Principles of AI Planning

Invariants

Malte Helmert   Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

December 1st, 2006

Example

Consider the goal formula

$$AonB \wedge BonC$$

regressed with operator

$$\langle AonC \wedge Aclear \wedge Bclear, AonB \wedge \neg Bclear \wedge Cclear\rangle$$

resulting in the new goal

$$AonC \wedge Aclear \wedge Bclear \wedge BonC.$$

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

- 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: partial valuations considered by satisfiability algorithms may represent unreachable states, and this may result in unnecessary search.

Invariants

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

Definition

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

Example

The formula $\neg(\mathit{AonB} \wedge \mathit{AonC})$ is an invariant in a blocks world task.

Remark

Invariants are usually proved inductively:

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

Invariants: the strongest invariant

Definition

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

The strongest invariant exactly characterizes the set of all states that are reachable from the initial state:
For all states s, $s \models \phi$ 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 $\phi \vee \phi \ldots$)

The strongest invariant for the blocks world

Let X be the set of blocks, for example $X = \{A, B, C, D\}$.
The conjunction of the following formulae is the strongest invariant for the set of all states for the blocks X.

For all $x \in X$: $\text{clear}(x) \leftrightarrow \bigwedge_{y \in X} \neg \text{on}(y, x)$

For all $x \in X$: $\text{ontable}(x) \leftrightarrow \bigwedge_{y \in X} \neg \text{on}(x, y)$

For all $x, y, z \in X$ with $y \neq z$: $\neg \text{on}(x, y) \vee \neg \text{on}(x, z)$

For all $x, y, z \in X$ with $y \neq z$: $\neg \text{on}(y, x) \vee \neg \text{on}(z, x)$

For all $n \geq 1$ and $x_1, \ldots, x_n \in X$:

$\neg(\text{on}(x_1, x_2) \wedge \text{on}(x_2, x_3) \wedge \cdots \wedge \text{on}(x_{n-1}, x_n) \wedge \text{on}(x_n, x_1))$

Invariants: connection to plan existence

Theorem

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

Proof.

Very easy! □

Theorem

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

Invariants: connection to plan existence

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 this month!)

Let $a' \notin A$ be a new state variable. Then a plan exists for $\mathcal{T} = \langle A, I, O, G \rangle$ iff $\top$ is the strongest invariant of the planning task $\mathcal{T}' = \langle A \cup \{a'\}, I \cup \{a' \mapsto 0\}, O \cup O', G \rangle$, where

$$O' = \{\langle G, a' \wedge \bigwedge_{a \in A} a \rangle\} \cup \{ \langle a', \neg a \rangle \mid a \in A \cup \{a'\} \}.$$

...

□

Invariants: connection to plan existence

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 this month!)

Let $a' \notin A$ be a new state variable. Then a plan exists for $\mathcal{T} = \langle A, I, O, G \rangle$ iff $\top$ is the strongest invariant of the planning task $\mathcal{T}' = \langle A \cup \{a'\}, I \cup \{a' \mapsto 0\}, O \cup O', G \rangle$, where

$$O' = \{\langle G, a' \wedge \bigwedge_{a \in A} a \rangle\} \cup \{ \langle a', \neg a \rangle \mid a \in A \cup \{a'\} \}.$$

...

□

Invariants: connection to plan existence

Proof continues...

(⇒): If a plan exists for $\mathcal{T}$, then the same plan is applicable in $\mathcal{T}'$. We can thus reach a state satisfying G in $\mathcal{T}'$.

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 $\mathcal{T}'$, then $\top$ is the strongest invariant for $\mathcal{T}'$.

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

Invariants: connection to plan existence

Proof continues...

(⇒): If a plan exists for $\mathcal{T}$, then the same plan is applicable in $\mathcal{T}'$. We can thus reach a state satisfying G in $\mathcal{T}'$.

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 $\mathcal{T}'$, then $\top$ is the strongest invariant for $\mathcal{T}'$.

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

Invariants: connection to plan existence

Proof continues...

(⇒): If a plan exists for $\mathcal{T}$, then the same plan is applicable in $\mathcal{T}'$. We can thus reach a state satisfying G in $\mathcal{T}'$.

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 $\mathcal{T}'$, then $\top$ is the strongest invariant for $\mathcal{T}'$.

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

Invariants: connection to plan existence

Proof continues...

($\Rightarrow$): If a plan exists for $\mathcal{T}$, then the same plan is applicable in $\mathcal{T}'$. We can thus reach a state satisfying G in $\mathcal{T}'$.
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 $\mathcal{T}'$, then $\top$ is the strongest invariant for $\mathcal{T}'$.

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

Computation of invariants: informally

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

Computation of invariants: informally

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

Computation of invariants
Example

Example

Let $C_0 = \{Aclear, \neg Bclear, AonB, \neg BonA, \neg AonT, BonT\}$ and $o = \langle Aclear \wedge AonB, Bclear \wedge \neg AonB \wedge AonT \rangle$.

1. $C_0 \cup \{Aclear \wedge AonB\}$ is satisfiable: o is applicable.
2. The 1-literal clauses $\neg Bclear$, $AonB$ and $\neg AonT$ 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 $s \models C_0$: $Aclear$, $Bclear$, $\neg AonB$, $\neg BonA$, $AonT$, $BonT$.
5. 2-literal clauses that are weaker than $\neg Bclear$ and now true are $\neg Bclear \vee Aclear$, $\neg Bclear \vee Bclear$, $\neg Bclear \vee \neg AonB$, $\neg Bclear \vee \neg BonA$, $\neg Bclear \vee AonT$, and $\neg Bclear \vee BonT$.

Example

Let $C_0 = \{Aclear, \neg Bclear, AonB, \neg BonA, \neg AonT, BonT\}$ and $o = \langle Aclear \wedge AonB, Bclear \wedge \neg AonB \wedge AonT \rangle$.

1. $C_0 \cup \{Aclear \wedge AonB\}$ is satisfiable: o is applicable.
2. The 1-literal clauses $\neg Bclear$, $AonB$ and $\neg AonT$ 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 $s \models C_0$: $Aclear$, $Bclear$, $\neg AonB$, $\neg BonA$, $AonT$, $BonT$.
5. 2-literal clauses that are weaker than $\neg Bclear$ and now true are $\neg Bclear \vee Aclear$, $\neg Bclear \vee Bclear$, $\neg Bclear \vee \neg AonB$, $\neg Bclear \vee \neg BonA$, $\neg Bclear \vee AonT$, and $\neg Bclear \vee BonT$.

Example

Let $C_0 = \{Aclear, \neg Bclear, AonB, \neg BonA, \neg AonT, BonT\}$ and $o = \langle Aclear \wedge AonB, Bclear \wedge \neg AonB \wedge AonT \rangle$.

1. $C_0 \cup \{Aclear \wedge AonB\}$ is satisfiable: o is applicable.
2. The 1-literal clauses $\neg Bclear$, $AonB$ and $\neg AonT$ 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 $s \models C_0$: $Aclear$, $Bclear$, $\neg AonB$, $\neg BonA$, $AonT$, $BonT$.
5. 2-literal clauses that are weaker than $\neg Bclear$ and now true are $\neg Bclear \vee Aclear$, $\neg Bclear \vee Bclear$, $\neg Bclear \vee \neg AonB$, $\neg Bclear \vee \neg BonA$, $\neg Bclear \vee AonT$, and $\neg Bclear \vee BonT$.

Example

Let $C_0 = \{\mathit{Aclear}, \neg \mathit{Bclear}, \mathit{AonB}, \neg \mathit{BonA}, \neg \mathit{AonT}, \mathit{BonT}\}$ and $o = \langle \mathit{Aclear} \wedge \mathit{AonB}, \mathit{Bclear} \wedge \neg \mathit{AonB} \wedge \mathit{AonT} \rangle$.

1. $C_0 \cup \{\mathit{Aclear} \wedge \mathit{AonB}\}$ is satisfiable: o is applicable.
2. The 1-literal clauses $\neg \mathit{Bclear}$, AonB and $\neg \mathit{AonT}$ 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 $s \models C_0$: Aclear, Bclear, $\neg \mathit{AonB}$, $\neg \mathit{BonA}$, AonT, BonT.
5. 2-literal clauses that are weaker than $\neg \mathit{Bclear}$ and now true are $\neg \mathit{Bclear} \vee \mathit{Aclear}$, $\neg \mathit{Bclear} \vee \mathit{Bclear}$, $\neg \mathit{Bclear} \vee \neg \mathit{AonB}$, $\neg \mathit{Bclear} \vee \neg \mathit{BonA}$, $\neg \mathit{Bclear} \vee \mathit{AonT}$, and $\neg \mathit{Bclear} \vee \mathit{BonT}$.

Example (continues...)

6. Similar 2-literal clauses are obtained from *AonB* and from $\neg$*AonT*.
7. By eliminating logically equivalent ones, tautologies, and clauses that follow from those in C_0 not falsified we get
$$C_1 = \{Aclear, \neg BonA, BonT, \\ \neg Bclear \vee \neg AonB, \neg Bclear \vee AonT, \\ AonB \vee Bclear, AonB \vee AonT, \\ \neg AonT \vee Bclear, \neg AonT \vee \neg AonB\}$$
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.

Example

Let $C_i = \{\neg \mathit{AinRome} \vee \neg \mathit{AinParis},$
$\neg \mathit{AinRome} \vee \neg \mathit{AinNYC},$
$\neg \mathit{AinParis} \vee \neg \mathit{AinNYC}\},$

$o = \langle \mathit{AinRome}, \mathit{AinParis} \wedge \neg \mathit{AinRome} \rangle.$

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

Example

Let $C_i = \{\neg \mathit{AinRome} \vee \neg \mathit{AinParis},$
$\neg \mathit{AinRome} \vee \neg \mathit{AinNYC},$
$\neg \mathit{AinParis} \vee \neg \mathit{AinNYC}\},$
$o = \langle \mathit{AinRome}, \mathit{AinParis} \wedge \neg \mathit{AinRome} \rangle.$

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

Example

Let $C_i = \{\neg \mathit{AinRome} \vee \neg \mathit{AinParis},$
$\neg \mathit{AinRome} \vee \neg \mathit{AinNYC},$
$\neg \mathit{AinParis} \vee \neg \mathit{AinNYC}\},$
$o = \langle \mathit{AinRome}, \mathit{AinParis} \wedge \neg \mathit{AinRome} \rangle.$

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

Example

Let $C_i = \{\neg \mathit{AinRome} \vee \neg \mathit{AinParis},$
$\neg \mathit{AinRome} \vee \neg \mathit{AinNYC},$
$\neg \mathit{AinParis} \vee \neg \mathit{AinNYC}\},$
$o = \langle \mathit{AinRome}, \mathit{AinParis} \wedge \neg \mathit{AinRome} \rangle.$

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

Example

Let $C_i = \{\neg \mathit{AinRome} \lor \neg \mathit{AinParis},$
$\neg \mathit{AinRome} \lor \neg \mathit{AinNYC},$
$\neg \mathit{AinParis} \lor \neg \mathit{AinNYC}\},$
$o = \langle \mathit{AinRome}, \mathit{AinParis} \land \neg \mathit{AinRome} \rangle.$

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

Example

Let $C_i = \{\neg \mathit{AinRome} \vee \neg \mathit{AinParis}, \neg \mathit{AinRome} \vee \neg \mathit{AinNYC}, \neg \mathit{AinParis} \vee \neg \mathit{AinNYC}\}$,

$o = \langle \mathit{AinRome}, \mathit{AinParis} \wedge \neg \mathit{AinRome} \rangle$.

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

Computation of invariants: function *preserved*

Test whether a clause remains true when operator is applied

Test if an operator preserves a clause

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

Test if an operator preserves a literal

def preserved-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 $l' \in L'$

Computation of invariants: function *preserved*

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

1. preserved($a \vee b$, C, $\langle \neg c, c \wedge d \rangle$) returns *true*
2. preserved($a \vee b$, C, $\langle \neg c, \neg a \wedge b \rangle$) returns *true*
3. preserved($a \vee b$, C, $\langle b, \neg a \rangle$) returns *true*
4. preserved($a \vee b$, C, $\langle \neg c, \neg a \rangle$) returns *true*
5. preserved($a \vee b$, C, $\langle c, \neg a \rangle$) returns *false*

Computation of invariants: function *preserved*

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

1. preserved($a \vee b$, C, $\langle \neg c, c \wedge d \rangle$) returns *true*
2. preserved($a \vee b$, C, $\langle \neg c, \neg a \wedge b \rangle$) returns *true*
3. preserved($a \vee b$, C, $\langle b, \neg a \rangle$) returns *true*
4. preserved($a \vee b$, C, $\langle \neg c, \neg a \rangle$) returns *true*
5. preserved($a \vee b$, C, $\langle c, \neg a \rangle$) returns *false*

Computation of invariants: function *preserved*

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

1. preserved($a \lor b$, C, $\langle \neg c, c \land d \rangle$) returns *true*
2. preserved($a \lor b$, C, $\langle \neg c, \neg a \land b \rangle$) returns *true*
3. preserved($a \lor b$, C, $\langle b, \neg a \rangle$) returns *true*
4. preserved($a \lor b$, C, $\langle \neg c, \neg a \rangle$) returns *true*
5. preserved($a \lor b$, C, $\langle c, \neg a \rangle$) returns *false*

Computation of invariants: function *preserved*

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

1. preserved($a \vee b$, C, $\langle \neg c, c \wedge d \rangle$) returns *true*
2. preserved($a \vee b$, C, $\langle \neg c, \neg a \wedge b \rangle$) returns *true*
3. preserved($a \vee b$, C, $\langle b, \neg a \rangle$) returns *true*
4. preserved($a \vee b$, C, $\langle \neg c, \neg a \rangle$) returns *true*
5. preserved($a \vee b$, C, $\langle c, \neg a \rangle$) returns *false*

Computation of invariants: function *preserved*

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

1. preserved($a \vee b$, C, $\langle \neg c, c \wedge d \rangle$) returns *true*
2. preserved($a \vee b$, C, $\langle \neg c, \neg a \wedge b \rangle$) returns *true*
3. preserved($a \vee b$, C, $\langle b, \neg a \rangle$) returns *true*
4. preserved($a \vee b$, C, $\langle \neg c, \neg a \rangle$) returns *true*
5. preserved($a \vee b$, C, $\langle c, \neg a \rangle$) returns *false*

Computation of invariants: function *preserved*

Correctness

Lemma

Let C be a set of clauses, $\phi = l_1 \vee \cdots \vee l_n$ a clause, and o an operator.

If preserved(ϕ, C, o) returns true, then app$_o(s) \models \phi$ for every state s such that $s \models C \cup \{\phi\}$ and app$_o(s)$ is defined.

Computation of invariants: function *preserved*

Why is *preserved* incomplete?

Example (incompleteness)

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

preserved($b \vee d \vee e$, $\emptyset$, o) returns false because the preserved-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.

1. $C =$ the set of 1-literal clauses that are true in the initial state.
2. For each operator o and clause $\phi \in C$, 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 $\phi \vee l$ 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 $\mathcal{O}(m^n)$ (where $m = |A|$ is the number of state variables) and hence polynomial.

Computation of invariants: the main procedure

Invariant computation

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 preserved($l_1 \vee \cdots \vee l_m$, C', o) = false:
            $C := C \setminus \{l_1 \vee \cdots \vee l_m\}$
            if $m < n$:
                for each literal l:
                    if $C' \cup \{c\} \models EPC_l(e) \vee (l \wedge \neg EPC_{\bar{l}}(e))$:
                        $C := C \cup \{l_1 \vee \cdots \vee l_m \vee l\}$
        $C := \{ \phi \in C \mid \neg\exists \phi' \in C : \phi' \models \phi \wedge \phi' \not\equiv \phi \}$
    until $C = C'$
    return C

Theorem

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 $o_1, \ldots, o_m \in O$: app$_{o_1;\ldots;o_m}(I) \models C$.

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.

. . . □

Proof continues...

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 preserved(ϕ, C, o) is true for all clauses $\phi \in C$ and operators $o \in O$.
- By the lemma, this means that $app_o(s) \models \phi$ for every state s such that $s \models C$ and $app_o(s)$ is defined.
- Since this is true for all clauses $\phi \in C$, 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?

1. Practical implementations of the algorithm use polynomial time approximations of the tests for satisfiability and $\models$.
2. The function *preserved* is incomplete for operators in general (but complete for STRIPS operators.) Making it complete makes it NP-hard.
3. 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.

Why is the strongest invariant not always found?

1. Practical implementations of the algorithm use polynomial time approximations of the tests for satisfiability and $\models$.
2. The function *preserved* is incomplete for operators in general (but complete for STRIPS operators.) Making it complete makes it NP-hard.
3. 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.

Why is the strongest invariant not always found?

1. Practical implementations of the algorithm use polynomial time approximations of the tests for satisfiability and $\models$.
2. The function *preserved* is incomplete for operators in general (but complete for STRIPS operators.) Making it complete makes it NP-hard.
3. 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.

Why is the strongest invariant not always found?

1. Practical implementations of the algorithm use polynomial time approximations of the tests for satisfiability and $\models$.
2. The function *preserved* is incomplete for operators in general (but complete for STRIPS operators.) Making it complete makes it NP-hard.
3. 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.

Computation of invariants
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:

$$
\begin{aligned}
C_0 &= \{a, \neg b, \neg c\} \\
C_1 &= \{\neg c, a \vee b, \neg b \vee \neg a\} \\
C_2 &= \{\neg b \vee \neg a, \neg c \vee \neg a, \neg c \vee \neg b\} \\
C_3 &= \{\neg b \vee \neg a, \neg c \vee \neg a, \neg c \vee \neg b\} \\
C_j &= C_2 \text{ for all } j \geq 2
\end{aligned}
$$

Invariants in satisfiability planning

Invariants in satisfiability planning

For every invariant $l_1 \vee \cdots \vee l_n$, add the clauses

$$l_1^t \vee \cdots \vee l_n^t$$

for all time points t.

Notice that the above formulae are logical consequences of Φ_i^{seq} and Φ_i^{par}, so the invariants do not change the set of valuations of these formulae.

Invariants are critical for the efficiency of satisfiability planning on many types of problems.

Invariants in backward search
Motivating example

Example

Regression of in(A, Freiburg) by
⟨in(A, Strassburg), ¬in(A, Strassburg) ∧ in(A, Paris)⟩
gives in(A, Freiburg) ∧ in(A, Strassburg)
No state satisfying in(A, Freiburg) ∧ in(A,Strassburg) makes sense if A denotes some usual physical object.

Invariants in backward search

Motivating example

Problem: Regression produces sets T of states such that

1. some states in T are not reachable from I, or
2. none of the states in T are reachable 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 $regr_o(\psi) \wedge \phi$ is satisfiable.

Invariants in backward search

Motivating example

Problem: Regression produces sets T of states such that

1. some states in T are not reachable from I, or
2. none of the states in T are reachable 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 $\mathit{regr}_o(\psi) \land \phi$ is satisfiable.

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 $\neg AonB \vee \neg AonC$ states that $AonB$ and $AonC$ are mutex.

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

Example

In blocks world, *BonA*, *ConA*, *DonA* and *Aclear* are mutex.

Invariants for problem reformulation
Multi-valued state variables

If a group of n literals $G = \{l_1, \dots, l_n\}$ over N different variables $A_G = \{a_1, \dots, a_n\}$ are mutually exclusive, then the planning task can be rephrased using a single multi-valued (i.e., non-binary) state variable v_G with $n + 1$ possible values in place of the n variables in A_G:

- n of the possible values represent situations in which exactly one of the literals in G is true.
- The remaining value represents situations in which none of the literals in G is true.

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

Summary

- Invariants are needed for making 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.

Summary

- Invariants are needed for making 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.

Summary

- Invariants are needed for making 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.

Summary

- Invariants are needed for making 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.

Summary

- Invariants are needed for making 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.

Summary

- Invariants are needed for making 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.