Discovering State Invariants

Fangzhen Lin
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
flin@cs.ust.hk

Abstract

We continue to advocate a methodology that we used earlier for pattern discovery through exhaustive search in selected small domains. This time we apply it to the problem of discovering state invariants in planning domains. State invariants are formulas that if true in a state, will be true in all successor states. In this paper, we consider the following four types of state invariants commonly found in AI planning domains: functional dependency constraints, constraints on mutual exclusiveness of categories, type information constraints, and domain closure axioms. As it turned out, for a class of action theories that include many planning benchmarks, for the first three types of constraints, whether they are state invariants can be verified by considering models whose domains are bounded by a small finite number. This forms the basis for a procedure that tries to discover state invariants by exhaustive search in small finite domains. An implementation of the procedure yields encouraging results in the blocks world and the logistics domain.

Introduction

In this paper we advocate a methodology for pattern discovery through exhaustive search in selected small domains. There are many problems with instances or parameters that have unbounded sizes. For example, the SAT problem can have instances with any numbers of variables and clauses, the blocks world can have problems with any number of blocks, and a sorting algorithm can accept any finite set of integers. For these problems, finding properties that are true for all instances can sometimes be done by exhaustive search in a selected few instances of small sizes. For example, to know that a block cannot be on top of itself, one only has to look at the case where there is just one block. Indeed, this is the strategy that we use (Lin 2003) for computing first-order successor state axioms (Reiter 2001) and STRIPS-like systems (Fikes & Nilsson 1971) from causal theories (Lin 1995). In this paper, we apply it to the problem of discovering *state invariants* in planning domains.

State invariants are formulas that if true in a state, will be true in every successor state. They are related to state constraints, but not the same. The latter are constraints that are true in all "legal" states.

We argued elsewhere (Lin 1995; 2003) that state constraints should be encoded as causal rules and given directly by the user as part of the domain specification. So instead of saying that they are sentences true in all "legal" states, one can define legal states as those that satisfy all state constraints.

State constraints are useful for a variety of reasons. They provide a more principled way of specifying the logical effects of actions. For instance, given a large C program, it is hard to figure out the effects of changing the value of a pointer on the values of all other pointers in the program. However, the underlying principle is very simple: when the value of a pointer changes, the values of all other pointers that point to the same memory location change as well. Put another way, the direct effect of the action of changing the value of a pointer to x is that the value of the pointer will be x. The indirect or side effects of this action are those derived from the constraint which says that if two pointers point to a common location, then their values must be the same. For a concrete proposal of how this will work out, see (Lin 2003).

Domain constraints can also be used to speed up certain domain independent planners (see, e.g. (Gerevini & Schubert 2000)). The basic idea is that if a state violates a domain constraint, then this state can be cut off from the search space.

However, if for whatever reason, the domain constraints are not directly given by the user, then one can try to learn it from dynamic laws such as those in the forms of successor state axioms or STRIPS-like systems. But without the complete information about what the legal initial states are, one can never be certain that what are learned are indeed state constraints. In this case, all that can be deduced are state invariants, and one can only hope that these state invariants will coincide with state constraints. This is the reason why we are concerning ourselves with discovering state invariants, rather than state constraints, here.

This paper is organized as follows. In section 2, we prove some simple theorems in many-sorted first-order logic. They will be used to show that some state invariants discovered by our procedure are provably correct in the general case. In section 3 we define formally action theories and their state invariants. In section 4 we give out four types of state invariants commonly found in planning domains, and in section 5, we describe our procedure for discovering them. In section 6, we give some experimental results in the blocks world and the logistics domain. In section 7 we discuss some related work and conclude this paper.

Some simple theorems in first-order logic

Before we proceed to discuss state invariant discovery in planning domains, some results from first-order logic will be essential.

In first-order logic, it is well-known that if a prenex formula of the form $(\exists x_1, ..., x_n)Q$, where Q has no quantifiers and functions, is satisfiable, then it is satisfiable in a structure whose domain has n elements. More generally, the following is true:

Proposition 1 *Suppose that B is a formula that does not have quantifiers and functions of positive arity, and that it mentions k constants. Suppose $\vec{x}$ is a tuple of n variables, and $\vec{y}$ another tuple of variables. If the prenex formula $\exists\vec{x}\forall\vec{y}.B$ is satisfiable, then it is satisfiable in a structure whose domain has at most $k+n$ elements, provided $k+n>0$. When $k+n=0$, the formula is satisfiable iff it is satisfiable in a structure whose domain has one element.*

Proof: Suppose M is a model of $\exists\vec{x}\forall\vec{y}.B$. Construct a structure M' as follows:

- Let D be the set of those elements in the domain of M that either interpret the constants in B or are witnesses for variables in $\vec{x}$ for the truth of $\exists\vec{x}\forall\vec{y}.B$. Clearly, the size of D is less than or equal to $k+n$.
- Let M' be the substructure of M induced by D.

Then M' is also a model of $\exists\vec{x}\forall\vec{y}.B$. ■

The dual of this proposition is the following:

Proposition 2 *Let B, $\vec{x}$, $\vec{y}$, k, and n be as in Proposition 1. If the prenex formula $\forall\vec{x}\exists\vec{y}.B$ is true in every structure whose domain has at most $k+n$ elements, then it is valid, provided $k+n>0$. When $k+n=0$, if the formula is true in every structure whose domain has one element, then it is valid.*

Frequently, we will be dealing with domains that have multiple sorts. We now extend these two results to a sorted first-order language with equality. In a sorted first-order language, the arity of a predicate is a tuple of sorts. For instance, in the logistics domain (Bacchus Fall 2001), the arity of at is $phyobj{*}place$, meaning that it is a predicate with two arguments, the first one is of sort $phyobj$ and the second $place$. When a variable is quantified, the sort over which the variable is ranged needs to be made explicit. For instance, to say that no object can be at more than two places (at the same time), we write

$$\forall(x, phyobj)(y, place)(z, place).$$
$$at(x,y) \wedge at(x,z) \supset y = z.$$

In the following, a pair of a variable and a sort such as $(x, place)$ is called a *variable-sort pair*.

Sorts are not required to be primitive. One sort may be a subsort of another. For instance, in the logistics domain, sort $phyobj$ contains $package$ and $vehicle$, and the latter contains $airplane$ and $truck$. In the following, we say that a term *may be* of sort g if it is declared to be of sort g', and g and g' intersect. For instance, a variable of sort $phyobj$ may be of sort $package$.

A *rank* τ of a language is a set of pairs such that for every primitive sort g in the language there is exactly one pair of the form (g,n) in it, where $n>0$ is an ordinal. We say that τ is a finite rank if for all $(g,n)\in\tau$, n is finite.

A first-order structure is said to be a *τ-structure* if for each $(g,n)\in\tau$, the domain of sort g in the structure has at most n elements. Similarly, a τ-model of a sentence (theory) is a τ-structure that satisfies the sentence (theory).

As an example, the language for the logistics domain has the following primitive sorts: $package$, $city$, $airport$, $truck$, $location$, and $airplane$. So a rank in this language could be

$$\{(package, 3), (city, 2), (airport, 2), (truck, 3),$$
$$(airplane, 1), (location, 3)\},$$

and a structure of this rank has at most three packages, two cities, etc.

Definition 1 *A formula φ is said to have rank τ if whenever φ has a model, it has a τ-model. In this case, we also call it a τ-formula.*

Notice that If both φ and ψ are τ-formulas, so is $\varphi\vee\psi$, but not necessarily $\varphi\wedge\psi$.

The following proposition extends Proposition 1 to the many sorted case:

Proposition 3 *Let B be a formula that does not have any quantifiers and functions of positive arity, and $\vec{x}$ and $\vec{y}$ tuples of variable-sort pairs. Then the prenex formula $(\exists\vec{x})(\forall\vec{y}).B$ is a τ-formula, where the rank τ is defined as follows: For each primitive sort g, if $k_g+n_g>0$, then $(g, k_g+n_g)\in\tau$, otherwise, $(g,1)\in\tau$, where k_g is the number of constants that may be of sort g in B, and n_g the number of variables that may be of sort g in $\vec{x}$.*

Proof: Similar to the proof of Proposition 1, noting that given any model M of $(\exists\vec{x})(\forall\vec{y}).B$, for each primitive sort g, the number of constants that are mapped to objects of sort g cannot be greater than k_g, and the number of objects of sort g that are witnesses for variables in $\vec{x}$ cannot be greater than n_g. ■

The dual of τ-formulas are τ-valid formulas.

Definition 2 *A formula φ is said to be τ-valid if whenever it is true in all τ-structures, it is true in all structures.*

Symmetrically, if both φ and ψ are τ-valid, so is $\varphi\wedge\psi$, but not necessarily $\varphi\vee\psi$. In fact, the following proposition is immediate.

Proposition 4 *For any rank τ, a formula φ is τ-valid iff $\neg\varphi$ is of rank τ.*

From Propositions 3 and 4, we have:

Proposition 5 *Let B be a formula that does not have any quantifiers and functions of positive arity, and $\vec{x}$ and $\vec{y}$ tuples of variable-sort pairs. The prenex formula*

$$(\forall \vec{x})(\exists \vec{y}).B$$

is τ-valid, where the rank τ is defined as follows: For each sort g, if $k_g + n_g > 0$, then $(g, k_g + n_g) \in \tau$, otherwise, $(g, 1) \in \tau$, where k_g is the number of constants that may be of sort g in B, and n_g the number of variables that may be of sort g in $\vec{x}$.

Action domains and state invariants

We now define formally what we mean by state invariants in an action domain. An action domain is described in a many-sorted first-order language called *domain language* that includes a set of predicates, which are used to describe states, and a set of functions. There is a special sort called "$action$". Functions whose range is of sort $action$ denote actions, and will be called action types below.

To specify the effects of actions in first-order logic, we extend the domain language by a new predicate $Poss$ of arity $action$, and for each predicate of arity $s_1 * \cdots * s_k$ a new predicate of the same name but with arity $s_1 * \cdots * s_k * action$. To distinguish between domain predicates and the new predicates with the same name, in the following we call the latter *successor state predicates*. For instance, in the blocks world, the domain predicate $on(x, y)$ means that block x is on top of block y in the current situation, and the successor state predicate $on(x, y, stack(u, v))$ means that x is on y in the successor situation of performing action $stack(u, v)$ in the current one.

The effects of actions can be specified in a number of ways. The standard way in classical AI planning is to use a STRIPS-like notation (Fikes & Nilsson 1971). The emerging standard PDDL (McDermott *et al.* 1998) is in this format. One can also use a specialized first-order logic like the situation calculus (McCarthy & Hayes 1969; Reiter 2001) and successor state axioms (Reiter 2001). However the actions are specified, for our purpose here, we assume that an action theory is a family of first-order theories, one for each action type as defined below.

Definition 3 *An action theory is a family of first-order theories $\{T_A \mid A$ is an action type$\}$, where for each action type A, T_A consists of the following axioms:*

- *An action precondition axiom of the form*

$$\forall \vec{x}.Poss(A(\vec{x})) \equiv \Psi, \tag{1}$$

  where Ψ is a formula in the domain language whose free variables are in $\vec{x}$. (Thus Ψ cannot mention $Poss$ and any successor state predicates.)

- *For each domain predicate F an axiom of the following form:*

$$(\forall \vec{x}, \vec{y}).F(\vec{x}, A(\vec{y})) \equiv \Phi_F(\vec{x}, \vec{y}), \tag{2}$$

  where $\vec{x}$ and $\vec{y}$ do not share common variables, and Φ_F is a formula in the domain language whose free variables are from $\vec{x}$ and $\vec{y}$.

Example 1 In the blocks world, for the action type $stack$, we have the following axioms (all free variables below are universally quantified from outside):

$$\begin{aligned}
&Poss(stack(x, y)) \equiv holding(x) \wedge clear(y),\\
&on(x, y, stack(u, v)) \equiv (x = u \wedge y = v) \vee on(x, y),\\
&ontable(x, stack(u, v)) \equiv ontable(x),\\
&handempty(stack(u, v)) \equiv true,\\
&holding(x, stack(u, v)) \equiv false,\\
&clear(x, stack(u, v)) \equiv clear(x) \wedge x \neq v.
\end{aligned}$$

The following definition captures the intuition that a state invariant is a formula that if true initially, will continue to be true after the successful completion of every possible action.

Definition 4 *Given an action theory $\{T_A \mid A$ is an action type$\}$, a formula W in the domain language is a state invariant if for each action type A,*

$$T_A \models \forall \vec{y}.W \wedge Poss(A(\vec{y})) \supset W(A(\vec{y})), \tag{3}$$

where $W(A(\vec{y}))$ is the result of replacing each atom $F(\vec{t})$ in W by $F(\vec{t}, A(\vec{y}))$, and $\models$ is the logical entailment in first-order logic.

The following propositions are immediate.

Proposition 6 *Let W' be a state invariant. For any formula W in the domain language, $W \wedge W'$ is a state invariant iff*

$$T_A \models (\forall \vec{y}).W' \wedge W \wedge Poss(A(\vec{y})) \supset W(A(\vec{y}))$$

for every action type A.

Proposition 7 *If $W \wedge W_1$ and $W \wedge W_2$ are both state invariants, then $W \wedge W_1 \wedge W_2$ is also a state invariant.*

The following will be our main theorem for proving that a formula is a state invariant.

Theorem 1 *Let W' be a state invariant. For any formula W in the domain language, $W \wedge W'$ is a state invariant iff for each action type A, the sentence*

$$\forall \vec{y}.W \wedge W' \wedge \Psi(\vec{y}) \supset \mathcal{R}(W, A(\vec{y})) \tag{4}$$

is valid, where Ψ is the action precondition of A as in the right side of (1), and $\mathcal{R}(W, A(\vec{y}))$, the regression of W over $A(\vec{y})$, is the result of replacing each atom $F(\vec{t})$ in W by $\Phi_F(\vec{t}, \vec{y})$ in the right of the axiom (2).

Proof: (Sketched) By Proposition 6, we show that

$$(\forall \vec{y}).W' \wedge W \wedge Poss(A(\vec{y})) \supset W(A(\vec{y})) \tag{5}$$

follows from T_A iff the sentence (4) is valid. From right to left is obvious. To prove from left to right, suppose (4) is not valid, and that M is a structure that satisfies its negation. Since (4) is a sentence in the domain language, it does not mention $Poss$ and any successor state predicates. So M can be extended to become a model of T_A in the extended language. The extended model is then a model of T_A that does not satisfy the sentence (5). ■

In the following, we shall call formula (4) the *state invariant condition* of W on A under W'. When W' is $true$, we simply call it the state invariant condition of W on A. Thus Theorem 1 says that if W' is a state invariant and W a formula, then $W \wedge W'$ is a state invariant iff for each action type A, the state invariant condition of W on A under W' is valid.

We shall now define a class of action domains and a class of formulas such that in these domains, the state invariant conditions of these formulas are always τ-valid for some finite τ.

Actions in many planning domains have only simple effects in the sense that if action $A(\vec{x})$ causes $F(\vec{y})$ to be true (or false), then $\vec{y}$ is a subset of $\vec{x}$. This compares to actions with, say universal effects. An example of the latter is the action of exploding a bomb, which will kill all those that are within a certain range of the bomb.

Definition 5 *An action theory is said to be* simple *if for each action type A:*

- *The formula Ψ in its action precondition axiom (1) has no quantifiers.*
- *The formula Φ_F in the axiom (2) for each F has no quantifiers, and (2) entails the following formula:*

$$(\forall \vec{x}, \vec{y}).\neg subset(\vec{x}, \vec{y}) \supset F(\vec{x}, A(\vec{y})) \equiv F(\vec{x}),$$

where $subset(\vec{x}, \vec{y})$, meaning $\vec{x}$ is a subset of $\vec{y}$, is the following formula:

$$\bigwedge_{(x,g)\in\vec{x}} \bigvee_{(y,g')\in\vec{y}} x = y.$$

Notice that for context-free action domains such as the blocks world and the logistics domain, successor state axioms (2) have the form:

$$F(\vec{x}, A(\vec{y})) \equiv E_1 \vee \cdots \vee E_n \vee (F(\vec{x}) \wedge \neg E_{n+1} \wedge \cdots \wedge \neg E_m),$$

where E_i's are conjunctions of equality atoms between variables in $\vec{x}$ and $\vec{y}$. For instance, in the blocks world, we have:

$$\begin{aligned} & clear(x, unstack(y, z)) \equiv \\ & \qquad (x = z) \vee (clear(x) \wedge y \neq x), \\ & on(x_1, x_2, unstack(y, z)) \equiv \\ & \qquad on(x_1, x_2) \wedge \neg(x_1 = y \wedge x_2 = z). \end{aligned}$$

Thus context-free action domains are simple according to our definition as long as action preconditions do not mention any quantifiers.

In the following, for two ranks τ_1 and τ_2, we say that $\tau_1 \leq \tau_2$ if for each sort g, if $(g, n_1) \in \tau_1$ and $(g, n_2) \in \tau_2$, then $n_1 \leq n_2$.

Theorem 2 *Suppose the given action theory is simple, and W is a conjunction of prenex formulas $(\forall \vec{x_1})B_1 \wedge \cdots \wedge (\forall \vec{x_k})B_k$, where B_i, $1 \leq i \leq k$, does not mention any quantifiers and functions of positive arity. Suppose the atoms in B_i are $F_{i1}(\vec{t_{i1}}), ..., F_{in_i}(\vec{t_{in_i}})$, and the actions in the domains are $A_1(\vec{y_1}), ..., A_m(\vec{y_m})$. For $1 \leq i \leq k$, $1 \leq j \leq n_i$, and $1 \leq w \leq m$, let τ_{ijw} be the rank defined as follows: for each primitive sort g, $(g, v) \in \tau_{ijw}$ for v defined to be the following number if it is greater than one, and $v = 1$ otherwise:*

(the number of constants that may be of sort g in W and Ψ_{A_w}) +
(the number of variables that may be of sort g in $\vec{x_i}$) +
(the number of variables that may be of sort g in $\vec{y_w}$) -
(the number of variables and constants that may be of sort g in $\vec{t_{ij}}$),

where Ψ_{A_w} is the precondition of $A_w(\vec{y_w})$. Now let τ be the smallest rank such that $\tau_{ijw} \leq \tau$ for all such i, j, w. Then the state invariant condition of W is τ-valid for every action type.

Proof: (Sketched) The state invariant condition of W on action $A_w(\vec{y_w})$ is:

$$\forall \vec{y_w}.W \wedge \Psi_{A_w} \supset \mathcal{R}(W, A(\vec{y_w})),$$

which is equivalent to

$$\begin{aligned} \forall \vec{y_w}.[(\forall \vec{x_1})B_1 \wedge \cdots \wedge (\forall \vec{x_k})B_k \wedge \Psi_{A_w}] \supset \\ [(\forall \vec{x_1})\mathcal{R}(B_1(\vec{x_1}), A(\vec{y_w})) \wedge \cdots \wedge \\ (\forall \vec{x_k})\mathcal{R}(B_k(\vec{x_1}), A(\vec{y_w}))] \end{aligned}$$

which is equivalent to the conjunction of the following formulas for $1 \leq i \leq k$:

$$\begin{aligned} (\forall \vec{y_w}, \vec{x_i})(\exists \vec{x'_1}, ..., \vec{x'_k}).\neg B_1(\vec{x'_1}) \vee \cdots \vee \neg B_k(\vec{x'_k}) \vee \\ \neg \Psi_{A_w} \vee \mathcal{R}(B_i(\vec{x_i}), A(\vec{y_w})), \end{aligned}$$

where $\vec{x'_i}$, $1 \leq i \leq k$, is a fresh new tuple of variable-sort pairs like $\vec{x_i}$ and $B_i(\vec{x'_i})$ is the result of replacing variables in $\vec{x_i}$ by the corresponding ones in $\vec{x'_i}$ in B_i. This formula is equivalent to the conjunction of the following formulas:

$$\begin{aligned} & (\forall \vec{y_w}, \vec{x_i})(\exists \vec{x'_1}, ..., \vec{x'_k}).subset(\vec{t_{i1}}, \vec{y_w}) \supset \\ & \qquad \neg B_1(\vec{x'_1}) \vee \cdots \vee \neg B_k(\vec{x'_k}) \vee \neg \Psi_{A_w} \vee \\ & \qquad \mathcal{R}(B_i(\vec{x_i}), A(\vec{y_w})), \\ & \qquad \cdots \\ & (\forall \vec{y_w}, \vec{x_i})(\exists \vec{x'_1}, ..., \vec{x'_k}).subset(\vec{t_{in_i}}, \vec{y_w}) \supset \\ & \qquad \neg B_1(\vec{x'_1}) \vee \cdots \vee \neg B_k(\vec{x'_k}) \vee \neg \Psi_{A_w} \vee \\ & \qquad \mathcal{R}(B_i(\vec{x_i}), A(\vec{y_w})), \\ & (\forall \vec{y_w}, \vec{x_i})(\exists \vec{x'_1}, ..., \vec{x'_k}). \\ & \quad \neg[subset(\vec{t_{i1}}, \vec{y_w}) \vee \cdots \vee subset(\vec{t_{in_i}}, \vec{y_w})] \supset \\ & \qquad \neg B_1(\vec{x'_1}) \vee \cdots \vee \neg B_k(\vec{x'_k}) \vee \neg \Psi_{A_w} \vee \\ & \qquad \mathcal{R}(B_i(\vec{x_i}), A(\vec{y_w})). \end{aligned}$$

Since the action theory is simple, the last sentence is equivalent to

$$\begin{aligned} & (\forall \vec{y_w}, \vec{x_i})(\exists \vec{x'_1}, ..., \vec{x'_k}). \\ & \quad \neg[subset(\vec{t_{i1}}, \vec{y_w}) \vee \cdots \vee subset(\vec{t_{in_i}}, \vec{y_w})] \supset \\ & \qquad \neg B_1(\vec{x'_1}) \vee \cdots \vee \neg B_k(\vec{x'_k}) \vee \neg \Psi_{A_w} \vee B_i(\vec{x_i}), \end{aligned}$$

which is valid. Again since the action theory is simple, Ψ_{A_w} is equivalent to a formula without any quantifiers, thus by Proposition 5, noting that by $subset(\vec{t_{ij}}, \vec{y_w})$, the variables and constants in t_{ij} that may be of sort g are subsumed by the variables that may be of sort g in y_w, the j-th formula above is τ_{ijw}-valid for $1 \leq j \leq i_n$. So all of them, as well as their conjunction, are τ-valid. ■

Example 2 Consider the following functional constraint in the logistics domain:

$$\forall(x_1, phyobj)(x_2, place)(x_3, place).\\ at(x_1, x_2) \wedge at(x_1, x_3) \supset x_2 = x_3. \quad (6)$$

First of all, notice that the sorts in this domain are as follows: primitive sorts: $package$, $city$, $airport$, $truck$, $location$, and $airplane$; composite sorts: $vehicle$ which contains $airplane$ and $truck$, $phyobj$ which contains $vehicle$ and $package$, and $place$ which contains $location$ and $airport$.

There are two atoms in this sentence: $F_1 = at(x_1, x_2)$ and $F_2 = at(x_1, x_3)$. There are six action types in the logistics domain (see (Bacchus Fall 2001)) $loadTruck$, $unloadTruck$, $loadAirplane$, $unloadAirplane$, $driveTruck$, and $flyAirplane$. For instance, the arity of $loadTruck(y_1, y_2, y_3)$ is $package * truck * place$), so for F_1 and $loadTruck$, the number v for sort $package$ in Theorem 2 is

(the number of constants that may be of sort $package$ in W and Ψ_{A_1}) +
(the number of variables that may be of sort $package$ in $(x_1, phyobj)(x_2, place)(x_3, place)$) +
(the number of variables that may be of sort $package$ in $(y_1, package)(y_2, truck)(y_3, place)$) -
(the number of variables that may be of sort $package$ in $(x_1, phyobj)(x_2, place)$)

which is equal to $0 + 1 + 1 - 1 = 1$. By similar calculation for other action types, we can use Theorem 2 to determine that the state invariant condition of (6) is $\{(package, 1), (city, 1), (airport, 1), (truck, 1), (airplane, 1), (location, 3)\}$-valid.

The calculation for constraints in the blocks world are simpler because there is only one sort. For instance, the state invariant conditions of the following functional dependency constraint

$$(\forall x, y, z).on(x, y) \wedge on(x, z) \supset y = z$$

are 3-valid for all action types.

Some typical types of state invariants in planning domains

To motivate, consider the following state constraints in the blocks world (all free variables in the formulas below are universally quantified from outside):

$$on(x, y) \wedge on(x, z) \supset y = z,$$
$$on(y, x) \wedge on(z, x) \supset y = z,$$
$$ontable(x) \supset \neg on(x, y),$$
$$holding(x) \vee ontable(x) \vee (\exists y)on(x, y).$$

The first two formulas can be called *functional dependency* constraints because they make certain parameters of a relation functional on others. The third formula can be called a *mutual exclusiveness constraint on categories or properties* because it says that if an object belongs to a class, then it cannot belong to another class (if a block is on the table, then it cannot be on another block). The last formula can be called a domain closure axiom because it says that an object must belong to one of the given classes (a block is either held by the robot, on the table, or on another block).

We claim that state constraints in planning domains typically are either these functional dependency constraints, mutual exclusiveness constraints on categories, constraints on type information (for domains with multiple sorts), or domain closure axioms.

In this section, we formally define these four types of constraints. As we shall see, if the given action theory is simple, then the state invariant conditions for the constraints of the first three types are τ-valid for some finite τ. This means that whether they are state invariants can be checked in finite domains.

Functional dependency constraints. For each n-ary predicate p, these are sentences of the following form:

$$(\forall \xi, \xi_1, \xi_2).p(\xi \cdot \xi_1) \wedge p(\xi \cdot \xi_2) \supset \xi_1 = \xi_2,$$

here ξ, ξ_1, ξ_2 are tuples of distinct variable-sort pairs such that $|\xi_1| = |\xi_2| = n - |\xi|$, and $\xi \cdot \xi_1$ and $\xi \cdot \xi_2$ are unions of ξ and ξ_1, ξ and ξ_2, respectively, such that the variables in ξ occur in the same positions in $\xi \cdot \xi_1$ and $\xi \cdot \xi_2$. In the blocks world, for predicate $holding(x)$, we have (there is only one sort in the blocks world, so we do not mention it explicitly):

$$(\forall x, y).holding(x) \wedge holding(y) \supset x = y.$$

For predicate $on(x, y)$, we have

$$(\forall x, y, z, t).on(x, y) \wedge on(z, t) \supset (x = z \wedge y = t), \quad (7)$$
$$(\forall x, y, z).on(x, y) \wedge on(x, z) \supset y = z, \quad (8)$$
$$(\forall x, y, z).on(y, x) \wedge on(z, x) \supset y = z. \quad (9)$$

In logistics domain, for predicate at with arity $phyobj * place$, all of the following are functional dependency constraints:

$$\forall(x, phyobj)(y, place)(z, place).\\ at(x, y) \wedge at(x, z) \supset y = z,$$
$$\forall(x, truck)(y, place)(z, place).\\ at(x, y) \wedge at(x, z) \supset y = z,$$
$$\forall(x, truck)(y, location)(z, place).\\ at(x, y) \wedge at(x, z) \supset y = z.$$

In general, one can restrict the first argument of at to any subsort of $phyobj$, and the second argument to any subsort of $place$.

By Thm 2, for simple action theories, the state invariant conditions of any conjunction of functional dependency constraints are τ-valid for some finite τ.

Mutual exclusiveness constraints of categories. An object can have certain attributes. For instance, in the blocks world, a block can be on the table, being held by the robot, or on top of some other blocks. Each of these attributes determines a category, and they may be mutually exclusive. For instance, a block cannot be both on the table and being held by the robot. In principle, any formula with exactly one free variable defines a category. But in planning domains, the ones that we are interested in normally have the following form: $(\exists \xi)p(\nu)$, where ν is a tuple of variables, and ξ is a subset of ν such that $|\xi| = |\nu| - 1$. In the blocks world, for block, we have the following categories:

$$ontable(x), holding(x), clear(x),$$
$$(\exists y)on(x,y), (\exists y)on(y,x).$$

In the logistics domain, each package can be at some place or inside a vehicle, so it has just two categories: $\exists(y, place)at(x,y)$ and $\exists(y, vehicle)in(x,y)$. For $vehicle$, syntactically there are two categories, $\exists(y, package)in(y,x)$ and $\exists(y, place)at(x,y)$. But only the latter one is meaningful.

Generally, if $C_1(x), ..., C_n(x)$ are categories for objects of sort g, then mutual exclusiveness constraints have the form:

$$\forall(x,g).C_{i_1}(x) \wedge \cdots \wedge C_{i_{k-1}}(x) \supset \neg C_{i_k}(x),$$

or equivalently in clausal form:

$$\forall(x,g).\neg C_{i_1}(x) \vee \cdots \vee \neg C_{i_k}(x).$$

For n categories, this would generate 2^n sentences. But n is normally quite small.

Sometimes we may also want to talk about categories about objects which are not explicitly represented in the domain. For instance, in the blocks world, the robot (or the agent) who is manipulating the blocks, is implicit. Nonetheless, we can still talk about its properties such as whether its hand is empty. In this case, the "categories" for it are sentences like $handempty$ and $\exists x.holding(x)$.

Notice that whether a constraint is a functional dependency one or a mutual exclusiveness one depends on the representation. For instance, if we have a predicate $at(x,y)$ in the blocks world to denote that the block x is at place y, which could be the robot's hand, the table, or the top of another block, then constraints like "if a block is on the table, then it is not held by the robot" becomes consequences of some functional dependency constraints.

If each category is a prenex formula of the form $(\exists\vec{x}).B$, then mutual exclusiveness constraints of categories will be prenex sentences of the form $\forall\vec{x}.B$. Thus by Theorem 2, for simple action theories, the state invariant conditions of any conjunction of these formulas are τ-valid for some finite rank τ.

Type information In the logistics domain, the arity of at is $phyobj{*}place$. However, we know that an airplane can only be at an airport. So we expect the following constraint to be true:

$$\forall(x, phyobj)(y, place).at(x,y) \wedge airplane(x) \supset airport(y).$$

In general, let F be a predicate that has at least two arguments of composite sorts. Then a type information constraint is a sentence of the form:

$$\forall\vec{x}.F(\vec{x}) \wedge g_1(x_1) \wedge \cdots \wedge g_k(x_k) \supset g_{k+1}(x_{k+1}),$$

where $x_1, ..., x_k, x_{k+1}$ are variables in $\vec{x}$, and for each $1 \leq i \leq k+1$, if $(x_i, g) \in \vec{x}$, then g_i is a subsort of g.

By Theorem 2, state invariant conditions for these constraints are again τ-valid for some finite τ for simple action theories.

Domain closure axioms for objects. These axioms say that an object must belong to one of the pre-defined categories. They are sentences of the following form:

$$\forall(x,g).C_1(x) \vee \cdots \vee C_n(x),$$

where $C_i(x)$'s are categories for objects of sort g. Since $C_i(x)$ is often a prenex formula of the form $\exists y.B$, state invariant conditions of domain closure axioms are in general not τ-valid for any finite τ. Thus checking whether they are state invariants cannot be reduced to propositional logic in general.

Discovering state invariants in finite domains

We now describe our state invariant discovery procedure. It requires two inputs: a set of sentences and a small set of small models, which are like first-order structures. In a nutshell, the procedure simply goes through the set of sentences looking for those that are state invariants in the given set of models.

Formally, given an action domain, a *model* of this domain is a pair (O, I), where O is a finite set of sorted objects called the *domain* of the model, and I a finite set of *atoms* in O called the *initial state* of the model, where an atomic formula $p(a_1, ..., a_n)$ is an atom in O if p is a predicate of arity $sort_1 * \cdots * sort_n$, and $a_i \in O$ is of type $sort_i$, for $1 \leq i \leq n$.

We make the closed-world assumption about the initial state I. If an atom is not in I, then it is assumed to be false. We assume that there are no constants and functions other than actions in the domain. Otherwise, a model will also need to specify their "meanings". All results in this paper can be extended straightforwardly to languages with constants, but not proper functions.

Given a model $M = (O, I)$, the *state space* $\mathcal{S}$ of the model is defined inductively to be the following set: (1) $I \in \mathcal{S}$, and (2) if $S \in \mathcal{S}$, and A is an action in O whose precondition is satisfied in S, then the resulting state of doing A in S is also in $\mathcal{S}$. Here an action in O is one whose arguments are objects in O.

Now given a set $\mathcal{H}$ of formulas, and a set of models, our algorithm for generating state invariants is very simple. First, it removes from $\mathcal{H}$ all formulas that are false in one of the states of the state space of one of the models, as we only want to consider state invariants that are at least true in the known legal states of the action domain. For the remaining ones, it checks whether they are state invariants in all of the given models. Here, informally, a formula is a state invariant in a model if the state invariant condition of the formula on every action type is true in all first-order structures with the same domain as the model. Notice that it does not hold that if a formula is true in the state space of a model, then it is a state invariant in this model. It may happen that if we change the initial state of the model, the formula may still be true in the initial state, but false in one of its successor states.

There are, however, some complications. Sometimes, none of the formulas in a set are state invariants on their own, but their conjunction is. For instance, in the blocks world,

Input $\mathcal{M}$ – a set of models; $\mathcal{H}$ – a set of sentences.

Output A sequence of state invariant sets in $\mathcal{M}$.

1. Let H_0 be the set of sentences in $\mathcal{H}$ that are true in the state space of every model in $\mathcal{M}$
2. Let $\Delta = \emptyset$
3. Let I be the set of sentences in H_0 that are state invariants in $\mathcal{M}$ under Δ
4. Let $i = 1$ and $H = H_0$.
5. While $I \neq \emptyset$ do
   - $I_i = I$, and $i = i + 1$
   - $\Delta = \Delta \cup I$
   - $H = H \setminus I$
   - Let I be the set of sentences in H that are state invariants in $\mathcal{M}$ under Δ
6. Let W be the conjunction of sentences in H.
7. If W is a state invariant in $\mathcal{M}$ under Δ, then return with $I_1, \cdots, I_{i-1}, \{W\}$
8. Let $D_1, ..., D_k$ be the maximal subsets of H such that $\bigwedge D_i$, $1 \leq i \leq k$, is a state invariant in $\mathcal{M}$ under Δ.
9. Return with the following sequence $I_1, ..., I_{i-1}, \{\bigwedge D_1, ..., \bigwedge D_k\}$

Figure 1: Procedure Discovering State Invariants

the constraint $(\forall x, y, z).on(x, y) \wedge on(x, z) \supset y = z$ is not a state invariant. The conjunction of it and some other similar constraints turns out to be. This means that we not only have to check whether an individual sentence in $\mathcal{H}$ is a state invariant, but also try all possible conjunctions. This greatly increases the complexity of our algorithm in the worst case.

What we do then is to generate state invariants in stages. In the first pass the procedure generates all formulas that are state invariants by themselves. Then the procedure is applied to the remaining formulas again, this time tries to see if any of them is a state invariant by assuming that the state invariants discovered earlier are true. This process is repeated until no more state invariants can be found, and then the conjunction of the remaining formulas is checked to see if it is a state invariant. Finally, as a last resort, all subsets of the remaining formulas are tried.

So the output of the procedure is really a sequence $(S_1, S_2, ..., S_n)$ of sets of formulas such that for each $1 \leq i \leq n$, and each sentence $\psi \in S_i$, the conjunction $\psi \wedge \bigwedge_{\varphi \in S_j, 1 \leq j < i} \varphi$ is a state invariant. In other words, each member of S_i is a state invariant when all sentences in $S_1 \cup \cdots \cup S_{i-1}$ are assumed to be true.

Figure 1 contains a sketch of our state invariant discovery procedure. In the procedure, a sentence W is said to be a state invariant in a set $\mathcal{M}$ of models under a set Δ of sentences if the conjunction of all the sentences in $\Delta \cup \{W\}$ is a state invariant in every model of $\mathcal{M}$.

Some experimental results

We have implemented the procedure outlined above in SWI-Prolog version 3.2.9[1]. In the following, we report some experimental results on the blocks world and the logistics domain.

As mentioned, the user needs to specify a set of sentences as potential state invariants. In our experiments, we run the procedure twice. On the first run, it is given the set consisting of functional dependency constraints, mutual exclusiveness constraints of categories, and constraints about types. Because these constraints are all τ-valid for some finite τ, when the set of models given to the procedure has a model of size τ' for every $\tau' \leq \tau$, the state invariants returned by the procedure are provably correct in the general case.

Then on second run, the procedure is given the set of all possible domain closure axioms, the same set of models, but with the set Δ in the procedure initialized to be the union of the sets of constraints in the sequence returned by the first run of the procedure (in Figure 1, Δ is initialized to be the empty set), as the conjunction of the sentences in the union is already shown to be a state invariant. The output on this run by the procedure is not guaranteed to be a sequence of state invariant sets in the general case. But in the blocks world and the logistics domain, they happen to be.

As for the set of models given as input to the procedure. There are two ways to do that. One can use Theorem 2 to find the smallest τ such that the state invariant conditions of all constraints except for the domain closure axioms are τ-valid, and select a model set that has a model of the size τ' for every $\tau' \leq \tau$. Or one can start with a small τ and models of sizes less than τ, and gradually increase τ and the set of models until one gets what one is looking for.

The blocks world

This domain has one sort, five predicates: $on(x, y)$, $ontable(x)$, $clear(x)$, $handempty$, and $holding(x)$, and four actions: $stack(x, y)$, $unstack(x, y)$, $pickup(x)$, and $putdown(x)$. This is a familiar action domain, so we omit its action theory here. The models given as input to the program are as follows:

$$([1], [ontable(1), clear(1), handempty]),$$
$$([1, 2], [ontable(1), ontable(2), clear(1), clear(2), handempty]),$$
$$([1, 2, 3], [ontable(1), ontable(2), ontable(3), clear(1), clear(2), clear(3), handempty]).$$

As mentioned, our program calls the procedure in Figure 1 twice.[2] The union of the state invariant sets returned by the first run of the procedure is as follows (we break a conjunction $W_1 \wedge \cdots \wedge W_k$ into its k conjuncts, and eliminate $\forall \vec{x}\vec{y}.C \vee D$ when $\forall \vec{x}.C$ is also in the union):

[1]SWI-Prolog is developed by Jan Wielemaker at University of Amsterdam

[2]The total running time was 25 seconds on a PIII notebook running Windows XP with 256MB of RAM.

$$
\begin{aligned}
\{&\neg handempty \vee \neg(\exists x) holding(x),\\
&\forall x.\neg clear(x) \vee \neg(\exists z) on(z,x),\\
&\forall x.\neg holding(x) \vee \neg clear(x),\\
&\forall x.\neg holding(x) \vee \neg(\exists y) on(x,y),\\
&\forall x.\neg holding(x) \vee \neg(\exists z) on(z,x),\\
&\forall x.\neg ontable(x) \vee \neg holding(x),\\
&\forall x.\neg ontable(x) \vee \neg(\exists y) on(x,y),\\
&\forall x_1, x_2. holding(x_1) \wedge holding(x_2) \supset x_1 = x_2,\\
&\forall x_1, x_2, x_3. on(x_1,x_2) \wedge on(x_1,x_3) \supset x_2 = x_3,\\
&\forall x_1, x_2, x_3. on(x_2,x_1) \wedge on(x_3,x_1) \supset x_2 = x_3\}.
\end{aligned}
$$

These sentences are functional dependency and mutual exclusiveness constraints. By Theorem 2, the state invariant condition of the conjunction of these sentences is 3-valid. Since there is a model whose domain size is k for each $1 \leq k \leq 3$, and this conjunction is a state invariant in all these models, so this conjunction is also a state invariant in the general case.

The union of the state invariant sets in the sequence returned by the second run of the procedure is as follows:

$$
\begin{aligned}
\{&\forall x. holding(x) \vee clear(x) \vee (\exists z) on(z,x),\\
&\forall x. ontable(x) \vee holding(x) \vee (\exists y) on(x,y),\\
&handempty \vee (\exists x) holding(x)\}.
\end{aligned}
$$

These sentences are domain closure axioms. They turned out to be state invariants in the general case as well.

While the state invariants returned by our program include many familiar state constraints in the blocks world, they are not complete in the sense that there are some illegal states that satisfy all of them. For instance, if B is the only block in the domain, then the state $\{on(B,B), handempty\}$ is apparently illegal but satisfies all the sentences in the above two sets. However, if we add the following sentences:

$$
\begin{aligned}
&(\forall x,y). above(x,y) \equiv (on(x,y) \vee\\
&\quad (\exists z)(on(x,z) \wedge above(z,y))),\\
&(\forall x,y). above(x,y) \supset \neg on(y,x),
\end{aligned}
$$

then we get a complete set of state invariants. It is not clear how sentences like these can be discovered as they involve new predicates, and are essentially second-order.

Finally, we want to point out that for the blocks world, the output is not sensitive to the particular initial states of the models provided to the program. As long as they are legal, the same output will be returned.

The logistics domain

This again is a familiar domain (see (Bacchus Fall 2001) for a description). The models given as the input to the program have the following ranks:

$$
\begin{aligned}
&\{(city,1),(location,1),(truck,1),(airplane,1),\\
&\quad (airport,1),(package,1)\},\\
&\{(city,1),(location,2),(truck,1),(airplane,1),\\
&\quad (airport,1),(package,1)\}\\
&\{(city,1),(location,3),(truck,1),(airplane,1),\\
&\quad (airport,1),(package,1)\}\\
&\{(city,2),(location,1),(truck,1),(airplane,1),\\
&\quad (airport,1),(package,1)\}\\
&\{(city,1),(location,1),(truck,1),(airplane,1),\\
&\quad (airport,2),(package,1)\}\\
&\{(city,1),(location,1),(truck,1),(airplane,1),\\
&\quad (airport,1),(package,2)\}.
\end{aligned}
$$

These models are chosen to ensure the correctness of the following set of state invariants returned by the first run of the procedure in Figure 1:[3]

$$
\begin{aligned}
\{&\forall (x1, place)(x2, city)(x3, city).\\
&\quad inCity(x1,x2) \wedge inCity(x1,x3) \supset x2 = x3,\\
&\forall (x2, phyobj)(x1, place).\\
&\quad at(x2,x1) \wedge airplane(x2) \supset airport(x1),\\
&\forall (x, package).\neg\exists(y, vehicle) in(x,y) \vee\\
&\quad \neg\exists(y, place) at(x,y),\\
&\forall (x1, package)(x2, vehicle)(x3, vehicle).\\
&\quad in(x1,x2) \wedge in(x1,x3) \supset x2 = x3,\\
&\forall (x1, phyobj)(x2, place)(x3, place).\\
&\quad at(x1,x2) \wedge at(x1,x3) \supset x2 = x3\}.
\end{aligned}
$$

The set of state invariants returned by the second run of the procedure is as follows:

$$
\begin{aligned}
\{&\forall (x, package).\exists(y, vehicle) in(x,y) \vee \exists(y, place) at(x,y),\\
&\forall (x, vehicle)\exists(y, place) at(x,y),\\
&\forall (x, place)\exists(y, city) inCity(x,y)\}.
\end{aligned}
$$

The union of these two sets of constraints turns out to be complete in the sense that a state is "legal" if it satisfies all the constraints in it.

Related work

In terms of discovering state invariants and state constraints in STRIPS-like action domains, previous work includes Zhang and Foo (1997), Gerevini and Schubert (1998; 2000), and Fox and Long (1998). However, some of the previous work concerns only with domain constraints in a particular model, i.e. constraints that are true in every state of the state space of the model. Besides using a uniformed method for discovering state invariants of various varieties, our system was also able to discover more constraints than the others. To the best of our knowledge, our system is the only one that can discover a complete set of state invariants in the logistics domain.

In terms of using small finite domains to discover general patterns, Lin (2003) considered how STRIPS-like systems compiled from a causal theory in some small domains can be provably correctly generalized to other domains. This work can be considered a continuation of (Lin 2003) in advocating pattern discovery using small domains.

[3] Again our program calls the procedure twice. The total running time was 62 seconds.

Conclusions

We have proposed a simple procedure for discovering state invariants in action domains. The procedure performs exhaustive search in the models given by the user as input. Many state invariants discovered this way are provably correct, provided the models given are general enough. Our experimental results showed that this procedure is effective even for discovering domain closure axioms, which are not τ-valid for any finite τ, in the blocks world and the logistics domain.

For future work, we are working on applying the methodology advocated in this paper to problems in other areas.

Acknowledgments

I would like to thank the anonymous reviewers of this paper for KR'04 for their comments. This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6182/01E.

References

Bacchus, F. Fall 2001. AIPS'00 planning competition. *AI Magazine* 22(3):47–56. See also http://www.cs.toronto.edu/aips2000.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to theorem proving in problem solving. *Artificial Intelligence* 2:189–208.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI–98), AAAI Press, Menlo Park, CA.*, 905–912.

Gerevini, A., and Schubert, L. 2000. Discovering state constraints in DISCOPLAN: Some new results. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI–2000), AAAI Press, Menlo Park, CA.*, 761–767.

Lin, F. 1995. Embracing causality in specifying the indirect effects of actions. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI–95), IJCAI Inc. Distributed by Morgan Kaufmann, San Mateo, CA.*, 1985–1993.

Lin, F. 2003. Compiling causal theories to successor state axioms and STRIPS-like systems. *Journal of Artificial Intelligence Research* 19:279–314.

McCarthy, J., and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. Edinburgh: Edinburgh University Press. 463–502.

McDermott, D. *et al.* 1998. PDDL – the planning domain definition language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

Zhang, Y., and Foo, N. 1997. Deriving invariants and constraints from action theories. *Fundamenta Informaticae* 30(1):109–123.