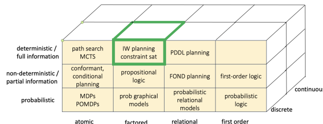
L04: Constraint Satisfaction and Factored Planning

AIMA4e: Chapter 6; Chapter 11.1–11.3

What you should know after this lecture

- Advanced CSP solution strategies:
  - Learning within a problem
  - Local search
- Basic idea of factored planning

Factored states and information



Factored, discrete states
Factored "observations" as constraints

Inference about state based on certain observations

- State space is <u>factored</u> into a set of state variables
- Observations are <u>constraints</u> on (pieces of information about) the values of those state variables
- Our objective is to figure out one or more possible states that are consistent with the observations.

Inference doesn't increase our information about the underlying state—just processes it into a more useful form

Constraint-satisfaction problem: formal definition

- $\mathcal{X}$ is a set of variables $\{X_1, \ldots, X_n\}$
- $\mathcal{D}$ is a set of domains $\{D_1, \ldots, D_n\}$, where $D_i = \{x_1, \ldots, x_k\}$ is the set of possible values of X_i
- $\mathcal{C}$ is a set of constraints:
  - *scope* : a tuple of variables
  - *relation* : a relation specifying tuples of values that this tuple of variables can legally take on

Define:

- *assignment* : mapping from variables to values
- *partial assignment*: only provides values for some variables
- *consistent assignment* : partial assignment that doesn't violate any constraints
- *solution* : complete assignment that doesn't violate any constraints

Forward checking

```
FC(X, a):
    for X_i ∈ UNASSIGNED-VAR(a) ∩ NEIGHBORS(X)
        REVISE(X_i, X)
        if D_i = { }: return 'failed'

BACKTRACK-FC(a)
    if COMPLETE(a): return a
    X = UNASSIGNED-VAR(a)
    for x ∈ DOMAIN-VALUES(X):
        if CONSISTENT(a, {X = x}):
            EXTEND(a, {X = x}); FC(X, a)
            r = BACKTRACK(a)
            if r ≠ 'failed': return r
            REMOVE(a, {X = x}); UNDO-FC(X, a)
    return 'failed'
```

Arc consistency

Can work harder to be sure that all arcs are consistent.

- See AC-3 alg in book.
- Roughly, keep doing REVISE until no domains change further.
- Completely solves some problems.
- BT-AC3 often more expensive than BT-FC.
- Can extend the idea to making k-tuples (for $k > 2$) of variables consistent.

Backjumping

Sometimes we have made a poor initial choice, but end up with endlessly considering assignments to irrelevant variables.

- Whenever a dead-end occurs at variable X, backtrack to the "most recent" variable that is connected to X in the constraint graph.
- Can be very helpful!

Requires careful bookkeeping to be sure all the right assignments and inferences are undone.

AIMA4e asserts that any assignment that is pruned by backjumping will also be pruned by forward-checking. Prove it to yourself!

Learning while searching

Idea: find assignments that are no good: not simply inconsistent themselves, but such that there is no possible way to assign the rest of the variables.

- conflict set for a variable X: Set of variables X' and values x' such that there is no assignment to X consistent with $X' = x'$. It's minimal if no subset of it is a conflict set.
- Once you discover a conflict set, don't ever try it again!
- Add a constraint that forbids this assignment and keep going. (But note that it's non-binary).

Identifying and recording only conflict sets which are known to be minimal constitutes deep learning. – Dechter, AIJ, 1990

Local search: a very different strategy!

- Start with a complete assignment, with constraint viols
- Until you reach a satisfying assignment: pick a variable and assign a new value.

Guidance helps! **Min-conflicts heuristic**:

- Randomly choose a variable that is in conflict (violating some constraint)
- Assign it the value that will minimize the total number of constraints violated.

Simulated annealing:

- Propose a move (variable and value) at random.
- If it reduces the number of conflicts, accept it.
- If it does not, accept anyway, with probability $e^{-\Delta/T}$ where Δ is number of conflicts added and T is a temperature parameter that is decreased over time.

Min conflict not guaranteed to find solution; simulated annealing is (eventually)

Message passing

When your constraint (hyper)graph is a tree (has no loops) then there's a super-cool algorithm!

- Pick any node to be root
- Construct a topological sort: every node is in the list after its parent.
- Starting at the *end* of the list, do, for each X
  $\textsc{revise}(parent(X), X)$
- Each X is left with a domain such that any value in the remaining domain is consistent with the whole subtree beneath it.
- After this $O(n)$ processing, select any value at root, and work forward selecting any consistent value. No backtracking needed.

But! What if you don't have a tree?

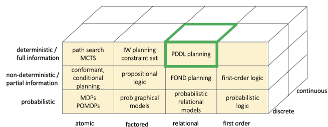
Two strategies:

- Make a tree by combining some variables into super-variables with the product of their domains.
- Find a cutset: a set of variables, such that if they were removed, the remaining (hyper)graph would be a tree.
  - Do backtracking on values of the variables in the cutset
  - Given an assignment to those variables, do message-passing to try to find assignment to the rest.

We will see these algorithms again in probabilistic inference!

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 $Aboard(blockA, 747_e35b2)$
  - 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.
- $G \subset \mathcal{S}$: 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 $T(s, a)$ where

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

as follows:

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

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

  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

  $$T(s, a) = s$$

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

Next time

- Powerful domain-dependent heuristics
- Actual PDDL syntax
- If time, Iterative Width planning