The Spurious Path Problem in Abstraction

Gaojian Fan and **Robert C. Holte**
University of Alberta, Edmonton, Alberta, Canada
{gaojian, rholte}@ualberta.ca

Abstract

Abstraction is a powerful technique in search and planning. A fundamental problem of abstraction is that it can create spurious paths, i.e., abstract paths that do not correspond to valid concrete paths. In this paper, we define spurious paths as a generalization of spurious states. We show that spurious paths can be categorized into two types: state-independent spurious paths and state-specific spurious paths. We present a practical method that eliminates state-independent spurious paths, as well as state-specific spurious paths when integrated with mutex detection methods. We provide syntactical conditions under which our method can remove state-independent spurious paths completely. We demonstrate that eliminating spurious paths can improve a heuristic substantially, even in abstract spaces that are free of spurious states and edges.

Introduction

Abstraction is a powerful technique in search and planning. The main applications of abstraction include plan refinement (Sacerdoti 1974; Bacchus and Yang 1994; Knoblock 1994; Seipp and Helmert 2013) and abstraction-based heuristics (Culberson and Schaeffer 1998; Hernádvölgyi and Holte 2000; Edelkamp 2001; Seipp and Helmert 2013; Helmert et al. 2014). Plan refinement first solves the problem in the abstract space and then uses the abstract plan as a skeleton to find a concrete plan. Heuristic search methods use the costs of abstract solutions as a heuristic to guide search.

A fundamental problem of abstraction is that it may create spurious paths. A spurious path is an abstract path that has no valid pre-image path. For plan refinement, spurious paths cause backtracking, i.e., re-planning in the abstract space (Bacchus and Yang 1994). For abstraction heuristics, spurious paths introduce short-cuts in the abstract space and produce heuristics that underestimate the true costs.

In this paper, we give a formal definition of spurious path. Our definition generalizes the definition of spurious states (Hernádvölgyi and Holte 1999; Zilles and Holte 2010). We introduce the concept of an *operator chain*, i.e., a sequence of operators that can be executed consecutively in the concrete space. Operator chains underlie our practical method, which we call SPECO (for Spurious Path Elimination with Chains of Operators), for eliminating spurious paths. To understand how effective SPECO could be, we analyze key properties of operator chains and derive the syntactical conditions under which SPECO can eliminate all spurious paths of a certain type, called *state-independent* spurious paths. After the formal analysis, we present SPECO and describe its integration into a heuristic search method. We demonstrate that spurious paths may exist in abstract spaces of typical search domains such as the Sliding Tile Puzzle. We test our method on the 8-Puzzle as a case study. The experimental results show that spurious paths are harmful and removing them substantially increases the heuristic quality, even in abstract spaces that are free of spurious states and edges.

State Spaces and Abstractions

A *state space* is a 4-tuple $\mathcal{S} = (S, L, c, \Pi)$. S is a set of *states*, L is a set of *labels*, $c : L \mapsto \mathbb{R}_0^+$ is a function giving the *cost* of a label, and $\Pi \subseteq S \times L \times S$ is a set of labelled edges. Each $(s, l, s') \in \Pi$ is a directed edge from state s to state s' with label l. As a more intuitive notation, we use $s \xrightarrow{l} s'$ to denote (s, l, s'). A label l is *non-deterministic* in $\mathcal{S}$ if there are two edges $s \xrightarrow{l} s'$ and $s \xrightarrow{l} s''$ such that $s' \neq s''$. Otherwise, the label is deterministic in $\mathcal{S}$. In the rest of the paper "s.t." stands for "such that."

A *path* w of length $k \in \mathbb{N}_0$ in state space $\mathcal{S} = (S, L, c, \Pi)$ is a sequence of states $(s_0, s_1, ..., s_k)$ s.t. for each $i \in \{1, 2, ..., k\}$ there exists a label l_i s.t. $s_{i-1} \xrightarrow{l_i} s_i \in \Pi$. Since there might be several transitions (with different labels and costs) from s to s' we define $c(s, s')$, the cost of the edge from s to s', as the minimum cost $c(l)$ over all labels l s.t. $s \xrightarrow{l} s' \in \Pi$. The cost of path w is then $\sum_{i=1}^{k} c(s_{i-1}, s_i)$. In a path the same state can appear more than once and self-loops are allowed, i.e., $s_i = s_{i+1}$ is allowed if there exists a label l s.t. $s_i \xrightarrow{l} s_{i+1} \in \Pi$. For states $s, s' \in S$, we say s' is *reachable* from s if there exists a path $(s_0, s_1, ..., s_k)$ such that $s = s_0$ and $s' = s_k$ for some $k \in \mathbb{N}_0$.

An *abstraction* of state space $\mathcal{S} = (S, L, c, \Pi)$ is a surjective mapping of states in S, ψ: $S \mapsto S'$ that induces an *abstract state space* $\mathcal{S}' = (S', L, c, \Pi')$ where $S' = \{\psi(s) \mid s \in S\}$ is the set of abstract states, and $\Pi' = \{\psi(s) \xrightarrow{l} \psi(s') \mid s \xrightarrow{l} s' \in \Pi\}$ is the set of abstract edges. We call the state space $\mathcal{S}$ the *concrete state space*, and the

states/edges/paths in $\mathcal{S}$ the concrete states/edges/paths. This definition corresponds to what Clarke et al. (1994) call the "minimal" abstraction of $\mathcal{S}$, since Π' only contains edges that are the images of the edges in Π.[1] If Π' were allowed to contain additional edges, it would be non-minimal in Clarke et al.'s terminology. We focus on minimal abstractions for two reasons. The first is to emphasize that spurious paths can arise even when every individual edge in Π' has a pre-image[2] in Π. The second reason is that the commonly used abstraction techniques, such as merge-and-shrink (Helmert et al. 2014), projection (Edelkamp 2001), and domain abstraction (Hernádvölgyi and Holte 2000), all produce minimal abstractions.

Spurious Paths

In this section, we formally define spurious paths as well as the special type of spurious path we focus on in this paper. In this paper "w.r.t." stands for "with respect to."

Definition 1. For $s \in S$, $k \in \mathbb{N}_0$ and abstraction ψ, an abstract path $(t_0, t_1, ..., t_k)$ is a *spurious path* w.r.t. s if

1. t_0 is reachable from $\psi(s)$;
2. there is no concrete path $(s_0, s_1, ..., s_k)$ s.t. $\psi(s_i) = t_i$ for all $i \in \{0, 1, ..., k\}$ with s_0 reachable from s;

In Definition 1, the first condition excludes abstract paths that are unreachable by forward search in the abstract space as we are only interested in spurious paths reachable in the abstract space. The second condition defines the key characteristic of a spurious path: it has no reachable concrete path pre-image.

Spurious paths of length 0 w.r.t. some $s \in S$ coincide with *spurious states* w.r.t. s, whose existence has been noted by several authors (e.g. (Hernádvölgyi and Holte 1999; Haslum, Bonet, and Geffner 2005)) and studied in depth by Zilles and Holte (2010) and Sadeqi (2014). Note that spurious states, as defined here, are entirely different from the "spurious states" that are often generated during regression search (Bonet and Geffner 2001; Alcázar et al. 2013).

Spurious paths of length 1 (spurious edges) w.r.t. some $s \in S$ have also been noted by previous authors (e.g. (Clarke, Grumberg, and Long 1994; Sharygina, Tonetta, and Tsitovich 2009)) and studied in some depth by Sadeqi (2014).

Spurious paths of length greater than 1 w.r.t. some $s \in S$ have been explicitly distinguished from paths containing a spurious edge by a few authors in the past (e.g. (Clarke et al. 2002; Qian and Nymeyer 2004; Smaus and Hoffmann 2008; Sharygina, Tonetta, and Tsitovich 2009)). The work we present in this paper is the first to distinguish two different types of such paths—state-specific and state-independent (defined below)—and to use this understanding to develop a new technique (operator chains) to eliminate spurious paths.

[1] For a minimal abstraction, Clarke et al. also require the set of abstract start states to be the exact image of the set of the concrete start states. Since we do not include the start states in our definition of a state space, we cannot express that requirement here.

[2] Here, the images are abstract edges and pre-images are concrete edges. Similarly, when we talk about paths, the images and the pre-images correspond to the abstract and concrete paths of the same length respectively.

(a) the concrete space (b) the abstract space

Figure 1: A state-independent (type (i)) spurious path.

Some properties studied previously are sufficient conditions for abstractions being free of spurious paths. An abstraction is *precondition-preserving* (PP) if for any label l and any state s, if there exists an abstract edge $\psi(s) \xrightarrow{l} t \in \Pi'$, then there exists a concrete edge $s \xrightarrow{l} s' \in \Pi$. If an abstraction is PP and all labels are deterministic in the abstract space, then the abstract space contains no spurious states (Zilles and Holte 2010). It is easy to prove that these conditions are also strong enough to prevent spurious paths:

Lemma 1. *If ψ is a precondition-preserving abstraction and all labels in the abstract space are deterministic, then the abstract space contains no spurious paths.*

Without determinism of the labels in the abstract space, PP alone does not guarantee there are no spurious paths. Consider the example in Figure 1. $S = \{a, b_1, b_2, c, d\}$, $L = \{x, y\}$ and $\Pi = \{a \xrightarrow{x} b_1, b_2 \xrightarrow{x} d, b_1 \xrightarrow{y} c\}$. The abstraction ψ maps the states as follows: $\psi(a) = A$, $\psi(b_1) = \psi(b_2) = B$, $\psi(c) = C$, and $\psi(d) = D$. This abstraction is PP but label y in the abstract space is not deterministic. The abstract path (A, B, D) is reachable from A but has no pre-image reachable from a, so it is a spurious path w.r.t. a. A stronger property called *bisimularity* does not require determinism. An abstraction is a bisimulation if for any states s_1, s_2 s.t. $\psi(s_1) = \psi(s_2)$, for every edge $s_1 \xrightarrow{l} s_1' \in \Pi$, there exists an edge $s_2 \xrightarrow{l} s_2' \in \Pi$ s.t. $\psi(s_1') = \psi(s_2')$ (Milner 1980; 1990; Helmert et al. 2014). A special type of bisimulation, namely, *goal-respecting* bisimulation, is used in merge-and-shrink to obtain perfect heuristics (Nissim, Hoffmann, and Helmert 2011). It is easy to prove that a bisimulation abstraction creates no spurious paths:

Lemma 2. *If ψ is a bisimulation, then the abstract space contains no spurious paths.*

Many abstractions do not have these strong properties and thus may contain spurious paths. We will now show that there are two types of spurious path. Abstractions are surjective mappings, thus any abstract state has a concrete state pre-image. Furthermore, every abstract edge also has a pre-image. Then why are there still spurious states and spurious edges (w.r.t. s) if all abstract states and abstract edges have pre-images? Because none of their pre-images are reachable (from s). For clarity, we rewrite condition 2 in Definition 1:

(i) there is no path $(s_0, s_1, ..., s_k)$ in $\mathcal{S}$ s.t. $\psi(s_i) = t_i$ for all $i \in \{0, 1, ..., k\}$, or

(ii) there exist one or more paths $(s_0, s_1, ..., s_k)$ in $\mathcal{S}$ s.t. $\psi(s_i) = t_i$ for all $i \in \{0, 1, ..., k\}$, but in all such paths, s_0 is unreachable from s.

We call spurious paths of type (i) *state-independent* spurious paths since those paths are spurious w.r.t. any state s,

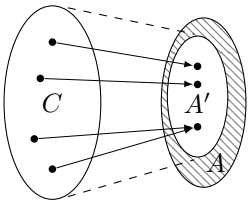


Figure 2: Surjectivity imposed by restricting A to A', i.e., excluding paths in the hatched area.

and we call spurious paths of type (ii) *state-specific* spurious paths. Spurious states and spurious edges are state-specific but never state-independent. However, spurious paths with more than 1 edge could be state-independent. The path (A, B, D) in Figure 1 is in fact a state-independent spurious path, since it does not have any pre-image at all. The path (A, B, C) is state-specific spurious path but not a state-independent spurious path. (A, B, C) has a pre-image (a, b_1, c) which is reachable from a but not from any other concrete state in the space.

In the next section, we develop the theoretical underpinnings for a method to identify and eliminate state-independent spurious paths. The implementation of this method in the context of Hierarchical IDA* (Holte, Grajkowski, and Tanner 2005), and a case study applying this implementation to the 8-Puzzle, are then given. This case study shows how harmful spurious paths can be: eliminating them reduces the number of 8-Puzzle states expanded by almost a factor of 10 in several cases.

State Space Representations and Operator Chains

For any abstraction ψ and any $k \in \mathbb{N}_0$, we have a mapping $\psi_k : C \mapsto A$ where C is the set of all concrete paths of length k and A is the set of all abstract paths of length k. Let $A' = \{\psi(w) \mid w \in C\}$, the set of images of concrete paths in C. The abstract paths in $A \backslash A'$ are the state-independent spurious paths of length k because they do not have pre-images. ψ_k is not surjective, but if we avoid the generation of the abstract paths in $A \backslash A'$, we will have imposed surjectivity on ψ_k. See Figure 2 for an illustration.

Can we impose surjectivity without enumerating C? The answer is yes (in practice only partially), because the state spaces are defined implicitly, by a state space representation.

An implicit representation of a state space is a triple, $\mathcal{R} = (\mathcal{L}, \mathcal{O}, c)$, where $\mathcal{L}$ is a formal language for describing states, $\mathcal{O}$ is a set of operators, and $c : \mathcal{O} \mapsto \mathbb{R}_0^+$ is a function giving the cost of each operator. Each operator o has a precondition, $pre(o)$, which specifies the set of states to which o can be applied, and a postcondition, which specifies the effects of applying o to a state. If operator o can be applied to state $s \in \mathcal{L}$, we write $s \in pre(o)$. We allow operators to be non-deterministic: applying an operator to a state can produce a set of states. We write $s' \in o(s)$ if state s' is among the states produced by applying operator o to state s, and use $post(o)$ to denote $\bigcup_{s \in pre(o)} o(s)$—the set of states that can be produced by applying o to states in $pre(o)$.

For example, in SAS$^+$ (Bäckström and Nebel 1995) and PSVN (Holte, Arneson, and Burch 2014), $\mathcal{L}$ is specified by fixing $n \in \mathbb{N}$, the number of state variables, and defining a finite set of possible values for each state variable. If D_i is the set of values for state variable i, then $\mathcal{L} = D_1 \times D_2 \times \ldots \times D_n$. The precondition of an operator in SAS$^+$ requires certain state variables to have specific values and the postcondition sets certain state variables to specific values. Preconditions and postconditions in PSVN are more expressive, allowing some state spaces to be represented with exponentially fewer operators than SAS$^+$, but are still executable in time linear in n. Operators in SAS$^+$ are always deterministic. In PSVN it is possible to define non-deterministic operators.

A state space representation $\mathcal{R} = (\mathcal{L}, \mathcal{O}, c)$ induces a state space $\mathcal{S} = (S, \mathcal{O}, c, \Pi)$ where $S = \mathcal{L}$ and $\Pi = \{s \xrightarrow{o} s' \mid o \in \mathcal{O}, s \in pre(o), \text{ and } s' \in o(s)\}$. Each operator $o \in \mathcal{O}$ therefore represents a set of edges in the state space, which we denote $\Pi_o = \{s \xrightarrow{o} s' \mid s \xrightarrow{o} s' \in \Pi\}$.

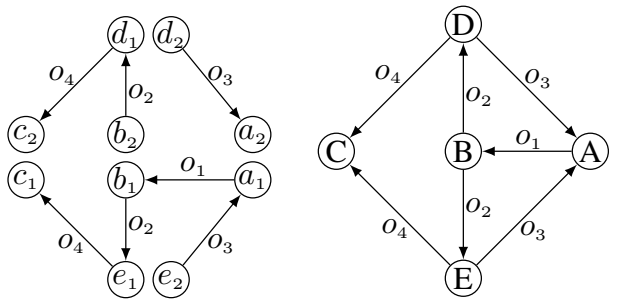
Although abstraction functions operate on the representation $\mathcal{R}$ and not directly on the induced state space $\mathcal{S}$, in the remainder of this paper we continue to refer to ψ as being applied to $\mathcal{S}$ to produce an abstract state space $\mathcal{S}' = (S', \mathcal{O}, c, \Pi')$ and we define the abstract operators in terms of the edges in $\mathcal{S}'$, as follows. For each operator $o \in \mathcal{O}$, abstraction ψ induces an *abstract operator*, denoted by $\psi(o)$, that corresponds to $\Pi'_o = \{t \xrightarrow{o} t' \mid t \xrightarrow{o} t' \in \Pi'\}$, the set of abstract edges with label o. We call operators in $\mathcal{O}$ the concrete operators. It is worth noting that two different concrete operators o_1 and o_2 can be mapped to an identical abstract operator, i.e., $\Pi'_{o_1} = \Pi'_{o_2}$. In addition, an abstraction may map a non-identity operator o to the abstract identity operator, i.e., for any $t \xrightarrow{o} t' \in \Pi'_o$, $t = t'$. Some search algorithms ignore duplicate operators and/or identity operators. Our search algorithm keeps all abstract operators including duplicate operators and identity operators. This ensures that our algorithm does not eliminate a valid solution path.

Let $k \in \mathbb{N}_0$ and π be the operator sequence $\pi = (o_1, o_2, ..., o_k)$. The precondition of π, $pre(\pi)$, is the set of states to which the whole sequence $(o_1, o_2, ..., o_k)$ can be applied, and the postcondition of π, $post(\pi)$, is the set of states can be produced by π when it is applied to states in $pre(\pi)$. We say $\pi = (o_1, o_2, ..., o_k)$ *generates* a concrete path $p = (s_0, s_1, ..., s_k)$ if $s_{i-1} \xrightarrow{o_i} s_i \in \Pi$ for $i \in \{1, 2, ..., k\}$. For an abstraction ψ, we say the abstract operator sequence $\psi(\pi) = (\psi(o_1), \psi(o_2), ..., \psi(o_k))$ *generates* an abstract path $w = (t_0, t_1, ..., t_k)$ if $t_{i-1} \xrightarrow{o_i} t_i \in \Pi'$ for $i \in \{1, 2, ..., k\}$.

Definition 2 (Operator Chain)**.** An operator sequence π is an *operator chain* if it generates a concrete path in $\mathcal{S}$.

In the rest of this section, if not specified otherwise, we tacitly assume $\mathcal{S} = (S, \mathcal{O}, c, \Pi)$ denotes the concrete state space induced by representation $\mathcal{R} = (\mathcal{L}, \mathcal{O}, c)$, ψ denotes an abstraction of $\mathcal{S}$, and $\mathcal{S}' = (S', \mathcal{O}, c, \Pi')$ denotes the abstract state space. In addition, $k \in \mathbb{N}_0$, $\pi = (o_1, o_2, ..., o_k)$ is an operator chain and $w = (t_0, t_1, ..., t_k)$ is an abstract path.

Returning to our discussion of Figure 2, if $w \in A'$ then w must have a concrete path pre-image in C and the concrete



(a) the concrete space (b) the abstract space

Figure 3: In the abstract space (b), $w_1 = (\text{B,D,A})$, $w_2 = (\text{B,E,A})$, $w_3 = (\text{D,A,B})$ and $w_4 = (\text{A,B,D,C})$ are state-independent spurious paths.

path must be generated by an operator chain. If an abstract path can only be generated by $\psi(\pi)$ for which π is not an operator chain, then the abstract path has no pre-image and is state-independent spurious.

Theorem 3. *If there is no operator chain π s.t. $\psi(\pi)$ generates the abstract path w, then w is a state-independent spurious path.*

Figure 3 shows an example in which spurious paths are generated by an operator sequence that is not the abstraction of an operator chain. In the example, the states and edges are shown in the figure. For $x \in \{a, b, c, d, e\}$ and $i \in \{1, 2\}$, the abstraction ψ maps the concrete state denoted by x_i to the abstract state denoted by X (the upper-case of x). Note that the postcondition of o_2 conflicts with the precondition of o_3, i.e., $post(o_2) \cap pre(o_3) = \emptyset$, so (o_2, o_3) is not an operator chain. This conflict, however, is hidden by the abstraction, so $(\psi(o_2), \psi(o_3))$ can generate state-independent spurious paths $w_1 = (\text{B,D,A})$ and $w_2 = (\text{B,E,A})$.

Reification and Abstract Determinism

The abstraction of an operator chain may still generate a state-independent spurious path. For example, in Figure 3, (o_3, o_1) is an operator chain, but $(\psi(o_3), \psi(o_1))$ still generates $w_3 = (\text{D,A,B})$ which is a spurious path. In fact, o_3 cannot be applied after o_1 is applied to d_2 because $o_3(d_2) = \{a_2\}$ and $pre(o_1) = \{a_1\}$. However, this fact is hidden by the abstraction since a_1 and a_2 are mapped to the same abstract state. Hence, $(\psi(o_3), \psi(o_1))$ can generate $w_3 = (\text{D,A,B})$, but (o_3, o_1) cannot generate a pre-image for w_3. The following definition concerns whether an operator chain can generate a pre-image for an abstract path.

Definition 3 (Reification)**.** Let $i \in \{1, 2, ..., k\}$. We say π *reifies* the first i steps in w if π generates a path $(s_0, s_1, ..., s_k)$ s.t. $\psi(s_j) = t_j$ for all $j \in \{0, 1, ..., i\}$. We say π reifies w if π reifies all k steps in w.

The definition immediately implies the following.

Lemma 4. *w is not state-independent spurious if and only if there exits an operator chain π that reifies w.*

Now consider again the operator chain $\pi_3 = (o_3, o_1)$ and the path w_3 in Figure 3. Since $d_1 \notin pre(\pi_3)$ and $d_2 \notin pre(\pi_3)$, π_3 does not reify the first step in w_3. Thus, π_3 does not reify w_3, and so we know w_3 is a spurious path. However, in the worst case, one has to check every state in an abstract path to verify whether an operator chain reifies the path. In general, even if π reifies the first i steps in w, it does not mean that π reifies the first $i+1$ steps in w. For example, $w_4 = (\text{A,B,D,C})$ in Figure 3 can be generated by $(\psi(o_1), \psi(o_2), \psi(o_4))$ whose pre-image $\pi_4 = (o_1, o_2, o_4)$ is an operator chain. We can see that $a_1 \in pre(\pi_4)$ and $b_1 \in o_1(a_1)$, which means π_4 reifies the first step in w_4. Furthermore, $c_1 \in post(\pi_4)$ and $\psi(c_1) = \text{C}$, i.e., π_4 can even generate a pre-image for the last state C in the path w_4. Unfortunately, w_4 still fails to reify the first two steps in w_4.

Note that π_4 actually reifies the path (A,B,E,C) which is different from w_4 on the third state, and the difference is caused by the non-determinism of o_2 in the abstract state space. It is easy to see that the determinism of operators in the abstract space is the key for the induction of operator chain reification. In particular, if $\pi = (o_1, o_2, ..., o_k)$ reifies the first i step in w, and o_{i+1} is deterministic in the abstract space, then π reifies the first $i+1$ step in w. This observation implies the following.

Lemma 5. *If π reifies the first step in w and all operators of π are deterministic in the abstract space, then π reifies w.*

If the abstraction is domain abstraction (Hernádvölgyi and Holte 2000), then the operators are deterministic in the abstract space if they are deterministic in the concrete space. For projection (Edelkamp 2001), if SAS$^+$ is used, the operators are guaranteed to be deterministic in the abstract space. If the formalism is PSVN, then projections may introduce non-determinism to the abstract space. Abstractions like merge-and-shrink (Helmert et al. 2014) or Cartesian abstraction (Seipp and Helmert 2013) may also make operators non-deterministic in the abstract space.

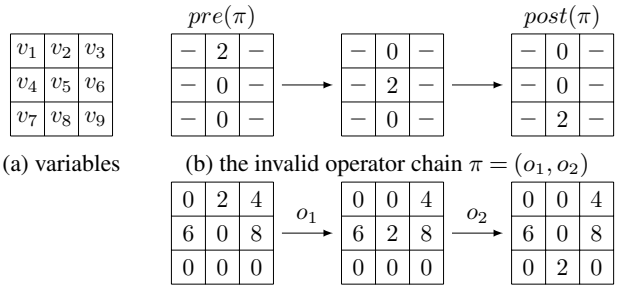
Syntactic Condition

When the operators are deterministic in the abstract space, we only need to check the first two states to know whether the operator chain reifies the path. However, this process is still dependent on the particular paths. For example, (o_3, o_1) in Figure 3 reifies (E,A,B) but not (D,A,B). One has to do the reification check for every path. To avoid this semantic dependence, we introduce the concept of an operator chain being *self-fulfilling*.

Definition 4 (Self-Fulfilling)**.** We say $\pi = (o_1, o_2, ..., o_l)$ is a *self-fulfilling* operator chain if $pre(o_1) = pre(\pi)$.

In PSVN or SAS$^+$, the precondition of an operator or an operator chain[3] can be represented by a simple expression. In these formalisms, the condition $pre(o_1) = pre(\pi)$ can be checked *syntactically* by comparing their corresponding expressions. If an operator chain is self-fulfilling and its abstract operators are deterministic, then any path generated by $\psi(\pi)$ is not state-independent spurious.

[3]Each operator chain forms a "macro-operator" (Holte and Burch 2014). The precondition of the operator chain is the precondition of the macro-operator.



(a) variables (b) the invalid operator chain $\pi = (o_1, o_2)$

(c) a spurious path generated by $\psi(\pi)$

Figure 4: An invaid operator chain $\pi = (o_1, o_2)$ in the 8-Puzzle. o_1 moves tile 2 from the upper-middle position to the center and o_2 moves tile 2 from the center to the bottom-middle position.

Theorem 6. *Let w be an abstract path generated by $\psi(\pi)$. If π is self-fulfilling and its operators are deterministic in the abstract space, then w is not state-independent spurious.*

Proof. $\psi(\pi)$ generates w, so $t_0 \xrightarrow{o_1} t_1 \in \Pi'$. There exists $s \xrightarrow{o_1} s' \in \Pi$ s.t. $\psi(s) = t_0$ and $\psi(s') = t_1$. Because π is self-fulfilling, π can be applied to s, which means it generates a path $(s_0, s_1, ..., s_l)$ in which $s_0 = s$. Since $s_0 = s$, $\psi(s_0) = t_0$. Because o_1 is deterministic in the abstract space, we have $\psi(s_1) = t_1$. π reifies the first step in w. In addition, all operators of π are deterministic in the abstract space, so π reifies the whole path w (Lemma 5). By Lemma 4, w is not state-independent spurious. □

State-Specific Spurious Paths

The definition of operator chain does not take reachability into consideration, so the abstraction of an operator chain may still generate state-specific spurious paths even if it does not generate state-independent spurious paths.

Consider the 8-Puzzle. A standard representation requires 9 state variables $v_1, v_2, ..., v_9$ for the 9 positions of the puzzle (see Figure 4(a)). The value sets for all variables are the same: $D = \{0, 1, 2, ..., 8\}$ in which each indicates what object (the blank (0) or a specific tile $(1, 2, ..., 8)$) is in a specific position of the puzzle. Now consider two operators o_1 and o_2. o_1 moves tile 2 from the upper-middle position to the center, so the precondition of o_1 requires $v_2 = 2$ and $v_5 = 0$, and the postcondition of o_1 exchanges the values of v_2 and v_5, i.e., $v_2 := 0$ and $v_5 := 2$. o_2 moves tile 2 from the center to the bottom-middle position, so its precondition is $(v_5 = 2, v_8 = 0)$ and its postcondition is $(v_5 := 0, v_8 := 2)$. o_2 can be applied immediately after o_1 because the precondition of o_2 does not conflict with the postcondition of o_1, so $\pi = (o_1, o_2)$ is an operator chain and can generate concrete paths. However, the precondition of π requires $(v_2 = 2, v_5 = 0, v_8 = 0)$. See Figure 4(b) ('$-$' indicates the variable is not included in either operator). Any states to which π can be applied must contain two blanks, which means the state cannot be reached from a standard 8-Puzzle state. Therefore, paths generated by $\psi(\pi)$ are state-specific spurious paths because they do have pre-images but all of them are unreachable concrete paths. To avoid these state-specific spurious paths, we integrate reachability with operator chains.

Definition 5 (Valid Operator Chain)**.** Let s be a concrete state. We say π is *valid* w.r.t. s if π generates a path $(s_0, s_1, ..., s_l)$ s.t. s_0 is reachable from s. Otherwise, we say π is *invalid* w.r.t. s.

The following theorem improves upon Theorem 3.

Theorem 7. *Let s be a concrete state. If there is no valid operator chain π w.r.t. s such that $\psi(\pi)$ generates the abstract path w, then w is a spurious path w.r.t. s.*

We can use *mutex sets* (Bonet and Geffner 2001), i.e., sets of variable assignments that are not achievable from the initial state, to discover invalid operator chains w.r.t. the initial state. In particular, if the precondition of an operator chain contains a mutex, we know the operator chain is invalid. Since it is usually not feasible to find all mutexes, we cannot discover all invalid operator chains w.r.t. the initial state. The most commonly used mutexes are mutex pairs, i.e., mutex sets of size 2. In the previous 8-Puzzle example, $(v_5 = 0, v_8 = 0)$ is a mutex pair, and it can be detected automatically by methods like h^2 (Haslum and Geffner 2000). With this knowledge, we know (o_1, o_2) is an invalid operator chain, so we can avoid using $(\psi(o_1), \psi(o_2))$ to generate abstract paths.

Hierarchical IDA* and SPECO

In this section, we present SPECO—the method for *Spurious Path Elimination by Chains of Operators*—and HIDA* the base search algorithm we apply SPECO on.

SPECO

We first set a limit $k \in \mathbb{N}$ on the length of operator chains we are willing to consider. In other words, we are interested in removing spurious paths of length k. An operator sequence is an operator chain if and only if it forms a "macro-operator" (Holte and Burch 2014). We use Holte and Burch's "move composition" process for building a macro-operator to check whether an operator sequence is an operator chain. After we get an operator chain π of length k, we check if its precondition contains a mutex detected by h^2 (Haslum and Geffner 2000). If the precondition of π does not contain a mutex detected by h^2 we put the π in a set $\mathcal{C}_k$. $\mathcal{C}_k$ contains all valid operator chains, but may also contains some invalid operator chains since h^2 is not guaranteed to detect all mutexes.

Consider forward search in the abstract space. Assume that we are expanding the abstract state t. Let $(\psi(o_1), \psi(o_2), ..., \psi(o_{k-1}))$ be the sequence of abstract operators that *precedes* t. For any operator $\psi(o)$ that is applicable to t, we check if $\pi = (o_1, o_2, ..., o_{k-1}, o) \in \mathcal{C}_k$. If $\pi \in \mathcal{C}_k$, then we apply $\psi(o)$ to t and continue the search, otherwise we do not apply $\psi(o)$ to t and move on to check the next operator applicable to t. We call this process SPECO. By Theorem 7, SPECO does not eliminate any non-spurious paths: for any non-spurious path $w = (t_0, t_1, ..., t_{k-2}, t, t')$, there must be a valid operator chain $\pi = (o_1, o_2, ..., o_{k-1}, o)$ s.t. $\psi(\pi)$ generates w.

Hierarchical IDA*

We apply SPECO in Hierarchical IDA* (HIDA*) (Holte, Grajkowski, and Tanner 2005). A hierarchy in HIDA* consists of multiple levels of state spaces. The bottom level is the concrete space. Each other level is an abstraction of the state space in the level immediately below it. Level 0 is the bottom level, level 1 is the first abstract level, level 2 is the second abstract level, and so on.

IDA* (Korf 1985) is used at all levels. The heuristic for IDA* at level i is obtained by IDA* search at level $i+1$. In other words, if ψ is the abstraction of level i that induces the abstract space at level $i+1$, $h_i(s) = d_{i+1}(\psi(s))$ where s is a state at level i, $h_i(\cdot)$ is the heuristic function for search at level i and $d_{i+1}(\cdot)$ is the function of the true cost to achieve the goal at level $i+1$. IDA* at the highest abstraction level is blind IDA* search, i.e., $h_i(s^*) = 0$ if s^* is a goal state, and $h_i(s) = \epsilon$ for any other state s where ϵ is the minimum cost over all edges. HIDA* terminates as soon as a concrete solution is found.

We choose HIDA* as the base search algorithm for two reasons. First, linear-space search algorithms such as IDA* require linear space to record the operator sequence preceding the current state. Second, hierarchical search (IDA* or A*) only computes the abstract distances that are needed for solving the concrete problem. This is different from pattern database (PDB) construction (Culberson and Schaeffer 1998) which requires an exhaustive search of the abstract space. See a discussion of the application of SPECO in PDB construction in the Discussion section below.

Cache Indexing

Heuristic values (at any level) in HIDA* can be cached to avoid re-solving abstract problems. The cache entries of HIDA* at level i are indexed by states at level i. In addition to the basic heuristic (abstract cost-to-goal), HIDA* also uses *P-g caching* to improve the heuristic (Holte et al. 1996; Holte, Grajkowski, and Tanner 2005). Let $bound$ be the cost bound of the current iteration of IDA* at the abstract level i and g be cost of the path from the abstract initial state to the current abstract state. For any state s at level i, P-g caching updates $h_i(s)$ to $bound - g$ if the value is larger than the abstract cost $d(\psi(s))$.

P-g caching is critical for improving the efficiency of HIDA*. However, if SPECO is used and cache entries are indexed by abstract states, P-g caching may create an inadmissible heuristic. Consider states t_0, t', t and $goal$ at the level i. t_0 is the abstract initial state and $goal$ is the abstract goal state. There are 4 operators a, b, c and d of cost 1 s.t. $\Pi' = \{t_0 \xrightarrow{a} t,\ t' \xrightarrow{b} t,\ t' \xrightarrow{c} t,\ t \xrightarrow{d} goal\}$. Suppose $\mathcal{C}_2 = \{(b, c), (c, d)\}$ and that a is applied earlier than b. Without SPECO, the goal will be reached when $bound = 2$. With SPECO, IDA* never allows the generation of the spurious path $(t_0, t, goal)$ because $(a, d) \notin \mathcal{C}_2$, so the goal will not be reached when $bound = 2$. However, when $bound = 3$, since a is applied before b, t is reached through operator a first. P-g caching makes $h_i(t) = bound - g = 2$ which is an overestimate of the cost from t to g. Later, when t is reached through b, $g = 2$ so $g + h_i(t) > bound$ and the search branch is pruned. IDA* will then continue to the next iteration with $bound = 4$ and P-g caching will make $h_i(t) = 3$ this time. This process will repeat forever and the goal will never be reached.

In order to solve this problem, one needs to distinguish t by its preceding operator sequences. We use the combination of t and π as the index of a cache entry for the cost-to-goal of t with preceding operator sequence π. In the previous example, we store two values $h_i(t, a)$ and $h_i(t, b)$ instead of one value $h_i(t)$. When $bound = 3$, P-g caching makes $h_i(t, a) = 2$ after applying operator a, but later when t is reached through b, $h_i(t, b) = 1$ is used. $h_i(t, b) + g = 3 \leq bound$, so the search will continue and the goal will be reached through a non-spurious path.

SPECO States

If the average number of different operator sequences preceding each state is B, then the search with SPECO will become B times more expensive than the normal search without spurious path elimination. We alleviate this overhead by using SPECO only on states on solution paths. At the abstract level i, we use normal search (i.e., without SPECO) to find a solution and then we check if the solution contains an operator sequence π' of length k such that $\pi' \notin \mathcal{C}_k$. Let $(t_0, t_1, ..., t_{k-1}, t_k)$ be the subsequence of the solution path generated by $\psi(\pi')$. If $\pi' \notin \mathcal{C}_k$, we mark the state t_{k-1} as a *SPECO state* and re-expand it. When expanding a SPECO state t_{k-1}, we perform SPECO and store its heuristic in $h_i(t_{k-1}, \pi)$ where π is the operator sequence of length $k-1$ that precedes t_{k-1}. When we say search with SPECO we mean the search with the application of SPECO only on SPECO states.

Our method can be viewed as an abstraction refinement method in the sense that every SPECO state t refines the normal abstract state t into P "states" $(t, \pi'_1), (t, \pi'_2), ..., (t, \pi'_P)$ where π'_i for $i \in \{1, 2, ..., P\}$ is an operator sequence that precedes t. The preceding operator sequence π' provides an additional precondition for applying an operator $\psi(o)$ to t—(π', o) must be an operator chain. Thus, this refinement helps us prune spurious paths.

A Case Study: 8-Puzzle

We illustrate that HIDA* with SPECO improves the heuristic quality substantially on the 8-Puzzle. We use domain abstractions that map the blank to itself ($\psi(0) = 0$) and map at least two tiles to the blank as well. This kind of abstraction induces abstract spaces that are *free of spurious states and spurious edges* (Sadeqi, Holte, and Zilles 2013) but still contain spurious paths of length 2. Figure 4(c) shows an example of such spurious paths where the abstraction maps the odd-number tiles to the blank and leaves the blank and the even-number tiles untouched. The path can be generated by $(\psi(o_1), \psi(o_2))$, but since (o_1, o_2) is an invalid operator chain, we know the path is a spurious path. We use "Blank-n" to denote the domain abstraction that maps the first n tiles to the blank.

We use a simple hierarchy that contains only two levels: the concrete level (Level-0) and the abstract level (Level-

Use Cache at Level-0							No Cache at Level-0				
Abst.	All	Level-0	Level-1	Time	Cache	Cache-1	All	Level-0	Level-1	Time	Cache-1
Blank-3	114,428	10,872	103,556	0.84	21,833	14,842	125,533	21,973	103,560	0.88	15,013
SPECO	170,606	1,363	169,243	1.38	33,626	32,271	171,607	2,755	168,852	1.38	32,362
Blank-4	46,318	26,355	19,963	0.34	17,692	3,005	84,406	64,481	19,925	0.47	3,022
SPECO	54,711	2,681	52,030	0.50	15,561	13,535	58,447	5,726	52,721	0.58	13,563
Blank-5	59,182	56,401	2,781	0.37	28,103	503	176,558	173,723	2,835	0.74	504
SPECO	22,065	5,913	16,152	0.22	8,432	4384	32,399	15,808	16,591	0.54	4,441
Blank-6	114,356	114,125	231	0.61	49,022	46	452,332	451,871	461	1.74	71
SPECO	27,763	22,931	4,832	0.33	14,540	1,014	90,941	86,021	4,920	2.26	1,118
Blank-7	192,789	192,781	8	0.92	71,786	5	1,004,426	1,004,417	9	3.72	5
SPECO	80,703	79,973	730	0.68	36,919	148	415,914	415,072	842	7.41	165

Table 1: The number of node expansions, search time (in seconds) and the number of cache entries used. A row starting with "SPECO", shows the data for HIDA* with SPECO using the same abstraction as the row above it.

1). We use HIDA* with/without SPECO to solve 100 random instances of the 8-Puzzle. We compare normal HIDA* and HIDA* with SPECO on domain abstractions Blank-n for $n \in \{3, 4, 5, 6, 7\}$. We compare them in terms of number of node expansions, search time, and number of cache entries. We use $k = 2$ in our experiment, i.e., we build the set $\mathcal{C}_2$ in a preprocessing step. Because all operator chains of length 2 in this domain are self-fulfilling and all operators are deterministic in the abstract spaces, by Theorem 6, operator chains in $\mathcal{C}_2$ do not permit any state-independent spurious paths.

Experiment Results

Table 1 shows the average number of node expansions, search time, and number of cache entries used for solving the 100 random instances. The table has two parts: HIDA* using a cache at the concrete level (on the left) and HIDA* using no cache at the concrete level (on the right). The columns "All" give the total number of node expansions at all levels of HIDA* and the columns "Level-0" and "Level-1" give the number of node expansions at the concrete level (Level-0) and the abstract level (Level-1). The average search time is given in the columns "Time". The column "Cache" (in the left sub-table) gives the number of cache entries used in both levels. "Cache-1" only counts cache entries in the abstract level. A row starting with an abstraction (e.g., Blank-3) indicates normal HIDA* with that abstraction, and the row below it, starting with "SPECO", indicates HIDA* with SPECO using the same abstraction.

Note that there are re-expansions when states get marked as SPECO states. In addition, a SPECO state can be expanded more than once since it has different preceding operators. Similarly, a SPECO state may need multiple cache entries because of different preceding operators. We count all of these re-expansions and cache entries for HIDA* with SPECO.

Results with a Concrete Level Cache. We will first analyze the performance of HIDA* with a concrete level cache. We observe a trade-off between the expansions in the concrete level and in the abstract level. On the one hand, the number of node expansions at the concrete level (Level-0) are much lower when SPECO is used. For the first three abstractions, the number of concrete node expansions with SPECO is about 10 times less than the number of concrete node expansions without spurious path elimination. This indicates that spurious paths existed and were substantially reducing heuristic values, and that SPECO was effective in removing them. On the other hand, the number of abstract node expansions is higher when SPECO is used due to the overhead of eliminating spurious paths. There are two reasons for this overhead. First, there are the re-expansions of SPECO states. Second, because SPECO forces the search to find *longer* (and more accurate) solutions that do not contain spurious short-cuts, there are more node expansions.

When only a few tiles are mapped to the blank, e.g., Blank-3 or Blank-4, normal HIDA* has fewer total node expansions than HIDA* with SPECO. When the abstraction maps more tiles to the blank, HIDA* with SPECO expands fewer in total. This is because when there are more tiles mapped to the blank, the abstraction is coarser and there are more spurious paths in the abstract space. These spurious paths introduce harmful short-cuts that can lower the heuristic, and normal HIDA* suffers from the low heuristic values. However, since SPECO removes all spurious paths (of length 2) in the abstract solutions, the abstract solutions do not contain spurious short-cuts (of length 2), so the heuristic values are closer to the true cost to reach the goal. As a result, even though the abstraction is coarse, search with SPECO at the abstract level can still produce a good quality heuristic that helps reduce the number of concrete level expansions. In fact, HIDA* with SPECO using abstraction Blank-5 or Blank-6 does fewer node expansions than normal HIDA* using finer abstractions Blank-3 and Blank-4.

The search time shows a similar trend to total node expansions. When the abstraction is finer-grained and contains fewer spurious paths, normal HIDA* is faster since the heuristic is still informative and SPECO costs more time than it saves. When the abstraction gets coarser, the spurious path problem becomes dominant. In such abstract spaces, removing spurious paths reduces the total search time.

We also record the number of cache entries used in HIDA*. For the four coarsest abstractions, HIDA* with

SPECO needs 2-3 times fewer total cache entries. On the abstract level, SPECO needs more cache entries (shown in column "Cache-1"). However, those SPECO cache entries store more accurate heuristic values that are important for reducing the total expansions and total cache usage.

Results with No Concrete Level Cache. Now we analyze HIDA* that does not use a cache in the concrete level. The advantage of disallowing concrete level cache is that it saves space (compare "Cache" on the left and "Cache-1" on the right).

First of all, although the data in the two sub-tables show some differences, we can see that the expansions in the sub-tables have the same trend, namely that HIDA* with SPECO outperforms normal HIDA* in terms of node expansions.

There are two important differences between using and not using a concrete level cache. First, without a concrete level cache, there is no P-g caching in the concrete level. This explains why the expansions are much higher when a concrete level cache is not used. The second difference is that when there is no concrete level cache, we have to acquire heuristic values from the abstract level cache, while with concrete level cache, we get heuristic from the concrete level cache. Acquiring the heuristic value from a cache entry of a SPECO state is slower because it needs to check the preceding operator. As a result, HIDA* with SPECO is slower even when it expands many fewer nodes than normal HIDA* which has no SPECO state. Our current implementation just uses a naive data structure for SPECO states, and it may be possible to alleviate the lookup time overhead by using a better data structure.

Related Work

Spurious path have been studied primarily in the setting of abstraction refinement. The most notable example is *Counter-Example Guided Abstraction Refinement (CEGAR)* (Clarke et al. 2000; Seipp and Helmert 2013). CEGAR iteratively refines its abstractions by fixing "errors" (a point at which the abstract solution cannot be converted to a concrete solution) in the abstract solution. Each refinement iteration first identifies an error in abstract abstraction, and then refines the abstraction so that the same error will never occur again. Similar ideas have been used in many other abstraction refinement methods (e.g., (Clarke et al. 2002; Smaus and Hoffmann 2008; Sharygina, Tonetta, and Tsitovich 2009)). One difference between refinements in CEGAR and our method is that in CEGAR the refinement tries to produce a non-spurious solution path while in our method the refinement only makes sure paths of fixed length k (e.g., $k = 2$ in our 8-Puzzle experiment) are not spurious. Thus, our refinement is less accurate but more efficient than CEGAR. Another important difference is that most abstraction refinements are predicate-based, i.e., an abstraction is refined by adding predicates that are ignored in the abstraction, while our refinement is based on operator sequences. *Coarse-to-Fine Dynamic Programming (CFDP)* (Raphael 2001) bears some resemblance to CEGAR and our refinement method in that it refines the abstraction by splitting the abstract states on the abstract solution path. However, CFDP refines the abstraction to ensure that the optimal abstract solution corresponds to the optimal concrete solution.

Abstraction refinement has also been used to improve heuristics. Smaus and Hoffmann (2008) apply predicate abstraction refinement for generating heuristic functions. Seipp and Helmert (2013) use the abstraction in last iteration of CEGAR to provide heuristics for optimal planning if the refinement fails to find a concrete path within the given time and space limit. A drawback of these methods is that the refinement only focuses on the region around the solution path from the abstract initial state to the abstract goal. Other regions of the state space are less refined. The resulting abstract space will provide better heuristic values for the states close to the abstract solution region but worse (lower) heuristic value for other states. Therefore, the concrete search tends to expand states outside the refined areas. Smaus and Hoffmann noticed this issue, and use refinement on multiple random solution paths in an effort to alleviate this effect. Our application of SPECO to refine abstraction for better heuristics for HIDA* does not have this problem, because we refine the solution path for every state on which the heuristic is requested. The refined abstract space has the same granularity (regarding spuriousness) for the all states that are evaluated. Thus, the search will not be misled into less refined areas. *Static Abstraction Guided* model-checking (Qian and Nymeyer 2004) is similar to our method because it also uses a pre-defined abstraction hierarchy, and uses abstract distances in a coarser abstraction as a heuristic to guide the search in the next finer abstract space. They also check spurious paths, but the spuriousness is defined with respect to the finer abstraction at the next level.

In addition to the precondition-preserving property and bisimularity, there are other abstraction properties related to spurious paths. Bäckström and Jonsson (2013) introduce a number of properties regarding abstraction, and an abstraction property $\mathbf{P}^m_{k\downarrow}$ was briefly discussed as their future work. $\mathbf{P}^m_{k\downarrow}$ implies any abstract path $(t_0, t_1, ..., t_k)$ can be refined into a sequence of concrete states $s_0, s_1, ..., s_k$ s.t. there is a concrete path of length at most m from s_{i-1} to s_i for all $i \in \{1, 2, ..., k\}$. Thus, $\mathbf{P}^1_{k\downarrow}$ means the abstraction is free of spuriosu paths of length k. The downward refinement property (DRP) (Bacchus and Yang 1994) is another abstraction property related to spurious paths. DRP requires any abstract solution to be refinable to a concrete solution.

There are many mutex detection methods invented for different planing systems and for different purposes (e.g., (Dawson and Siklóssy 1977; Blum and Furst 1997; Gerevini and Schubert 2000; Rintanen 2000; Bonet and Geffner 2001; Haslum and Geffner 2000; Helmert 2009; Sadeqi 2014)). We use mutex information to discover invalid operator chains. This is related to Haslum, Bonet and Geffner (2005)'s work that avoids applying an operator to a state if the state is mutex with the precondition of the operator, and Alcázar et al. (2013)'s action pruning using *e-deletion*, which also uses mutexes to identify "impossible" actions. Both the methods check if applying an operator leads to a state that contains mutex. Haslum et al.'s method is different from Alcázar et al.'s and ours in that their method checks the application

of an operator to a particular abstract state during search, whereas e-deletion and our method do not rely on particular states and can precompute the inapplicability of operators in a preprocessing step.

Discussion

In CEGAR-like abstraction refinements, in order to discover an error, one has to applying the operator sequence used in the abstract solution to the initial state to generate a concrete solution. As soon as an error is found, the path generation has to stop, and a refinement needs to be done, because the error prevents the applications of subsequent operators in the operator sequence. Thus, CEGAR-like refinement only fixes the *first* error in an abstract path, and ignore any other errors after the first one. SPECO, however, does not have this restriction, because it does not use concrete path generation to discover errors. SPECO can pick any error in the path to fix. The current implementation fixes the *last* error. In fact, our preliminary experiments showed that fixing the last error in a path resulted in fewer node expansions than fixing the first error. It is worth exploring which spurious error in the path should be fixed first in abstraction refinement.

If the spurious path elimination is more about improving a heuristic than finding a concrete solution, like our method and others' (Smaus and Hoffmann 2008; Seipp and Helmert 2013; Qian and Nymeyer 2004), the abstraction refinement could be more "relaxed", because it should care more about the solution cost than the actual solution. For example, we have shown that w_4 in Figure 3 is a spurious path. Our current method removes this path. However, if C is the abstract goal state, w_4 is not a short-cut (C can be reached from A through the path (A,B,E,C) which has the same cost as w_4), i.e., keeping w_4 in the abstract space will not harm the heuristic quality. This fact can be exploited to avoid unnecessary refinement. In addition, since we only care about solution cost, we could use *label reduction*, i.e., a mapping from the original label set to a new (smaller) label set, to reduce the complexity of refinement. For instance, the current SPECO may change a state t to two states (t, o_1) and (t, o_2) (assume o_1 and o_2 are preceding operators of t), but with the label reduction $\phi(o_1) = \phi(o_2) = o$ we would only need to change t to one state (t, o).

To get the set of operator chains $\mathcal{C}_k$, one needs to check, in the worst case, all $|\mathcal{O}|^k$ operator sequences. Since we use PSVN in our experiments, the size of $\mathcal{O}$ is not a problem for computing $\mathcal{C}_k$. This computation, however, would be infeasible if $|\mathcal{O}|$ is larger (even for small k). To avoid this problem, instead of precomputation of $\mathcal{C}_k$, we could apply the "macro-operator" composition process on operator sequences during the search. This would make it possible to apply SPECO on a problem represented by SAS$^+$, which often has exponentially many more operators than PSVN.

SPECO does not have to be used in conjunction with HIDA*, we originally used it during pattern database (PDB) construction to improve the heuristic quality of the PDB. A PDB requires a solution path for every abstract state, so to obtain the most accurate heuristic every abstract state should be expanded as a SPECO state. This makes the abstract space much larger than it would normally be. We did not keep this abstract space as our final PDB. Instead, we stored the minimum cost for an abstract state t over all preceding operator sequences of t, i.e., $h(t) = \min_{\pi \in Q} h(t, \pi)$ where Q is the set of preceding operator sequences of t. The final PDB therefore had the same size as the normal PDB, but it provided a better heuristic. Our experiments with SPECO applied to PDB construction were conducted on the 3×4 Sliding Tile Puzzle. Similar to the HIDA* experiment, we used domain abstractions in which 5 tiles were mapped to the blank (so there were 6 blanks in each abstract state) and varied which tiles were mapped to the blank. For SPECO, we set $k = 2$ to eliminate spurious paths of length 2. The number of concrete node expansions was between 126 and 762 times smaller when the SPECO PDB was used compared to the normal PDB, depending on which 5 tiles were mapped to the blank. However, the intermediate abstract space was about 17 times larger when SPECO was used. This increase may be acceptable in some circumstances, especially considering the speedup obtained, but our conclusion was that SPECO is better suited to hierarchical search methods such as HIDA* than PDB construction.

Conclusion

We define a spurious path as a generalization of a spurious state. We showed that spurious paths can be categorized into two types: state-independent and state-specific. We presented SPECO—a novel method based on syntactical checking that eliminates state-independent spurious paths, as well as state-specific spurious paths when integrated with mutex detection methods. We gave syntactical conditions under which our method can remove state-independent spurious paths completely. We demonstrated that spurious paths may exist in abstract spaces of typical search domains such as the Sliding Tile Puzzle. We tested our method on 8-Puzzle as a case study. The results show that spurious paths are harmful and removing them substantially increases the heuristic quality.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We thank Martin Wehrle for pointing us to related work in the model-checking literature, and Mehdi Sadeqi for providing the h^2 program that computes mutexes for PSVN. We gratefully acknowledge the funding provided by Canada's NSERC.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *Proc. 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*.

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

Bäckström, C., and Jonsson, P. 2013. Bridging the gap between refinement and heuristics in abstraction. In *Proc. 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2261–2267.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS^+ planning. *Computational Intelligence* 11:625–656.

Blum, A., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

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

Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2000. Counterexample-guided abstraction refinement. In *Proc. 12th International Conference on Computer Aided Verification (CAV 2000)*, 154–169.

Clarke, E. M.; Gupta, A.; Kukula, J. H.; and Strichman, O. 2002. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. 14th International Conference on Computer Aided Verification (CAV 2002)*, 265–279.

Clarke, E. M.; Grumberg, O.; and Long, D. E. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16(5):1512–1542.

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

Dawson, C., and Siklóssy, L. 1977. The role of preprocessing in problem solving systems. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI 1977)*, 465–471.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. 6th European Conference on Planning (ECP 2001)*, 13–24.

Gerevini, A., and Schubert, L. K. 2000. Discovering state constraints in DISCOPLAN: some new results. In *Proc. 17th National Conference on Artificial Intelligence (AAAI 2000)*, 761–767.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. 5th International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*, 140–149.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proc. 20th National Conference on Artificial Intelligence (AAAI 2005)*, 1163–1168.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM* 61(3):16.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Hernádvölgyi, I. T., and Holte, R. C. 1999. PSVN: A vector representation for production systems. Technical Report TR-99-04, Department of Computer Science, University of Ottawa.

Hernádvölgyi, I. T., and Holte, R. C. 2000. Experiments with automatically created memory-based heuristics. In *Proc. 4th Symposium on Abstraction, Reformulation and Approximation (SARA 2000)*, 281–290.

Holte, R. C., and Burch, N. 2014. Automatic move pruning for single-agent search. *AI Communication* 27(4):363–383.

Holte, R. C.; Arneson, B.; and Burch, N. 2014. PSVN manual. Technical Report TR14-03, Computing Science Department, University of Alberta.

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proc. 13th National Conference on Artificial Intelligence (AAAI 1996)*, 530–535.

Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical heuristic search revisited. In *Proc. 6th Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, 121–133.

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

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97 – 109.

Milner, R. 1980. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer.

Milner, R. 1990. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. Elsevier and MIT Press. 1201–1242.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *Proc. 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1983–1990.

Qian, K., and Nymeyer, A. 2004. Abstraction-based model checking using heuristical refinement. In *Proc. 2nd International Conference on Automated Technology for Verification and Analysis (ATVA 2004)*, 165–178.

Raphael, C. 2001. Coarse-to-fine dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 23(12):1379–1390.

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *Proc. 17th National Conf. on Artificial Intelligence (AAAI 2000)*, 806–811.

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

Sadeqi, M.; Holte, R. C.; and Zilles, S. 2013. Using coarse state space abstractions to detect mutex pairs. In *Proc. 10th Symposium on Abstraction, Reformulation and Approximation (SARA 2013)*, 104–111.

Sadeqi, M. 2014. *Mutex Pair Detection for Improving Abstraction-Based Heuristics*. Ph.D. Dissertation, University of Regina.

Seipp, J., and Helmert, M. 2013. Counterexample-guided Cartesian abstraction refinement. In *Proc. 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 347–351.

Sharygina, N.; Tonetta, S.; and Tsitovich, A. 2009. The synergy of precise and fast abstractions for program verification. In *Proc. 2009 ACM Symposium on Applied Computing*, 566–573.

Smaus, J., and Hoffmann, J. 2008. Relaxation refinement: A new method to generate heuristic functions. In *Model Checking and Artificial Intelligence, 5th International Workshop (MoChArt 2008)*, 147–165.

Zilles, S., and Holte, R. C. 2010. The computational complexity of avoiding spurious states in state space abstraction. *Artificial Intelligence* 174(14):1072–1092.