Introduction to Artificial Intelligence

Planning

Ing. Tomas Borovicka

Department of Theoretical Computer Science (KTI), Faculty of Information Technology (FIT)
Czech Technical University in Prague (CVUT)

BIE-ZUM, LS 2013/14, 6. lecture

https://edux.fit.cvut.cz/courses/BIE-ZUM/

Summary of Previous Lecture

- Evolutionary Computation
    - Family of global meta-heuristic or stochastic optimization methods.
    - Algorithms typically imitate some principle of natural evolution as method to solve optimization problems.
    - Genetic Algorithm
        - Universal ”black box” solver for optimization using binary strings.
    - Genetic Programming
        - Approach for general automatic programming.
        - Originally used to evolve Lisp programs.
    - Evolutionary Programming
        - Motivation was to generate alternative approach to artificial intelligence.
        - Early versions applied to the evolution of transition table of finite state machines.
    - Evolution Strategies
        - Based on the concept of the ”evolution of evolution”.

General Problem Solving

- Traditional approach to solve a problem:

  Problem ⟶ Human ⟶ **Design of special algorithm** ⟶ Solution

- Disadvantages:
  - design and implementation of specialized algorithm is expensive,
  - optimization and tuning of the algorithm is time consuming.
- **General Problem Solving**:

  Problem ⟶ Human ⟶ Problem formalization ⟶ **Solver** ⟶ Solution

  - The user must only formalize the problem.

General Problem Solving

General principle:

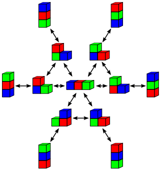
Problem $\longrightarrow$ Model $\longrightarrow$ Language $\longrightarrow$ Solver $\longrightarrow$ Solution

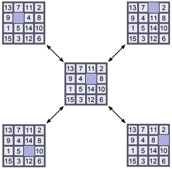
1. Having a **problem** P,
2. choose a **model** M,
3. choose a **language** L for definition of model M,
4. choose a **solver** S for solving problems in L,
5. leave solver **automatically solve** $S(L(P))$,
6. **interpret** $S(L(P))$ in the context of P.

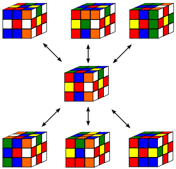
Automatic Planning

- Sliding puzzles,
- Rubik's cube,
- Blocks world,

All these problems can be simply defined using state space, where each state represents some **structured configuration** and actions make changes in this configuration.







Planning vs Search

Planning problem is essentially search problem. However there are differences that make reasonable to treat them in a different way.

	Search	**Planning**
States	data structures	logical sentences
Actions	code	preconditions / effects
Goal	code	logical sentences
Plan	sequence of actions	constrains on actions

The main benefits are:

- Unified action and goal representation (logical language).
- Independent solution of sub-goals .
- Flexible construction of solution, relax requirement for sequential construction.

Classical Planning

Assumptions are:

1. Environment is deterministic
2. Environment is observable
3. Environment is static (in response to the agent's actions)

Planning Problem

Objective is to get tea, biscuits and a book.

- Initial state:

  The agent is at *home* without tea, biscuits, book.

- Goal state:

  The agent is at *home* with tea, biscuits, book.

States

States can be represented by predicates such as:

- At(x) - agent is at a specific position x.
- Have(y) - agent has an item y.
- Sells(x,y) - some location x sells item y.

Actions

- Go(y) - agent goes to y,

  causes At(y) to be true.
- Buy(z) - agent buys z,

  causes Have(z) to be true.
- Steal(z) - agent steals z,

  causes Have(z) to be true.

Problem Representation

- Initially attempted to represent planning problems through variants of predicate calculus, like first order logic and propositional calculus.
- **Initial State**: We are home we do not have tea, we do not have biscuits and we do not have a book.

$$At(Home, s_0) \wedge \neg Have(Tea, s_0) \wedge \neg Have(Biscuits, s_0) \wedge \neg Have(Book, s_0)$$

- **Goal State**: There exists some state where we are at home, we have a tea, we have biscuits and we have a book.

$$\exists s At(Home, s) \wedge Have(Tea, s) \wedge Have(Biscuits, s) \wedge Have(Book, s)$$

Operators

$$\forall a, s\ Have(Tea, Result(a, s)) \iff [(a = Buy(Tea) \wedge At(TeaShop, s)) \\ \wedge (Have(Tea, s) \wedge a \neq Drop(Tea))]$$

- Result(a,s) names the situation resulting from executing the action *a* in the situation *s*.
- Drop(z) - agent drops z,
  - causes Have(z) to be false.

The frame problem

- We have to write rules for all things that does not change.
- Everything that just will not change, we have to explicitly specify in predicate logic to say that it will not change.
- Makes representation of a problem very complex.

Resolution: If very few things are changing at a time, then, it is always easier to model it as changes rather than anything else.

STRIPS

- STanford Research Institute Problem Solver

A STRIPS instance is composed of:

- An initial state.
- The goal states.
- The set of actions (operators) described by
  - preconditions - must be satisfied before the action is performed.
  - effects - established after the action is performed.

STRIPS

STRIPS

STRIPS instance is a quadruple $(P, A, \mathcal{I}, G)$, where

- P is a set of conditions,
- A is a set of actions (operators) described by $\text{precond}(a) \subseteq P$ and $\text{effect}(a) \subseteq P$ (in original version $\text{add}(a) \subseteq P$, $\text{del}(a) \subseteq P$),
- $\mathcal{I} \subseteq P$ is **initial state**,
- $G \subseteq P$ is **goal state**.

Planning problem $\neq$ state space.

STRIPS: Block World Example 1

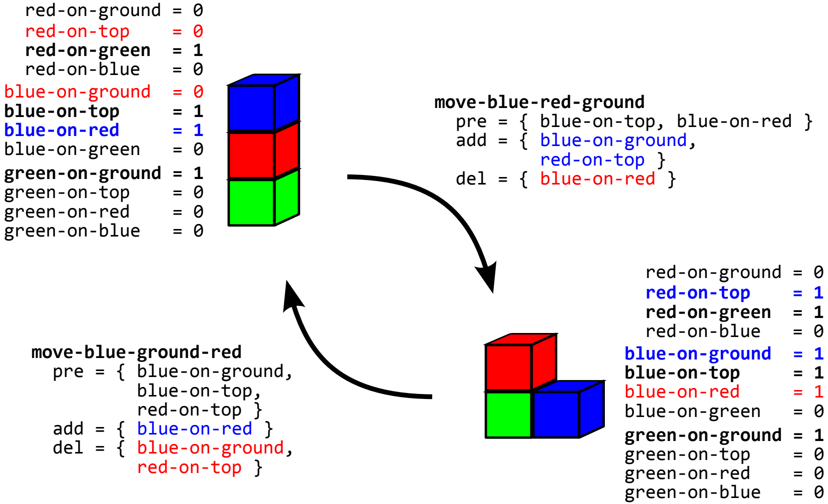
$P = \{$*red-on-ground*, *red-on-top*, *red-on-green*, *red-on-blue*,
green-on-ground, *green-on-top*, *green-on-red*, *green-on-blue*,
blue-on-ground, *blue-on-top*, *blue-on-red*, *blue-on-green* $\}$

$A = \{$*move-red-ground-green*, *move-red-ground-blue*, *move-red-green-blue*,
move-red-blue-green, *move-red-green-ground*, *move-red-blue-ground*,
move-green-ground-red, *move-green-ground-blue*, $\ldots\}$

$\mathcal{I} = \{$*red-on-green*, *green-on-ground*, *blue-on-top*, *blue-on-red* $\}$

$G = \{$*green-on-red*, *red-on-ground*, *blue-on-top*, *blue-on-green* $\}$
move-blue-ground-green $)$

STRIPS: Block World Example 1



red-on-ground = 0
red-on-top = 0
red-on-green = 1
red-on-blue = 0
blue-on-ground = 0
blue-on-top = 1
blue-on-red = 1
blue-on-green = 0
green-on-ground = 1
green-on-top = 0
green-on-red = 0
green-on-blue = 0

move-blue-red-ground
pre = { blue-on-top, blue-on-red }
add = { blue-on-ground, red-on-top }
del = { blue-on-red }

red-on-ground = 0
red-on-top = 1
red-on-green = 1
red-on-blue = 0
blue-on-ground = 1
blue-on-top = 1
blue-on-red = 1
blue-on-green = 0
green-on-ground = 1
green-on-top = 0
green-on-red = 0
green-on-blue = 0

move-blue-ground-red
pre = { blue-on-ground, blue-on-top, red-on-top }
add = { blue-on-red }
del = { blue-on-ground, red-on-top }

Block World Example 2

- **Objects** : $U = \{R, G, B\}$,
- **Predicates**: $P = \{on, on\text{-}ground, on\text{-}top, distinct\}$,

Problem in STRIPS $(P', A, \mathcal{I}, G)$:

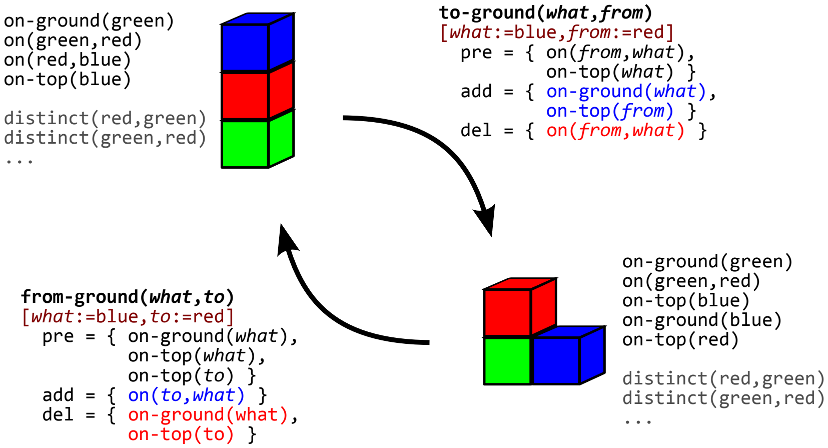
$$\begin{aligned}
P' &= \{on(x,y), on\text{-}ground(x), on\text{-}top(x), distinct(x,y) \mid x, y \in \{R, G, B\}\}, \\
A &= \{move(what, from, to), from\text{-}ground(what, to), to\text{-}ground(what, from)\} \\
\mathcal{I} &= \{on\text{-}ground(G), on(G,R), on(R,B), on\text{-}top(B)\} \\
&\qquad \cup \{distinct(x,y) \mid x, y \in \{R, G, B\}, x \neq y\} \\
G &= \{on\text{-}ground(R), on(R,G), on(G,B), on\text{-}top(B)\},
\end{aligned}$$

STRIPS: Block World Example 2

For each action we define preconditions and effects (add and del):

- *move*$(what, from, to)$
  - pre$(move) = \{on(from, what), on\text{-}top(what), on\text{-}top(to), distinct(what, to)\}$,
  - add$(move) = \{on(to, what), on\text{-}top(from)\}$,
  - del$(move) = \{on(from, what), on\text{-}top(to)\}$,
- *from-ground*$(what, to)$
  - pre$(from\text{-}ground) = \{on\text{-}ground(what), on\text{-}top(what), on\text{-}top(to)\}$
  - add$(from\text{-}ground) = \{on(to, what)\}$
  - del$(from\text{-}ground) = \{on\text{-}ground(what), on\text{-}top(to)\}$
- *to-ground*$(what, from)$
  - pre$(to\text{-}ground) = \{on(from, what), on\text{-}top(what)\}$
  - add$(to\text{-}ground) = \{on\text{-}ground(what), on\text{-}top(from)\}$
  - del$(to\text{-}ground) = \{on(from, what)\}$

STRIPS: Block World Example 2



```
on-ground(green)
on(green,red)
on(red,blue)
on-top(blue)

distinct(red,green)
distinct(green,red)
...
```

```
to-ground(what,from)
[what:=blue,from:=red]
  pre = { on(from,what),
          on-top(what) }
  add = { on-ground(what),
          on-top(from) }
  del = { on(from,what) }
```

```
on-ground(green)
on(green,red)
on-top(blue)
on-ground(blue)
on-top(red)

distinct(red,green)
distinct(green,red)
...
```

```
from-ground(what,to)
[what:=blue,to:=red]
  pre = { on-ground(what),
          on-top(what),
          on-top(to) }
  add = { on(to,what) }
  del = { on-ground(what),
          on-top(to) }
```

Representing States

- States are represented by positive function-free literals (atoms).

  Initial:

  $$At(Home) \land Sells(BS, Book) \land Sells(TS, Tea) \land Sells(TS, Biscuits)$$

  Goal:

  $$At(Home) \land Have(Tea) \land Have(Biscuits) \land Have(Book)$$

- Closed World: unmentioned literals are false.
- In later definition states can also contain variables

  $$At(x) \land Sells(x, Tea).$$

Representing Actions

- **Action description** - serves as a name.
- **Precondition** - a conjunction of positive literals.
- **Effect** - a conjunction of positive and negative literals.
  - The original version had an *add* list and *del* list (effect of $P \land \neg Q$ means add P, delete Q)

$$\begin{aligned} &Op(ACTION : Go(There), \\ &\quad PRECOND : At(here) \land Path(here, there), \\ &\quad EFFECT : At(there) \land \neg At(here)) \end{aligned}$$

Representing Plans

- A set o steps, where each step is one of the operators of the problem.
- A set of step ordering constrains. Each ordering constrain is of the form $S_i \prec S_j$ indicating S_i must occur sometime before S_j
- A set of variable binding constraints of the form $v = x$ where v is a variable in some step, and x is either a constant or another variable.
- A set of causal links written as $S \xrightarrow{c} S'$ indicating S satisfies the precondition c for S'.

Example

Actions:

$$
\begin{aligned}
&Op(ACTION : RightShoe,\\
&\quad PRECONDITION : RightSockOn,\\
&\quad EFECT : RightShoeOn)\\
&Op(ACTION : RightSock,\\
&\quad EFECT : RightShoeOn)\\
&Op(ACTION : LeftShoe,\\
&\quad PRECONDITION : LeftSockOn,\\
&\quad EFECT : LeftShoeOn)\\
&Op(ACTION : LeftShoe,\\
&\quad EFECT : LeftShoeOn)
\end{aligned}
$$

Example: Initial Plan

$$
\begin{array}{l}
\mathit{Plan}(\\
\quad \mathit{STEPS} : \{ \\
\qquad S1 : \mathit{Op}(\mathit{ACTION} : \mathit{start}), \\
\qquad S2 : \mathit{Op}(\mathit{ACTION} : \mathit{finish}, \\
\qquad\quad \mathit{PRECOND} : \mathit{RightShoeOn} \wedge \mathit{LeftShoeOn})\}, \\
\quad \mathit{ODRERINGS} : \{S_1 \prec S_2\}, \\
\quad \mathit{BINDINGS} : \{\}, \\
\quad \mathit{LINKS} : \{\})
\end{array}
$$

Partial Order Planning

- We add steps from the given set of actions in order to satisfy not achieved preconditions.
- We finish when all preconditions of each step has been satisfied.
- Any topologically ordered sequence of actions is solution.

Example

- Initial State:

$$
\begin{aligned}
&Op(ACTION : Start,\\
&\quad EFFECT : At(Home) \land Sells(BS, Book)\\
&\quad \land Sells(TS, Tea) \land Sells(TS, Biscuits))
\end{aligned}
$$

- Goal:

$$
\begin{aligned}
&Op(ACTION : Finish,\\
&\quad PRECOND : At(Home) \land Have(Tea)\\
&\quad \land Have(Biscuits) \land Have(Book)
\end{aligned}
$$

Example

- Actions

$$\begin{aligned} &Op(ACTION : Go(There), \\ &\quad PRECOND : At(here), \\ &\quad EFFECT : At(there) \wedge \neg At(here)) \end{aligned}$$

$$\begin{aligned} &Op(ACTION : Buy(x), \\ &\quad PRECOND : At(store) \wedge Sells(store, x), \\ &\quad EFFECT : Have(x)) \end{aligned}$$

Partial Order Planning: Pseudo-code Sketch

Algorithm 1 Partial Order Planning (POP)

```
plan ← INIT_MINIMAL_PLAN(initial,goal)
loop
    if SOLUTION(plan) then return plan
    end if
    $S_{need}$, c ← SELECT_SUBGOAL(plan)
    CHOOSE_OPERATOR(plan,OPERATORS,$S_{need}$,c)
    RESOLVE_THREATS(plan)
end loop
```

Algorithm 2 Partial Order Planning (POP)

```
function SELECT_SUBGOAL(plan)
    pick a step $S_{need} \in STEPS$ with a precondition c that has not been achieved
    return $S_{need}$, c
end function
```

Algorithm 3 Partial Order Planning (POP)

```
function CHOOSE_OPERATOR($plan$, $OPERATORS$, $S_{need}$, $c$)
    choose s step $S_{add} \in OPERATORS \cup STEPS$ that has $c$ as an effect
    if $S_{add} = \{\}$ then FAIL
    end if
    add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS
    add the ordering constrain $S_{add} \prec S_{need}$ to ORDERINGS
    if $S_{add}$ is newly added step from operators then
        add $S_{add}$ to STEPS
        add $S_{start} \prec S_{add} \prec S_{goal}$ to ORDERINGS
    end if
end function
```

Algorithm 4 Partial Order Planning (POP)

```
function RESOLVE_THREATS(plan)
    for all $S_{threat}$ that threatens a link $S_i \xrightarrow{c} S_j \in$ LINKS do
        Promote: add $S_{threat} \prec S_i$ or Demote: add $S_j \prec S_{threat}$
        if ¬ CONSISTENT(plan) then FAIL
        end if
    end for
end function
```

Partially Instantiated Operators

- So far we have not mentioned anything about binding constraints.
- Should an operator that has the effect, say, $\neg At(x)$, be considered a threat to the condition $At(Home)$?

Dealing with possible threats:

- Resolve with an equality constraint,
  - bind x with something that resolves the threat (e.g. $x = TS$)
- Resolve with an inequality constrain.
  - Add constrain that x can not be bound to *Home*.
- Resolve later.
  - Ignore possible threats. If $x = Home$ is added later into the plan, try to resolve by promotion or demotion.

Algorithm 5 Partial Order Planning (POP)

```
function CHOOSE_OPERATOR($plan$, $OPERATORS$, $S_{need}$, $c$)
    choose s step $S_{add} \in OPERATORS \cup STEPS$ that has $c$ as an effect such that u = UNIFY(c,c',BINDINGS)
    if $S_{add} = \{\}$ then FAIL
    end if
    add $u$ to BINDINGS
    add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS
    add the ordering constrain $S_{add} \prec S_{need}$ to ORDERINGS
    if $S_{add}$ is newly added step from operators then
        add $S_{add}$ to STEPS
        add $S_{start} \prec S_{add} \prec S_{goal}$ to ORDERINGS
    end if
end function
```