Classical Planning

Chapter 10

Outline

- ♢ Search vs. planning
- ♢ PDDL operators
- ♢ Planning algorithms
- ♢ Situation calculus

Search vs. planning

Planning: Devising a sequence of actions to achieve one's goal.

We have seen two planning agents so far:

♢ Search-based problem solving agent
- can find sequences of actions that result in a goal state.
- but deals with atomic states (needs good domain-specific heuristics)

♢ hybrid logical agents: logical condition-action rule + searching
- can find plans without domain-specific heuristics
  (uses domain-independent heuristics based on the
  logical structure of the problem)
- but relies on ground (variable-free) propositional inference
  (it may be over worked when there are many actions and states.)

We want representation for planning problems
that scales up to problems unable to be handled by earlier approaches.

Classical planning environment

The assumptions for classical planning problems

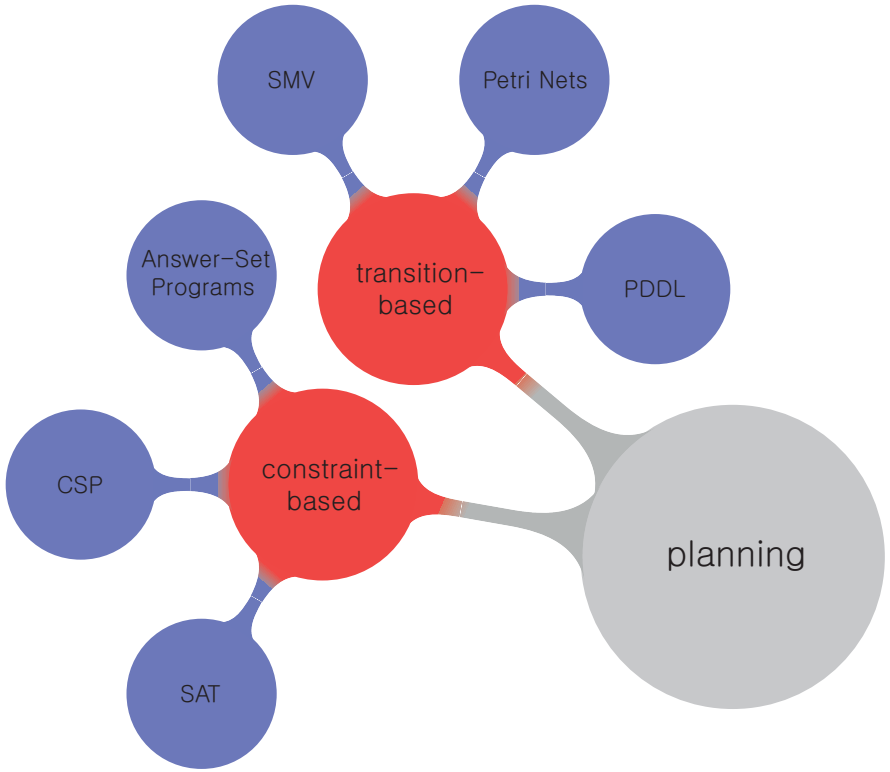
- ♢ Fully observable
  we see everything that matters
- ♢ Deterministic
  the effects of actions are known exactly
- ♢ Static
  no changes to environment other than those caused by agent actions
- ♢ Discrete
  changes in time and space occur in quantum amounts
- ♢ Single agent
  no competition or cooperation to account for

How to represent Planning Problem

What is a good representation?
– Expressive enough to describe a wide variety of problems
– Restrictive enough for efficient algorithms to operate on it
– Planning algorithm should be able to take advantage
  of the *logical structure* of the problem

Historical AI planning languages
– STRIPS was used in classical planners
  Stanford Research Institute Problem Solver
– ADL addresses expressive limitations of STRIPS
  Action Description Language
  Adds features not in STRIPS
    negative literals, quantified variables, conditional effects, equality
We'll look at a simpler version of PDDL (currently used widely)

How to represent Planning Problem



Different strengths and advantages; No single “right” language.

* figure adopted from [J. Rintane, "Algorithms for Classical Planning." IJCAI/AAAI 2011]

PDDL

PDDL = Planning Domain Definition Language

$\leftarrow$ standard encoding language for "classical" planning tasks

Components of a PDDL planning task:

$\diamondsuit$ Objects: Things in the world that interest us. $\diamondsuit$ Predicates: Properties of objects that we are interested in;

can be true or false. $\diamondsuit$ Initial state: The state of the world that we start in. $\diamondsuit$ Goal specification: Things that we want to be true. $\diamondsuit$ Actions/Operators: Ways of changing the state of the world.

PDDL can also be described as what we need for search algorithm:

1) initial state
2) actions that are available in a state
3) result of applying an action
4) and goal test

PDDL operators: States

States: ♢ Factored representation used:
Each state is represented as a collection of variables

♢ Represented as a conjunction of fluent (ground, functionless atoms).

♢ Database semetics is used:
– Closed-world assumption: Fluent that are not mentioned are false.
– Unique names assumption: States named differently are distinct.

Following are not allowed:
$At(x, y)$ (non-ground), $\neg Poor$ (negation),
and $At(Father(Fred), Sydney)$ (use function symbol)

PDDL operators: States

PDDL state representation allows alternative algorithms
    it can be manipulated either by logical inference techniques or by
    set operations (sets may be easier to deal with)

* Fluents: Predicates and functions whose values vary from by time (situation).
Ground term: a term with no variables.
Literal: positive or negative atomic sentence.

Initial state is conjunction of ground atoms.

Goal is also a conjunction ($\wedge$) of literal (positive or negative) that may contain variables.

The problem is solved when we can find a sequence of actions that end in a state s that entails the goal.
Ex. state $Rich \wedge Famous \wedge Miserable$ entails the goal $Rich \wedge Famous$,
Ex. state $Plane(Plane_1) \wedge At(Plane_1, SFO)$ entails the goal $At(p, SFO) \wedge$

$Plane(p)$.

PDDL operators: Action schema

Actions: described by a set of action schemas

Implicitly define the $ACTIONS(s)$ and $RESULT(s, a)$.

Classical planning concentrates on problems where most actions leave most things unchanged.

PDDL specify the result of an action in terms of what changes;
everything that stays the same is left unmentioned.

Action schema: composes of

1) Action name,
2) List of all the variable used in the schema,
3) A Precondition
   defines the states in which the action can be executed
   conjunctions of literals
4) An Effect
   defines the result of executing the action
   conjunctions of literals

PDDL operators: Action schema

Action schema for flying a plane from one location to another:

$Action(Fly(p, from, to)$,

PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airprot(to)$

EFFECT: $\neg At(p, from) \wedge At(p, to)$

Value subtituted for the variables:

$Action(Fly(P_1, SFO, JFK)$,

PRECOND: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airprot(JFK)$

EFFECT: $\neg At(P_1, SFO) \wedge At(p, JFK)$

PDDL operators: Action schema

Action a is applicable in state s if the preconditions are staisfied by s.

i.e., s entails the precondition of a

i.e., $s \models q$ iff every positive literal in q is in s and every negated literal in q is not.

$$a \in Actions(s) \Leftrightarrow s \models Precond(a)$$

When an action schema a contains variables, it may have multiple applicable instantiations.

– If an action a has v variables, then, in a domain with k unique names of objects, it takes $O(v^k)$ time in the worst case to find the applicable ground actions.

– Impractical when v and k are large

PDDL operators: Result

Result of executing action a in state s is defined as a state s'

Result s' is represented by the set of fluents formed by
1. Starting with s,
2. Remove the fluents that are negative literals in the action's effects.
   What we call the delete list or DEL(a) 3. Add the fluents that are positive literals in the actions effects
   What we call the add list or ADD(a)

$Result(s, a) = (s - Del(a)) \cup Add(a)$
where $Del(a)$ is the list of literals which appear negatively in the effect of a, and $Add(a)$ is the list of positive literals in the effect of a.

Ex. in $Fly(P_1, SFO, JFK)$ we remove $At(P_1, SFO)$ and add $At(P_1, JFK)$.

*In PDDL the times and states are implicit in the action schemas:
– The precondition always refers to time t and the effect to time $t + 1$.

PDDL: Cargo transportation planning problem

Init (At (C_1, SFO) $\wedge$ At (C_2, JFK) $\wedge$ At (P_1, SFO) $\wedge$ At (P_2, JFK)
$\wedge$Cargo(C_1) $\wedge$ Cargo(C_2) $\wedge$ Plane(P_1) $\wedge$ Plane(P_2)
$\wedge$Airport (JFK) $\wedge$ Airport (SFO))
Goal(At (C_1, JFK) $\wedge$ At (C_2, SFO))
Action(Load(c, p, a),
PRECOND: At (c, a) $\wedge$ At (p, a) $\wedge$ Cargo(c) $\wedge$ Plane(p) $\wedge$ Airport (a)
EFFECT: $\neg$ At (c, a) $\wedge$ In (c, p))
Action(Unload(c, p, a),
PRECOND: In (c, p) $\wedge$ At (p, a) $\wedge$ Cargo(c) $\wedge$ Plane(p) $\wedge$ Airport (a)
EFFECT: At (c, a) $\wedge$ $\neg$In (c, p))
Action(Fly (p, from, to),
PRECOND: At (p, from) $\wedge$ Plane(p) $\wedge$ Airport (from) $\wedge$ Airport (to)
EFFECT: $\neg$ At (p, from) $\wedge$ At (p, to))

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

At means "available for use at a given location."

In problem of spurious actions such as Fly(P1,JFK,JFK),

It is common to ignore such problems, because they seldom cause incorrect plans to be produced.

An example domain: the blocks world

The domain consists of:

1. A table, a set of cubic blocks and a robot arm;
2. Each block is either on the table or stacked on top of another block
3. The arm can pick up a block and move it to another position
   either on the table or on top of another block;
4. The arm can only pick up one block at time, so it cannot pick up a block
   which has another block on top.

A goal is a request to build one or more stacks of blocks specifed in terms of what blocks are on top of what other blocks

State descriptions

Blocks are represented by constants $A, B, C, \ldots etc.$ and an additional constant $Table$ representing the table.

The following predicates are used to describe states:

$On(b, x)$ block b is on x, where x is either another block or the table

$Clear(x)$ there is a clear space on x to hold a block

Planning algorithms

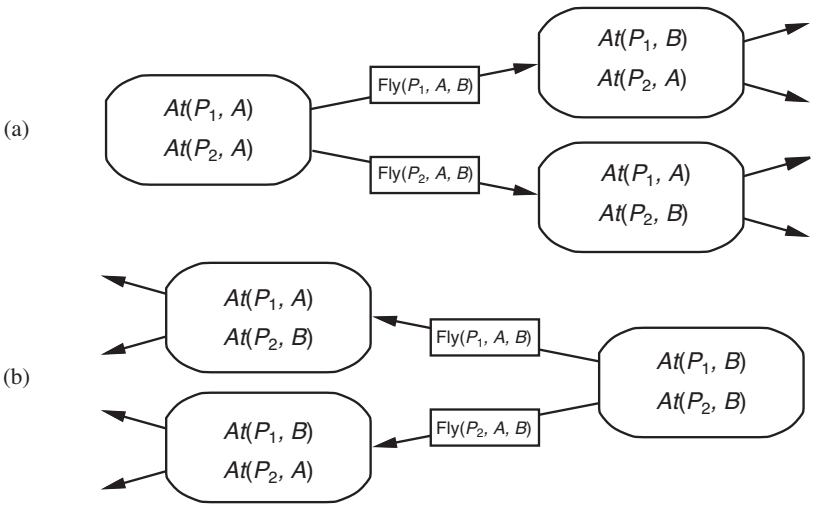
Planning problem = planning domain + initial state + goal

So far, in the search lectures, we only looked at forward search from the initial state to a goal state

One of the nice advantages of the declarative representation of action schema is that we can also search backward from the goal, looking for the initial stat.

(a) Forward (progression) state-space search

(b) Backward (regression) relevant-state search



Planning algorithms: Forward state-space search

Forward (progression) state-space search:

- ♢ Search through the space of states, starting in the initial state and using the problem's actios to search forward for a member of the set of goal states.
- ♢ Prone to explore irrelevant actions
- ♢ Planning problems often have large state space.
- ♢ The state-space is concrete,
  i.e. there is no unassigned variables in the states.

Planning algorithms: Backward state-space search

Backward (regression) relevant-state search:

♢ Search through set of relevant states,
staring at the set of states representing the goal and
using the inverse of actions to search backward for the init. state.

♢ Only considers actions that are relevant to the goal (or current state).

♢ Works only when we know how to regress from a state
description to the predecessor state description.

Ex. n-queens problem: not easy to describe states that are one move away from goal.

♢ Need to deal with partially instantiated actions and
state, not just ground ones.
(Some variable may not be assigned values.)

♢ Keeps the branching factor lower than forward search, for most problem domains

However, uses state sets rather than individual states - good heuristics hard to get.

Reason majority of current systems favor forward search.

Planning algorithms: Backward state-space search

The PDDL representation was designed to make it easy to regress actions
- if a domain can be expressed in PDDL, then we can do regression search

Given a ground goal description g and a ground action a, the regression from g over a gives us a state description $g!$ defined by

$$g! = (gADD(a)) \cup Precond(a)$$

* Note that DEL(a) does not appear in the formula
while we know the fluents in $DEL(a)$ are no longer true after the action, we don't know whether or not they were true before (do nothing)

Planning algorithms: Backward state-space search

Deciding which actions are candidates to regress over:

– In the forward direction we chose actions that were $applicable$

+ those actions that could be the next step in the plan.

– In backward search we want actions that are $relevant$

+ those actions that could be the $last$ step in a plan leading up to the current goal state.

+ at least one of the actions effects (+ or -) must unify with an element of the goal.

+ action must not have any effect (+ or -) that negates an element of the goal.

PDDL: Block-world problem

Start State Goal State

Initial state:

$$On(A, Table) \wedge On(B, Table) \wedge On(C, A)$$

Goal:

$$On(A, B) \wedge On(B, C)$$

PDDL: Block-world problem

Init (On(A, Table) $\wedge$ On(B, Table) $\wedge$ On(C, A)
$\wedge$ Block(A) $\wedge$ Block(B) $\wedge$ Block(C) $\wedge$ Clear(B) $\wedge$ Clear(C))
Goal(On(A, B) $\wedge$ On(B, C))
Action(Move(b, x, y),
PRECOND: On(b, x) $\wedge$ Clear(b) $\wedge$ Clear(y) $\wedge$ Block(b) $\wedge$ Block(y) $\wedge$
$(b \neq x) \wedge (b \neq y) \wedge (x \neq y)$,
EFFECT: On(b, y) $\wedge$ Clear(x) $\wedge$ $\neg$On(b, x) $\wedge$ $\neg$Clear(y))
Action(MoveToTable(b, x),
PRECOND: On(b, x) $\wedge$ Clear(b) $\wedge$ Block(b) $\wedge$ $(b \neq x)$,
EFFECT: On(b,Table) $\wedge$ Clear(x) $\wedge$ $\neg$On(b, x))

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence [MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)].

Goal stack planning

We work backwards from the goal,
looking for an operator which has one or more of the goal literals
as one of its effects and then trying to satisfy the
preconditions of the operator.

The preconditions of the operator become subgoals that must be satisfed.

We keep doing this until we reach the initial state.

Goal stack planning uses a stack to hold goals
and actions to satisfy the goals,
and a knowledge base to hold the current state, action schemas and domain axioms

Goal stack is like a node in a search tree;
if there is a choice of action, we create branches

Goal stack planning pseudocode

Push the original goal on the stack.

Repeat until the stack is empty:
- If stack top is a compound goal, push its unsatisfed subgoals on the stack.
- If stack top is a single unsatisfed goal,
  + replace it by an action that makes it satisfed and
  + push the action's precondition on the stack.
- If stack top is an action,
  + pop it from the stack,
  + execute it and change the knowledge base by the action's effects.
- If stack top is a satisfed goal, pop it from the stack.

Heuristics for planning

An admissible heuristic can be derived by defining a relaxed problem that is easier to solve.

1. Heuristics that add edges to the graph.

♢ Ignore precondition heuristic:
Drop (all) preconditions from actions (adds edges to graphs).

from

$Action(Fly(p, from, to)$,
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to)$

to

$Action(Fly(p, from, to)$,
PRECOND:
EFFECT: $\neg At(p, from) \wedge At(p, to)$

Still expensive (NP-hard)

It is also possible to ignore only selected preconditions of actions.

Heuristics for planning cont.

♢ Ignore delete list heuristics:

Removing all negative literals from effects (makes monotonic progress towards the goal).

from

$Action(Fly(p, from, to)$,
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to)$

to

$Action(Fly(p, from, to)$,
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\wedge At(p, to)$

The relaxed problems are still expensive (NP-hard).

Approximate solution can be found in polynomial time by hill-climbing

Heuristics for planning cont.

2. Heuristics that decrease the number of states by forming a state abstraction

a many-to-one mapping from states in the ground representation of the problem to the abstract representation.

♢ Ignore some fluents

Ex. Drop all the At fluents except for the ones involving one plane and one package at each of the 5 airports.