Classical Planning
via State-space search

COMP3431
Malcolm Ryan
What is planning?

- Planning is the AI approach to control
- Deliberation about action
- Key ideas
  - We have a **model** of the world
  - Model describes **states** and **actions**
  - Give the planner a **goal** and it outputs a **plan**
  - Aim for **domain independence**
- Planning is search
Some History

1969 **GPS**, Newell and Simon
   State-space search, means-ends analysis

1971 **STRIPS**, Fikes and Nilsson
   Introduced STRIPS notation for actions

1975 **NOAH**, Sacerdoti
   **NONLIN**, Tate
   First plan-space search planners

1989 **ADL**, Pednault
   An extension of the STRIPS action notation
Further History

1991 **SNLP**, Soderland and Weld
   based on McAllester and Rosenblitt
   Plan-space search make easy

1995 **SATPlan**, Kautz and Selman
   Planning as a satisfiability problem

1995 **GraphPlan**, Blum and Furst
   A return to state-space search, using planning graphs.

1998 **PDDL**, McDermott et al
   Extending ADL to include… ?
Classical planning

• **Classical planning** is the name given to early planning systems (before about 1995)

• Most of these systems are based on the Fikes & Nilsson’s STRIPS notation for actions (1971)

• Includes both **state-space** and **plan-space** planning algorithms.
The Model

- Planning is performed based on a given model of the world.
- A model $\Sigma$ includes:
  - A set of states, $S$
  - A set of actions, $A$
  - A transition function, $\gamma : S \times A \to S$
Restrictions on the Model

1. $S$ is finite
2. $\Sigma$ is fully observable
3. $\Sigma$ is deterministic
4. $\Sigma$ is static (no external events)
5. $S$ has a factored representation
6. Goals are restricted to reachability.
7. Plans are ordered sequences of actions
8. Actions have no duration
9. Planning is done offline
Example: Blocks World

- blue
- red
- green

table
Example: Blocks World

- $S$ = the set of all different configurations of the blocks
- $A$ = the set of “move” actions
- $\gamma$ describes the outcomes of actions

move(red,blue,green)
States, actions and goals

- States, actions and goals are described in the language of **symbolic logic**.
- **Predicates** denote particular features of the world:
  - Eg, in the blocks world:
    - on(block1, block2)
    - on_table(block)
    - clear(block)
Representing States

- States are described by conjunctions of ground predicates (possibly negated).
- Eg:
  \[ \text{on(blue, red)} \land \neg \text{on(green, red)} \]
- The **closed world assumption** (CWA) is employed to remove negative literals:
  \[ \text{on(blue, red)} \]
- The state description is **complete**.
Representing Goals

- The goal is the specification of the task
- A goal is a usually conjunction of predicates:
  \[\text{on}(\text{red}, \text{green}) \land \text{on}_\text{table}(\text{green})\]
- The CWA does **not** apply.
- So the above goal could be satisfied by:
  \[\text{on}(\text{red}, \text{green}) \land \text{on}_\text{table}(\text{green}) \land \text{on}(\text{blue}, \text{red}) \land \text{clear}(\text{blue}) \land \ldots\]
Actions are described in terms of **preconditions** and **effects**.

Preconditions are predicates that must be true **before** the action can be applied.

Effects are predicates that are made true (or false) **after** the action has executed.

Sets of similar actions can be expressed as a **schema**.
STRIPS operators

• An early but still widely used form of action description is as “STRIPS operators”.
• Three parts:
  Precondition A conjunction of predicates
  Add-list The set of predicates made true
  Delete-list The set of predicates made false
Blocks World Action Schema

move\(^{(block, from, to)}\)

- **Pre:**
  
  on\(^{(block, from)}\), clear\(^{(block)}\),
  
  clear\(^{(to)}\)

- **Add:**
  
  on\(^{(block, to)}\), clear\(^{(from)}\)

- **Del:**
  
  on\(^{(block, from)}\), clear\(^{(to)}\)
Blocks World Actions

• Note that this action schema defines many actions:
  move(red, blue, green)
  move(red, green, blue)
  etc...

• We also need to define:
  move_to_table(block, from)
  move_from_table(block, to)
Representing Plans

• A plan is simply a sequence of actions. eg
  \[ \pi = \text{move\_from\_table(red, blue)}, \]
  \[ \text{move(red, blue, green)}, \]
  \[ \text{move\_to\_table(red, green)} \]

• We require that every action in the sequence is applicable, i.e. its precondition is true before it is executed.
Reasoning with STRIPS

• An action $a$ is **applicable** in state $s$ if its precondition is satisfied, ie:
  \[ \text{pre}^+(a) \subseteq s \]
  \[ \text{pre}^-(a) \cap s = \emptyset \]

• The result of executing $a$ in $s$ is given by:
  \[ \gamma(s,a) = (s - \text{del}(a)) \cup \text{add}(a) \]

• This is called **progressing** $s$ through $a$
Progression example

• Taking the earlier example:

\[
S = \text{on}(\text{red}, \text{blue}), \ \text{on\_table}(\text{blue}),
\quad \text{clear}(\text{red}), \ \text{on\_table}(\text{green}),
\quad \text{clear}(\text{green})
\]
\[
a = \text{move}(\text{red}, \text{blue}, \text{green})
\]
Progression example

1. Check action is applicable:
   on(red, blue), clear(red), clear(green)

2. Delete predicates from delete-list:
   on(red, blue), on_table(blue),
   clear(red), on_table(green),
   clear(green)

3. Add predicates from add-list:
   on_table(blue), clear(red),
   on_table(green), on(red, green),
   clear(blue)
Progression example 2

- Consider instead the action
  \[ a = \text{move\_from\_table(blue, green)} \]
- This has precondition:
  \[ \text{pre}(a) = \text{on\_table(blue)}, \text{clear(blue)}, \text{clear(green)} \]
- This action cannot be executed as \text{clear(blue)} is not in s.
- i.e. it is not applicable
Reasoning with STRIPS

- We can also **regress** states.
- If we want to achieve goal $g$, using action $a$, what needs to be true beforehand?
- An action $a$ is **relevant** for $g$, if:
  
  $g \cap \text{add}(a) \neq \emptyset$
  
  $g \cap \text{del}(a) = \emptyset$

- The result of regressing $g$ through $a$ is:
  
  $\gamma^{-1}(g,a) = (g - \text{add}(a)) \cup \text{pre}(a)$
Regression Example

• Take the goal:
  \[ g = \text{on(red, green), on\_table(blue)} \]
• Regress through action:
  \[ a = \text{move(red, blue, green)} \]
Regression Example

1. Check action is relevant:
   \[ g \cap \text{add}(a) = \{\text{on(red, green)}\} \neq \emptyset \]
   \[ g \cap \text{del}(a) = \emptyset \]

2. Remove predicates from add list:
   \text{on(red, green), on_table(blue)}

3. Add preconditions:
   \text{on_table(blue), on(red, blue), clear(red), clear(green)}
Regression example 2

- Consider instead the action
  \[ a = \text{move\_to\_table}(\text{red}, \text{blue}) \]
- This has effects:
  \[ \text{add}(a) = \text{on\_table}(\text{red}), \text{clear}(\text{blue}) \]
- This action is not relevant as it does not achieve any of the goal predicates, ie:
  \[ g \cap \text{add}(a) = \emptyset \]
Regression Example 3

- Consider instead the goal
  \[ g = \text{clear(blue)}, \text{clear(green)} \]
- Now \( a = \text{move(red, blue, green)} \) achieves \text{clear(blue)} but is not relevant, as it conflicts with the goal:
  \[ g \cap \text{del}(a) = \{\text{clear(green)}\} \neq \emptyset \]
Planning as state-space search

• Imagine a directed graph in which nodes represent states and edges represent actions.

• An edge joins two nodes if there is an action that takes you from one state to the other.
Graph of state space
Forward/Backward chaining

• Planning can be done as **forward** or **backward chaining**.

• Forward chaining starts at the initial state and searches for a path to the goal using progression.

• Backward chaining starts at the goal and searches for a path to the initial state using regression.
Non-deterministic programming

• I will show planning algorithms using **non-deterministic** pseudocode
• The **choose** command will allow us to choose one of several paths to take.
• The **fail** command will indicate that a particular choice was unsuccessful.
Non-deterministic programming

• When we fail, we **backtrack** to the most recent choice with more options, and choose again
• Executing the code is then actually a search process
• **Prolog** is an example of such a language
Example

SumToSeven()

    choose \( a \in \{1,2,3,4,5\} \)
    choose \( b \in \{1,2,3,4,5\} \)
    if \( a + b == 7 \) then return \{a,b\}
    else fail
Example 2

SumToN(n)

\textbf{choose} \ a \in \ \{1,2,3,4,5\} \\
d = n - a \\
if (d < 0) \text{ then fail} \\
else if (d == 0) \text{ then return } \{a\} \\
\textbf{else} \text{ return } \{a\} \cup \text{SumToN}(d)
Forward Search

Forward-search(s, g)
    if s satisfies g then return empty plan
    applicable = \{a \mid a \text{ is applicable in } s\}
    if applicable = \emptyset then fail
    choose action a ∈ applicable
    s’ = γ(s,a)
    π’ = Forward-search(s’,g)
    return a.π’
Backward Search

Backward-search(s, g)
   if s satisfies g then return empty plan
   relevant = \{a \mid a \text{ is relevant to } g\}
   if relevant = \emptyset then fail
   choose action a ∈ relevant
   g’ = γ^{−1}(g,a)
   π’ = Backward-search(s,g’)
   return π’.a
Pruning Search

• As in any search problem, an important element is to **prune** the search.
• Both of these algorithms have the potential to waste time exploring **loops**.
• A record should be kept of already **visited states** and actions that return to these states should be pruned.
• Backwards search can produce inconsistent goals (usually not pruned)
Instantiating Schema

• When using action schema often it is more efficient to instantiate schema variables on the fly, by unification, rather than generating and testing all instances.
Planning as Search

- Forward-search, backward-search and most other planning algorithms can be described in a similar structure:
  1. Generate possible branches.
  2. Prune those that are no good.
  3. Select one remaining branch
  4. Recurse
Bi-directional search

• We can apply bidirectional search to state-space planning, doing both progression from the start state and regression from the goal.

• Typical heuristic:
  – try both, then repeat whichever expansion took less time (as it is likely to again)
Heuristics for planning

• “cost to goal” = number of plan steps
• relaxed measure:
  number of plan steps **disregarding delete effects**
  \[
  h_1(s,g) = \max \{ h_1(s,p) \mid p \in g \} \\
  h_1(s,p) = 0, \quad \text{if } p \in s \\
  h_1(s,p) = \infty, \quad \text{if } p \text{ is not in } \text{add}(a) \text{ for any } a \\
  h_1(s,p) = \min(1 + h_1(s,\text{pre}(a)), p \in \text{add}(a))
  \]
Heuristics for planning

• $h_1$ can be extended to $h_2$, $h_3$, … considering pairs of propositions, triplets, etc.
• All $h_k$ are admissible
• $h_{k+1}$ dominates $h_k$
• Computing $h_k$ is $O(n^k)$ with $n$ propositions
• Generally $k \leq 2$ is used