y'_i \notin \Sigma$ then $y'_i \in \{y_k \mid k \in M\}$.*
2. *If $y_i \in \Sigma$ then $i \in M$.*
3. *If $i \neq j$ and $y_i = y_j$ then both $i \in M$ and $j \in M$.*

Then the projection defined by M is precondition-preserving and therefore DPP.

Proof. Let $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$ and $s = \langle \sigma_1, \ldots, \sigma_n \rangle \in \Sigma^n$. We have to prove that

1. $\psi(\langle y_1, \ldots, y_n \rangle) \to \psi(\langle y'_1, \ldots, y'_n \rangle)$ is an operator.
2. If $\psi(s) = t$ and t matches $\psi(\langle y_1, \ldots, y_n \rangle)$ then s matches $\langle y_1, \ldots, y_n \rangle$.

The first property follows immediately from the first condition.

For the second property, assume that $\psi(s) = t$, t matches $\psi(\langle y_1, \ldots, y_n \rangle)$, but s does not match $\langle y_1, \ldots, y_n \rangle$. Let $M = \{i_1, \ldots, i_m\}$. Then $t = \langle \sigma_{i_1}, \ldots, \sigma_{i_m} \rangle$ and $\psi(\langle y_1, \ldots, y_n \rangle) = \langle y_{i_1}, \ldots, y_{i_m} \rangle$. Since s does not match $\langle y_1, \ldots, y_n \rangle$, one of the following two cases must occur.

Case (a). There is an $i \in \{1, \ldots, n\}$ with $y_i \in \Sigma$ and $\sigma_i \neq y_i$.

Case (b). There are $i, j \in \{1, \ldots, n\}$ with $i \neq j$, $y_i = y_j$, and $\sigma_i \neq \sigma_j$.

Assume Case (a) occurs. Since $y_i \in \Sigma$ we have $i \in M$ by the second condition. $\sigma_i \neq y_i$ then implies that t does not match $\psi(\langle y_1, \ldots, y_n \rangle)$, which is a contradiction.

So Case (b) occurs. Since $i \neq j$ and $y_i = y_j$, both i and j belong to M (by the third condition). $\sigma_i \neq \sigma_j$ then implies that t does not match $\psi(\langle y_1, \ldots, y_n \rangle)$. Again we obtain a contradiction.

Hence s matches $\langle y_1, \ldots, y_n \rangle$, which proves the second property. □

For illustration, consider again Rubik's Cube. Every variable in the standard state space encoding of this problem domain [Kor97] 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). Either one of these sets can take the role of M in Theorem 8. Therefore we can project out either all corner cubie variables or all edge cubie variables without violating the DPP property.

Note that a minor adaptation of the proof of Theorem 8 yields the following observation.

Corollary 9. *Let (Σ, n, O) be a representation for a state space. Suppose there is an $M = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ such that, for each $o = \langle y_1, \ldots, y_n \rangle \rightarrow \langle y'_1, \ldots, y'_n \rangle \in O$, either $\langle y_{i_1}, \ldots, y_{i_m} \rangle = \langle y'_{i_1}, \ldots, y'_{i_m} \rangle$ or the following three conditions are satisfied for all $i, j \in \{1, \ldots, n\}$.*

1. *If $i \in M$ and $y'_i \notin \Sigma$ then $y'_i \in \{y_k \mid k \in M\}$.*
2. *If $y_i \in \Sigma$ then $i \in M$.*
3. *If $i \neq j$ and $y_i = y_j$ then both $i \in M$ and $j \in M$.*

Then the projection defined by M is DPP.

A situation in which Corollary 9 applies and Theorem 8 does not apply is a standard encoding of the Towers of Hanoi. In this puzzle, one has d disks of strictly increasing size, which can be distributed and stacked onto p distinguishable pegs, where the only condition is that no disk is placed on top of a smaller disk.

Assuming $d = 2$ and $p = 3$, we can represent the puzzle with the state space (Σ, n, O) with $\Sigma = \{1, 2, 3\}$ and $n = 2$. A state $s = (a, b)$ indicates that the smaller disk is placed on peg a and the larger disk is placed on peg b, for $a, b \in \Sigma$. The set O would then contain the following operators.

(moving the smaller disk)	(moving the larger disk)
$\langle 1, x_1\rangle \to \langle 2, x_1\rangle$	$\langle 1, 2\rangle \to \langle 1, 3\rangle$
$\langle 1, x_1\rangle \to \langle 3, x_1\rangle$	$\langle 1, 3\rangle \to \langle 1, 2\rangle$
$\langle 2, x_1\rangle \to \langle 1, x_1\rangle$	$\langle 2, 1\rangle \to \langle 2, 3\rangle$
$\langle 2, x_1\rangle \to \langle 3, x_1\rangle$	$\langle 2, 3\rangle \to \langle 2, 1\rangle$
$\langle 3, x_1\rangle \to \langle 1, x_1\rangle$	$\langle 3, 1\rangle \to \langle 3, 2\rangle$
$\langle 3, x_1\rangle \to \langle 2, x_1\rangle$	$\langle 3, 2\rangle \to \langle 3, 1\rangle$

Theorem 8 would not help to create DPP abstractions since the operators moving the larger disk contain constants in every position, *i.e.*, the second condition of Theorem 8 would require the constructed projection to keep both state variables.

However, if one projects out the second state variable, which represents the location of the larger disk, the operators moving the larger disk all become identity operators, so they are harmless. For the operators moving the smaller disk the original conditions of Theorem 8 still apply. Hence Corollary 9 says that ignoring the second state variable yields a DPP projection.

5.2. Domain abstraction

A typical example in which the commonly used domain abstractions never violate the DPP property is the sliding-tile puzzle, *cf.* [SS06].

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 the names of some 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 this example can be generalized to a criterion for DPP domain abstractions, let us look at how operators are typically defined for this problem domain. For the 8-puzzle, for instance, one typically uses 9 variables. For each of the 9 puzzle 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, by the same reasoning, 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.

Theorem 10. *Let (Σ, n, O) be a representation for a state space. Suppose there is a set $\Sigma_0 \subseteq \Sigma$ such that for all operators $o = \langle y_1, \ldots, y_n\rangle \to \langle y'_1, \ldots, y'_n\rangle \in O$ and all $i, j \in \{1, \ldots, n\}$ the following two conditions are fulfilled.*

1. *If $y_i \in \Sigma$ then $y_i \in \Sigma_0$.*
2. *If $y_i = y_j$ and $i \neq j$ then $y_i \in \Sigma_0$.*

Then every domain abstraction ψ with $\psi(\sigma) = \sigma$ for all $\sigma \in \Sigma_0$ and $\psi(\sigma) \notin \Sigma_0$ for all $\sigma \in \Sigma \setminus \Sigma_0$ is precondition-preserving and therefore DPP.

Proof. Let $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$ and $s = \langle \sigma_1, \ldots, \sigma_n \rangle \in \Sigma^n$. We have to prove that

1. $\psi(\langle y_1, \ldots, y_n \rangle) \to \psi(\langle y'_1, \ldots, y'_n \rangle)$ is an operator.
2. If $\psi(s) = t$ and t matches $\psi(\langle y_1, \ldots, y_n \rangle)$ then s matches $\langle y_1, \ldots, y_n \rangle$.

For the first property, we must show, for each $i \in \{1...n\}$, that either $\psi(y'_i) \in \psi(\Sigma)$ or $\psi(y'_i) = \psi(y_j)$ for some $j \in \{1...n\}$. If $y'_i \in \Sigma$ then $\psi(y'_i) \in \psi(\Sigma)$. On the other hand, if $y'_i \notin \Sigma$ then because o is an operator, $y'_i = y_j$ for some $j \in \{1...n\}$, and the required equality follows because domain abstractions map variables to themselves, *i.*e., $\psi(y'_i) = y'_i$ and $\psi(y_j) = y_j$.

For the second property, assume that $\psi(s) = t$, t matches $\psi(\langle y_1, \ldots, y_n \rangle)$, but s does not match $\langle y_1, \ldots, y_n \rangle$. Since $t = \langle \psi(\sigma_1), \ldots, \psi(\sigma_n) \rangle$ and $\psi(\langle y_1, \ldots, y_n \rangle) = \langle \psi(y_1), \ldots, \psi(y_n) \rangle$, we know that $\psi(y_i) \in \Sigma$ implies $\psi(y_i) = \psi(\sigma_i)$, where $i \in \{1, \ldots, n\}$.

Since s does not match $\langle y_1, \ldots, y_n \rangle$, one of the following two cases must occur.

Case (a). There is an $i \in \{1, \ldots, n\}$ with $y_i \in \Sigma$ and $\sigma_i \neq y_i$.

Case (b). There are $i, j \in \{1, \ldots, n\}$ with $i \neq j$, $y_i = y_j$, and $\sigma_i \neq \sigma_j$.

Assume Case (a) occurs. Since $y_i \in \Sigma$ we have $y_i \in \Sigma_0$ by the first condition. Then $\psi(y_i) = y_i$ because of the choice of ψ. $y_i \in \Sigma$ also implies $\psi(y_i) = \psi(\sigma_i)$ by our reasoning above. Hence $\psi(\sigma_i) = y_i \in \Sigma_0$. As $\psi(\sigma_i) \in \Sigma_0$, the choice of ψ implies that $\sigma_i \in \Sigma_0$ and hence $\sigma_i = \psi(\sigma_i) = y_i$, which is a contradiction.

So Case (b) occurs. Since $i \neq j$ and $y_i = y_j$, our second condition requires $y_i \in \Sigma_0$ and hence $\psi(y_i) = \psi(y_j) = y_i = y_j$. We know that $\psi(y_i) = \psi(\sigma_i)$ and $\psi(y_j) = \psi(\sigma_j)$ and thus $\psi(\sigma_i) = \psi(\sigma_j)$. By the same reasoning as in Case (a) we obtain $\psi(\sigma_i) = \sigma_i$ and $\psi(\sigma_j) = \sigma_j$ and therefore $\sigma_i = \sigma_j$, which is a contradiction.

Hence s matches $\langle y_1, \ldots, y_n \rangle$, which proves the second property. □

For the standard representation of the 8-puzzle, this theorem would apply to $\Sigma_0 = \{B\}$. 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 maps the blank symbol to itself will be DPP.

Similarly to the projection case, a minor adaptation of the proof of Theorem 10 yields a slightly stronger statement.

Corollary 11. *Let (Σ, n, O) be a representation for a state space. Suppose there is a set $\Sigma_0 \subseteq \Sigma$ and a domain abstraction ψ with $\psi(\sigma) = \sigma$ for all*

$\sigma \in \Sigma_0$ and $\psi(\sigma) \notin \Sigma_0$ for all $\sigma \in \Sigma \setminus \Sigma_0$ such that, for each operator $o = \langle y_1, \ldots, y_n \rangle \to \langle y'_1, \ldots, y'_n \rangle \in O$, either $\psi(\langle y_1, \ldots, y_n \rangle) = \psi(\langle y'_1, \ldots, y'_n \rangle)$ or the following two conditions are satisfied for all $i, j \in \{1, \ldots, n\}$.

1. *If $y_i \in \Sigma$ then $y_i \in \Sigma_0$.*
2. *If $y_i = y_j$ and $i \neq j$ then $y_i \in \Sigma_0$.*

Then ψ is DPP.

6. Encoding matters

As Theorems 8 and 10 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. 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 [Ama68, Ben89, Kor80, Low88, Van92]).

We now give an example of a planning domain, the Blocks World with table positions that was introduced in Section 2, illustrating that, if one deviates from standard STRIPS encodings, quite intuitive encodings that match Theorem 10 can be found. The lesson is that a careful design of problem domain encodings may make the design of DPP abstractions easier.

The standard STRIPS encoding of the Blocks World with table positions does not match the provably sufficient conditions allowing for DPP abstractions as given in Theorems 8 and 10, and, indeed, as was seen in Section 2, projections on this encoding do introduce spurious states.[7]

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

[7]Domain abstraction of a STRIPS-encoded state space is the same as projection since all variables are binary-valued, so there is no known tractable way to automatically get DPP abstractions in this problem domain—at least for the standard encoding.

natural way such that Theorem 10 becomes applicable, *i.e.*, DPP domain abstractions can be easily defined.

The encoding for k blocks and q 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 defined by a set O of operators (described below).

Intuitively, a state has q sections of $k+1$ variables each. The p^{th} section represents the stack of blocks in table position p, where zeroes 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 in which they are stacked 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 $k-3$ zeroes.

The set O has a total of $q(q-1)k(k+1)/2$ operators. There are $q(q-1)$ pairs of different table positions, and for each pair there is one operator for every pair of numbers (h_i, h_j) with $0 < h_i \leq k$, $0 \leq h_j < k$, and $h_i + h_j \leq k$, where h_i is the number of blocks stacked on position i and h_j is the number of blocks stacked on position j; the operator moves the topmost block on the (non-empty) stack in position i to the top of the (non-full) stack in position j.

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

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

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

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

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

Now obviously the names of blocks do not occur in the precondition of any operator; moreover, no variable occurs twice in the precondition of any operator. Hence Theorem 10 is applicable where

$$\Sigma_0 = \{0, 1, \ldots, k\}\,.$$

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

It is important to note that a DPP abstraction does not guarantee that the heuristic produced by the abstraction is superior to the heuristic produced by a non-DPP abstraction of a different state-space representation. However, a DPP abstraction does guarantee that the heuristic values are not artificially reduced by shortcuts and, if pattern databases are being used, it guarantees that no preprocessing time or memory is wasted in generating and storing spurious states.

7. Related work

In the heuristic search literature, the problem of violating the DPP property has been addressed by Holte and Hernádvölgyi [HH04] 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. [HBG05] report the problem of heuristic values being too small because of spurious states and experimentally show that enforcing "mutex" constraints sometimes substantially speeds up A*. Most recently Haslum et al. [HBH$^+$07]—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

mutex methods 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 [BY91, BY94]. 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.

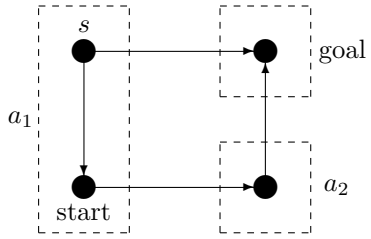


Figure 4: A state space and abstraction that is DPP but not DRP.

Figure 4 shows a state space (the dark circles are its states) and an abstraction (dotted boxes) that is DPP but not DRP. There is an abstract solution that uses one operator to go directly from the abstract start state a_1 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 a_2, 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 5, some studies of DRP look for easily tested properties of a state space that guarantee DRP. The original DRP paper [BY91] 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 [Has07], which also points out, as we have done in Section 5, 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.

8. 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.

We studied the problem of avoiding spurious states on a formal level, our main contributions being as follows.

- We introduced a formal criterion, called the downward path-preserving (DPP) property, that defines abstractions that do not give rise to 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 Blocks World variant with table positions 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 8 and 10 yield very promising criteria for the design of problem domain encodings, yet they have strong limitations. It unlikely that every typical benchmark domain can be encoded in a way that our theorems apply, and, even if we knew that a given state space 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.

While DPP abstractions do not in general guarantee better search performance than non-DPP abstractions, they have the desirable property that the resulting heuristic values are not artificially reduced by shortcuts and, in the case of memory-based heuristics, do not waste time or memory in generating and storing spurious states.

Acknowledgements

We thank the anonymous referees for their careful reading of a previous version of this paper. Their suggestions improved the paper substantially. In particular, we would like to thank the anonymous reviewer who suggested refactoring our proofs of Theorems 8 and 10 along the current lines. We would also like to thank Marcel Ball for providing the experimental results in Section 2.

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Ingenuity Centre for Machine Learning (AICML).

References

[Ama68] Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence*, volume 3, pages 131–171. Edinburgh University Press, 1968.

[Bäc92] Christer Bäckström. Equivalence and tractability results for SAS^+ planning. In *Proceedings of the 3rd International Conference on Principles on Knowledge Representation and Reasoning*, pages 126–137, 1992.

[Ben89] D. Paul Benjamin. *Change of Representation and Inductive Bias*. Kluwer Academic Publishers, Norwell, MA, USA, 1989.

[BG01] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[BN95] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656, 1995.

[BY91] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 286–293. Morgan Kaufmann, 1991.

[BY94] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[CS96] Joseph Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Proceedings of the 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, volume 1081 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 1996.

[CS98] Joseph Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Ede01] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the European Conference on Planning*, pages 13–24, 2001.

[FN71] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[Has07] Patrik Haslum. Reducing accidental complexity in planning problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1898–1903, 2007.

[HBG05] Patrik Haslum, Blai Bonet, and Hector Geffner. New admissible heuristics for domain-independent planning. In *Proceedings of the 20th AAAI Conference on Artificial Intelligence*, pages 1163–1168, 2005.

[HBH+07] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 1007–1012, 2007.

[HG00] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems*, pages 140–149, 2000.

[HH99] István Hernádvölgyi and Robert Holte. PSVN: A vector representation for production systems. Technical Report TR-99-04, Department of Computer Science, University of Ottawa, 1999.

[HH00] István Hernádvölgyi and Robert C. Holte. Experiments with automatically created memory-based heuristics. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation*, volume 1864 of *Lecture Notes in Artificial Intelligence*, pages 281–290. Springer, 2000.

[HH04] Robert Holte and István Hernádvölgyi. Steps towards the automatic creation of search heuristics. Technical Report TR04-02, Department of Computing Science, University of Alberta, 2004.

[HHH07] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceed-*

ings of the 17th International Conference on Automated Planning and Scheduling, pages 176–183, 2007.

[HMZM96] Robert C. Holte, Taieb Mkadmi, Robert M. Zimmer, and Alan J. MacDonald. Speeding up problem-solving by abstraction: A graph-oriented approach. *Artificial Intelligence*, 85:321–361, 1996.

[Kno94] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[Kor80] Richard E. Korf. Towards a model of representation changes. *Artificial Intelligence*, 14(1):41–78, 1980.

[Kor97] Richard Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the 14th AAAI Conference on Artificial Intelligence*, pages 700–705, 1997.

[Low88] Michael R. Lowry. Invariant logic: A calculus for problem reformulation. In *Proceedings of the 7th AAAI Conference on Artificial Intelligence*, pages 14–18, 1988.

[McD99] Drew McDermott. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159, 1999.

[McD00] Drew McDermott. The 1998 AI planning systems competition. *AI Magazine*, 2(2):35–55, 2000.

[Pea84] Judea Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[Pri93] Armand Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.

[Sac74] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Sch78] Thomas Schaefer. The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 216–226. ACM, 1978.

[SM95] Shinichi Shimozono and Satoru Miyano. Complexity of finding alphabet indexing. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, E78-D(1):13–18, 1995.

[SS06] Jerry Slocum and Dic Sonneveld. *The 15 Puzzle*. Slocum Puzzle Foundation, 2006.

[Van92] Jeffrey Van Baalen. Automated design of specialized representations. *Artificial Intelligence*, 54(1):121–198, 1992.

A. Proofs of the Results in Section 4

Theorem 1. 1. *REACHABLE is PSPACE-complete 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. (1) To prove the first assertion, note that reachability for SAS$^+$—a subproblem of ours—is already *PSPACE*-complete [Bäc92]. Since transforming SAS$^+$ problems into our notation requires only polynomial space (we omit the details), our REACHABLE problem is *PSPACE*-hard. That REACHABLE is actually contained in *PSPACE* follows by simply adopting the analogous proof for SAS$^+$ from [BN95].

(2) The second assertion is a trivial and well-known fact—reachability can be tested in time linear in the number of edges.

(3) 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. □

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.*

Proof.

(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

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

  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 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 only 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 component 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 6.

This completes the proof of Assertion 1.

(2) We consider implicit state space representation only; the explicit case works analogously.

The proof proceeds similarly to that of Assertion 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 *PSPACE*-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 of Assertion 2. □

Corollary 3. 1. *IS-DPP is PSPACE-complete 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) PSPACE-hardness follows from Theorem 2 and Theorem 1.1. To see that IS-DPP is in *PSPACE*, it suffices to show that the complement of IS-DPP is in *NPSPACE*, since *NPSPACE* = *PSPACE* and *PSPACE* is closed under complementation.

A non-deterministic Turing machine for the complement of IS-DPP works as follows, given a state space $\mathcal{S}$ in PSVN notation and an abstraction mapping ψ of type domain abstraction or of type projection.

1. The machine non-deterministically generates two states s^* and s in $\mathcal{S}$.
2. The machine tests with a polynomial space algorithm whether or not $\psi(s)$ is reachable from $\psi(s^*)$. (Such an algorithm exists, because potential witnessing paths $(\psi(s_0), \psi(s_1), \ldots, \psi(s_z))$ can be constructed stepwise in polynomial space by always just memorizing the current path index j and the current state $\psi(s_j)$ on the path and non-deterministically generating the next state $\psi(s_{j+1})$.) If $\psi(s)$ is not reachable from $\psi(s^*)$ then the machine returns 'no' and stops. If $\psi(s)$ is reachable from $\psi(s^*)$ then the machine goes to stage 3.
3. The machine uses a polynomial space algorithm in order to decide whether or not there is an s' such that both $\psi(s') = \psi(s)$ and s' is reachable from s^*. (Such an algorithm exists by a straightforward application of by Theorem 1.1.) If such an s' exists then the machine returns 'no' and stops. If no such s' exists then the machine returns 'yes' and stops.

Hence the machine returns 'yes' if and only if the states s and s^* that it generates non-deterministically witness that *(i)* $\psi(s)$ is reachable from $\psi(s^*)$ and *(i)* no pre-image s' of $\psi(s)$ under ψ is reachable from s^*; in other words, they witness that ψ is not DPP. Obviously, it requires only a polynomial amount of space.

(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. □

Theorem 4. 1. *EXIST-DPP is PSPACE-complete 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-complete for the case of domain abstraction and either type of state space representation if the dimension n is fixed with* $n > 2$ a priori.

Proof of (1) and (2).

(1) We first prove that EXIST-DPP for the case of projection and implicit state space representation is in *PSPACE*. It suffices to show containment in *NPSPACE*. A non-deterministic Turing machine given a state space $\mathcal{S}$ simply generates a projection mapping ψ (i.e., a subset of the given state variables) as a potential witness. It then passes both $\mathcal{S}$ and ψ on to a polynomial space algorithm A for IS-DPP, which exists because of Corollary 3. The non-deterministic Turing machine simply returns the answer that A returns. Obviously, this Turing machine non-deterministically decides EXIST-DPP (for projection) using only a polynomial amount of space.

Next we prove *PSPACE*-hardness. 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

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

where $\sigma \in \Sigma$ is a fixed symbol and a appears in the j^{th} position in the precondition. 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 in the precondition of any operator, and *(ii)* neither a nor b ever appear in the postcondition of any operator in O. In particular, all operators in $\hat{O}$ with b in their postcondition must have a occurring in their precondition.

It is not hard to prove that projections ignoring any of the first $n - 1$ variables will always be non-DPP with respect to $\hat{\mathcal{S}}$. To see this, let $s^\circ = \langle \sigma_1, \ldots, \sigma_{j-1}, a, \sigma_{j+1}, \ldots, \sigma_n \rangle$ for some $j \in \{1, \ldots, n-1\}$, where all the σ_i are elements of Σ. Let $s^* = \langle \sigma_1, \ldots, \sigma_{j-1}, \sigma_j, \sigma_{j+1}, \ldots, \sigma_n \rangle$ for some $\sigma_j \in \Sigma$. If ψ ignores the j^{th} state variable then $\psi(s^*) = \psi(s^\circ)$. Moreover, in the abstract space defined by ψ, the state $\psi(\langle b, \ldots, b \rangle)$ is reachable from $\psi(s^*)$, since $\langle b, \ldots, b \rangle$ is reachable from s° and $\psi(s^*) = \psi(s^\circ)$. However, $\langle b, \ldots, b \rangle$ is not reachable from s^* in the original state space $\hat{\mathcal{S}}$, because all operators in $\hat{O}$ with b in their postcondition must have a occurring in their precondition. Hence the abstract state $\psi(\langle b, \ldots, b \rangle)$ is spurious for 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).

(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. $\square$*(1) and (2)*

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.*[8]

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 $\varphi(A_1) \cap \varphi(A_2) = \emptyset$.*[9]

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 [Sch78] showed the following lemma.

Lemma A.1 (Schaefer, 1978). *NOT-ALL-EQUAL-3SAT is NP-complete.*

[8]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 does not satisfy at least one literal per clause in H.

[9]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?

This was exploited by Shimozono and Miyano [SM95] to show that the problem SHI-MIY95 is *NP*-complete, too.

Lemma A.2 (Shimozono & Miyano, 1995). *SHI-MIY95 is NP-complete.*

The reduction used by Shimozono and Miyano to prove Lemma A.2 is essential for our proof of Theorem 4.3; hence we give their proof here in some detail.

Proof of Lemma A.2. The proof is done by polynomial reduction from NOT-ALL-EQUAL-3SAT. This is sufficient because of Lemma A.1.

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

- $\Sigma = \{\mathtt{t},\mathtt{f}\} \cup \{x_j, \overline{x_j} \mid 1 \leq j \leq \jmath\}$,
- $\Gamma = \{0, 1\}$,
- $A_1 = \{\mathtt{tft}\} \cup \{x_j\overline{x_j}x_j \mid 1 \leq j \leq \jmath\} \cup \{l_{i1}l_{i2}l_{i3} \mid 1 \leq i \leq \imath\}$,
- $A_2 = \{\mathtt{ttt},\mathtt{fff}\}$.

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.[10]

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 $\rho = (\rho(x_1), \ldots, \rho(x_\jmath)) \in \{0, 1\}^\jmath$ be an assignment of the variables $x_1, \ldots, x_\jmath$, such that H is satisfied by ρ with at least one satisfied and at

[10] We omit the details that are not relevant for our proof of Theorem 4.3.

least one unsatisfied literal per clause. Let $\varphi(\mathtt{t}) = 1$, $\varphi(\mathtt{f}) = 0$, and $\varphi(x_j) = \rho(x_j)$, $\varphi(\overline{x_j}) = 1 - \rho(x_j)$ for $1 \leq j \leq \jmath$. Now it is not hard to prove that φ is a homomorphism from Σ^* to Γ^* with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$.

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). The details in which these differ from SHI-MIY95 are highlighted in bold.

Definition 9. 1. *The decision problem ALPH-INDEX is defined as follows. Given two finite alphabets Σ, Γ with $|\Sigma| > |\Gamma|$, given $\boldsymbol{n \in \mathbb{N}}$, $\boldsymbol{n > 2}$, 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 $\boldsymbol{\varphi : \Sigma^n \to \Gamma^n}$ with*

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

2. *The decision problem ALPH-INDEX′ is defined as follows. Given two finite alphabets Σ, Γ with $|\Sigma| > |\Gamma|$, $\boldsymbol{n \in \mathbb{N}}$, $\boldsymbol{n > 2}$, 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 $\boldsymbol{\varphi : \Sigma^n \to \Gamma^n}$ with*

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

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 A.2 to show the following main lemma for the proof of Theorem 4.3 (and for the proof of Corollary 5.3).

Lemma A.3. 1. *ALPH-INDEX is NP-complete, even for fixed $n > 2$.*

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

Proof.

(1) Containment in *NP* is easily verified by observing that a mapping $\varphi : \Sigma \to \Gamma$ can be generated non-deterministically; it can then be checked in polynomial time whether φ is a solution for the given problem instance. It remains to prove *NP*-hardness.

The essence of the *NP*-hardness proof is the observation (from the proof of Lemma A.2) 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:

(a) all strings in the sets A_1 and A_2 always have the same length n ($n = 3$);
(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 a 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 A.2.

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 $\varphi(A_1) \cap \varphi(A_2) = \emptyset$. In particular, there is a surjective homomorphism $\varphi : \Sigma^n \to \Gamma^n$ with $\varphi(A_1) \cap \varphi(A_2) = \emptyset$ or $\varphi(A_1) = \varphi(A_2)$.[11]

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.

(2) Containment in *NP* is easily verified by observing that a mapping $\varphi : \Sigma \to \Gamma$ can be generated non-deterministically; it can then be checked in

[11] 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.

polynomial time whether φ is a solution for the given problem instance. It remains to prove *NP*-hardness.

Modifying 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 a 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 A.2), 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. Containment in *NP* is verified by observing that a mapping ψ (as a potential witness) can be constructed non-deterministically; then, according to Corollary 3.3, it can be checked in polynomial time whether ψ is a DPP abstraction. It remains to prove *NP*-hardness.

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.

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 $A_1 \cap A_2 = \emptyset$, $A_i = \{s_1^i, \ldots, s_{k_i}^i\}$ 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

$$\begin{aligned} \Pi \;=\; & \{(s_{k_1}^1, s_1^1)\} \cup \{(s_j^1, s_{j+1}^1) \mid 1 \le j < k_1\} \cup \\ & \{(s_{k_2}^2, s_1^2)\} \cup \{(s_j^2, s_{j+1}^2) \mid 1 \le j < k_2\} \end{aligned}$$

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. □

Corollary 5.
1. *IS-DPP$_{s^*}$ is PSPACE-complete for the case of implicit state space representation and either type of abstraction.*
2. *EXIST-DPP$_{s^*}$ is PSPACE-complete for the case of projection and implicit state space representation.*
3. *EXIST-DPP$_{s^*}$ is NP-complete for the case of domain abstraction and either type of state space representation if the dimension n is fixed with $n > 2$ a priori.*

Proof. The proofs showing containment in *PSPACE* or *NP* can be adapted from the corresponding proofs for the s^*-free versions of these decision problems. In what follows, we only prove hardness.

(1) PSPACE-hardness 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$_{s^*}$.

(2) PSPACE-hardness 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.

(3) To prove *NP*-hardness, we polynomially reduce the problem ALPH-INDEX′ (for fixed $n > 2$) to 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 $\{s_1^2, \ldots, s_{k_2}^2\}$. □