Principles of AI Planning

9. Interlude: Finite-domain representation

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel and Robert Mattmüller

November 27th, 2017

Invariants

Invariants

- When we as humans reason about planning tasks, we implicitly make use of "obvious" properties of these tasks.
  - Example: we are never in two places at the same time
- We can express this as a logical formula φ that is true in all reachable states.
  - Example: $\varphi = \neg(\textit{at-uni} \land \textit{at-home})$
- Such formulae are called invariants of the task.

Computing invariants

How does an automated planner come up with invariants?

- Theoretically, testing if an arbitrary formula φ is an invariant is as hard as planning itself.
- Still, many practical invariant synthesis algorithms exist.
- To remain efficient (= polynomial-time), these algorithms only compute a subset of all useful invariants.
- Empirically, they tend to at least find the "obvious" invariants of a planning task.

Invariant synthesis algorithms

Most algorithms for generating invariants are based on a generate-test-repair paradigm:

- Generate: Suggest some invariant candidates, e. g., by enumerating all possible formulas φ of a certain size.
- Test: Try to prove that φ is indeed an invariant. Usually done inductively:
  1. Test that initial state satisfies φ.
  2. Test that if φ is true in the current state, it remains true after applying a single operator.
- Repair: If invariant test fails, replace candidate φ by a weaker formula, ideally exploiting why the proof failed.

Invariant synthesis: references

We discussed invariant synthesis in detail in previous courses on AI planning, but this year we will focus on other aspects of planning.

Literature on invariant synthesis:

- DISCOPLAN (Gerevini & Schubert, 1998)
- TIM (Fox & Long, 1998)
- Edelkamp & Helmert's algorithm (1999)
- Rintanen's algorithm (2000)
- Bonet & Geffner's algorithm (2001)
- Helmert's algorithm (2009)

Exploiting invariants

Invariants have many uses in planning:

- Regression search:
  Prune states that violate (are inconsistent with) invariants.
- Planning as satisfiability:
  Add invariants to a SAT encoding of a planning task to get tighter constraints.
- Reformulation:
  Derive a more compact state space representation (i. e., with lower percentage of unreachable states).

We now briefly discuss the last point, since it leads to planning tasks in finite-domain representation, which are very important for the next chapters.

Planning tasks in finite-domain representation

Mutexes

Invariants that take the form of binary clauses are called mutexes because they state that certain variable assignments cannot be simultaneously true and are hence mutually exclusive.

Example (Blocksworld)

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 (Blocksworld)

Every pair in $\{B\text{-}on\text{-}A, C\text{-}on\text{-}A, D\text{-}on\text{-}A, A\text{-}clear\}$ is 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 $\mathscr{D}_v$ for the domain of v.

Example

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

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 $s : V \to \bigcup_{v \in V} \mathscr{D}_v$ such that $s(v) \in \mathscr{D}_v$ for all $v \in V$.

Example

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

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 $d \in \mathscr{D}_v$.

Example

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

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 $d \in \mathscr{D}_v$.

Example

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

$\leadsto$ definition of finite-domain operators follows from this

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 $\Pi = \langle V, I, O, \gamma \rangle$ 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
- γ is a formula over V describing the goal states.

Relationship to propositional planning tasks

Definition (induced propositional planning task)

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

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

- $\leadsto$ 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 $\Pi = \langle V, I, O, \gamma \rangle$ is called an SAS$^+$ planning task iff there are no conditional effects in O and all operator preconditions in O and the goal formula γ 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$^+$

Summary

- Invariants are common properties of all reachable states, expressed as logical formulas.
- A number of algorithms for computing invariants exist.
- These algorithms will not find all useful invariants (which is too hard), but try to find some useful subset within reasonable (polynomial) time.
- Mutexes are invariants that express that certain pairs of state variable assignments are mutually exclusive.
- Groups of mutexes can be used for problem reformulation, transforming a planning task into finite-domain representation (FDR).
- Many planning algorithms are more efficient when working on these FDR tasks (rather than the original tasks) because they contain fewer unreachable states.