State-Set Search

Bo Pang and **Robert C. Holte**
Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
(bpang,rholte@ualberta.ca)

Abstract

State-set search is state space search when the states being manipulated by the search algorithm are sets of states from some underlying state space. State-set search arises commonly in planning and abstraction systems, but this paper provides the first formal, general analysis of state-set search. We show that the state-set distance computed by planning systems is different than that computed by abstraction systems and introduce a distance in between the two, d_{ww}, the maximum admissible distance. We introduce the concept of a multi-abstraction, which maps a state to more than one abstract state in the same abstract space, describe the first implementation of a multi-abstraction system that computes d_{ww}, and give initial experimental evidence that it can be superior to domain abstraction.

Introduction

This paper investigates state space search in the setting where the states being manipulated by the search algorithm are, in fact, sets of states from some underlying state space. To maintain a clear distinction between these two types of states, we always use "state" to refer to the states in the underlying space and "state-set" to refer to the states the search algorithm is manipulating. Hence we refer to this type of state space search as state-set search.

State-set search occurs frequently in the literature. Although classical planning systems search forward from a fully-specified start state, and therefore do not do state-set search, there are several situations in which planners do state-set search. The first is conformant planning, which Bonet and Geffner (2000) define as having a set of start states instead of just one. This is supported by the SAS+ formalism, which allows state variables to be assigned a special value (u) to indicate that the actual value of the variable is not known (Bäckström 1995). Any system that fully supports SAS+ must support state-set search. Another situation in which planners must reason about state-sets is in regression planning (Rintanen 2008). A planning goal is a set of conditions to be satisfied; this most often defines a state-set, not one particular state. Regression planning systems search backwards from the goal until the start condition is satisfied. Even if the start condition is a fully specified state, all the intermediate reasoning during search involves state-sets. If bidirectional search (Kaindl and Kainz 1997) were to be used in a planning context, it would involve regression planning and therefore involve state-set search.

The other commonly occurring situation that involves state-set search is search in abstract spaces. In most types of abstraction, an abstraction of a state space S is another state space whose individual states represent sets of states in S (Yang et al. 2008).

Despite its being commonplace, state-set search has never been subject to a general analysis. Our main contribution is to give a formal definition and analysis that is not specific to how the state-sets are represented. The most interesting outcome of this analysis is that there are at least four natural, distinct ways to define the notions of "path" and "distance" in a space of state-sets. We show that planning systems use a different notion of path and distance than abstraction systems and that Haslum and Geffner's (2000) method for computing the h^m heuristic cannot be used in a forward search setting. We show that there is an admissible abstract distance, d_{ww}, capable of returning larger values than today's abstraction systems, and also that d_{ww} does not permit efficiency to be gained in hierarchical search by alternating search directions (Larsen et al. 2010). Finally, we introduce the idea of a "multi-abstraction", which maps a state to more than one abstract state (state-set) in the same abstract space.

The paper ends with a description of the first implementation of a multi-abstraction system that computes d_{ww}. An initial experiment shows that the new system is superior to domain abstraction in terms of nodes generated and CPU time while using no more memory.

Formal Analysis

Space precludes giving proofs for most of the formal results in this paper. Most are straightforward applications of elementary facts about set intersection and containment.

Definition 1 (State-set) *Let S be a non-empty set of states. A state-set (with respect to S) is a non-empty subset of S. We equate state $s \in S$ with the state-set $\{s\}$.*

Notationally, we allow a function $f : A \to B$ to be applied to a subset $A' \subseteq A$ instead of just an element of A, with $f(A') = \{f(a) \mid a \in A'\}$. Under this convention $f(\emptyset) = \emptyset$.

Definition 2 (State Multimap) *Let S be a non-empty set of states. A state multimap ω on S is a function from S to 2^S, the powerset of S. If P is a state-set, $\omega(P) = \bigcup_{s \in P} \omega(s)$.*

Definition 3 (State Space, Operator) *A state space is a pair $\mathcal{S} = (S, \Omega)$ where S is a non-empty set of states, and Ω is a set of state multimaps on S called operators. For each operator $\omega \in \Omega$ the set of states to which ω can be applied is $PRE_\omega = \{s \in S \mid \omega(s) \neq \emptyset\}$ and the set of states that can possibly be produced by ω is $POST_\omega = \omega(S)$. Without loss of generality we assume for all $\omega \in \Omega$ that PRE_ω is non-empty and therefore it and $POST_\omega$ are both state-sets.*

By defining an operator to be a multimap, we permit one operator to generate more than one successor of a state. This is not to be interpreted nondeterministically—the operator produces all the successors not just one of them.

These definitions do not allows costs to be associated with operators. We leave this extension for future work.

Definition 4 (State Distance) *Let $\mathcal{S} = (S, \Omega)$ be a state space. A finite sequence of operators $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ is applicable to state $s \in S$ iff $\pi(s) = \omega_z(\omega_{z-1}...\omega_2(\omega_1(s))...) \neq \emptyset$. π is a path from state $s \in S$ to state $t \in S$ iff $t \in \pi(s)$. The distance $d(s,t)$ between two states is the length of the shortest path from s to t (∞ if no such path exists).*

Definition 5 (State-set Space) *Let $\mathcal{S} = (S, \Omega)$ be a state space. The state-set space induced by $\mathcal{S}$ is $\mathcal{SS} = (2^S, \Omega)$.*

State-set Matching, Paths, and Distances

Definition 6 (Strong/Weak Match) *Let $\mathcal{S} = (S, \Omega)$ be a state space and P and Q state-sets w.r.t. S. We say that P strongly matches Q iff $P \subseteq Q$, and that P weakly matches Q iff $P \cap Q \neq \emptyset$. We will say simply that P matches Q where strongly/weakly is determined by the context or where either definition can be used.*

Definition 7 *Let $\mathcal{S} = (S, \Omega)$ be a state space and P a state-set w.r.t. S. Operator ω is strongly/weakly applicable to P iff P strongly/weakly matches PRE_ω.*

In this and all other definitions, theorems *etc.*, "strongly/weakly" means that the definition, theorem, *etc.* holds whenever either one of "strongly" or "weakly" is used throughout. For example, Definition 7 is actually two definitions: ω is strongly applicable to P iff P strongly matches PRE_ω, and ω is weakly applicable to P iff P weakly matches PRE_ω.

Definition 7 is shown graphically in Figure 1. State-sets are depicted with circles. State-sets representing operator preconditions are shaded. The upper part of Figure 1 shows that operator ω's precondition contains state-set P; ω is therefore strongly applicable to P. In the lower part of Figure 1 operator ω's precondition intersects P but does not contain it; ω is therefore only weakly applicable to P. In both parts of the figure $Q = \omega(P) = \omega(P \cap PRE_\omega)$.

Definition 8 *Let $\mathcal{S} = (S, \Omega)$ be a state space and P a state-set w.r.t. S. A finite sequence of operators $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ is strongly/weakly applicable to P iff $z = 1$ and ω_1 is strongly/weakly applicable to P, or $z > 1$, ω_1 is strongly/weakly applicable to P, and the sequence $(\omega_2, \ldots, \omega_z)$ is strongly/weakly applicable to $\omega_1(P)$.*

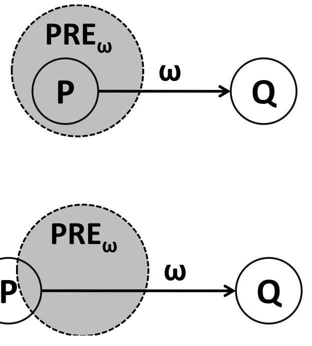

Figure 1: (upper) Operator ω is strongly applicable to P. (lower) Operator ω is weakly applicable to P.

These definitions immediately imply the following.

Corollary 1 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S such that $P \supseteq Q$, and $\pi \in \Omega^+$ a finite sequence of operators that is strongly/weakly applicable to both P and Q. Then $\pi(P) \supseteq \pi(Q)$.*

Corollary 2 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S such that $P \supseteq Q$, and $\pi \in \Omega^+$ a finite sequence of operators that is weakly applicable to Q. Then π is weakly applicable to P.*

Corollary 3 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S such that $P \supseteq Q$, and $\pi \in \Omega^+$ a finite sequence of operators that is strongly applicable to P. Then π is strongly applicable to Q.*

Definition 9 (State-set Distance) *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S, and $\pi \in \Omega^+$ a finite sequence of operators that is applicable to P. π is a path from P to Q iff $\pi(P)$ matches Q. The distance from P to Q, denoted $d(P,Q)$, is the length of the shortest path from P to Q (∞ if no such path exists). We say that Q is reachable from P if there exists a path from P to Q.*

There are actually four different definitions of path, distance, and reachable here, depending on whether strong or weak matching is used to test if π is applicable to P and whether strong or weak matching is used to test if $\pi(P)$ matches Q. The two definitions we will focus on are:

- When strong matching is used throughout Definition 9, a path from P to Q is applicable to all states in P and guaranteed to map each state in P to a state in Q. We call such paths "strong paths" and denote the corresponding distance as $d_{ss}(P,Q)$.
- When weak matching is used throughout Definition 9, a path from P to Q maps at least one state in P to a state in Q. We call such paths "weak paths" and denote the corresponding distance as $d_{ww}(P,Q)$.

Figure 2 depicts these two definitions, with the upper part showing a strong path from P to Q and the lower part showing a weak path from P to Q. Because every strong path is also a weak path, $d_{ww}(P,Q) \leq d_{ss}(P,Q)$. It can easily happen that $d_{ss}(P,Q)$ is infinite but $d_{ww}(P,Q)$ is finite. For example, this would happen with the sliding tile puzzle if P contained states in which the blank was in different locations.

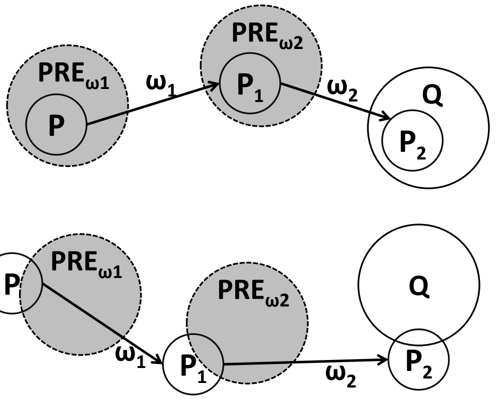

Figure 2: Strong (upper) and weak (lower) paths from P to Q.

Properties of Weak/Strong Paths and Distances

Theorem 4 *Let $\mathcal{S} = (S,\Omega)$ be a state space, P, Q, and R state-sets w.r.t. S such that $R \supseteq Q$, and π a weak/strong path from P to Q. Then π is a weak/strong path from P to R and $d_{ww/ss}(P,R) \leq d_{ww/ss}(P,Q)$.*

It is because of Theorem 4 that the h^m method can be applied in backwards search. See Corollary 16.

Theorem 5 *Let $\mathcal{S} = (S,\Omega)$ be a state space, P, Q, and R state-sets w.r.t. S such that $R \supseteq P$, and π a weak path from P to Q. Then π is a weak path from R to Q and $d_{ww}(R,Q) \leq d_{ww}(P,Q)$.*

The preceding two theorems show that if we have two state-sets, P and Q, and "abstract" them to supersets $P' \supseteq P$ and $Q' \supseteq Q$, then the "abstract" distance $d_{ww}(P',Q')$ is a lower bound on $d_{ww}(P,Q)$. This is not true for d_{ss}, since Theorem 5 does not hold for strong paths. Indeed, as the next theorem shows, the opposite holds. It immediately follows that d_{ss} cannot, in general, be used in abstraction systems.

Theorem 6 *Let $\mathcal{S} = (S,\Omega)$ be a state space, P, Q, and R state-sets w.r.t. S such that $R \subseteq P$, and π a strong path from P to Q. Then π is a strong path from R to Q and $d_{ss}(R,Q) \leq d_{ss}(P,Q)$.*

It is because of Theorem 6 that the h^m method cannot be applied in forward search. But to prove that we also need to prove that h^m is based on strong paths/distances. This is Corollary 16 below.

Theorem 7 $d_{ww}(P,Q) = \min_{p \in P, q \in Q} d(p,q)$.

This theorem shows that d_{ww} has the property that is needed to guarantee that "abstracting" states (or state-sets) by mapping them to supersets and using the distances between supersets as an estimate of the distances between the original states (or state-sets) will be an admissible heuristic. The other distance measures defined in Definition 9 do not have this property, so a general-purpose abstraction system cannot use them if admissibility is required. Moreover, $d_{ww}(P,Q)$ is the largest distance from P to Q that is guaranteed to be an admissible estimate of the distance between a subset of P and a subset of Q.

The following shows that d_{ss} obeys the triangle inequality.

Lemma 8 *Let $\mathcal{S} = (S,\Omega)$ be a state space, and P, Q and R state-sets w.r.t. S, Then $d_{ss}(P,Q) \leq d_{ss}(P,R) + d_{ss}(R,Q)$.*

The same is not true of d_{ww}, and therefore it is not a true distance metric. Figure 3 shows an example in which d_{ww}violates the triangle inequality. $d_{ww}(P,Q) = 1$ (using operator ω) but $d_{ww}(P,R) = 0$ because $P \cap R \neq \emptyset$ and $d_{ww}(R,Q) = 0$ as well. Hence $d_{ww}(P,Q) > d_{ww}(P,R) + d_{ww}(R,Q)$. Since the standard proof of the consistency of heuristics defined by distances in an abstract space relies on the abstract distances obeying the triangle inequality, the fact that d_{ww} does not obey the triangle inequality raises doubts about the consistency of heuristics based on computing d_{ww} in an abstract space. We return to this issue later.

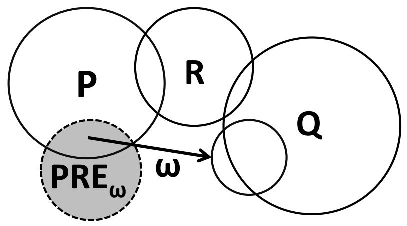

Figure 3: Example of d_{ww}violating the triangle inequality.

Theorem 9 *Let $\mathcal{S} = (S,\Omega)$ be a state space, P and Q state-sets w.r.t. S, $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ a shortest weak/strong path from P to Q, and R_i the i^{th} intermediate state-set along the path (i.e., $R_i = (\omega_1, \omega_2, \ldots, \omega_i)(P)$ for $i \in \{1, \ldots, z-1\}$). Then $d_{ww/ss}(R_i,Q) = z - i$, $d_{ss}(P,R_i,) = i$, and $d_{ww}(P,R_i,) \leq i$.*

The surprising part of Theorem 9 is the last part, that $d_{ww}(P,R_i)$ can be less than i. Figure 4 illustrates how this can happen. The shortest weak path from P to Q passes through R_1 and then R_2 but there is weak path directly from P to R_2 so $d_{ww}(P,R_2) = 1$ even though R_2 is distance two from P on the shortest path to Q. As the figure shows, this happens because the path directly from P to R_2 intersects with R_2 in a state-set (X) from which Q cannot be reached.

Because $d_{ww/ss}(R_i,Q) = z - i$, systems like HIDA* (Holte, Grajkowski, and Tanner 2005) and Hierarchical A* (Holte et al. 1996), which compute abstract distances by searching in the forward direction (from start to goal), will work correctly for any method that abstracts states (or state-sets) by mapping them to supersets. By contrast, because $d_{ww}(P,R_i)$ might be less than i, systems that

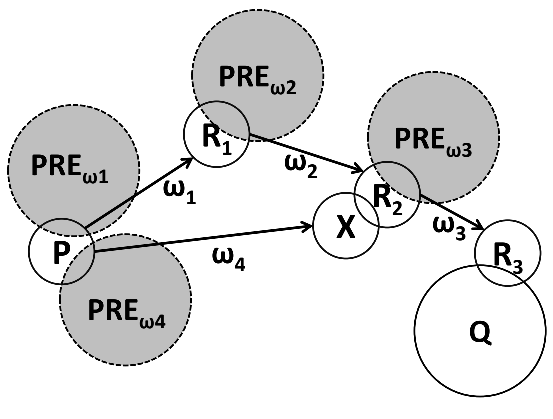

Figure 4: The shortest weak path from P to R_2 on the way to Q is not necessarily the shortest weak path from P to R_2.

compute abstract distances by searching backwards in the abstract space, such as Switchback (Larsen et al. 2010), cannot, in general, be used in conjunction with d_{ww} because the weak distances-from-goal they cache for intermediate state-sets along the paths they find, such as R_2 in Figure 4,[1] may be larger than the actual weak distance from the goal. However, there are special circumstances that permit these efficiencies to be obtained even though weak distances are being computed. These are captured in the next definition and theorem.

Definition 10 (Simple Space) *Let $\mathcal{S} = (S, \Omega)$ be a state space and $\mathcal{SS} = (2^S, \Omega)$ the state-set space induced by S. State-set $P \in \mathcal{SS}$ is "simple" if for all state-sets Q and R reachable from P, either $Q = R$ or $Q \cap R = \emptyset$. $\mathcal{SS}$ is simple if all state-sets $P \in \mathcal{SS}$ are simple.*

Theorem 10 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S, P a simple state-set, $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ a weak path from P to Q, and R_i the i^{th} intermediate state-set along the path (i.e., $R_i = (\omega_1, \omega_2, \ldots, \omega_i)(P)$ for $i \in \{1, \ldots, z-1\}$). Then $d_{ww}(P, R_i,) = i$.*

Inverting Operators and Paths

Definition 11 (Inverse Operator) *Let $\mathcal{S} = (S, \Omega)$ be a state space and $\omega \in \Omega$ an operator. The inverse of ω, denoted ω^{-1}, is defined to be the state multimapping, $\omega^{-1} : POST_\omega \to PRE_\omega$, such that, for any state $r \in POST_\omega$, $\omega^{-1}(r)$ is defined to be $\{s \in PRE_\omega \mid r \in \omega(s)\}$. If $r \notin POST_\omega$ then $\omega^{-1}(r) = \emptyset$.*

The definition immediately implies the following.

Lemma 11 *Let $\mathcal{S} = (S, \Omega)$ be a state space and $\omega \in \Omega$ an operator. Then:*

1. *ω^{-1} is an operator in the sense of Definition 3.*
2. *$(\omega^{-1})^{-1} = \omega$, the inverse of ω^{-1}, is ω.*
3. *For any state $s \in PRE_\omega$ and any state $t \in \omega(s)$, $s \in \omega^{-1}(t)$.*
4. *$PRE_{\omega^{-1}} = POST_\omega$.*
5. *$POST_{\omega^{-1}} = PRE_\omega$.*
6. *For any state-set P to which ω is strongly/weakly applicable, ω^{-1} is strongly applicable to $\omega(P)$ and $(P \cap PRE_\omega) \subseteq \omega^{-1}(\omega(P))$.*
7. *For any state-set P to which ω^{-1} is strongly/weakly applicable, ω is strongly applicable to $\omega^{-1}(P)$ and $(P \cap POST_\omega) \subseteq \omega(\omega^{-1}(P))$.*

The last two parts of Lemma 11 show that ω^{-1} in not guaranteed to be a true inverse of ω and that neither is ω guaranteed to be a true inverse of ω^{-1}. To be true inverses they should exactly reverse each other's actions, which would require $(P \cap PRE_\omega) = \omega^{-1}(\omega(P))$ and $(P \cap POST_\omega) = \omega(\omega^{-1}(P))$.

Theorem 12 (Inverses of Weak Paths) *Let $\mathcal{S} = (S, \Omega)$ be a state space, P a state-set, $O = \langle \omega_1, \ldots, \omega_k \rangle$ an operator sequence that is weakly applicable to P, and Q any state-set such that $Q \cap O(P) \neq \emptyset$. Then $O^{-1} = \langle \omega_k^{-1}, \ldots, \omega_1^{-1} \rangle$ is weakly applicable to Q and $P \cap O^{-1}(Q) \neq \emptyset$.*

Planning as State-Set Search

Backstrom (1995) describes two planning formalisms that explicitly support reasoning about state-sets (called partial states by Backstrom), Grounded TWEAK and SAS+. Although these are planning formalisms, and not planning systems, they formally define the semantics of operator applicability and goal testing. The following theorems show that in both formalisms strong matching is used for both operator applicability and goal testing, and therefore systems based on these formalisms will be computing strong paths.

Theorem 13 *Grounded TWEAK, as described by Backstrom (1995), uses strong matching for testing operator applicability and goal satisfaction.*

Proof Sketch. In Grounded TWEAK a state is a set of literals, positive or negative facts that are known to hold in the state. Atoms that have no corresponding literal in a state may be either true or false. Hence, every "state" s in the Grounded TWEAK parlance represents the set of states, in our parlance, for which the literals in s are true and the atoms having no corresponding literal in s have all possible combinations of truth value assignments. Operator preconditions are also sets of literals, and precondition p is satisfied by state s if s contains all the literals in p. From a state-set point of view, this means that s must be a subset of p (having all the literals in p, and possibly more, means that s represents a subset of the states that p represents). In other words, Grounded TWEAK does strong matching to determine if an operator applies to a state. The goal is also a set of literals, and state s satisfies the goal if s contains all the literals in the goal. Again, this is strong matching. The only point that remains to be proved to finish the proof of the theorem is that the action of operators on a "state" (state-set) in Grounded TWEAK is exactly as defined for state-sets in Definition 3. This is true, but we omit the proof. □

[1]In this example, Switchback is searching backwards from the goal, P, towards Q, so the distance from P to R_2 is R_2's distance to goal.

Theorem 14 *SAS+, as described by Backstrom (1995), uses strong matching for testing operator applicability and goal satisfaction.*

Proof Sketch. States in SAS+ are exactly as we define states in Definition 12 below: vectors of a fixed length, with each position in the vector having its value drawn from a finite set of possible values. In addition, SAS+ has a special symbol (u) that allows a state to specify that the value in one or more of its positions is unknown. An SAS+ state with one or more u values therefore represents the set of states which agree on the known values and have all possible combinations of values for the unknown positions.

Operator preconditions, in our sense, are divided into preconditions and prevail conditions in SAS+, but here we will call them all preconditions. An SAS+ state s satisfies the preconditions p of an operator if the values specified by p are known in s. From a state-set point of view, this means that s must be a subset of p (having known values that agree with those specified in p, and possibly more known values, means that s represents a subset of the states that p represents). Hence, SAS+ does strong matching to determine if an operator applies to a state. The goal is defined exactly like a precondition, and matching a state to the goal is done exactly as matching a state to a precondition is done, hence SAS+ also does strong matching to determine if a state matches the goal. The only point that remains to be proved to finish the proof of the theorem is that the action of operators on a "state" (state-set) in SAS+ is exactly as defined for state-sets in Definition 3. This is true, but we omit the proof. □

In regression planning, as defined by Haslum and Geffner (2000), a path (and distance) from state-set P to state-set Q is computed by searching backwards, with reverse operators, from Q until a superset of P is reached. We call them reverse operators, not inverses, because they behave slightly differently than the inverse operators of Definition 11. In particular, if ω^r is the reverse of operator ω, then $PRE_{\omega^r} \supseteq POST_\omega$[2] and $POST_{\omega^r} \subseteq PRE_\omega$,[3] whereas equality is required in these equations for inverse operators. These properties of reverse operators are sufficient to prove the following.

Theorem 15 *The length of the shortest path from P to Q found by regression planning, as defined by Haslum and Geffner (2000), is $d_{ss}(P,Q)$.*

The h^m distance from P to Q defined by Haslum and Geffner (2000) is a variation on regression planning in which, under certain circumstances, a state-set R reached during the regression search from Q to P is replaced by a superset $R' \supseteq R$.[4]

[2]states in $POST_\omega$ contain all of the atoms in $Add(\omega)$ and none of the atoms in $Del(\omega)$ but a state can be regressed through ω as long as it has none of the atoms in $Del(\omega)$ and at least one of the atoms in $Add(\omega)$. Hence ω^r can be applied to states that do not contain all the atoms in $Add(\omega)$ and are therefore not in $POST_\omega$.

[3]The states that are in PRE_ω but not in $POST_{\omega^r}$ are those that contain all the atoms in $Add(\omega)$.

[4]Haslum and Geffner's paper is written in the classical planning setting, which assumes the start state (P here, s_0 in their notation) is a fully specified state, but their definition and proof that it is a lower bound have no reliance on this assumption and so can be applied without modification to the more general setting where P is a state-set.

Corollary 16 *$h^m(P,Q)$, as defined by Haslum and Geffner (2000), is a lower bound on $d_{ss}(P,Q)$.*

Proof. This follows immediately from Theorem 15 and Theorem 4. □

If the same idea was used in forward planning, *i.e.,* replacing a state R reached while searching forward from P to Q with a superset $R' \supseteq R$, the distance calculated would be an upper bound on $d_{ss}(P,Q)$, not a lower bound (*cf.* Theorem 6).

The State-Set View of Abstraction

In order to formally analyze existing abstraction systems, we first need to formally define them. Our formalization is a blend of ideas from Zilles and Holte (2010) and Yang *et al.* (2008).

Definition 12 *A vector state space is a triple $\mathcal{S} = (n, \{D_1, D_2, \ldots, D_n\}, \Omega)$ where $n \in \mathbb{N}$, each D_i is a finite set called a "domain", and Ω is a set of operators. The set of states in $\mathcal{S}$ is $S = D_1 \times D_2 \times \ldots \times D_n$.*

We consider two types of abstraction of vector state spaces.

Domain abstraction. A domain abstraction ψ of vector state space $\mathcal{S} = (n, \{D_1, D_2, \ldots, D_n\}, \Omega)$ is defined by a set $\{\psi_1, \psi_2, \ldots, \psi_n\}$ of mappings $\psi_i : D_i \rightarrow E_i$ where $E_i \subseteq D_i$ and, for at least one i, $E_i \neq D_i$. State $\langle \sigma_1, \ldots, \sigma_n \rangle \in D_1 \times D_2 \times \ldots \times D_n$ is mapped by ψ to abstract state $\langle \psi_1(\sigma_1), \ldots, \psi_n(\sigma_n) \rangle$.

Projection. A projection abstraction ψ of vector state space $\mathcal{S} = (n, \{D_1, D_2, \ldots, D_n\}, \Omega)$ is defined by a subset $\{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}$. State $\langle \sigma_1, \ldots, \sigma_n \rangle \in D_1 \times D_2 \times \ldots \times D_n$ is mapped by ψ to abstract state $\langle \sigma_{i_1}, \ldots, \sigma_{i_m} \rangle$.

In both types of abstraction, each abstract state represents a set of states, *i.e.,* is a state-set over $D_1 \times D_2 \times \ldots \times D_n$. The abstract state spaces created by both types of abstraction are simple. For projection this follows from the fact that if two abstract states are not the same, they must differ in the value of at least one variable. No state can have two different values for the same variable and therefore two different state-sets created by projection cannot have any state in common.

The abstract state spaces created by domain abstraction are also simple, for a similar reason. Two different abstract states, α_1 and α_2, created by the same domain abstraction ψ must differ in at least one position. Let i be a position in which they differ ($\alpha_1[i] \neq \alpha_2[i]$). Because ψ_i maps each value in the original domain D_i to one value in the abstract domain E_i, $\alpha_1[i] \neq \alpha_2[i]$ implies that there cannot exist a state s such that $\psi_i(s[i]) = \alpha_1[i]$ and $\psi_i(s[i]) = \alpha_2[i]$. Therefore there is no s mapped to both α_1 and α_2, so $\alpha_1 \cap \alpha_2 = \emptyset$.

If ψ is any abstraction mapping of state space $\mathcal{S}$, and ω is an operator that can be applied to state s, then it is required by the definition of an abstraction that ω be applicable to $\psi(s)$, the abstract state corresponding to s. This immediately implies that abstraction systems use weak matching to define if an operator is applicable, since there may be another state, t, such that $\psi(t) = \psi(s)$ but ω is not applicable to t.

A path from P to Q in an abstract space is a sequence of operators π such that $\pi(P) = Q$. This requirement for exact matching is more demanding than strong matching. However, for the types of abstraction we are considering (projection and domain abstraction) exact matching, strong matching, and weak matching are all equivalent when used to test if $\pi(P)$ matches Q.

Having concluded that projection and domain abstraction systems use weak matching to test operator applicability and the equivalent of weak matching to test if $\pi(P)$ matches Q, one is tempted to conclude that these systems compute d_{ww}. This is not true: $d_{abs}(P, Q)$, the distance from P to Q computed by a projection or domain abstraction system, can be strictly smaller than d_{ww}, as the following example illustrates.

Example 17 *A state is a 3-tuple of binary variables and there are only two operators, $\omega_1 : \langle 1,1,1\rangle \rightarrow \langle 0,0,1\rangle$ and $\omega_2 : \langle 1,0,1\rangle \rightarrow \langle 1,0,0\rangle$. If the first state variable is projected out, states $\langle 0,0,1\rangle$ and $\langle 1,0,1\rangle$ are mapped to the same abstract state, $\langle 0,1\rangle$, creating a path of length 2 from abstract state $\langle 1,1\rangle$ to $\langle 0,0\rangle$: $\omega_1(\langle 1,1\rangle) = \langle 0,1\rangle$ and $\omega_2(\langle 0,1\rangle) = \langle 0,0\rangle$. Thus $d_{abs}(\langle 1,1\rangle, \langle 0,0\rangle) = 2$. However, $d_{ww}(\langle 1,1\rangle, \langle 0,0\rangle) = \infty$. The difference in distances is because of a subtle difference between how $\omega_1(\langle 1,1\rangle)$ is defined in the projected space and how it is defined from a state-set point of view. In the latter, $\omega_1(\langle 1,1\rangle)$ is not $\langle 0,1\rangle$, it is $\langle 0,0,1\rangle$. This is because $\langle 1,1\rangle$ denotes the state-set $\{\langle 0,1,1\rangle, \langle 1,1,1\rangle\}$ and ω_1 maps that state-set to the state-set $\{\langle 0,0,1\rangle\}$. No operator is applicable to $\{\langle 0,0,1\rangle\}$, hence $d_{ww}(\langle 1,1\rangle, \langle 0,0\rangle) = \infty$.*

An analogous example can be given to show that the distances computed by domain abstraction systems can be strictly less than d_{ww}. We record these observations in the following theorem.

Theorem 18 *Let $d_{abs}(P, Q)$ be the distance from P to Q computed by a projection or domain abstraction system. Then $d_{abs}(P, Q) \leq d_{ww}(P, Q)$ and there exist projections and domain abstractions such that $d_{abs}(P, Q) < d_{ww}(P, Q)$ for some P and Q.*

Proof. $d_{abs}(P, Q) > d_{ww}(P, Q)$ is impossible because d_{abs} is admissible and Theorem 7 established that d_{ww} is the largest distance that is guaranteed to be admissible. Example 17, along with its analog for domain abstraction, proves the second part of the theorem. $\square$

The reasoning underlying Example 17 applies broadly: any state-set space that imposes constraints on state-set reachability beyond those implied by the operator preconditions themselves runs the risk of having to approximate the state-set produced by applying an operator with a superset in order to enforce the extra constraints. Doing this can create paths that would not otherwise exist, which can reduce distances and make state-sets reachable that would not be reachable otherwise ("spurious states" (Zilles and Holte 2010)).

On the other hand, abstraction systems that exactly compute d_{ww} run a different risk: if operators are able to reduce the cardinality of a state-set, as happens in Example 17, the number of reachable state sets might become very large. In the worst case, a sequence of operator applications might lead to a state in the original state space making the reachable portion of the abstract space a superset of the original space. This can be viewed as the "problem" that h^m solves in the context of regression planning: when the cardinality of a state-set gets too small, its cardinality is increased artificially by replacing it by a superset (actually, several supersets—see the next paragraph). Here we see that the same "problem" may occur in any abstraction system that attempts to compute d_{ww}.

The analogy with h^m leads to a final point that may give an advantage to abstraction systems that exactly compute d_{ww} over existing abstraction systems. When h^m reaches a state-set P whose cardinality is too small, it does not replace it with just one superset of a sufficiently large cardinality. It enumerates all supersets of P having a sufficiently large cardinality and uses the maximum of their distances to Q as a lower bound estimate for P's distance to Q. An abstraction system could do exactly the same: instead of computing one superset (abstraction) of the given state-set P, several could be computed and the maximum taken over all the distances thus computed. When the abstract state spaces are simple, this idea is exactly equal to using multiple abstractions (Holte et al. 2006). However, in non-simple abstract spaces, a given state (or state-set) could have multiple abstract images that are reachable from one another. We call such an abstraction mapping a "multi-abstraction". Multi-abstractions might result in memory savings over having multiple non-overlapping abstract spaces.

There is an additional reason to consider looking up several supersets of a given state (or state-set). If a consistent heuristic is desired (*e.g.,* if A* search is being done) doing just one lookup is more likely to result in inconsistency than if several lookups are done. Figure 5 illustrates this. Suppose $\mathtt{a}$ and $\mathtt{b}$ are two states and $\omega_1(\mathtt{a})=\mathtt{b}$. $h(\mathtt{a})$ is computed as $d_{ww}(A, G) = 3$ (the operator sequence is $\omega_1, \omega_2, \omega_3$), where A is a state-set containing $\mathtt{a}$ and G is the goal state-set. If $h(\mathtt{b})$ is computed as $d_{ww}(B_2, G) = 1$, an inconsistency in the heuristic values will occur. If it were computed as $d_{ww}(B_1, G)$ or $max(d_{ww}(B_1, G), d_{ww}(B_2, G))$ the heuristic values would be consistent. If the max was taken over all state-sets containing $\mathtt{b}$, consistency would be guaranteed.

We now report on our first efforts to implement multi-abstractions that compute d_{ww} exactly.

An Implementation of d_{ww} Multi-abstraction

In this section we describe an initial implementation of a multi-abstraction system for a vector state space $\mathcal{S}$ (*cf.* Definition 12) in which all n state variables have the same domain D and there is a set X of variable symbols distinct

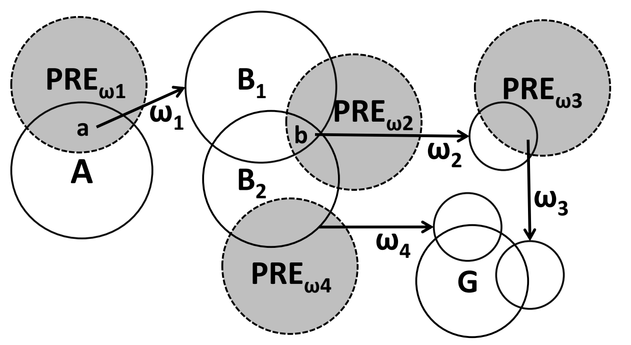

Figure 5: Potential for $d_{ww}$to produce an inconsistent heuristic.

from the values in D. Any vector V defined over $D \cup X$ in which no $x \in X$ occurs twice denotes the set of states s such that if $V_i \in D$ then $s_i = V_i$ (*i.e., s* agrees with V wherever V specifies a constant). If we replace any constant in V by a variable symbol not already in V, the resulting vector defines a superset of V. This is the how we will define an abstraction of a state-set in our implementation.

An operator is defined as a pair $\langle L, R\rangle$, where each of L and R is, like the vector V just described, a vector of length n over $D \cup X$ in which no $x \in X$ occurs twice. L is the operator's preconditions. The set of states defined by L for operator ω is, by definition, PRE_ω. R defines $POST_\omega$. In the PSVN language we are using for our implementation (Hernádvölgyi and Holte 1999; Zilles and Holte 2010) vectors L and R also define how an operator acts on a state.

Given a state-set P defined as a vector of length n over $D \cup X$ in which no $x \in X$ occurs twice, operator $\omega = \langle L, R\rangle$ is strongly applicable to P iff $L_i \in D \implies P_i = L_i$ (for all i) and is weakly applicable to P iff $L_i \in D$ and $P_i \in D \implies P_i = L_i$ (for all i).

Our abstraction method assumes that, in any set of state vectors reachable from one other, each value $d_j \in D$ occurs the same number of times, n_j. An abstraction is specified by a vector $\langle m_1, \ldots, m_{|D|}\rangle$ with $0 \leq m_j \leq n_j$ for $j \in \{1, \ldots, |D|\}$. m_j specifies how many occurrences of value a_j are to be replaced by a variable symbol. This is a multi-abstraction because, in general, there will be several ways to choose m_j of the n_j occurrences of d_j. For example, suppose $D = \{1, 2, 3\}$, the state is $\langle 1, 1, 1, 2, 3, 3\rangle$ and the abstraction is $\langle 2, 0, 1\rangle$. There are three ways to choose two of the three 1's and two ways to choose one of the two 3's, so this abstraction maps $\langle 1, 1, 1, 2, 3, 3\rangle$ to six different abstract states ($\langle 1, x_1, x_2, 2, 3, x_3\rangle$, $\langle x_1, 1, x_2, 2, 3, x_3\rangle$, $\langle 1, x_1, x_2, 2, x_3, 3\rangle$, *etc.*). By construction, all of them contain the state $\langle 1, 1, 1, 2, 3, 3\rangle$.

As an initial evaluation of this state-set abstraction method we have compared it with domain abstraction on the 8-puzzle. We use HIDA* (Holte, Grajkowski, and Tanner 2005) as our search algorithm for the reasons given in the discussion of Theorem 9. See Algorithms 1-2. In these algorithms there is a cache that contains two values for each state vector x. cache[x].dist is the best known estimate of the distance from x to the goal state, Goal. If this distance is known to be an exact distance, and not just a lower bound, cache[x].exact is true. cache[x].dist is initialized to 0, cache[x].exact to false. The same cache is used for all levels of abstraction. Algorithms 1 and 2 are almost identical to normal HIDA* in their overall structure, but there are two subtleties in Algorithm 2 that cause it to compute d_{ww}: line 1 tests if the goal has been reached by **intersecting** the current state-set with the goal, and in computing the successors of state-set s (line 7) **weak matching** is used to test if an operator applies to s.

Algorithm 1 HIDA*(Start, Goal)

```
1: bound ← h(Start, Goal)
2: while Goal not found do
3:     bound ← DFS(Start, Goal, 0, bound)
4: end while
```

Algorithm 2 DFS(Start, Goal, g, bound)

```
 1: if $Start \cap Goal \neq \emptyset$ then
 2:     cache[Start].dist ← 0
 3:     cache[Start].exact ← true
 4:     return cache[Start].dist and success // goal found
 5: end if
 6: newbound ← ∞
 7: for each $x \in$ successors(s) do
 8:     //P-g caching
 9:     cache[x].dist ← max(cache[x].dist, bound - g, h(x,Goal))
10:     f ← g + cache[x].dist
11:     // Optimal path caching
12:     if (f == bound) and (cache[x].exact == true) then
13:         cache[Start].dist ← cache[x].dist + 1
14:         cache[Start].exact ← true
15:         return cache[Start].dist and success
16:     end if
17:     if f ≤ bound then
18:         f ← DFS(x,Goal,g+1,bound)
19:     end if
20:     if Goal is found then
21:         cache[Start].dist ← f + 1
22:         cache[Start].exact ← true
23:         return cache[Start].dist and success
24:     end if
25:     if f < newbound then
26:         newbound ← f
27:     end if
28: end for
29: return newbound and failure
```

Three points in Algorithm 3 are unique to state-set multi-abstraction and d_{ww}. Line 8 looks in the cache for all entries that are supersets of the given state-set (x); each of these gives a lower bound on the (weak) distance from x to Goal so the maximum such distance can be returned (line 9). Line 4 represents a new implementation decision that arises with multi-abstractions: how many, and which, of the abstractions of a state-set, should have exact distances to Goal computed? In our implementation, we compute an exact distance to Goal for just one abstraction of state-set x; given the abstraction $\langle m_1, \ldots, m_{|D|}\rangle$ abstract(x) returns the abstraction of x in which the rightmost m_j occurrences of d_j have been replaced by variable symbols. Finally, note that Goal is **not** abstracted: exactly the same Goal is used at every abstraction level.

Algorithm 3 h(x, Goal)

```
1: if at top abstraction level then
2:     return 0
3: end if
4: absStart ← abstract(x)
5: if cache[absStart].exact == false then
6:     HIDA*(absStart, Goal)
7: end if
8: $hSet$ ← { cache[y].dist | x ⊆ y }
9: return $\max_{v \in hSet} v$
```

We use a state space as small as the 8-puzzle so that we can evaluate all possible state-set abstractions and domain abstractions of a particular granularity in a reasonable amount of time. This experiment is a proof-of-principle, not a thorough empirical study.

In order to fairly compare our state-set method with domain abstraction, we carefully chose state-set abstractions and domain abstractions so that the abstract space at each level is the same size in both cases. Level 1 is always defined by the domain abstraction that maps tiles 1-4 to 1. The mapping from Level 2 to Level 3 is chosen so that Level 3 is also always the same. For domain abstraction we consider six alternative abstractions of the same granularity to map Level 1 to Level 2 and two alternative abstractions to map from Level 3 to Level 4, giving a total of 12 combinations. We do the same for state-set abstraction: 6 different alternative for mapping Level 1 to Level 2 and two different alternatives for mapping from Level 3 to Level 4.

The 12 abstraction hierarchies of each type are evaluated by using each to solve 1000 randomly generated solvable start states. The experiment were run on an 2.5 GHz Intel E5200 with 6GB of RAM. We recorded the average number of nodes generated to solve each start state (total over all abstraction levels and the "base level" 8-puzzle space), the average CPU time, and the number of entries stored in the caches at all levels.

Figure 6 shows the average number of nodes generated (y-axis) for each of the 1000 instances (the x-axis). There are two data points for each instance; a black circle showing the average number of nodes generated over the 12 alternative state-set abstractions and a grey cross showing the average number of nodes generated over the 12 alternative domain abstractions. The instances are ordered on the x-axis according to their state-set average. We see that state-set abstraction generates fewer nodes than domain abstraction, with the advantage increasing as problems become more difficult. The CPU time plot had the same qualitative characteristics. The number of nodes cached using state-set abstraction was the same as, or smaller than, the number cached using domain abstraction.

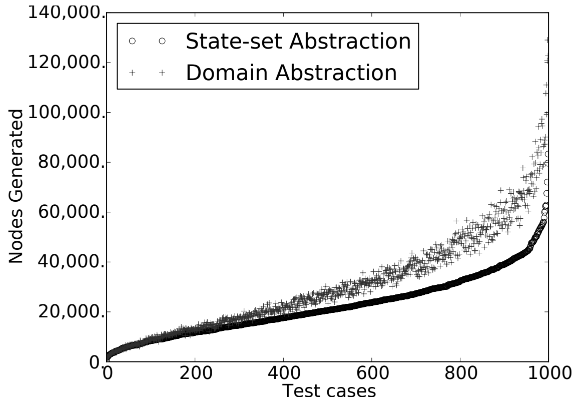

Figure 6: Average Nodes Generated.

Figure 7 plots the number of nodes generated for the worst and best abstractions of each type, where "worst" means the abstraction that resulted in the most nodes generated. The plots for state-set abstraction are shown with black, and the plots for domain abstraction are shown in grey. For the easiest problems, there is no difference between the two types of abstraction: their best abstractions are equally good and their worst abstractions are equally bad. As the problems get harder, the plots for domain abstraction rise more quickly than the corresponding plots for state-set abstraction. For the hardest problems the worst state-set abstraction outperforms the best domain abstraction. The superior performance of state-set abstraction in this experiment is not due to d_{ww} returning larger heuristic values than d_{abs}: the state-set abstraction at Level 2 actually returns slightly smaller values, on average, than the domain abstraction, resulting in more nodes being generated at Level 1. However, the number of nodes generated at Level 2 is much smaller for state-set abstraction, resulting in an overall reduction in nodes generated when all levels are considered.

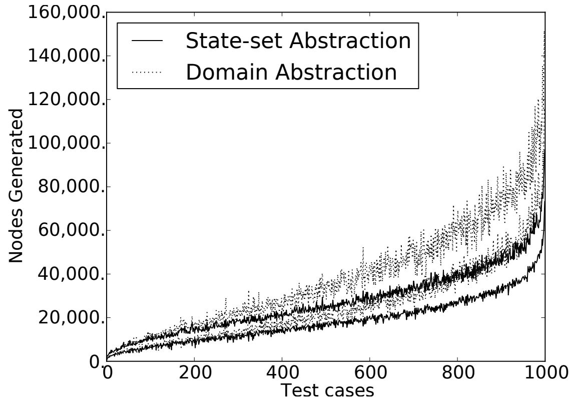

Figure 7: Best and Worst Abstractions of each type.

Conclusions

The formal, general analysis of state-set search in this paper highlights the similarities and differences between the paths and distances computed by planning systems and abstraction systems, exposes both limitations and special, advantageous properties of current techniques, and suggests new, potentially powerful techniques such as multi-abstraction and the d_{ww} distance.

Acknowledgements

The authors thank the reviewers for their comments on the submitted version and gratefully acknowledge the research support provided by Canada's Natural Sciences and Engineering Research Council (NSERC).

References

Bäckström, C. 1995. Expressive equivalence of planning formalisms. *Artif. Intell.* 76(1-2):17–34.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *AIPS*, 52–61.

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

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

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *National Conference on Artificial Intelligence (AAAI-96)*, 530–535.

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence* 170:1123–1136.

Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical heuristic search revisited. In *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, volume 3607 of *LNAI*, 121–133. Springer.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research* 7:283–317.

Larsen, B. J.; Burns, E.; Ruml, W.; and Holte, R. 2010. Searching without a heuristic: Efficient use of abstraction. In *AAAI*.

Rintanen, J. 2008. Regression for classical and nondeterministic planning. In *Proceeding of the 18th European Conference on Artificial Intelligence*, 568–572. IOS Press.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.

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