Principles of AI Planning

9. Invariants

Malte Helmert

Albert-Ludwigs-Universität Freiburg

December 5th, 2008

Principles of AI Planning

December 5th, 2008 — 9. Invariants

Spurious formulae in regression planning

Example

Consider the goal formula

$$A\text{-}on\text{-}B \land B\text{-}on\text{-}C$$

regressed with operator

$$\langle A\text{-}on\text{-}C \land A\text{-}clear \land B\text{-}clear, A\text{-}on\text{-}B \land \neg B\text{-}clear \land C\text{-}clear\rangle$$

resulting in the new subgoal

$$A\text{-}on\text{-}C \land A\text{-}clear \land B\text{-}clear \land B\text{-}on\text{-}C.$$

It is intuitively clear that no state satisfying this formula is reachable by any plan from a legal blocks world state.

Spurious formulae cause unnecessary search

- Goal formulae and formulae obtained by regressing them often represent some states that are not reachable from the initial state.
- If none of the states is reachable from the initial state, there are no plans reaching the formula.
- We would like to have reachable states only, if possible.
- The same problem shows up in satisfiability planning (discussed later in the course):
  partial valuations considered by satisfiability algorithms may represent unreachable states, and this may result in unnecessary search.

Restricting search to reachable sets

Goal: Restriction to states that are reachable.

Problem: Testing reachability is computationally as complex as testing whether a plan exists.

Solution: Use an approximate notion of reachability.

Implementation: Compute in polynomial time formulae that characterize a superset of the reachable states.

Invariants

Definition (invariant)

A formula φ is an invariant of $\langle A, I, O, G\rangle$ if $s \models \varphi$ for every state s reachable from I.

Example

The formula $\neg(A\text{-}on\text{-}B \wedge A\text{-}on\text{-}C)$ is an invariant in a well-formed blocks world task.

Remark

Invariants are usually proved inductively:

- Prove that φ is true in the initial state.
- Prove that operator application preserves φ.

Strongest invariants

Definition (strongest invariant)

An invariant φ is the strongest invariant of $\langle A, I, O, G\rangle$ iff for any invariant ψ, $\varphi \models \psi$.

The strongest invariant exactly characterizes the set of all states that are reachable from the initial state:
For all states s, $s \models \varphi$ if and only if s is reachable.

Remark

There are infinitely many strongest invariants for any given planning task, but they are all logically equivalent.
(If φ is a strongest invariant, then so is $\varphi \wedge \top$, $\varphi \vee \varphi$, ...)

Example: strongest invariant for blocks world

Example (blocks world)

Let X be the set of blocks of a well-formed blocks world task Π, for example $X = \{A, B, C, D\}$.
The conjunction of the following formulae is the strongest invariant for Π:

For all $x \in X$: $clear(x) \leftrightarrow \bigwedge_{y \in X} \neg on(y, x)$
For all $x \in X$: $ontable(x) \leftrightarrow \bigwedge_{y \in X} \neg on(x, y)$
For all $x, y, z \in X$ with $y \neq z$: $\neg on(x, y) \vee \neg on(x, z)$
For all $x, y, z \in X$ with $y \neq z$: $\neg on(y, x) \vee \neg on(z, x)$
For all $n \geq 1$ and $x_1, \ldots, x_n \in X$:
$\neg(on(x_1, x_2) \wedge on(x_2, x_3) \wedge \cdots \wedge on(x_{n-1}, x_n) \wedge on(x_n, x_1))$

Strongest invariants: connection to plan existence

Theorem (strongest invariants vs. plan existence)

Let φ be the strongest invariant for $\Pi = \langle A, I, O, G \rangle$.
Then Π has a plan if and only if $G \wedge \varphi$ is satisfiable.

Proof.

Obvious. □

Strongest invariants: complexity

Theorem (complexity of computing strongest invariants)

Computing the strongest invariant φ is PSPACE-hard.
Even deciding whether or not $\top$ is the strongest invariant is already PSPACE-hard.

Proof.

By reduction from the plan existence problem.
Fact: Testing plan existence for $\langle A, I, O, G \rangle$ is PSPACE-hard.
(We'll show this later in the course!)

Let $a' \notin A$ be a new state variable. Then a plan exists for $\Pi = \langle A, I, O, G \rangle$ iff $\top$ is the strongest invariant of the planning task
$\Pi' = \langle A \cup \{a'\}, I \cup \{a' \mapsto 0\}, O \cup O', G \rangle$, where
$O' = \{\langle G, a' \wedge \bigwedge_{a \in A} a \rangle\} \cup \{ \langle a', \neg a \rangle \mid a \in A \cup \{a'\} \}$.
...

Strongest invariants: complexity (ctd.)

Proof (ctd.)

($\Rightarrow$): If a plan exists for Π, then the same plan is applicable in Π'. We can thus reach a state satisfying G in Π'.
From this state, we can reach *any* state s by first applying $\langle G, a' \wedge \bigwedge_{a \in A} a \rangle$ and then applying the operators $\langle a', \neg a \rangle$ for each variable a with $s(a) = 0$. (If $s(a') = 0$, the corresponding operator must be applied last.)
If *all* states are reachable in Π', then $\top$ is the strongest invariant for Π'.

($\Leftarrow$) (by contraposition): If Π is not solvable, then no state satisfying G is reachable in Π. In that case, no state satisfying G is reachable in Π', and thus a' cannot be made true in Π'. Thus, $\neg a'$ is an invariant in Π' which is stronger than $\top$, so $\top$ is not the strongest invariant in Π'. □

Invariant synthesis: example run

Compute sets C_i of *n*-literal clauses characterizing (giving an upper bound!) the states that are reachable in up to i steps.

Example

$$
\begin{aligned}
C_0 &= \{a, \neg b, c\} && \sim \{101\} \\
C_1 &= \{a \vee b, \neg a \vee \neg b, c\} && \sim \{101, 011\} \\
C_2 &= \{\neg a \vee \neg b, c\} && \sim \{001, 011, 101\} \\
C_3 &= \{\neg a \vee \neg b, c \vee a\} && \sim \{001, 011, 100, 101\} \\
C_4 &= \{\neg a \vee \neg b\} && \sim \{000, 001, 010, 011, 100, 101\} \\
C_5 &= \{\neg a \vee \neg b\} && \sim \{000, 001, 010, 011, 100, 101\} \\
C_i &= C_5 \text{ for all } i > 5 &&
\end{aligned}
$$

$\neg a \vee \neg b$ is the only invariant found.

Invariant synthesis algorithm (informally)

- Start with all 1-literal clauses true in the initial state.
- Repeatedly test every operator vs. every clause to check whether the clause can be shown to be true after applying the operator:
  - One of the literals in the clause is necessarily true: retain.
  - Otherwise, if the clause is too long: forget it.
  - Otherwise, replace the clause by new clauses obtained by adding literals that are now true.
- When all clauses are retained, stop: they are invariants.

Blocks world example

Example (blocks world)

Let $C_0 = \{A\text{-}clear, \neg B\text{-}clear, A\text{-}on\text{-}B, \neg B\text{-}on\text{-}A, \neg A\text{-}on\text{-}T, B\text{-}on\text{-}T\}$ and $o = \langle A\text{-}clear \wedge A\text{-}on\text{-}B, B\text{-}clear \wedge \neg A\text{-}on\text{-}B \wedge A\text{-}on\text{-}T\rangle$.

1. $C_0 \cup \{A\text{-}clear \wedge A\text{-}on\text{-}B\}$ is satisfiable: o is applicable.
2. The 1-literal clauses $\neg B\text{-}clear$, $A\text{-}on\text{-}B$ and $\neg A\text{-}on\text{-}T$ become false when o is applied.
3. They are not thrown away, though: they are replaced by weaker clauses.
4. Literals true after applying o in state s such that $s \models C_0$: $A\text{-}clear$, $B\text{-}clear$, $\neg A\text{-}on\text{-}B$, $\neg B\text{-}on\text{-}A$, $A\text{-}on\text{-}T$, $B\text{-}on\text{-}T$.
5. 2-literal clauses that are weaker than $\neg B\text{-}clear$ and now true are $\neg B\text{-}clear \vee A\text{-}clear$, $\neg B\text{-}clear \vee B\text{-}clear$, $\neg B\text{-}clear \vee \neg A\text{-}on\text{-}B$, $\neg B\text{-}clear \vee \neg B\text{-}on\text{-}A$, $\neg B\text{-}clear \vee A\text{-}on\text{-}T$, and $\neg B\text{-}clear \vee B\text{-}on\text{-}T$.

Blocks world example (ctd.)

Example (ctd.)

6. Similar 2-literal clauses are obtained from $A\text{-}on\text{-}B$ and from $\neg A\text{-}on\text{-}T$.
7. By eliminating logically equivalent ones, tautologies, and clauses that follow from those in C_0 not falsified we get

   $$\begin{aligned} C_1 = \{&A\text{-}clear, \neg B\text{-}on\text{-}A, B\text{-}on\text{-}T, \\ &\neg B\text{-}clear \vee \neg A\text{-}on\text{-}B, \neg B\text{-}clear \vee A\text{-}on\text{-}T, \\ &A\text{-}on\text{-}B \vee B\text{-}clear, A\text{-}on\text{-}B \vee A\text{-}on\text{-}T, \\ &\neg A\text{-}on\text{-}T \vee B\text{-}clear, \neg A\text{-}on\text{-}T \vee \neg A\text{-}on\text{-}B\}\end{aligned}$$

   for distance 1 states.
8. Some clauses in C_1 can be refined further by checking other operators whose preconditions are consistent with C_1.

   With a bit more computation, C_i settles to a set containing all invariants for two blocks.

Simple travel example

Example (simple travel)

$$\begin{aligned} \text{Let } C_i = \{&\neg AinRome \vee \neg AinParis, \\ &\neg AinRome \vee \neg AinNYC, \\ &\neg AinParis \vee \neg AinNYC\}, \\ o = \ &\langle AinRome, AinParis \wedge \neg AinRome\rangle. \end{aligned}$$

- Does o preserve truth of $\neg AinParis \vee \neg AinNYC$?
- Because o makes $\neg AinParis$ false, we must show that $\neg AinNYC$ is true after applying o.
- But $\neg AinNYC$ is not even mentioned in o!
- However, since $AinRome$ is the precondition of o and $\neg AinRome \vee \neg AinNYC$ was true before applying o, we can infer that $\neg AinNYC$ was true before applying o.
- Since o does not make $\neg AinNYC$ false, it is true also after applying o, and then so is $\neg AinParis \vee \neg AinNYC$.

Invariant synthesis: function *preserves-clause*

Test if an operator preserves a clause

def preserves-clause($l_1 \vee \cdots \vee l_n$, C, o):
  for each $l \in \{l_1, \ldots, l_n\}$:
    if not preserves-literal(C, o, $\{l_1, \ldots, l_n\} \setminus \{l\}$, l):
      return false
  return true

Test if an operator preserves a literal

def preserves-literal(C, o, L', l):
  $\langle c, e\rangle := o$
  $C_{\bar{l}} := C \cup \{c\} \cup \{EPC_{\bar{l}}(e)\}$
  return $C_{\bar{l}}$ is unsatisfiable
    or $C_{\bar{l}} \models EPC_{l'}(e)$ for some $l' \in L'$
    or $C_{\bar{l}} \models l' \wedge \neg EPC_{\overline{l'}}(e)$ for some $l' \in L'$

Function *preserves-clause*: examples

Let $C = \{c \vee b\}$.

- preserves-clause($a \vee b$, C, $\langle \neg c, c \wedge d\rangle$) returns **true**
- preserves-clause($a \vee b$, C, $\langle \neg c, \neg a \wedge b\rangle$) returns **true**
- preserves-clause($a \vee b$, C, $\langle b, \neg a\rangle$) returns **true**
- preserves-clause($a \vee b$, C, $\langle \neg c, \neg a\rangle$) returns **true**
- preserves-clause($a \vee b$, C, $\langle c, \neg a\rangle$) returns **false**

Correctness of function *preserves-clause*

Lemma (correctness of *preserves-clause*)

Let C be a set of clauses, $\varphi = l_1 \vee \cdots \vee l_n$ a clause, and o an operator. If preserves-clause(φ, C, o) returns **true**, *then $app_o(s) \models \varphi$ for every state s such that $s \models C \cup \{\varphi\}$ and $app_o(s)$ is defined.*

(Proof omitted.)

Incompleteness of function *preserves-clause*

Example (incompleteness of *preserves-clause*)

Let $o = \langle a, \neg b \wedge (c \rhd d) \wedge (\neg c \rhd e)\rangle$.

preserves-clause($b \vee d \vee e$, $\emptyset$, o) returns **false** because the preserves-literal check for $l = b$ fails:

- Operator o can make b false.
- It is not guaranteed that d is true in the resulting state.
- It is not guaranteed that e is true in the resulting state.

However, $d \vee e$ is true after applying o, and hence $b \vee d \vee e$ will be true as well.

Invariant synthesis: outline of main procedure

1. C = the set of 1-literal clauses true in the initial state.
2. For each operator o and clause $\varphi \in C$, test if φ remains true when o is applied.
3. If not, remove φ, and if the number of literals in φ is less than n, add clauses $\varphi \vee l$ for each literal l which is guaranteed to be true after applying o.
4. Remove all dominated invariants.
5. Repeat from step 2 if C has changed in the previous two steps.
6. Otherwise every clause in C is an invariant.

For any fixed limit n on the size of the clauses, the number of iterations is $O(m^n)$ (where $m = |A|$ is the number of state variables) and hence polynomial.

Invariant synthesis: the main procedure

Invariant synthesis

def invariants(A, I, O, n):
  $C := \{ a \in A \mid I \models a \} \cup \{ \neg a \mid a \in A, I \not\models a \}$
  repeat:
    $C' := C$
    for each $l_1 \vee \cdots \vee l_m \in C'$ **and** $o = \langle c, e \rangle \in O$
        with preserves-clause($l_1 \vee \cdots \vee l_m$, C', o) = **false**:
      $C := C \setminus \{l_1 \vee \cdots \vee l_m\}$
      if $m < n$:
        for each literal l:
          if $C' \cup \{c\} \models EPC_l(e) \vee (l \wedge \neg EPC_{\bar{l}}(e))$:
            $C := C \cup \{l_1 \vee \cdots \vee l_m \vee l\}$
    $C := \{ \varphi \in C \mid \neg\exists \varphi' \in C : \varphi' \models \varphi$ and $\varphi' \not\equiv \varphi \}$
  until $C = C'$
  return C

Invariant synthesis: correctness

Theorem (correctness of *invariants*)

The procedure invariants(A, I, O, n) returns a set C of clauses with at most n literals such that for any applicable operator sequence $o_1, \dots, o_m \in O$: $app_{o_1 \dots o_m}(I) \models C$.

Proof.

A $I \models C$:
- The initial state satisfies the initial set of 1-literal clauses.
- All modifications to the clause set only make it logically weaker (i.e., $C' \models C$ after each iteration of the main loop.)
- Thus the initial state satisfies the resulting clause set C by induction over the number of iterations.

...

Invariant synthesis: correctness (ctd.)

Proof (ctd.)

B If $s \models C$ and $app_o(s)$ is defined, then $app_o(s) \models C$.
- In the last iteration of the procedure, no formula is removed from $C = C'$, and hence preserves-clause(φ, C, o) is true for all clauses $\varphi \in C$ and operators $o \in O$.
- By the lemma, this means that $app_o(s) \models \varphi$ for every state s such that $s \models C$ and $app_o(s)$ is defined.
- Since this is true for all clauses $\varphi \in C$, we get $app_o(s) \models C$ for every state s such that $s \models C$ and $app_o(s)$ is defined.

From A and B, the theorem follows by induction over the length of the operator sequence. □

Why is the strongest invariant not always found?

- The function *preserves-clause* is incomplete for general operators (but complete for STRIPS operators.)
  Making it complete makes it NP-hard.
- The strongest invariant may require arbitrarily long clauses, so the restriction to clauses of any fixed length makes it impossible to represent it.

Example

The acyclicity of the *on* relation in the blocks world needs clauses of length n when there are n blocks.

- Practical implementations of the algorithm use polynomial time approximations of the tests for satisfiability and $\models$.

Invariant synthesis: example

Initial state: $I \models a \wedge \neg b \wedge \neg c$

Operators: $o_1 = \langle a, \neg a \wedge b \rangle$,
$o_2 = \langle b, \neg b \wedge c \rangle$,
$o_3 = \langle c, \neg c \wedge a \rangle$

Computation: Find invariants with at most 2 literals:

$$\begin{array}{rcl} C_0 & = & \{a, \neg b, \neg c\} \\ C_1 & = & \{\neg c, a \vee b, \neg b \vee \neg a\} \\ C_2 & = & \{\neg b \vee \neg a, \neg c \vee \neg a, \neg c \vee \neg b\} \\ C_3 & = & \{\neg b \vee \neg a, \neg c \vee \neg a, \neg c \vee \neg b\} \\ C_i & = & C_2 \text{ for all } i \geq 2 \end{array}$$

Invariants for regression: motivating example

Example

Regression of in(A, Freiburg) by
⟨in(A, Strasbourg), ¬in(A, Strasbourg) ∧ in(A, Paris)⟩
gives in(A, Freiburg) ∧ in(A, Strasbourg)

No state satisfying in(A, Freiburg) ∧ in(A,Strasbourg) makes sense if *A* denotes some usual physical object.

Exploiting invariants for regression

Problem: Regression produces sets T of states such that
- some states in T are unreachable from I, or even
- all states in T are unreachable from I.

The first is not always a serious problem (but may worsen the quality of distance estimates, for example.)

Solution: Use invariants to avoid formulae that do not represent any reachable states.
1. Compute invariant φ.
2. Do only regression steps such that $regr_o(\psi) \wedge \varphi$ is satisfiable.

Exploiting invariants in satisfiability planning

- Invariants are very useful in the planning as satisfiability framework (SAT planning), where they help reduce the search space for the SAT solver.
- We will discuss SAT planning later in this course.

Invariants for problem reformulation: mutexes

Binary clause invariants are called mutexes because they state that certain variable assignments cannot be simultaneously true and are hence mutually exclusive.

Example

The invariant $\neg$*A-on-B* $\vee$ $\neg$*A-on-C* states that *A-on-B* and *A-on-C* are mutex.

Often, a larger set of literals is mutually exclusive because every pair of them forms a mutex.

Example

In blocks world, *B-on-A*, *C-on-A*, *D-on-A* and *A-clear* are mutex.

Encoding mutex groups as finite-domain variables

Let $L = \{l_1, \dots, l_n\}$ be mutually exclusive literals over n different variables $A_L = \{a_1, \dots, a_n\}$.

Then the planning task can be rephrased using a single finite-domain (i.e., non-binary) state variable v_L with $n + 1$ possible values in place of the n variables in A_L:

- n of the possible values represent situations in which exactly one of the literals in L is true.
- The remaining value represents situations in which none of the literals in L is true.
  - Note: If we can prove that one of the literals in L has to be true in each state, this additional value can be omitted.

In many cases, the reduction in the number of variables can dramatically improve performance of a planning algorithm.

Finite-domain state variables

Definition (finite-domain state variable)

A finite-domain state variable is a symbol v with an associated finite domain, i. e., a non-empty finite set.

We write $\mathcal{D}_v$ for the domain of v.

Example

$v =$ *above-a*, $\mathcal{D}_{above\text{-}a} = \{\mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{nothing}\}$

This state variable encodes the same information as the propositional variables *B-on-A*, *C-on-A*, *D-on-A* and *A-clear*.

Finite-domain states

Definition (finite-domain state)

Let V be a finite set of finite-domain state variables.

A state over V is an assignment $s : V \to \bigcup_{v \in V} \mathcal{D}_v$ such that $s(v) \in \mathcal{D}_v$ for all $v \in V$.

Example

$s = \{$*above-a* $\mapsto$ nothing, *above-b* $\mapsto$ a, *above-c* $\mapsto$ b,
below-a $\mapsto$ b, *below-b* $\mapsto$ c, *below-c* $\mapsto$ table$\}$

Finite-domain formulae

Definition (finite-domain formulae)

Logical formulae over finite-domain state variables V are defined as in the propositional case, except that instead of atomic formulae of the form $a \in A$, there are atomic formulae of the form $v = d$, where $v \in V$ and $d \in \mathcal{D}_v$.

Example

The formulae (*above-a* = nothing) $\vee \neg$(*below-b* = c) corresponds to the formula *A-clear* $\vee \neg$*B-on-C*.

Finite-domain effects

Definition (finite-domain effects)

Effects over finite-domain state variables V are defined as in the propositional case, except that instead of atomic effects of the form a and $\neg a$ with $a \in A$, there are atomic effects of the form $v := d$, where $v \in V$ and $d \in \mathcal{D}_v$.

Example

The effect (*below-a* := table) $\wedge$ ((*above-b* = a) $\rhd$ (*above-b* := nothing)) corresponds to the effect
A-on-T $\wedge \neg$*A-on-B* $\wedge \neg$*A-on-C* $\wedge \neg$*A-on-D* $\wedge$ (*A-on-B* $\rhd$ ($\neg$*A-on-B* $\wedge$ *B-clear*)).

$\rightsquigarrow$ definition of finite-domain operators follows

Planning tasks in finite-domain representation

Definition (planning task in finite-domain representation)

A deterministic planning task in finite-domain representation or FDR planning task is a 4-tuple $\Pi = \langle V, I, O, G \rangle$ where

- V is a finite set of finite-domain state variables,
- I is an initial state over V,
- O is a finite set of finite-domain operators over V, and
- G is a formula over V describing the goal states.

Relationship to propositional planning tasks

Definition (induced propositional planning task)

Let $\Pi = \langle V, I, O, G \rangle$ be an FDR planning task.
The induced propositional planning task Π' is the (regular) planning task $\Pi' = \langle A', I', O', G' \rangle$, where

- $A' = \{(v, d) \mid v \in V, d \in \mathcal{D}_v\}$
- $I'((v, d)) = 1$ iff $I(v) = d$
- O' and G' are obtained from O and G by replacing
  - each atomic formula $v = d$ with the proposition (v, d), and
  - each atomic effect $v := d$ with the effect $(v, d) \wedge \bigwedge_{d' \in \mathcal{D}_v \setminus \{d\}} \neg(v, d')$.

- $\leadsto$ can define operator semantics, plans, relaxed planning graphs, ... for Π in terms of its induced propositional planning task

SAS$^+$ planning tasks

Definition (SAS$^+$ planning task)

An FDR planning task $\Pi = \langle V, I, O, G \rangle$ is called an SAS$^+$ planning task iff there are no conditional effects in O and all operator preconditions in O and the goal formula G are conjunctions of atoms.

- analogue of STRIPS planning tasks for finite-domain representations
- induced propositional planning task of a SAS$^+$ planning task is STRIPS
- FDR tasks obtained by invariant-based reformulation of STRIPS planning task are SAS$^+$

Literature on invariant synthesis

DISCOPLAN (Gerevini & Schubert, 1998)

- many classes of invariants (not just mutexes), but not general clausal invariants
- generate/test/repair approach (similar to the algorithm presented here)
- limited to STRIPS
- works directly with schematic operators
- usually fast, but too expensive for some large tasks

Literature on invariant synthesis (ctd.)

TIM (Fox & Long, 1998)

- mutexes + some additional invariants
- not a generate/test/repair approach (or at least, not described as such)
- limited to STRIPS
- works directly with schematic operators

Literature on invariant synthesis (ctd.)

Edelkamp & Helmert's algorithm (1999)

- only mutexes
- specifically tailored towards FDR reformulation
- generate/test/repair approach
  (similar to the algorithm presented here)
- limited to STRIPS
- works directly with schematic operators
- fast, but limitations in PDDL support
  (even in addition to being STRIPS only)

Literature on invariant synthesis (ctd.)

Rintanen's algorithm (2000)

- general clausal invariants
  - however, speed unclear for general invariants
    (beyond mutexes)
- generate/test/repair approach
- limited to STRIPS
- works with schematic operators

The algorithm presented in this section is essentially Rintanen's algorithm, translated to non-schematic operators.

Literature on invariant synthesis (ctd.)

Bonet & Geffner's algorithm (2001)

- mutexes only
- generate/test approach (without repair stage)
- limited to STRIPS
- works with propositional representation (not schematic)
- can be seen as simpler version of Rintanen's algorithm
- quite expensive for very large planning tasks
- developed for additional pruning in regression search

Literature on invariant synthesis (ctd.)

Helmert's algorithm (2009)

- only mutexes
- specifically tailored towards FDR reformulation
- generate/test/repair approach
  (similar to the algorithm presented here)
- not limited to STRIPS
- works directly with schematic operators
- fast

Summary

- Invariants help make backward search and satisfiability planning more efficient and (in the case of mutexes) can be used for problem reformulation.
- We gave an algorithm for computing a class of invariants.
  1. Start with 1-literal clauses true in the initial state.
  2. Repeatedly weaken clauses that could not be shown to be invariants.
  3. Stop when all clauses are guaranteed to be invariants.
- The algorithm runs in polynomial time if the satisfiability and logical consequence tests are approximated by a polynomial time algorithm and the size of the invariant clauses is bounded by a constant.