Course on Automated Planning: Intro to Planning

Hector Geffner
ICREA & Universitat Pompeu Fabra
Barcelona, Spain

Planning: Motivation

How to develop systems or 'agents'
that can make decisions on their own?

Wumpus World PEAS description

Performance measure

gold +1000, death -1000

-1 per step, -10 for using the arrow

Environment

Squares adjacent to wumpus are smelly

Squares adjacent to pit are breezy

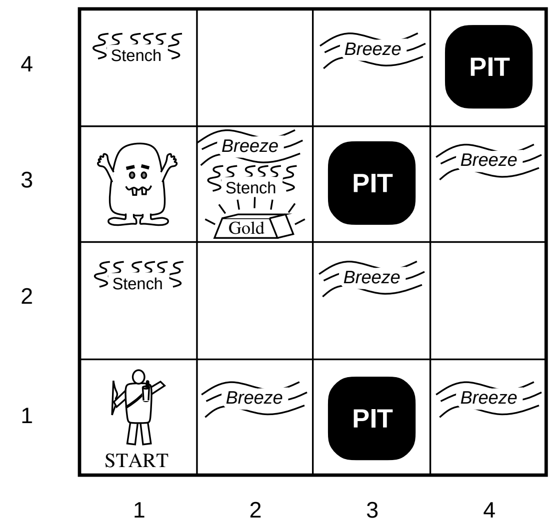
Glitter iff gold is in the same square

Shooting kills wumpus if you are facing it

Shooting uses up the only arrow

Grabbing picks up gold if in same square

Releasing drops the gold in same square



Actuators Left turn, Right turn,
Forward, Grab, Release, Shoot

Sensors Breeze, Glitter, Smell

Autonomous Behavior in AI: The Control Problem

The key problem is to select **the action to do next**. This is the so-called **control problem**. Three approaches to this problem:

- **Programming-based:** Specify control by hand
- **Learning-based:** Learn control from experience
- **Model-based:** Specify problem by hand, derive control automatically

Approaches not orthogonal though; and successes and limitations in each . . .

Settings where greater autonomy required

- **Robotics**
- **Video-Games**
- **Web Service Composition**
- **Aerospace**
- **Manufacturing**
- $\vdots$

Solution 1: Programming-based Approach

Control specified by programmer; e.g.,

- *don't move into a cell if not known to be safe (no Wumpus or Pit)*
- *sense presence of Wumpus or Pits nearby if this is not known*
- *pick up gold if presence of gold detected in cell*
- ...

Advantage: domain-knowledge easy to express

Disadvantage: cannot deal with situations not anticipated by programmer

Solution 2: Learning-based Approach

- **Unsupervised** (Reinforcement Learning):
  - ▷ penalize agent each time that it 'dies' from Wumpus or Pit
  - ▷ reward agent each time it's able to pick up the gold, . . .
- **Supervised** (Classification)
  - ▷ learn to classify actions into good or bad from info provided by teacher
- **Evolutionary:**
  - ▷ from pool of possible controllers: try them out, select the ones that do best, and mutate and recombine for a number of iterations, keeping best

Advantage: does not require much knowledge in principle

Disadvantage: in practice though, right features needed, incomplete information is problematic, and unsupervised learning is slow . . .

Solution 3: Model-Based Approach

- specify model for problem: actions, initial situation, goals, and sensors
- let a solver compute controller automatically

$$\begin{array}{c} \textit{Actions} \\ \textit{Sensors} \\ \textit{Goals} \end{array} \longrightarrow \boxed{\;\textit{SOLVER}\;} \longrightarrow \textit{CONTROLLER} \quad \begin{array}{c} \xrightarrow{actions} \\ \xleftarrow{observations} \end{array} \quad \textit{World}$$

Advantage: flexible, clear, and domain-independent

Disadvantage: need a model; computationally intractable

Model-based approach to intelligent behavior called **Planning** *in AI*

Basic State Model for Classical AI Planning

- finite and discrete state space S
- a **known initial state** $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each $s \in S$
- a **deterministic transition function** $s' = f(a, s)$ for $a \in A(s)$
- positive **action costs** $c(a, s)$

A **solution** is a sequence of applicable actions that maps s_0 into S_G, and it is **optimal** if it minimizes **sum of action costs** (e.g., $\#$ of steps)

Different **models** obtained by relaxing assumptions in **bold** . . .

Uncertainty but No Feedback: Conformant Planning

- finite and discrete state space S
- a **set of possible initial state** $S_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each $s \in S$
- a **non-deterministic** transition function $F(a, s) \subseteq S$ for $a \in A(s)$
- uniform action costs $c(a, s)$

A **solution** is still an **action sequence** but must achieve the goal for **any possible initial state and transition**

More complex than **classical planning**, verifying that a plan is **conformant** intractable in the worst case; but special case of **planning with partial observability**

Planning with Markov Decision Processes

MDPs are **fully observable, probabilistic** state models:

- a state space S
- initial state $s_0 \in S$
- a set $G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- **transition probabilities** $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- action costs $c(a, s) > 0$

– **Solutions** are **functions (policies)** mapping states into actions

– **Optimal** solutions minimize **expected cost** to goal

Partially Observable MDPs (POMDPs)

POMDPs are **partially observable, probabilistic** state models:

- states $s \in S$
- actions $A(s) \subseteq A$
- transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$

- initial **belief state** b_0
- final **belief states** b_F
- **sensor model** given by probabilities $P_a(o|s)$, $o \in Obs$

– **Belief states** are probability distributions over S

– **Solutions** are policies that map belief states into actions

– **Optimal** policies minimize **expected** cost to go from b_0 to b_F

Models, Languages, and Solvers

- A **planner** is a **solver over a class of models;** it takes a model description, and computes the corresponding controller

$$\mathit{Model} \Longrightarrow \boxed{\mathit{Planner}} \Longrightarrow \mathit{Controller}$$

- Many models, many solution forms: uncertainty, feedback, costs, . . .

- Models described in suitable **planning languages** (Strips, PDDL, PPDDL, . . .) where **states** represent interpretations over the language.

Language for Classical Planning: Strips

- A **problem** in Strips is a tuple $P = \langle F, O, I, G \rangle$:
  - ▷ F stands for set of all **atoms** (boolean vars)
  - ▷ O stands for set of all **operators** (actions)
  - ▷ $I \subseteq F$ stands for **initial situation**
  - ▷ $G \subseteq F$ stands for **goal situation**

- Operators $o \in O$ **represented** by
  - ▷ the **Add** list $Add(o) \subseteq F$
  - ▷ the **Delete** list $Del(o) \subseteq F$
  - ▷ the **Precondition** list $Pre(o) \subseteq F$

From Language to Models

A Strips problem $P = \langle F, O, I, G \rangle$ determines **state model** $\mathcal{S}(P)$ where

- the states $s \in S$ are **collections of atoms** from F
- the initial state s_0 is I
- the goal states s are such that $G \subseteq s$
- the actions a in $A(s)$ are ops in O s.t. $Prec(a) \subseteq s$
- the next state is $s' = s - Del(a) + Add(a)$
- action costs $c(a, s)$ are all 1

– (Optimal) **Solution** of P is (optimal) **solution** of $\mathcal{S}(P)$

– Slight language extensions often convenient (e.g., **negation** and **conditional effects**); some required for describing richer models (costs, probabilities, ...).

Example: Blocks in Strips (PDDL Syntax)

```
(define (domain BLOCKS)
  (:requirements :strips) ...
  (:action pick_up
             :parameters (?x)
             :precondition (and (clear ?x) (ontable ?x) (handempty))
             :effect (and (not (ontable ?x)) (not (clear ?x)) (not (handempty)) (hol
  (:action put_down
             :parameters (?x)
             :precondition (holding ?x)
             :effect  (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
  (:action stack
             :parameters (?x ?y)
             :precondition (and (holding ?x) (clear ?y))
             :effect  (and (not (holding ?x)) (not (clear ?y)) (clear ?x)(handempty)
                           (on ?x ?y))) ...
(define (problem BLOCKS_6_1)
   (:domain BLOCKS)
   (:objects F D C E B A)
   (:init (CLEAR A) (CLEAR B) ...  (ONTABLE B) ... (HANDEMPTY))
   (:goal (AND (ON E F) (ON F C) (ON C B) (ON B A) (ON A D))))
```

Example: Logistics in Strips PDDL

```
(define (domain logistics)
  (:requirements :strips :typing :equality)
  (:types airport - location  truck airplane - vehicle  vehicle packet - thing  thin
  (:predicates (loc-at ?x - location ?y - city) (at ?x - thing ?y - location) (in ?x
  (:action load
    :parameters (?x - packet ?y - vehicle)
    :vars (?z - location)
    :precondition (and (at ?x ?z) (at ?y ?z))
    :effect (and (not (at ?x ?z)) (in ?x ?y)))
  (:action unload ..)
  (:action drive
    :parameters (?x - truck ?y - location)
    :vars (?z - location ?c - city)
    :precondition (and (loc-at ?z ?c) (loc-at ?y ?c) (not (= ?z ?y)) (at ?x ?z))
    :effect (and (not (at ?x ?z)) (at ?x ?y)))
...
(define (problem log3_2)
  (:domain logistics)
  (:objects packet1 packet2 - packet  truck1 truck2 truck3 - truck  airplane1 - airp
  (:init (at packet1 office1) (at packet2 office3) ...)
  (:goal (and (at packet1 office2) (at packet2 office2))))
```

Example: 15-Puzzle in PDDL

```
(define (domain tile)
  (:requirements :strips :typing :equality)
  (:types tile position)
  (:constants blank - tile)
  (:predicates (at ?t - tile ?x - position ?y - position)
       (inc ?p - position ?pp - position)
       (dec ?p - position ?pp - position))
  (:action move-up
    :parameters (?t - tile ?px - position ?py - position  ?bx - position ?by - posit
    :precondition (and (= ?px ?bx) (dec ?by ?py) (not (= ?t blank)) ...)
    :effect (and (not (at blank ?bx ?by)) (not (at ?t ?px ?py)) (at blank ?px ?py) (
   ...
(define (domain eight_tile) ..
  (:constants t1 t2 t3 t4 t5 t6 t7 t8 - tile    p1 p2 p3 - position)
  (:timeless (inc p1 p2) (inc p2 p3) (dec p3 p2) (dec p2 p1)))

(define (situation eight_standard)
  (:domain eight_tile)
  (:init (at blank p1 p1) (at t1 p2 p1) (at t2 p3 p1) (at t3 p1 p2) ..)
  (:goal (and (at t8 p1 p1) (at t7 p2 p1) (at t6 p3 p1) ..)
```

Computation: how to solve Strips planning problems?

- **Key issue:** exploit two roles of **language:**
  - **specification:** concise model description
  - **computation:** reveal useful heuristic info

- **Two traditional approaches:** search vs. decomposition
  - explicit **search** of the state model $S(P)$ direct but not effective til recently
  - **near decomposition** of the planning problem thought a better idea

Computational Approaches to Classical Planning

- **Strips algorithm** (70's): Total ordering planning backward from Goal; work always on **top** subgoal in stack, delay rest
- **Partial Order (POCL) Planning** (80's): work on **any** subgoal, resolve threats; UCPOP 1992
- **Graphplan** (1995 – . . .): build graph containing all possible **parallel** plans up to certain length; then extract plan by searching the graph backward from Goal
- **SatPlan** (1996 – . . .): map planning problem given horizon into SAT problem; use state-of-the-art SAT solver
- **Heuristic Search Planning** (1996 – . . .): search state space $\mathcal{S}(P)$ with heuristic function h extracted from problem P
- **Model Checking Planning** (1998 – . . .): search state space $\mathcal{S}(P)$ with 'symbolic' BrFS where sets of states represented by formulas implemented by BDDs

State of the Art in Classical Planning

- significant **progress** since Graphplan (Blum & Furst 95)
- **empirical methodology**
  - ▷ standard PDDL language
  - ▷ planners and benchmarks available; competitions
  - ▷ focus on performance and scalability
- **large problems solved** (non-optimally)
- different **formulations** and **ideas**

We'll focus on **two formulations:**

- (Classical) Planning as **Heuristic Search**, and
- (Classical) Planning as **SAT**