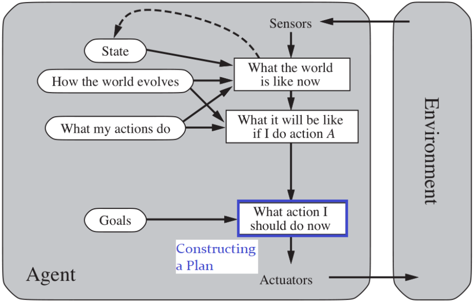
Artificial Intelligence

Classical Planning: Task Representation

Jiamou Liu
The University of Auckland

Recap: Goal-based Agents

Example. [Goal-based agents] A goal-based agent program maintains explicit information about the situations that are desired by the agent.

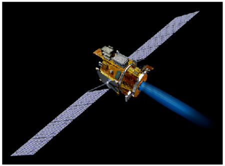


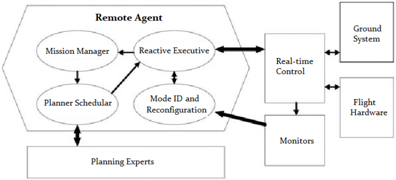
Planning is the task of devising a sequence of actions to reach a goal.

Application Scenario 1: Deep Space 1

Remote Agent, an AI system that operates NASA's Deep Space 1 spacecraft 1999.

- Remote Agent is an independent onboard mission control system to navigate and control the spacecraft.
- Traditional spacecraft control: Ground control sends a plan. If anything goes wrong, spacecraft calls ground control and wait for an updated plan.
- Remote Agent: Ground control sends a high-level goal. Remote Agent generates plan autonomously.
- Remote Agent's main components:
  - Planner (planning)
  - Executive (execution)
  - Diagnostic system (inference)





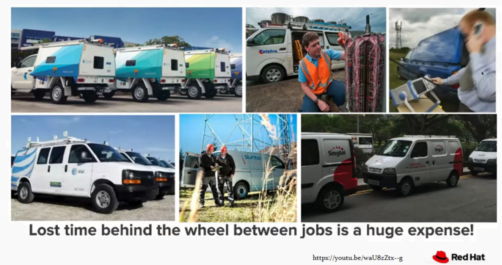
The Remote Agent embedded in Deep Space 1's Flight Software

http://web.csulb.edu/~wmartinz/rssc/content/new-millennium-remote-agent-architecture.html

Application Scenario 2: OptaPlanner

OptaPlanner, developed by Red Hat, facilitates domain-independent planning tasks for business operations, such as scheduling, routing, resource allocations, etc.

- A large telecom service provider with 70000+ technicians who work as "man in a van".
- The task is to schedule jobs for the technicians, under constraints.
- Large amount of constraints: Location, skills, availability, job windows, etc.
- OptaPlanner helps to save USD 200M+ cost, reduce 25% driving time, and improve quality of service[1].

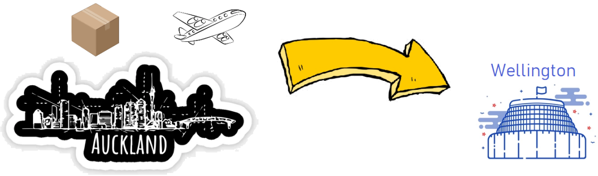


[1]Claimed by Red Hat.

Classical Planning

Example. [package delivery] Suppose we want to send a package from Auckland to Wellington, and another package from Wellington back to Auckland. Devise a plan for this job.

- Initial state: Package 1 at Auckland, Package 2 at Wellington, Plane at Auckland.
- Goal state: Package 1 at Wellington, Package 2 at Auckland.
- Actions:
  - Load package to plane
  - Unload package from plane
  - Fly plane from one city to another
- Plan: A sequence of actions that takes the initial state to the goal state.



Classical planning is a planning task with the following assumptions:

- Finite state space, with a initial state and goal states.
- Perfect information: The agent can tell which state we are in.
- Deterministic actions: Each action in a state has one outcome, which can be foreseen by the agent.
- Nothing changes unless the agent changes.
- Goals must be achieved.

Classical planning is a planning task with the following assumptions:

- Finite state space, with a initial state and goal states.
- Perfect information: The agent can tell which state we are in.
- Deterministic actions: Each action in a state has one outcome, which can be foreseen by the agent.
- Nothing changes unless the agent changes.
- Goals must be achieved.

Note.

- A plan is a sequence of actions that could take us from the initial state to the goal state.
- This is basically a search problem, i.e., exploring the state space hoping to find a sequence of actions that lead from the initial state to the goal state.

Classical planning is a planning task with the following assumptions:

- Finite state space, with a initial state and goal states.
- Perfect information: The agent can tell which state we are in.
- Deterministic actions: Each action in a state has one outcome, which can be foreseen by the agent.
- Nothing changes unless the agent changes.
- Goals must be achieved.

Note.

- A plan is a sequence of actions that could take us from the initial state to the goal state.
- This is basically a search problem, i.e., exploring the state space hoping to find a sequence of actions that lead from the initial state to the goal state.

Two paradigms for planning: search and inference.

- Search: A search strategy such as A^* requires domain-specific heuristics to function efficiently.
- Inference: SATPlan uses domain-independent heuristics for inference, but relies on propositional logic which may be space inefficient.

We will describe these paradigms in detail in future lectures.

Planning Task Representation

We need a domain-independent language that allows efficient representation of classical planning tasks.

To define a task, we need to specify

1. States
2. Actions (state transitions)
3. Initial state
4. Goal state

In the following we will describe these components using first-order logic.

1. States: A state is a conjunction of ground, functionless atoms.

E.g. [air cargo]

- $At(P, AKL)$ is a state.
- $At(Plane_1, AKL) \wedge At(Plane_2, WLG)$ is a state.
- $At(x, y)$ is not a state.
- $\neg At(Plane_1, WLG)$ is not a state.
- $At(Pilot(Plane_1), AKL)$ is not a state.

Note. We apply the following assumptions as for first-order inference

- Close-world assumption: Any atomic sentence not appearing in the state is assumed to be false.
- Domain-closure assumption: All elements of the domain are expressed using constants.

2. Actions: An action schema consists of:

- Action name
- List of variables used in the schema (assumed to be universally quantified).
- Preconditions $Precond(a)$: Conjunction of literals (positive or negative); defines the states in which the action can apply
- Effects $Effects(a)$: Conjunction of literals (positive or negative); defines the changes made as the action is executed.

  The delete set $Del(a)$ contains the negative literals in $Effects(a)$.
  The add set $Add(a)$ contains the positive literals in $Effects(a)$

E.g. [air cargo] An action schema for *Fly*:

$$\begin{aligned}&Action(Fly(p, from, to),\\ &\quad Precond : At(p, from) \wedge Plane(p) \wedge Airport(From) \wedge Airport(to)\\ &\quad Effect : \neg At(p, from) \wedge At(p, to))\end{aligned}$$

The schema can be (universally) instantiated:

$$\begin{aligned}&Action(Fly(P, AKL, WLG),\\ &\quad Precond : At(P, AKL) \wedge Plane(P) \wedge Airport(AKL) \wedge Airport(WLG)\\ &\quad Effect : \neg At(P, AKL) \wedge At(P, WLG))\end{aligned}$$

Example. [air cargo] Action schemas for the delivery task:

$$
\begin{aligned}
&Action(Load(c,p,a),\\
&\quad Precond: At(c,a) \wedge At(p,a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)\\
&\quad Effect: \neg At(c,a) \wedge In(c,p))\\
&Action(Unload(c,p,a),\\
&\quad Precond: In(c,p) \wedge At(p,a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)\\
&\quad Effect: At(c,a) \wedge \neg In(c,p))\\
&Action(Fly(p,from,to),\\
&\quad Precond: At(p,from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)\\
&\quad Effect: \neg At(p,from) \wedge At(p,to))
\end{aligned}
$$

- **3. Initial state:** A conjunction of ground atoms.

  E.g. [air cargo]

$$\begin{aligned}&Init(At(C_1, AKL) \land At(C_2, WLG) \land At(P, AKL)\\&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P)\\&\quad \land Airport(AKL) \land Airport(WLG))\end{aligned}$$

  Close-world assumption means that all literals not appearing in the state is assumed to be false.

- **3. Initial state:** A conjunction of ground atoms.

  E.g. [air cargo]

$$
\begin{aligned}
Init(&At(C_1, AKL) \wedge At(C_2, WLG) \wedge At(P, AKL)\\
&\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P)\\
&\wedge Airport(AKL) \wedge Airport(WLG))
\end{aligned}
$$

  Close-world assumption means that all literals not appearing in the state is assumed to be false.

- **4. Goal:** A conjunction of literals (positive or negative)

  E.g. $Goal(At(C_1, WLG) \wedge At(C_2, AKL))$.

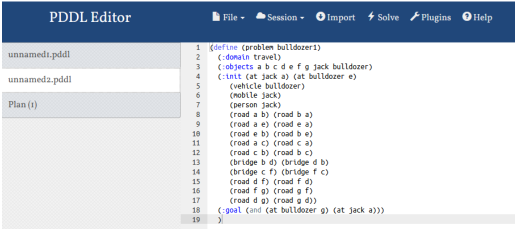
PDDL

Planning Domain Definition Language (PDDL) is a domain-independent language that implements the framework above[2].

- Online PDDL editor: `http://editor.planning.domains/`
- Load/Save `.pddl` file

A PDDL definition consists of a domain file and a problem file.

- Domain file: Specifies predicates (states), and actions
- Problem file: Specifies initial state and goal.



[2] Developed in 1998 with inspiration from earlier problem solving systems such as STRIPS (Stanford Research Institute Problem Solver), which forms a subset of PDDL.

1. Domain definition

```
(define (domain DOMAIN_NAME)
    (:requirements [:strips] [:equality] [:typing] [:adl])
    (:predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN)
                 (PREDICATE_2_NAME ?A1 ?A2 ... ?AN) ...)
    [(:constants CONSTANT_1 CONSTANT_2 ...)]
    (:action ACTION_1_NAME
        [:parameters (?P1 ?P2 ... ?PN)]
        [:precondition PRECOND_FORMULA]
        [:effect EFFECT_FORMULA])
    (:action ACTION_2_NAME ...)
...)
```

E.g. [air cargo]

```
(define (domain cargo)
  (:requirements :strips)
  (:predicates
    (at ?thing ?place) (plane ?pl) (airport ?a)
    (in ?thing ?place) (cargo ?thing)
  )
...)
```

Action definition:

- Precondition formula:
  - An atomic formula: `(PREDICAT_NAME ARG1 ... ARG_N)` or
  - A conjunction of atomic formulas: `(and ATOM1 ... ATOM_N)`
- Effect formula:
  - An added atom: `(PREDICATE_NAME ARG1 ... ARG_N)`
  - A deleted atom: `(not (PREDICATE_NAME ARG1 ... ARG_N)`
  - A conjunction of atomic effects: `(and ATOM1 ... ATOM_N)`

E.g. [air cargo]

```
(:action fly
  :parameters (?pl ?from ?to)
  :precondition (and (plane ?pl) (airport ?from) (airport ?to)
                     (at ?pl ?from))
  :effect (and (at ?pl ?to) (not (at ?pl ?from))
          )
)
```

2. Problem definition

```
(define (problem PROBLEM_NAME)
    (:domain DOMAIN_NAME)
    (:objects OBJ1 OBJ2 ... OBJ_N)
    (:init ATOM1 ATOM2 ... ATOM_N)
    (:goal CONDITION_FORMULA)
)
```

E.g. [air cargo]

```
(define (problem task1)
    (:domain cargo)
    (:objects c1 c2 akl wlg p)
    (:init (at c1 akl) (at c2 wlg) (at p akl)
       (cargo c1) (cargo c2) (plane p)
       (airport akl) (airport wlg))
    (:goal (and (at c1 wlg) (at c2 akl)) )
)
```

More Examples

Example. [spare tire] Changing a flat tire.

- Initial state: flat tire on the axle and a good spare tire in the trunk
- Goal: good spare tire properly mounted onto the car's axle
- Actions: Removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended.

First-order logic description:

$$
\begin{aligned}
&Action(Remove(obj, loc),\\
&\quad PRECOND: At(obj, loc),\\
&\quad EFFECT: \neg At(obj, loc) \wedge At(obj, Ground))\\
&Action(PutOn(t, Axle),\\
&\quad PRECOND: Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle),\\
&\quad EFFECT: \neg At(t, Ground) \wedge At(t, Axle))\\
&Action(LeaveOvernight,\\
&\quad PRECOND: ,\\
&\quad EFFECT: \neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)\\
&\qquad\qquad \wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk))\\
&Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk))\\
&Goal(At(Spare, Axle))
\end{aligned}
$$

PDDL domain:

```
(define (domain tire)
  (:requirements :strips)
  (:predicates (at ?thing ?place) (tire ?tr))
  (:constants Flat Spare Axle Trunk Ground)
  (:action remove
    :parameters (?obj ?loc)
    :precondition (at ?obj ?loc)
    :effect (and (not (at ?obj ?loc)) (at ?obj Ground))
  )
  (:action putOn
    :parameters (?tr)
    :precondition (and (tire ?tr) (at ?tr Ground) (not (at Flat Axle)))
    :effect (and (not (at ?tr Ground)) (at ?tr Axle))
  )
  (:action leaveOvernight
    :parameters ()
    :effect (and (not (at Spare Ground)) (not (at Spare Axle))
            (not (at Spare Trunk)) (not (at Flat Axle))
            (not (at Flat Ground))  (not (at Flat Trunk)))
  )
)
```

PDDL problem:

```
(define (problem tire1)
  (:domain tire)
  (:objects tr)

  (:init (tire Flat) (tire Spare) (at Flat Axle)
         (at Spare Trunk)
  )
  (:goal (at Spare Axle)
  )
)
```

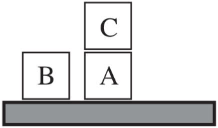
Solve to get plan:

```
(remove Flat Axle) (remove Spare Trunk) (PutOn Spare)
```

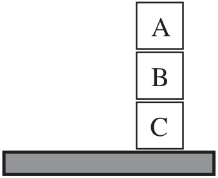
Example. [blocks world] Stackable blocks on a table. A robot arm can pick up the top block of a stack and move it to another position.

Predicates: $onTable(x), on(x,y), clear(x)$

$$
\begin{aligned}
&Action(MoveToTable(x,y),\\
&\qquad PRECOND: clear(x) \land on(x,y),\\
&\qquad EFFECT: clear(y) \land onTable(x) \land \neg on(x,y))\\
&Action(MoveToBlock1(x,y,z)\\
&\qquad PRECOND: clear(x) \land clear(z) \land on(x,y),\\
&\qquad EFFECT: clear(y) \land on(x,z) \land \neg clear(z) \land \neg on(x,y))\\
&Action(MoveToBlock2(x,y),\\
&\qquad PRECOND: clear(x) \land clear(y) \land onTable(x),\\
&\qquad EFFECT: on(x,y) \land \neg clear(y) \land \neg onTable(x))\\
&\quad Init(onTable(B) \land on(C,A) \land onTable(A) \land clear(C) \land clear(B))\\
&\quad Goal(on(A,B) \land on(B,C))
\end{aligned}
$$



Start State


Goal State

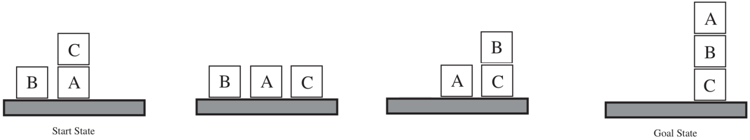
PDDL domain:

```
(define (domain blocks_world)
  (:requirements :strips)
  (:predicates (on-table ?x) (on ?x ?y) (clear ?x))

  (:action MoveToTable
    :parameters (?x ?y)
    :precondition (and (clear ?x) (on ?x ?y))
    :effect (and (clear ?y) (on-table ?x) (not (on ?x ?y))))

  (:action MoveToBlock1
    :parameters (?x ?y ?z)
    :precondition (and (clear ?x) (clear ?z) (on ?x ?y))
    :effect (and (clear ?y) (on ?x ?z) (not (clear ?z)) (not (on ?x ?y))))

  (:action MoveToBlock2
    :parameters (?x ?y)
    :precondition (and (clear ?x) (clear ?y) (on-table ?x))
    :effect (and (on ?x ?y) (not (clear ?y)) (not (on-table ?x)))
  )
)
```

PDDL problem:

```
(define (problem bwtask1)
    (:domain blocks_world)
    (:objects a b c)
    (:init
        (on-table b) (on c a) (on-table a) (clear b) (clear c)
    )
    (:goal (and
            (on a b) (on b c)
         )
    )
)
```

Solve to get plan: (MoveToTable c a) (MoveToBlock2 b c) (MoveToBlock2 a b)



Summary of The Topic

The following are the main knowledge points covered:

- Classical planning: Finding a sequence of actions to get from an initial state to a goal state, in a finite, deterministic, fully observable state space.
- Task description: A first-order logic language for describing domain-independent tasks
    - States: Conjunction of ground, functionless atoms
    - Action schema:
        - Parameters
        - Preconditions
        - Effects
    - Initial state
    - Goal
- PDDL implementation:
    - Domain file
    - Problem file