L15: Planning with factored representations

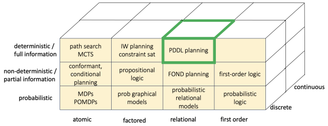
AIMA4e: Chapter 11.1–11.3; 11.5

What you should know after this lecture

- STRIPS planning representation takes advantage of relations, factoring, locality, and sparsity to make transition model compact
- STRIPS models enable powerful domain-independent heuristics
- We can model partial information, and other extensions, in this formalism

Factored states and information

Factored, discrete states
Compact, sparse representation of T
Construct heuristics via relaxation



Making plans in complex domains

- We have seen how to frame planning for an agent as searching for a path through a state space.
- We have also seen how to describe states using a factored representation in terms of variables and values
- Can we combine them? Yes, with the following advantages:
  - Factoring state representations lets us compactly describe the goals and transition model
  - Factored structure enables a lot of relaxations that lead to powerful domain independent heuristics

"Classical planning" framework

- Make some structural assumptions about the domain
  - sparsity of effect: any action taken by an agent doesn't change many aspects of the environment state
  - locality of dependence: what effects an action will have depend only on a few aspects of the environment state
- Leads us to a special-purpose (but still domain independent) representation language for describing $\mathcal{S}$, $\mathcal{A}$, T, and G that
  - Is highly compact (and therefore learnable from few samples)
  - Can be used to plan efficiently
- Language is called STRIPS; standardized syntax and variations in PDDL (planning domain description language)

Planning domain description language

For now we are following syntax from AIMA—we'll show later what the "real" syntax is like.

Domain specification

- predicates: symbols, like *On* or *Airport*
- object variables: symbols, like x
- fluents: atoms, like $On(x, y)$
  // These are the factors of our state representation
- operators: schematic, factored, description of T, like

  $Unload(obj, plane, loc)$

  - preconditions: $Aboard(obj, plane), At(plane, loc)$
  - effect: $At(obj, loc), \neg Aboard(obj, plane)$

Planning domain description language

A ground fluent is a predicate applied to a tuple of constant symbols.

Problem specification

- constants: symbols, like *blockA* or *747_e35b2*
- initial state: set of ground fluents that are true in the initial state; assume all other ground fluents are false.
  (This is often called the closed world assumption.)
- goal: conjunction (set) of ground fluents

Path-search problem given PDDL domain and problem

Mapping this back into the representation we used for path search problems

- $\mathcal{S}$:
  - Plug all combinations of constants into all predicates to get all ground fluents, like $\mathit{Aboard}(\mathit{blockA}, \mathit{747\_e35b2})$
  - A state is an assignment of **True** or **False** to each ground fluent.
  - It is often most efficient to represent a state as the set of ground fluents that have the value **True**.
- $\mathcal{A}$: Plug all combinations of constants into all operators to get all ground operators. These are the possible actions.
- $\mathsf{G} \subset \mathcal{S}$: All states in which all ground fluents in the goal are assigned to **True**
- s_0: The initial state, set of ground fluents that are true initially

State transition function

Define $\mathsf{T}(s, a)$ where

- s : set of true ground fluents
- a : ground operator instance

as follows:

- If $\mathit{preconditions}(a) \subseteq s$ then

$$\mathsf{T}(s, a) = s - \mathit{del}(a) \cup \mathit{add}(a)$$

  where $\mathit{add}(a)$ are positive fluents in $\mathit{effects}(a)$ and $\mathit{del}(a)$ are negated fluents in $\mathit{effects}(a)$
- Otherwise, the operator a is not applicable in state s, and we can think of it as having no effect, so

$$\mathsf{T}(s, a) = s$$

Planning algorithms

Given a domain and problem description, how do we find a plan?

- Forward best-first search with heuristics that take advantage of the structured representation
- Regression (or backward chaining), works backwards from the goal, states in the search space are actually sets of fluents representing sub-goals (not environment states)
- Reduction to propositional satisfiability.

Why is this formalism useful?

- The domain description is independent of the particular universe of objects (constants)
- Similar in some ways to a graph neural network (you can think of nodes for fluents in the problem instance; the operator description specifies connectivity (which other fluents the new value of a fluent depends on) and parameters (what those fluent values actually are.)
- Generalizes broadly
- Takes advantage of sparsity
  - The effects of most actions don't depend on most factor values
  - Relatively few factors are affected by any action
- Provides leverage for defining effective domain-independent heuristics

Delete relaxation

- The thing that makes planning difficult is interference among the operators—executing an action might potentially undo some effect that you had already achieved or wanted to maintain from the initial state.
- A relaxation of the planning problem is to assume that this never happens, by ignoring the delete effects of an operator, so that our update is:

$$s' = s \cup add(a)$$

- In this relaxation, a fluent never become false once it becomes true! So, e.g. a robot can be in multiple locations. Weird, but convenient.
- An even more relaxed relaxation: allow all actions whose preconditions are satisfied to be executed in parallel!

The relaxed planning graph (RPG) is computed by computing a sequence of relaxed, parallel state updates.

Example relaxed plan graph

Left
- P = In(Robot, R_2)
- D = In(Robot, R_2)
- A = In(Robot, R_1)

Suck(R_1)
- P = In(Robot, R_1)
- D = ∅ [empty set]
- A = Clean(R_1)

Right
- P = In(Robot, R_1)
- D = In(Robot, R_1)
- A = In(Robot, R_2)

Suck(R_2)
- P = In(Robot, R_2)
- D = ∅ [empty set]
- A = Clean(R_2)

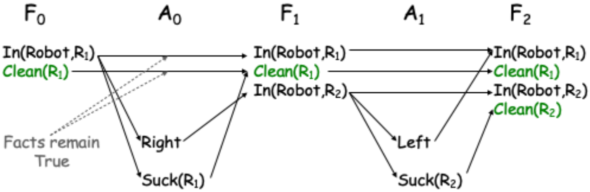


Image source: J.C.Latombe, Stanford

Compute relaxed planning graph

COMPUTE-RPG($s_0, \mathcal{A}, G$)

// s_0 and G are both sets of ground fluents
// $\mathcal{A}$ is a set of ground operator descriptions
$F_0 = s_0; t = 0$
while $G \nsubseteq F_t$
    $A_t = \{a \in \mathcal{A} \mid \mathit{pre}(a) \subseteq F_t\}$ // Do all applicable actions!
    $F_{t+1} = F_t \cup \bigcup_{a \in A_t} \mathit{add}(a)$ // Add all add effects!
    if $F_{t+1} = F_t$: **return** // Goal is infeasible ☹
    $t = t + 1$
return $F_0, \ldots, F_t, A_0, \ldots, A_{t-1}$,

Heuristics based on RPG

- Add up the levels at which each goal fluent appear: not admissible

$$H_{\text{add}}(s, G) = \sum_{f \in G} \operatorname*{argmin}_{t} f \in F_t$$

- Max of the levels at which each goal fluent appear: admissible but weak

$$H_{\max}(s, G) = \max_{f \in G} \operatorname*{argmin}_{t} f \in F_t$$

- Optimal solution to the delete-relaxation problem (without parallel actions): still NP-hard!
- H_{ff}: Approximate solution to the delete-relaxation problem, searching backward in the RPG for a relaxed plan

Computing H_{ff}

$H_{ff}(s, G, \mathit{RPG})$

$M = \max_{f \in G} \mathit{RPG.level}(f);\ \mathtt{plan} = \{\}$

for $t \in 0 \dots M$:   // Fluents we need to make true at each level

    $G_t = \{f \in G \mid \mathit{RPG.level}(f) = t\}$

for $t = M \dots 1$:

    for $f \in G_t$:   // Find any applicable a with result f

        $a = \{a \mid \mathit{RPG.level}(a) = t - 1, f \in \mathit{add}(a)\}[0]$

        $\mathtt{plan} = \mathtt{plan} \cup \{a\}$ // Add action a to plan

        for $p \in \mathit{pre}(a)$

            $G_{\mathit{RPG.level}(p)} = G_{\mathit{RPG.level}(p)} \cup \{p\}$

return $|\mathtt{plan}|$

$\mathit{RPG.level}(f) = \min_t f \in F_t$

$\mathit{RPG.level}(a) = \min_t a \in A_t$

Extensions

There are lots of extensions to classical planning!

- Conformant planning: have, for each predicate `P`, `BP` and `BNotP`.
- Temporal planning: discrete time steps, actions take time
- Cost-sensitive planning: add action costs
- Conditional planning: add <u>observe</u> actions

Actual PDDL syntax example

LISP and prefix syntax used to be a thing!

```
(:action unload
  :parameters (?obj ?plane ?loc)
  :precondition (and
     (package ?obj)
     (plane ?plane)
     (location ?loc)
     (at ?plane ?loc)
     (aboard ?obj ?plane))
  :effect (and
     (not (aboard ?obj ?plane))
     (at ?obj ?loc)))
```

Next time

- Action planning in continuous state and action spaces