Principles of AI Planning

9. Invariants

Malte Helmert
Albert-Ludwigs-Universität Freiburg

December 5th, 2008

Invariants

Motivation
Definition
Example
Complexity

Algorithms
Idea
Example
Invariant test
Main procedure
Example

Applications
Regression & SAT planning
Reformulation

Conclusion
Literature & summary

Spurious formulae in regression planning

Example
Consider the goal formula

\[ A-on-B \land B-on-C \]

regressed with operator

\[ \langle A-on-C \land A-clear \land B-clear, A-on-B \land \neg B-clear \land C-clear \rangle \]

resulting in the new subgoal

\[ A-on-C \land A-clear \land B-clear \land B-on-C. \]

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

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

**Remark**
Invariants are usually proved inductively:
- Prove that $\varphi$ is true in the initial state.
- Prove that operator application preserves $\varphi$.

---

Strongest invariants

**Definition (strongest invariant)**
An invariant $\varphi$ is the strongest invariant of $\langle A, I, O, G \rangle$ iff for any invariant $\psi$, $\varphi \models \psi$.

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 $\varphi$ is a strongest invariant, then so is $\varphi \land \top$, $\varphi \lor \varphi$, ...)

---

Example: strongest invariant for blocks world

**Example (blocks world)**
Let $X$ be the set of blocks of a well-formed blocks world task $\Pi$, for example $X = \{A, B, C, D\}$.

The conjunction of the following formulae is the strongest invariant for $\Pi$:

For all $x \in X$:
- $\text{clear}(x) \leftrightarrow \bigwedge_{y \in X} \neg \text{on}(y, x)$
- $\text{ontable}(x) \leftrightarrow \bigwedge_{y \in X} \neg \text{on}(x, y)$
- For all $x, y, z \in X$ with $y \neq z$:
  - $\neg \text{on}(x, y) \lor \neg \text{on}(x, z)$
  - $\neg \text{on}(x, y) \lor \neg \text{on}(x, z)$
  - For all $n \geq 1$ and $x_1, \ldots, x_n \in X$:
    - $\neg (\text{on}(x_1, x_2) \land \text{on}(x_2, x_3) \land \cdots \land \text{on}(x_{n-1}, x_n) \land \text{on}(x_n, x_1))$
Strongest invariants: connection to plan existence

Theorem (strongest invariants vs. plan existence)
Let $\varphi$ be the strongest invariant for $\Pi = \langle A, I, O, G \rangle$.
Then $\Pi$ has a plan if and only if $G \land \varphi$ is satisfiable.

Proof.
Obvious.

Strongest invariants: complexity

Theorem (complexity of computing strongest invariants)
Computing the strongest invariant $\varphi$ 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 $O' = \{\langle G, a' \land \bigwedge_{a \in A} a \rangle\} \cup \{\langle a', \neg a \rangle \mid a \in A \cup \{a'\}\}.$

Proof (ctd.)
($\Rightarrow$): If a plan exists for $\Pi$, then the same plan is applicable in $\Pi'$. We can thus reach a state satisfying $G$ in $\Pi'$.
From this state, we can reach any state $s$ by first applying $\langle G, a' \land \bigwedge_{a \in A} a \rangle$ 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 $\Pi'$, then $\top$ is the strongest invariant for $\Pi'$.
($\Leftarrow$) (by contraposition): If $\Pi$ is not solvable, then no state satisfying $G$ is reachable in $\Pi$. In that case, no state satisfying $G$ is reachable in $\Pi'$, and thus $a'$ cannot be made true in $\Pi'$. Thus, $\neg a'$ is an invariant in $\Pi'$ which is stronger than $\top$, so $\top$ is not the strongest invariant in $\Pi'$.

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

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

$\neg a \lor \neg b$ 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 \( C_0 = \{ \neg B\text{-clear}, A\text{-on-}B, \neg B\text{-on-}A, \neg A\text{-on-}T, B\text{-on-}T \} \) and \( o = (A\text{-clear} \land A\text{-on-}B, B\text{-clear} \land \neg A\text{-on-}B \land A\text{-on-}T) \).

1. \( C_0 \cup \{ A\text{-clear} \land A\text{-on-}B \} \) is satisfiable: \( o \) is applicable.
2. The 1-literal clauses \( \neg B\text{-clear}, A\text{-on-}B \) and \( \neg A\text{-on-}T \) 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 \( s \models C_0: A\text{-clear}, B\text{-clear}, \neg A\text{-on-}B, \neg B\text{-on-}A, A\text{-on-}T, B\text{-on-}T \).
5. 2-literal clauses that are weaker than \( \neg B\text{-clear} \) and now true are \( \neg B\text{-clear} \lor A\text{-clear}, \neg B\text{-clear} \lor B\text{-clear}, \neg B\text{-clear} \lor \neg A\text{-on-}B, \neg B\text{-clear} \lor \neg B\text{-on-}A, \neg B\text{-clear} \lor A\text{-on-}T, \) and \( \neg B\text{-clear} \lor B\text{-on-}T \).

Blocks world example (ctd.)

Example (blocks world)

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

\[ C_1 = \{ A\text{-clear}, \neg B\text{-on-}A, B\text{-on-}T, \neg B\text{-clear} \lor \neg A\text{-on-}B, \neg B\text{-clear} \lor A\text{-on-}T, A\text{-on-}B \lor B\text{-clear}, A\text{-on-}B \lor A\text{-on-}T, \neg A\text{-on-}T \lor B\text{-clear}, \neg A\text{-on-}T \lor \neg A\text{-on-}B \} \]

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

Simple travel example

Example (simple travel)

Let \( C_i = \{ \neg Ain\text{-Rome} \lor \neg Ain\text{-Paris}, \neg Ain\text{-Rome} \lor \neg Ain\text{-NYC}, \neg Ain\text{-Paris} \lor \neg Ain\text{-NYC} \} \) and \( o = (Ain\text{-Rome}, Ain\text{-Paris} \land \neg Ain\text{-Rome}) \).

- Does \( o \) preserve truth of \( \neg Ain\text{-Paris} \lor \neg Ain\text{-NYC} \)?
- Because \( o \) makes \( \neg Ain\text{-Paris} \) false, we must show that \( \neg Ain\text{-NYC} \) is true after applying \( o \).
- However, since \( Ain\text{-Rome} \) is the precondition of \( o \) and \( \neg Ain\text{-Rome} \lor \neg Ain\text{-NYC} \) was true before applying \( o \), we can infer that \( \neg Ain\text{-NYC} \) was true before applying \( o \).
- Since \( o \) does not make \( \neg Ain\text{-NYC} \) false, it is true also after applying \( o \), and then so is \( \neg Ain\text{-Paris} \lor \neg Ain\text{-NYC} \).
Invariant synthesis: function *preserves-clause*

Test if an operator preserves a clause

```python
def preserves-clause(l₁ ∨ ... ∨ lₙ, C, o):
    for each l ∈ {l₁, ..., lₙ}:
        if not preserves-literal(C, o, {l₁, ..., lₙ} \ {l}, l):
            return false
    return true
```

Test if an operator preserves a literal

```python
def preserves-literal(C, o, L', l):
    ⟨c, e⟩ := o
    C_l := C ∪ {c} ∪ {EPC_l(e)}
    return C_l is unsatisfiable
    or C_l ⊨ EPC_{l'}(e) for some l' ∈ L'
    or C_l ⊨ l' ∧ ¬EPC_{l'}(e) for some l' ∈ L'
```

Correctness of function *preserves-clause*

**Lemma (correctness of preserves-clause)**

Let C be a set of clauses, \( \varphi = l₁ ∨ ... ∨ lₙ \) a clause, and o an operator. If preserves-clause(\( \varphi, C, o \)) returns true, then \( \text{app}_o(s) \models \varphi \) for every state s such that \( s \models C \cup \{ \varphi \} \) and \( \text{app}_o(s) \) is defined.

(Proof omitted.)

Function *preserves-clause*: examples

Let \( C = \{ c \lor b \} \).

- preserves-clause(\( a \lor b, C, (\neg c, c \land d) \)) returns true
- preserves-clause(\( a \lor b, C, (\neg c, \neg a \land b) \)) returns true
- preserves-clause(\( a \lor b, C, (b, \neg a) \)) returns true
- preserves-clause(\( a \lor b, C, (\neg c, \neg a) \)) returns true
- preserves-clause(\( a \lor b, C, (c, \neg a) \)) returns false

Incompleteness of function *preserves-clause*

**Example (incompleteness of preserves-clause)**

Let \( o = \langle a, \neg b \land (c \triangleright d) \land (\neg c \triangleright e) \rangle \).

preserves-clause(\( b \lor d \lor e, \emptyset, o \)) 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, \( d \lor e \) is true after applying o, and hence \( b \lor d \lor e \) 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 \( \varphi \) remains true when \( o \) is applied.
3. If not, remove \( \varphi \), and if the number of literals in \( \varphi \) is less than \( n \), add clauses \( \varphi \lor l \) 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

Definition: \( \text{invariants}(A, I, O, n) \):

\[
\begin{align*}
C & := \{ a \in A \mid I \models a \} \cup \{ \neg a \mid a \in A, I \not\models a \} \\
\text{repeat} & \quad C' := C \\
& \quad \text{for each } l_1 \lor \cdots \lor l_m \in C' \text{ and } o = (c, e) \in O \\
& \quad \quad \text{with } \text{preserves-clause}(l_1 \lor \cdots \lor l_m, C', o) = \text{false}: \\
& \quad \quad \quad C := C \setminus \{ l_1 \lor \cdots \lor l_m \} \\
& \quad \quad \quad \text{if } m < n: \\
& \quad \quad \quad \quad \text{for each literal } l: \\
& \quad \quad \quad \quad \quad \text{if } C' \cup \{ e \} \models \text{EPC}(e) \lor (l \land \neg \text{EPC}(e)): \\
& \quad \quad \quad \quad \quad \quad C := C \cup \{ l_1 \lor \cdots \lor l_m \lor l \} \\
& \quad \quad \quad \quad \quad \quad \text{if } \varphi \in C \land \exists \varphi' \in C : \varphi' \models \varphi \text{ and } \varphi' \not\equiv \varphi \} \\
& \quad \quad \quad \text{until } C = C' \\
& \quad \text{return } C
\end{align*}
\]

Invariant synthesis: correctness

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

Proof.

A. \( I \models C \):

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

B. If \( s \models C \) and \( \text{app}_o(s) \) is defined, then \( \text{app}_o(s) \models C \).

1. In the last iteration of the procedure, no formula is removed from \( C = C' \), and hence \( \text{preserves-clause}(\varphi, C, o) \) is true for all clauses \( \varphi \in C \) and operators \( o \in O \).
2. By the lemma, this means that \( \text{app}_o(s) \models \varphi \) for every state \( s \) such that \( s \models C \) and \( \text{app}_o(s) \) is defined.
3. Since this is true for all clauses \( \varphi \in C \), we get \( \text{app}_o(s) \models C \) for every state \( s \) such that \( s \models C \) and \( \text{app}_o(s) \) 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 \textit{preserves-clause} is incomplete for general operators (but complete for STRIPS operators.)
  Making it complete makes it \textit{NP-hard}.
- The strongest invariant may require \textit{arbitrarily long clauses}, so the restriction to clauses of any \textit{fixed length} makes it impossible to represent it.

Example

The acyclicity of the \textit{on} relation in the blocks world needs clauses of length \(n\) when there are \(n\) blocks.

- Practical implementations of the algorithm use \textit{polynomial time approximations} of the tests for satisfiability and \(\models\).

Invariant synthesis: example

Initial state: \(I \models a \land \neg b \land \neg c\)

Operators:

\(o_1 = \langle a, \neg a \land b \rangle\),
\(o_2 = \langle b, \neg b \land c \rangle\),
\(o_3 = \langle c, \neg c \land a \rangle\)

Computation: Find invariants with at most 2 literals:

\[
C_0 = \{a, \neg b, \neg c\} \\
C_1 = \{\neg c, a \lor b, \neg b \lor \neg a\} \\
C_2 = \{\neg b \lor \neg a, \neg c \lor \neg a, \neg c \lor \neg b\} \\
C_3 = \{\neg b \lor \neg a, \neg c \lor \neg a, \neg c \lor \neg b\} \\
C_i = C_2 \text{ for all } i \geq 2
\]

Invariants for regression: motivating example

Example

Regression of \textit{in}(A, \text{Freiburg}) by
\(\langle \text{in}(A, \text{Strasbourg}), \neg \text{in}(A, \text{Strasbourg}) \land \text{in}(A, \text{Paris}) \rangle\)
gives \textit{in}(A, \text{Freiburg}) \land \textit{in}(A, \text{Strasbourg})

No state satisfying \textit{in}(A, \text{Freiburg}) \land \textit{in}(A, \text{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 \textit{unreachable} from \(I\), or even
- all states in \(T\) are \textit{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 \(\varphi\).
2. Do only regression steps such that \(\text{regr}_o(\psi) \land \varphi\) 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 \( \neg A-on-B \lor \neg A-on-C \) 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 \( L = \{ l_1, \ldots, l_n \} \) be mutually exclusive literals over \( n \) different variables \( A_L = \{ a_1, \ldots, a_n \} \).

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 \( D_v \) for the domain of \( v \).

Example
\( v = \text{above-a} \), \( D_{\text{above-a}} = \{ b, c, d, \text{nothing} \} \)

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 \( s: V \rightarrow \bigcup_{v \in V} D_v \) such that \( s(v) \in D_v \) for all \( v \in V \).

Example
\[ s = \{ \text{above-a} \mapsto \text{nothing}, \text{above-b} \mapsto \text{a}, \text{above-c} \mapsto \text{b}, \text{below-a} \mapsto \text{b}, \text{below-b} \mapsto \text{c}, \text{below-c} \mapsto \text{table} \} \]

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 \( d \in D_v \).

Example
The formulae (\( \text{above-a} = \text{nothing} \)) \( \lor \lnot(\text{below-b} = \text{c}) \) corresponds to the formula \( A\text{-clear} \lor \lnot B\text{-on-C} \).

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 \( \lnot a \) with \( a \in A \), there are atomic effects of the form \( v := d \), where \( v \in V \) and \( d \in D_v \).

Example
The effect (\( \text{below-a} := \text{table} \)) \( \land ((\text{above-b} = \text{a}) \triangleright (\text{above-b} := \text{nothing})) \) corresponds to the effect \( A\text{-on-T} \land \lnot A\text{-on-B} \land \lnot A\text{-on-C} \land \lnot A\text{-on-D} \land (A\text{-on-B} \triangleright (\lnot A\text{-on-B} \land B\text{-clear})). \)

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 \( \Pi' \) is the (regular) planning task \( \Pi' = \langle A', I', O', G' \rangle \), where

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

\( \Rightarrow \) can define operator semantics, plans, relaxed planning graphs, \ldots for \( \Pi \) in terms of its induced propositional planning task

SAS\textsuperscript{+} planning tasks

Definition (SAS\textsuperscript{+} planning task)
An FDR planning task \( \Pi = \langle V, I, O, G \rangle \) is called an SAS\textsuperscript{+} 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\textsuperscript{+} planning task is STRIPS
- FDR tasks obtained by invariant-based reformulation of STRIPS planning task are SAS\textsuperscript{+}

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

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)

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.

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

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.