State-based Discovery and Verification of Propositional Planning Invariants

(*Submitted to the 2006 International Conference on Artificial Intelligence*)

Proshanto Mukherji*
Department of Computer Science
University of Rochester
Rochester NY 14627
Tel: (585) 329-0109
Fax: (585) 273-4556
mukherji@cs.rochester.edu

Lenhart K. Schubert
Department of Computer Science
University of Rochester
Rochester NY 14627
Tel: (585) 275-8845
Fax: (585) 273-4556
schubert@cs.rochester.edu

Abstract

Planning invariants are formulae that are true in every reachable state of a planning world. We describe a novel approach to the problem of discovering such invariants in propositional form—by analyzing only a set of reachable states of the planning domain, and not its operators. Our system works by exploiting perceived patterns of propositional covariance across the set of states: It hypothesizes that strongly-defined patterns represent features of the planning world.

We demonstrate that, in practice, our system overwhelmingly produces correct invariants. Moreover, we compare it with a well-known system from the literature that uses complete operator descriptions, and show that it discovers a comparable number of invariants, and moreover, does so orders of magnitude faster.

We also show how an existing operator-based invariant finder can be used to verify the correctness of the invariants we find, should operator information be available. We show that such hybrid systems can efficiently produce verifiably true invariants.

Keywords. domain analysis for planning and scheduling, domain-independent classical planning, invariant discovery.

Introduction

Planning invariants are formulae that are true in every reachable state of a planning world. They are characteristics of reachable states, and thus can be used to reduce the size of the search space in planning. A number of studies (e.g., (Gerevini & Schubert 1998; Kautz & Selman 1998; Koehler & Hoffmann 2000; Porteous, Sebastia, & Hoffmann 2001)) have demonstrated empirically that the use of certain classes of invariants can significantly speed up the planning process. This is true whether the constraints are added manually, as in (Kautz & Selman 1998), or by automated preplanners such as DISCOPLAN (Gerevini & Schubert 1998; 2001), Rintanen's (2000) algorithm, or TIM (Fox & Long 1998; 2000).

Most systems that try to find such invariants automatically do so by analyzing the operators of the planning world. In this paper, we take a complementary approach: We discover invariants by analyzing a set of the reachable states of the system, rather than by examining the operators. Our model is that of an observer who sees patterns of covariance across the observed states, and hypothesizes that they represent features of the underlying generative system—in other words, that they are invariants of the planning domain. For instance, in a propositional Blocks World, she might be struck by the fact that the propositions `on-a-b` and `on-b-a` never co-occurred in the same state, and thus hypothesize the invariant $\neg$ `on-a-b`$\vee\neg$ `on-b-a`.

The drawback of a state-based method is that, being inductive, it is not sound in the deductive sense. It is possible that it will find "invariants" that are true in the states it is given, but not true of the world in general. However, as we will demonstrate, in domains where operator-based methods are applicable, they can be modified to efficiently *verify* the correctness of the invariants hypothesized by our state-based methods. Moreover, in complex, real-world domains, it may be impossible or inefficient to apply operator-based techniques. In such worlds, state-based methods provide statistical invariants—formulae that are *probably* true in any reachable state—which can be used to guide the search of probabilistic or priority-based planners.

State-based methods may thus either be used independently, to produce statistical invariants, or combined with standard operator-based methods (where these are applicable) to quickly produce verified invariants. The latter hybrid approach can equivalently be viewed as using state-based methods as a preprocessing step to speed up the sound operator-based invariant discovery method.

Standalone State-based Methods Standalone state-based invariant detection has many advantages:

1. State-based approaches require less information than operator-based ones—only a set of reachable states, rather than the complete operators (which implicitly describe *all* reachable states). They are thus applicable even if the operators are unknown or only partially known.
2. State-based systems make no use of the STRIPS assumption, since they require only state descriptions. Operator-based methods, on the other hand, rely heavily on this assumption. If the world can change in ways the operators don't allow, then deduction based on operator preconditions and effects is unsound. In fact, the STRIPS assumption is quite unrealistic; realistic worlds change through

*Presenting Author.

other agencies than that of the planner. Thus such methods are more easily extensible to more realistic planning worlds.

3. Lastly, operator-based methods tend to use declarative bias to guide their search through the space of possible invariants. For instance DISCOPLAN (Gerevini & Schubert 1998; 2000) searches only for invariants of specific syntactic forms that the authors believe *a priori* to be useful. Our approach uses correlations in the data to guide the search instead. Thus it finds useful invariants that operator-based methods might miss.

Hybrid Methods If operator information is in fact available, then standard operator-based methods can be used to quickly verify the correctness of the hypotheses produced by our method. Such an hybrid approach might turn out to be the best of both worlds: It would yield provably correct invariants, and at the same time use inductive state-based reasoning for efficiency. Moreover, since invariant detection in general is PSPACE-complete (Rintanen 2000), no practical system can hope to find all invariants, which indicates that the radically different state-based approach might also help operator-based systems find invariants they would otherwise miss.

Many of the invariant identification systems in the literature (e.g. DISCOPLAN, TIM) generate invariants in first-order form. State-based approaches to that problem have been studied (see for example (Mukherji & Schubert 2003; 2005a)), but, in this paper, we will tackle the related problem of discovering *propositional* invariants from states. Propositional invariants are of particular interest because many modern planners—for example GRAPHPLAN (Blum & Furst 1995), and the family of satisfiability-based planners (Kautz & Selman 1992)—do propositional planning. Moreover, propositional worlds lack the rich relational structure of first-order ones, and thus pose a very different set of challenges for state-based invariant discovery. This leads to the employment of significantly different algorithms, which could potentially find invariants that first-order systems miss.

The rest of this paper is organized as follows. In the next section, we describe our methods. This section has three parts: In the first part we present our algorithm for (standalone) state-based propositional invariant discovery; in the second, we describe a standard operator-based propositional invariant finder (Rintanen's iterative system (Rintanen 2005; 2000)); and finally, in the third part, we describe how this operator-based system can be modified to verify the candidate invariants our system hypothesizes. We then go on to report and discuss the results obtained when we apply our methods to some common planning domains; we compare them with the invariants obtained by Rintanen's method, and demonstrate that state-based methods might offer significant speedups over operator-based methods with little loss of accuracy. Finally, we state our conclusions and discuss ways that the system might be extended in the future.

Methods

In this section, we first describe our standalone state-based propositional invariant discovery algorithm, then describe a well-known operator-based system from the literature, and finally show how it can be modified to verify the invariants found by our state-based technique.

Propositional Invariant Discovery

Our model of invariant discovery is that of an observer who is "struck" by patterns in the set of state descriptions it is given and conjectures and evaluates potential invariants on that basis. The output is a list of propositional clauses representing the set of invariants hypothesized.

We take all states to be finite and fully observable. Each state is described by a set of positive propositions. Function-free first-order domains are propositionalized in the obvious way before the algorithm is run.

Our approach works by mapping propositional literals to an intermediate representation—which we call a "state set"—that can be manipulated independently. This representation is crucial to our technique, because it permits us to map logical operations to set operations, and to do a very large number of inferences at the same time.

An Intermediate Representation: State Sets

Let $S_1, S_2, \ldots, S_k$ be the given state descriptions. We treat each as a set of the propositional atoms that are true in it. Let Σ be the set of literals that occur in at least one of these state-descriptions and the negations of such literals. Formally,

$$\Sigma = \{\ell \mid \ell \in \bigcup_{i=1}^{k} S_i \text{ or } \neg\ell \in \bigcup_{i=1}^{k} S_i\}$$

For every clause φ consisting of literals from Σ, the *state-set* of φ, denoted $f(\varphi)$, is the set of the indices of those states in which φ is satisfied. That is,

$$f(\varphi) = \{i \mid S_i \models \varphi\}$$

For example, suppose in a Blocks World domain with two blocks, we are given the following three state descriptions:

$$\begin{array}{lcl} S_1 & = & \{\text{on-A-B}, \text{on-B-Tbl}, \text{clear-A}, \text{clear-Tbl}\} \\ S_2 & = & \{\text{on-A-Tbl}, \text{on-B-Tbl}, \text{clear-A}, \text{clear-B}, \text{clear-Tbl}\} \\ S_3 & = & \{\text{on-A-Tbl}, \text{on-B-A}, \text{clear-B}, \text{clear-Tbl}\} \end{array}$$

where the atoms have the obvious meanings.

Here, clear-Tbl is true in every state, so $f(\text{clear-Tbl}) = \{1, 2, 3\}$. on-A-B is only true in S_1, so $f(\text{on-A-B}) = \{1\}$. Similarly, $f(\text{on-A-B} \vee \neg\text{clear-A}) = \{1, 3\}$.

Operations

There is a clear connection between syntactic operations on clauses and set operations on their state-sets. The following lemmas should be intuitively obvious; for formal proofs see (Mukherji & Schubert 2005b).

Lemma 1 (Disjunction). *If $\varphi_1, \varphi_2, \ldots \varphi_n$ are clauses of literals from Σ then*

$$f(\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n) = f(\varphi_1) \cup f(\varphi_2) \cup \ldots \cup f(\varphi_n)$$

Lemma 2 (Negation). *If φ is a clause of literals from Σ then*

$$f(\neg\varphi) = \{1, 2, \ldots, k\} \setminus f(\varphi)$$

State-based Invariant Discovery

Overview Lemmas 1 and 2 show we can manipulate state sets *independently of the clauses that map to them*, and then reverse the mapping f to get the actual logical formulae. We use this observation to come up with an algorithm for (stand-alone) state-based propositional invariant discovery. Essentially, the algorithm works as follows: First, it computes the state-sets of all ℓ in Σ. It then repeatedly combines these state-sets using the union operation; any new state-sets created in this process may themselves be combined at later steps. Finally, it reverses the mapping f to find the clauses that correspond to the complete state-set $\{S_1, \ldots, S_k\}$—in other words, those that are true in all the given states—and outputs these as potential invariants.

The key observation here is that, in general, arbitrarily many propositions map to the same state set. The ability to manipulate state-sets independently of the clauses that map to them thus greatly decreases the number of operations that have to be performed: In computing the union of two state-sets, we implicitly compute the disjunctions of all pairs of clauses that map to them.

Algorithm A general algorithm for generating invariants is shown in Figure 1. The function names written in fixed-width font below refer to generic subroutines that can be implemented in different ways, and can be used to control the running time of the algorithm and the richness and number of the invariants produced, and to make fine-grained trade-offs between various quality metrics. These are discussed in detail below.

This algorithm first computes Σ from the given state descriptions. It then initializes $\mathcal{S}$—a collection of state-sets—to those corresponding to all single-literal sentences. It then introduces two important families of variables: $\mathcal{F}$ and $\mathcal{H}$. For each state-set $s \in \mathcal{S}$, $\mathcal{H}(s)$ is a set of clauses whose state-set is s, and $\mathcal{F}(s)$ is a set of (unordered) pairs of state sets that union to s. $\mathcal{F}(s)$ always starts out as just $\{s, \emptyset\}$, but every time the algorithm unions two state-sets s_1 and s_2 giving s, $\mathcal{F}(s)$ is updated by the addition of the element $\{s_1, s_2\}$.

The main loop of the algorithm then begins, and proceeds as follows: It repeatedly updates $\mathcal{S}$ with state-sets created by unioning sets that are already in $\mathcal{S}$ and that also satisfy the generic subroutine `allow_union`. Each time it does so, it updates $\mathcal{F}$. This continues as long as the generic subroutine `keep_iterating` continues to return true.

When the main loop is complete, the algorithm maps each state-set in $\mathcal{S}$ to the set of clauses that have been found to be satisfied in it. This is done as follows: The state-sets $s \in \mathcal{S}$ are considered in increasing order of size. For each one, the elements of $\mathcal{F}(s)$ are considered one by one. If the element is of the form $\{s, \emptyset\}$, then the corresponding set of clauses is just:

$$\{(\ell) \mid \ell \in \Sigma \text{ and } f(\ell) = s\}$$

If, on the other hand, the element is of the form $\{s_1, s_2\}$, then the corresponding clauses are:

$$\{(c_1 \vee c_2) \mid c_1 \in \mathcal{H}(s_1) \text{ and } c_2 \in \mathcal{H}(s_2)\}$$

Note that since the algorithm proceeds in increasing order of state-set size, the values of $\mathcal{H}(s_1)$ and $\mathcal{H}(s_2)$ are certain to have been finalized before $\mathcal{H}(s)$ is considered.

The generic subroutine `allow_disjunct` can be used as a filter on the syntax of the disjuncts produced; it disallows all those for which it fails.

Finally, the system outputs all the sentences that correspond to the set $\{1, 2, \ldots, k\}$, which is the set of all the states. That is to say, it outputs the set of clauses it has found that are true in *every* state.

We will now describe the generic subroutines `keep_iterating`, `allow_union` and `allow_disjunct` in the algorithm above, and show how they can be used to control the efficiency and scope of the system.

Fine-Tuning the System With perfectly permissive implementations of `allow_union` and `allow_disjunct`, the system might output a very large number of invariants. This set might be too large to be useful in practice.

This is why we have incorporated the three generic subroutines `keep_iterating`, `allow_union` and `allow_disjunct` into the algorithm. They have many uses: They serve as filters built into the state set-generation mechanism to prevent the explicit enumeration of clauses already created; they provide a degree of control over the running time of the algorithm and the forms of the hypotheses generated; and they permit the system to use other constraints and metrics to fine-tune the set of constraints produced.

`keep_iterating` controls how often the main loop of the algorithm is run. In its least restrictive form, it simply continues to return true until an iteration occurs in which $\mathcal{S}$ remains unchanged. Another possibility is to have it run at most a constant number T times. Such a metric would improve the speed of the algorithm, and also incidentally restrict the maximum number of literals permitted in an invariant.

`allow_union` controls how new state-sets are added to $\mathcal{S}$. We use it to bias state-set creation in two ways: First, we don't allow state-sets s_1 and s_2 to combine if the resulting union is smaller than some threshold; this prevents the system from wasting time with sets that are very small, and thus unlikely to be useful. Second, we use it to prohibit state-sets that overlap "too much" from combining, since such a combination would not result in a significantly larger state-set.

The subroutine `allow_disjunct` can be used to constrain the syntactic form of the disjunctions permitted. We use it to eliminate tautologies by forbidding clauses that contain both an atom and its negation, and also to restrict the number of disjuncts allowed in a clause. If the world is a propositionalized first-order world, then first-order syntactic information is used here as well—for instance, to forbid clauses from combining unless their first-order equivalents have arguments in common.

Finally, if such first-order information is available, we make use of it in a post-processing step to increase the size of the set of invariants found. We generalize the invariants we find to first-order form, and then re-specialize these

`find-propositional-invariants` ($S_1, \ldots, S_k$: state descriptions)

1. $\Sigma \leftarrow \{\ell \mid \ell \in \bigcup_{i=1}^{k} S_i \text{ or } \neg\ell \in \bigcup_{i=1}^{k} S_i\}$
2. $\mathcal{S} \leftarrow \{f(\ell) \mid \ell \in \Sigma\}$ $\quad \triangleright$ a collection of state sets
3. for each $\ell \in \Sigma$, $\mathcal{F}(f(\ell)) \leftarrow \{\{f(\ell), \emptyset\}\}$
4. for each $\ell \in \Sigma$, $\mathcal{H}(f(\ell)) \leftarrow \mathcal{H}(f(\ell)) \cup \{\ell\}$.
5. $\mathcal{H}(\emptyset) \leftarrow \{\text{false}\}$
6. while `keep_iterating`
7. for each $\{\kappa, \tau\} \in \{\{\kappa, \tau\} \mid \kappa, \tau \in \mathcal{S}$ and $\{\kappa, \tau\} \notin \mathcal{F}(\kappa \cup \tau)$ and `allow_union`$(\kappa, \tau)\}$
8. $\mathcal{S} \leftarrow \mathcal{S} \cup \{\kappa \cup \tau\}$
9. $\mathcal{F}(\kappa \cup \tau) \leftarrow \mathcal{F}(\kappa \cup \tau) \cup \{\{\kappa, \tau\}\}$
10. for each $s \in \mathcal{S}$ in increasing order of $\|s\|$
11. for each $\{\kappa, \tau\} \in \mathcal{F}(s)$
12. $\mathcal{H}(\kappa \cup \tau) \leftarrow \mathcal{H}(\kappa \cup \tau) \cup \left\{ \varphi_\kappa \vee \varphi_\tau \;\middle|\; \begin{array}{l} \varphi_\kappa \in \mathcal{H}(\kappa) \text{ and } \varphi_\tau \in \mathcal{H}(\tau) \text{ and} \\ \texttt{allow\_disjunct}(\varphi_\kappa, \varphi_\tau) \end{array} \right\}$
13. output $\mathcal{H}(\{1, 2, \ldots, k\})$

Figure 1: The algorithm for finding propositional invariants

to give a set of propositional invariants that have the same first-order signature as ones found, but were not themselves found. We discard those that are not present in all the state-descriptions supplied, and then add the remainder to the list of invariants. For instance, if ¬ `on-a-b`∨¬ `on-b-a` was detected, we would add additional invariants like ¬ `on-c-d`∨¬ `on-d-c`, which could potentially have been missed by the heuristics used.

Complexity The running time of the algorithm can be controlled by the generic subroutines described above. However, even using perfectly permissive versions of these functions, its running time is exponential only in the number of *states*, and not in the number of literals. This is because the number of states k serves as an upper bound on the number of literals in a clause.

Moreover, given some fixed k, it is theoretically possible to pre-compute the union relationships between all the state-sets there could be, making it unnecessary to run the first part of the algorithm at all. It would, however, be necessary to assume that every state-set was present, which makes such a scheme too costly in practice.

Soundness and Completeness The algorithm is (a) sound and (b) complete in the following restrictive senses: (a) every hypothesis it outputs is in fact true in all k of the states it is given; and (b) with perfectly permissive subroutines, it finds every such clause of length upto k.

Here we merely state these theorems. For complete proofs please see (Mukherji & Schubert 2005b).

Theorem 3 (Soundness). *If the algorithm returns a hypothesis of the form*

$$\varphi = \bigvee_{i=1}^{m} \ell_i$$

where $(\forall i : 1 \leq i \leq m)[\ell_i \in \Sigma]$*, then,*

$$(\forall j : 1 \leq j \leq k)\ [S_j \models \varphi]$$

Theorem 4 (Completeness). *Let φ be any clause of $m \leq k$ literals from Σ—i.e.*

$$\varphi = \bigvee_{i=1}^{m} \ell_i$$

where $(\forall i : 1 \leq i \leq m)[\ell_i \in \Sigma]$*. If φ is true in all k states given, then the algorithm outputs φ (using perfectly permissive subroutines).*

Rintanen's Operator-based Approach

Given the STRIPS assumption, operator-based approaches have the potential to be sound. However, since the general problem of invariant detection is PSPACE-complete, any practical system that tries to maintain soundness must compromise on completeness. In this section, we will describe one operator-based algorithm that makes this tradeoff.

Rintanen (2005; 2000) has given a sound, operator-based algorithm for discovering invariants. The invariants it finds are framed as propositional clauses that are at most n literals long, for some constant n. The algorithm is polynomial-time bounded (although sometimes slow in practice, as we demonstrate later) and sound, but not complete.

In what follows, operators are thought of as $\langle \text{precondition}, \text{effect} \rangle$ pairs, where both the precondition and the effect are conjunctive propositional formulas. For any operator o, precondition(o) is the set of literals in o's precondition, and effect(o) is the set of literals in its effect.

In overview, the algorithm works as follows: it is iterative, and at each stage i maintains a set C_i of clauses. For each i, C_i contains only clauses that the algorithm knows to be true

in every state within distance i of the initial state (the distance between two states s_1 and s_2 is defined as the length of the shortest sequence of operator applications that maps s_1 to s_2). To compute C_{i+1}, the algorithm examines the clauses of C_i one by one, and tries to prove that they remain true after another operator application. Clauses that cannot be proved to be in C_{i+1} are not necessarily dropped—instead, they are weakened and re-tried. The algorithm halts when two successive sets $C_{i'}$ and $C_{i'+1}$ are found to be identical; it returns the final set $C_{i'}$.

The tight link between the sets C_i and distance i from the initial state yields a simple inductive proof of soundness (Rintanen 2005).

The algorithm outlined above is shown in Figure 2; we now proceed to a more detailed description. The typewritten functions in the algorithm denote generic functions, which will be described in more detail later.

The algorithm starts by setting C_0 to the set of all one-literal clauses that are true in the initial state. It then enters the main loop, which terminates only when C_i and C_{i+1} are equal.

In the main loop, it proceeds to construct successive C_{i+1}'s from the corresponding C_i's. To do so, it first sets C_{i+1} to C_i, and then tries each clause c_j of C_{i+1} (lines 3–15), the aim being to see if c_j can be proved to remain true after one further operator application.

This proof is attempted as follows: The algorithm considers every operator o and tries (using `preserved`) to prove that c_j must remain true after the application of o. If it can do so for every o, it falls through the intervening loops and c_j is retained in C_{i+1} in line 14. If, on the other hand, this proof attempt fails for some o, then c_j is weakened (by means of the function `weaken`) and the resulting (weaker) clauses are checked—and, if necessary, further weakened—until a set is found of clauses that do remain true after o. This set is then checked against falsification by the remaining operators in the same way, and the set that results used to replace c_j in C_{i+1} in line 14.

The generic function `preserved`—which tries to prove that a clause c, assumed to be true in any world satisfying all the clauses in a set C, must remain true after one application of an operator o—is clearly crucial. Rintanen's version is shown in Figure 3. This algorithm is self-explanatory: it tries four different ways to prove that c must remain true after o; if they all fail, it concludes that c need not remain true.

The other generic function in Figure 2 is `weaken`. In Rintanen's system, this function just checks if the original clause already has the maximum n disjuncts—in which case it returns $\emptyset$—and, if not, returns the set of all clauses obtained by disjoining one literal to the original clause.

Operator-based Verification

Each C_i in Rintanen's algorithm is just a list of propositional clauses, of varying sizes, that the algorithm knows to be true in all states at distance i from the initial-state. Thus this algorithm can be used soundly to verify a set of candidate propositional hypotheses $\mathcal{H}$ as follows:

1. Ensure that all the candidate hypotheses are true in the initial state
2. Initialize C_0 to $\mathcal{H}$, and
3. Replace `weaken` with a function that always returns $\emptyset$.

Line 1 and 2 above ensure that C_0 satisfies the corresponding invariant; Rintanen's algorithm is constructed so as to ensure that this invariant is maintained by later C_is. Finally, line 3 ensures that unverifiable candidates are dropped, rather than weakened.

This verification method is sound but not, of course, complete (verification of an invariant is PSPACE-complete in general). Thus it is possible that there will be false negatives—true candidate invariants that fail of verification.

We know that all the invariants we find are true in the initial state (indeed, they are true in all k of the states we are given). Thus Line 1 is satisfied. We can thus use this method to verify the invariants our state-based method produces.

Experiments

We evaluated our system in four standard planning problems: one instance of the Blocks world, two of the ATT Logistics world and one of the Towers of Hanoi world. We also used Rintanen's algorithm to verify the candidate invariants we produced as described above. We compare the results with those obtained by Rintanen's operator-based algorithm running by itself in the same domains.

Our Blocks world had 4 blocks and a table; our small Logistics world had 1 package, 2 cities, 2 airports, 2 other locations, 1 airplane, and 1 truck; our large Logistics world had 8 packages, 3 cities, 3 airports, 3 other locations, 3 airplanes, and 3 trucks; and our Towers of Hanoi world had 3 disks and 3 pegs.

We obtained a set of "random" reachable states by randomly performing sequences of valid operations from the initial state. We then randomly selected a subset of these states to operate on. We used 12 states in the Blocks world, the small Logistics world, and the Towers of Hanoi world, and 16 states in the large Logistics world. The experiments were conducted on a 2 GHz Pentium IV with 512 MB of RAM.

Table 1 summarizes the invariants found. For each domain it shows how many candidate invariants our state-based method generated and how many of these were verified by the operator-based verification scheme described above. It also reports how many invariants the operator-based approach was able to discover on its own. The last column shows what percentage of the invariants detected by Rintanen's algorithm the hybrid system was able to generate and verify.

Table 2 gives a comparison of the time taken by the two systems in milliseconds. For our system, it shows the amount of time taken for each of the two steps: state-based candidate invariant generation and operator-based verification. It also gives the time taken by Rintanen's algorithm running independently on the same domain, and finally reports the speedup factor achieved by our generate-and-verify approach.

```
invariants(Σ: set of atoms, I: initial state, O: operators, n: max disjunct size)
1.  C_0 ← {σ | σ ∈ Σ and I ⊨ σ} ∪ {¬σ | σ ∈ Σ and I ⊭ σ}
2.  repeat with i ← 0, 1, ...
3.      C_{i+1} ← C_i
4.      for each clause c_j = ℓ_1 ∨ ... ∨ ℓ_m in C_{i+1}
5.          for each operator o ∈ O
6.              N ← {c_j}
7.              repeat
8.                  N' ← N
9.                  for each clause c_k ∈ N
10.                     if not(preserved(c_k, C_i, o)) then
11.                         N ← (N \ {c_k}) ∪ weaken(c_k, Σ, n)
12.                     endif
13.             until (N = N')
14.         C_{i+1} ← (C_{i+1} \ {c_j}) ∪ N
15. until C_i = C_{i+1}
16. return C_i
```

Figure 2: Rintanen's sound operator-based algorithm

Domain	Our Algorithm			Rintanen	%Found
	#Found	#Verified	%Verified		& Verified
Blocks	218	206	94.5	219	93.2
Logistics (small)	224	209	93.3	231	90.5
Logistics (large)	990	901	91.0	1385	65.0
Towers of Hanoi	81	77	95.1	108	71.3

Table 1: Number of invariants detected and verified

Evaluation

In evaluating these results, we will discuss three questions, based on the roles we set out for state-based invariant discovery methods in the Introduction.

1. How good is the state-based system in isolation?
2. How does the preliminary state-based candidate identification affect the performance of the operator-based invariant detector?
3. How well does the invariant-based verifier work?

Question 1 corresponds to the first potential role we described for state-based methods in the Introduction—that of standalone invariant detector. Questions 2 and 3 correspond to two natural views of the hybrid invariant detection process. If we view the state-based system primarily as a technique to improve the performance of the sound operator-based invariant discovery system, then it is natural to ask, with Question 2, how much value this enhancement adds. If, on the other hand, we think of operator-based verification as a post-processing filter for invariants discovered by the state-based method, then Question 3 is more pertinent.

Standalone State-Based Invariant Detection Table 1 shows that between 91% and 95% of the invariants produced by the state-based method in these domains were subsequently verified. Moreover, it is entirely possible—indeed, very likely—that Rintanen's algorithm failed to verify some candidates that were in fact true invariants. Thus the vast majority of the candidates that our system produced without the use of operator information were genuine, and represented real features of the planning domain.

The time taken for the state-based production of invariants varied from 30–340 msec. Rintanen's algorithm on the same domain required between 20550 and 494490 msec. Our algorithm thus ran between 685 and 3962 times faster than Rintanen's well-known one.

State-based Preprocessing In this section, we evaluate our system with respect to Question 2 above. If our aim is to find verifiably true invariants, how well does the hybrid system do in comparison with Rintanen's operator-based algorithm in isolation?

From Tables 1 and 2, we observe that the hybrid system found between 65% and 90.5% of the true invariants that Rintanen's algorithm did. It is important to remember, however, that this decrease does not entail the loss of any form of "completeness"; Rintanen's algorithm in its original form potentially misses many invariants too. The hybrid system

`preserved`($c = \ell_1 \vee \ldots \vee \ell_r$: a clause, C: a set of clauses, o: an operator)

1. if ($\exists c' \in C$) s.t. [$c' \models \neg \wedge_{(p \in \text{precondition}(o))}\ p$] then
2. $\quad\triangleright$ Operator o is not applicable
3. $\quad$return true
4. else if ($\nexists i \in \{1 \ldots r\}$) s.t. [$\neg\, \ell_i \in \text{effect}(o)$] then
5. $\quad\triangleright$ No literal of c is falsified by o
6. $\quad$return true
7. else if ($\exists j \in \{1 \ldots r\}$) s.t. [$j \neq i$ & $\ell_j \in \text{effect}(o)$] then
8. $\quad\triangleright$ o forces some other literal ℓ_j of c to be true
9. $\quad$return true
10. else if ($\exists k \in \{1 \ldots r\}$) s.t. $k \neq i$ and $\neg\, \ell_k \notin \text{effects}(o)$ and

$$\left[\begin{array}{c} \ell_k \in \text{precondition}(o) \text{ or} \\ (\exists p_1 \ldots p_m \in \text{precondition}(o)) \text{ and } (\neg\, p_1 \vee \ldots \vee \neg\, p_m \vee \ell_k) \in C \end{array}\right]$$

11. $\quad\triangleright$ Some other literal ℓ_k of c was true and hasn't been made false
12. $\quad$return true
13. else
14. $\quad$return false
15. endif

Figure 3: Rintanen's version of the generic function `preserved`

Domain	Our Algorithm			Rintanen	Speedup
	Generation	Verification	Total		Factor
Blocks	40	1060	1100	32640	30
Logistics (small)	80	200	280	316960	1132
Logistics (large)	340	48600	48940	494490	10
Towers of Hanoi	30	70	100	20550	206

Table 2: Time (msec) taken to find and/or verify invariants

found and verified upto 35% fewer invariants than the pure operator-based one, but did so upto 1132 times faster.

Operator-based Verification Here we discuss our results in the light of Question 3. If we treat Rintanen's algorithm as a filter, removing from the list of invariants discovered by our state-based system those that are not easily verifiable, how effective is it?

We have already remarked that Rintanen's algorithm was able to verify most of the invariants our state-based system finds (91%–95%). It is very hard to tell what fraction of those labeled "unverifiable" really were so. Without this figure, it is impossible fully to evaluate the performance of the verifier, but it seems likely that it is doing a good job, since, intuitively, an error rate of 5%–10% in our method seems plausible in the small subsets of states we used.

The speed of the verifier was less impressive. Certainly, it ran many times faster than Rintanen's basic algorithm, but this was still 2–143 times slower than the state-based system in isolation.

Discussion

In cases where statistical invariants are useful (for instance, in probabilistic or priority-based planners) there are significant speed advantages to using the standalone state-based system. The efficiency gains are great enough that the system can often be run multiple times on different subsets of states to increase the number of invariants found and refine the statistics of those generated.

In cases where statistical invariants cannot be used, either the hybrid approach or the pure operator-based one must be used. There is a trade-off between the number of invariants found (higher for the pure operator-based system) and efficiency (higher for the hybrid system). However, the efficiency savings are so great that there is room to improve the size of the hypothesis set generated by the hybrid system by running the algorithm repeatedly, as above.

Finally, the Rintanen system is an effective verifier—it seems to generate only a small proportion of false negatives—but its running time is a very significant drag on the efficiency of the hybrid system. Perhaps other operator-based system (such as DISCOPLAN) might be more effi-

ciently adapted to the purpose.

Conclusions

We have described a system that finds propositional planning invariants from state descriptions, without requiring knowledge of the planning operators. Although an inductive process like ours cannot guarantee the correctness of the laws it finds, we have shown that, in practice, the system is robust enough to overwhelmingly identify only correct invariants.

Our approach's weaker knowledge requirement means that invariants can be identified in more realistic settings, such as systems without the STRIPS assumption, or whose operators are only partially known. We have shown that our system generates sets of invariants comparable with those of systems in the literature that do require full operator knowledge. Moreover, we have demonstrated efficiency improvements of many orders of magnitude over one such system.

We have described one way to combine our state-based method with an operator-based verifier for domains where "likely" invariants are not good enough, and in which information about the operators is in fact available. We have shown that the verifier is able to verify a large proportion of the candidates found by our methods. We compared this hybrid system to a well-known operator-based system, and showed that it achieved up to a thousand-fold improvement in efficiency at the price of a relatively small ($<$35%) drop in number of invariants found.

One interesting direction for future research involves generalizing the propositional invariants found to first-order form. We are working on a Bayesian model of type-structure that will automatically find optimal generalizations for the formulae. We hope that such a system might find invariants that other systems miss.

Acknowledgments This work was supported in part by NSF grant IIS-0328849.

References

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, 1636–1642.

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

Fox, M., and Long, D. 2000. Utilizing automatically inferred invariants in graph construction and search. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 102–111. AAAI Press.

Gerevini, A., and Schubert, L. K. 1998. Inferring state constraints for domain-independent planning. In *AAAI/IAAI*, 905–912.

Gerevini, A., and Schubert, L. K. 2000. Discovering state constraints in DISCOPLAN: Some new results. In *AAAI/IAAI*, 761–767.

Gerevini, A., and Schubert, L. 2001. DISCOPLAN: An efficient on-line system for computing planning domain invariants. In *Proceedings of the 6th European Conference on Planning (ECP-01)*. Toledo, Spain: Springer-Verlag.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, 359–363.

Kautz, H., and Selman, B. 1998. The role of domain-specific axioms in the planning as satisfiability framework. In *Proceedings of AIPS-98*.

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research (JAIR)* 12:338–386.

Mukherji, P., and Schubert, L. K. 2003. Discovering laws as anomalies in logical worlds. Technical Report 828, University of Rochester.

Mukherji, P., and Schubert, L. K. 2005a. Discovering planning invariants as anomalies in state descriptions. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*. Monterey, CA: AAAI Press.

Mukherji, P., and Schubert, L. K. 2005b. State-based discovery and verification of propositional invariants. Technical Report 871, University of Rochester.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering and usage of landmarks in planning. In *Proceedings of the 6th European Conference on Planning (ECP-01)*. Toledo, Spain: Springer-Verlag.

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, 806–811.

Rintanen, J. 2005. Introduction to automated planning (unpublished manuscript).