CS370

Symbolic Programming

Declarative Programming

LECTURE 16: Planning

Jong C. Park
park@cs.kaist.ac.kr
Computer Science Department
Korea Advanced Institute of Science and Technology
http://nlp.kaist.ac.kr/~cs370

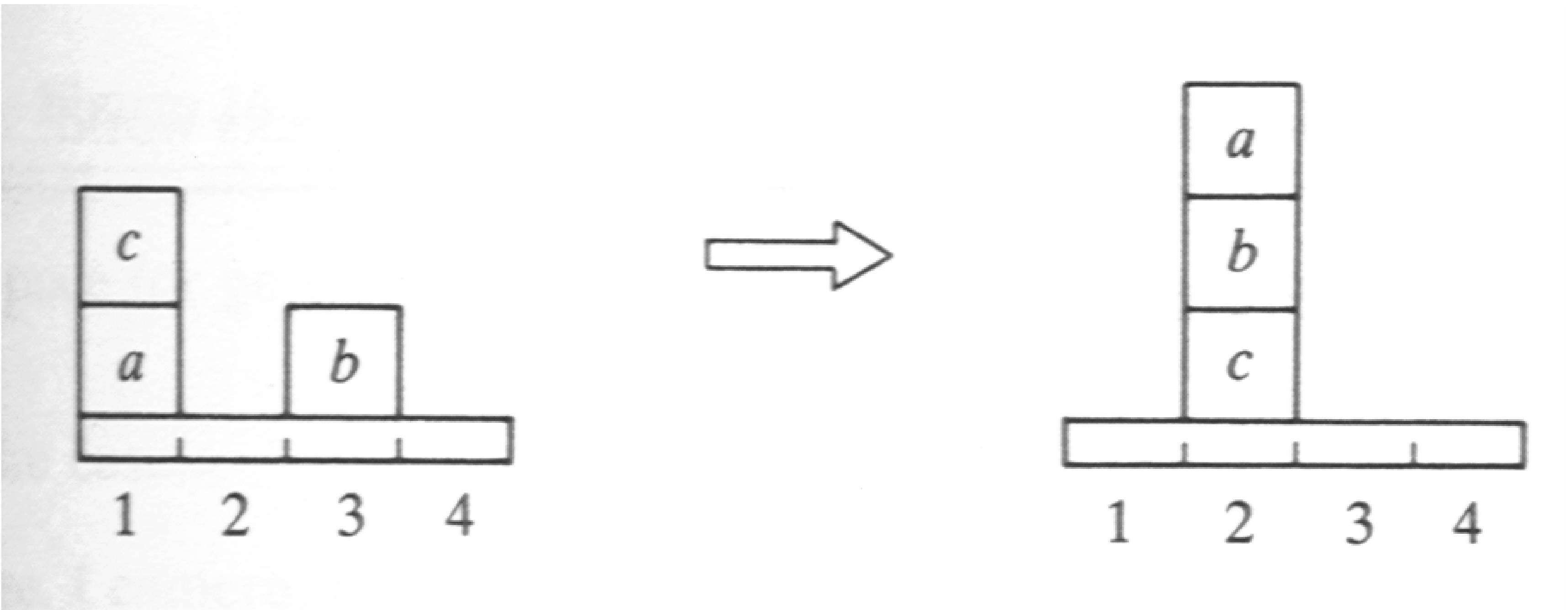
Planning

- Representing actions
- Deriving plans by means-ends analysis
- Protecting goals
- Procedural aspects and breadth-first regime
- Goal regression
- Combining means-ends planning with best-first heuristic
- Uninstantiated actions and partial-order planning

Representing actions

⊙ Planning task: an example (Figure 17.1)

[clear(2),clear(4),clear(b),clear(c),on(a,1),on(b,3),on(c,a)]



Representing actions

Specification of an action

- precondition: can(Action,Cond)
- add-list: add(Action,AddRels)
- delete-list: del(Action,DelRels)
- anything else?
  -

Type of action in the blocks world

- move(Block,From,To)

Representing actions

⊙A definition of the planning space

```
can(move(Block,From,To),
      [clear(Block), clear(To), on(Block,From)]) :-
   block(Block), object(To), To \== Block, object(From),
   From \== To, Block \== From.
adds(move(X,From,To),
       [on(X,To),clear(From)]).
deletes(move(X,From,To),
          [on(X,From),clear(To)].
object(X) :- place(X) ; block(X).
block(a).  block(b).      block(c).
place(1).  place(2).      place(3).      place(4).
state1([clear(2),clear(4),clear(b),clear(c),on(a,1),on(b,3)
   ,on(c,a)]).
```

```
c
a   b
= = = =
1 2 3 4
```

Representing actions

Camera Manipulation (Figure 17.3)

Opening the case	Rewinding film
Closing the case	Removing battery or film
Opening a slot	Inserting new battery or film
Closing a slot	Taking pictures

Initial State

[camera_in_case, slot_closed(film), slot_closed(battery), in(film), film_at_end, in(battery)]

Goal State

[in(film), film_at_start, film_unused, in(battery), ok(battery), slot_closed(film), slot_closed(battery)]

Deriving plans by means-ends analysis

The principle of means-ends planning

- To solve a list of goals **Goals** in state **State**, leading to state **FinalState**, do:

  If all the **Goals** are true in **State** then **FinalState** = **State**.

  Otherwise do the following steps.

  (1) Select a still unsolved goal **Goal** in **Goals**.

  (2) Find an action **Action** that adds **Goal** to the current state.

  (3) Enable **Action** by solving the precondition **Condition** of **Action**, giving **MidState1**.

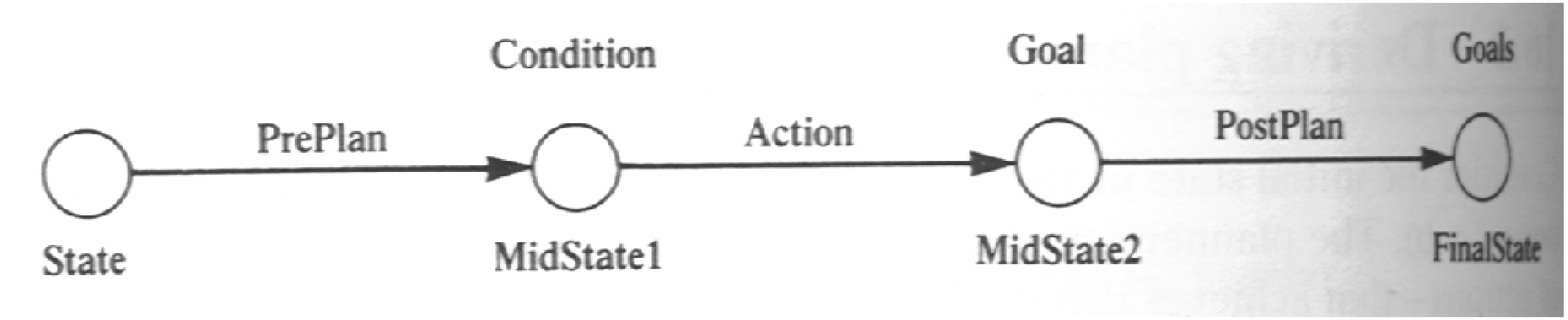
  (4) Apply **Action** to **MidState1**, giving **MidState2** (in **MidState2**, **Goal** is true).

  (5) Solve **Goals** in **MidState2**, leading to **FinalState**.

Deriving plans by means-ends analysis

The principle of means-ends planning

- select an unsolved goal: on(a,b)
- find an action that achieves it: move(a,From,b)
- enable the action by satisfying its precondition: [clear(a),clear(b),on(a,From)]
- find an action that achieves the relation clear(a): move(Block,a,To)



Deriving plans by means-ends analysis

⊙ Means-ends planner (Figure 17.5)

```
% plan(State,Goals,Plan,FinalState)
plan(State,Goals,[ ],State) :- satisfied(State,Goals).
plan(State,Goals,Plan,FinalState) :-
  conc(PrePlan,[Action|PostPlan],Plan),
  select(State,Goals,Goal),
  achieves(Action,Goal),
  can(Action,Condition),
  plan(State,Condition,PrePlan,MidState1),
  apply(MidState1,Action,MidState2),
  plan(MidState2,Goals,PostPlan,FinalState).
```

Deriving plans by means-ends analysis

Means-ends planner

```
?- Start =
   [clear(2),clear(4),clear(b),clear(c),on(a,1),on(b,3),
   on(c,a)], plan(Start, [on(a,b)], Plan, FinState).
Plan = [move(c,a,2),move(a,1,b)]
FinState =
   [on(a,b),clear(1),on(c,2),clear(a),clear(4),clear(c),on(b,
   3)]
?- Start =
   [camera_in_case, slot_closed(film), slot_closed(battery),
   in(film), film_at_end,in(battery)],
    plan(Start, [ok(battery)], FixBattery,_).
FixBattery = [open_case, open_slot(battery),
   remove(battery), insert_new(battery)]
```

Protecting goals

⦿ Strange behavior

```
?- plan(Start,[on(a,b),on(b,c)],Plan,_).
Plan = [move(b,3,c),
        move(b,c,3),
        move(c,a,2),
        move(a,1,b),
        move(a,b,1),
        move(b,3,c),
        move(a,1,b)]
```

```
c
a   b
= = = =
1 2 3 4
```

Protecting goals

⦿ Strange behavior

```
?- plan(Start,[clear(2),clear(3)],Plan,_).
     move(b,3,2)
     move(b,2,3)
     move(b,3,2)
     move(b,2,3)
     . . .
```

```
c
a   b
= = = =
1 2 3 4
```

Protecting goals

- **Protect the goals that are already achieved**

```
plan(InitialState,Goals,Plan,FinalState) :-
    plan(InitialState,Goals,[ ],Plan,FinalState).
plan(State,Goals,_,[ ],State) :- satisfied(State,Goals).
plan(State,Goals,Protected,Plan,FinalState) :-
    conc(PrePlan,[Action|PostPlan],Plan),
    select(State,Goals,Goal),
    achieves(Action,Goal), can(Action,Condition),
    preserves(Action,Protected),
    plan(State,Condition,Protected,PrePlan,MidState1),
    apply(MidState1,Action,MidState2),
   plan(MidState2,Goals,[Goal|Protected],PostPlan,FinalState).
```

Protecting goals

- **Protect the goals that are already achieved**

```
preserves(Action,Goals) :-
    deletes(Action,Relations),
    not (member(Goal,Relations),
           member(Goal,Goals)).
```

Procedural aspects and breadth-first regime

⦿ The search behavior

- globally depth-first w.r.t. action sequencing
- locally breadth-first w.r.t. preplan expansion

```
PrePlan = [ ];
PrePlan = [ _ ];
PrePlan = [ _, _ ];
PrePlan = [ _, _, _ ];
. . .
```

Procedural aspects and breadth-first regime

⊙Forcing into the breadth-first regime

- minimize the length of plans

```
plan(State,Goals,Plan,FinState) :-
  conc(Plan,_,_),
  conc(PrePlan,[Action|PostPlan],Plan),
  ...

plan(Start,[clear(2),clear(3)],Plan,_)
Plan = [move(b,3,4)]
```

Procedural aspects and breadth-first regime

⦿Forcing into the breadth-first regime

- Incompleteness

  plan(Start,[on(a,b),on(b,c)],Plan,_)

  move(c,a,2)
  move(b,3,a)
  move(b,a,c)
  move(a,1,b)

```
c
a   b
= = = =
1 2 3 4
```

- Why?
  - It does not suggest all relevant actions to the planning process. That is, the planner considers only those actions that pertain to the current goal and disregards other goals (locality).

  → We need to enable interaction between different goals.

Goal regression

⦿ Regressing goals through actions

- We are interested in a list of goals Goals being true in some state S.
- Question:
  - What goals Goals0 have to be true in S0 to make Goals true in S, where the action A leads state S0 to state S?

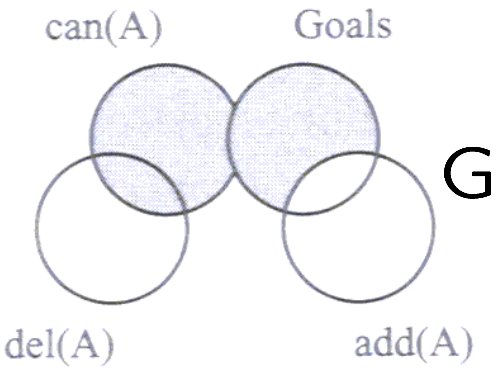
$$\text{state S0: Goals0} \xrightarrow{\quad\quad} \text{state S: Goals}$$

A

- Properties of Goals0
  - Action A must be possible in S0 (Goals0 must imply the precondition for action A).
  - For each goal G in Goals either action A adds G, or G is in Goals0 and A does not delete G.

⊙Regressing goals through actions

- Regressing Goals through action A
  - to determine Goals0 from given Goals and action A



Goals0 = can(A) ∪ Goals - add(A)

Figure 17.7 Relations between various sets of conditions in goal regression through action A. The shaded area represents the resulting regressed goals **Goals0**: **Goals0** = **can**(A) ∪ **Goals** − **add**(A). Notice that the intersection between **Goals** and the delete-list of A must be empty.

Goal regression

⦿ Goal regression for planning

- To achieve a list of goals Goals from some initial situation StartState, do:
  - If Goals are true in StartState then the empty plan suffices;
  - Otherwise select a goal G in Goals and an action A that adds G; then regress Goals through A obtaining NewGoals and find a plan for achieving NewGoals from StartState.

Goal regression

⦿ Goal regression for planning

- We use the relation below for compatible goals.
  - impossible(Goal, Goals)

```
impossible(on(X, X), _).
impossible(on(X, Y), Goals) :-
  member(clear(Y), Goals)
  ;
  member(on(X,Y1), Goals), Y1 \== Y
  ;
  member(on(X1, Y), Goals), X1 \== X.
impossible(clear(X), Goals) :-
  member(on(_, X), Goals).
```

Goal regression

⦿ Planner with goal regression (Figure 17.8)

```
plan(State,Goals,[ ]) :- satisfied(State,Goals).
plan(State, Goals, Plan) :-
    conc(PrePlan, [Action], Plan),
    select(State, Goals, Goal), achieves(Action, Goal),
    can(Action, Condition), preserves(Action, Goals),
    regress(Goals, Action, RegressedGoals),
    plan(State, RegressedGoals, PrePlan).
satisfied(State, Goals) :- delete_all(Goals, State, [ ]).
regress(Goals, Action, RegressedGoals) :-
    add(Action, NewRels), delete_all(Goals, NewRels,
    RestGoals),
    can(Action, Condition),
    addnew(Condition, RestGoals, RegressedGoals).
```

Combining means-ends planning with best-first heuristic

⦿ Guidance with domain-specific knowledge

- The top-most on relations should be achieved last.
- The selection of goals that are true in the initial state should be deferred.
- Alternative actions can be rated.
  - Some actions achieve several goals simultaneously, some action's precondition may be easier to satisfy.
- Continue working on the goal that looks easiest among the alternative regressed goal sets.

Combining means-ends planning with best-first heuristic

⦿ Making use of the best-first search program

- Define a successor relation between the nodes in the state space.
  - s(Node1,Node2,Cost)
- Define the goal nodes of the search by relation goal(Node)
- Define a heuristic function by relation h(Node, HeuristicEstimate)
- Define the start node of the search.

How would you do it?

Combining means-ends planning with best-first heuristic

Formulating the state space

- Make goal sets correspond to nodes in the state space.
- Define a link between two goal sets Goals1 and Goals2 if there is an action A such that
  - A adds some goal in Goals1,
  - A does not destroy any goal in Goals1, and
  - Goals2 is a result of regressing Goals1 through action A, as defined by the relation regress, as in regress(Goals1, A, Goals2).

Combining means-ends planning with best-first heuristic

⦿ Formulating the state space

- s(Goals1, Goals2, 1) :-
  member(Goal, Goals1),
  achieves(Action, Goal),
  can(Action, Condition),
  preserves(Action, Goals1),
  regress(Goals1, Action, Goals2).

⦿ The program finds a sequence of states, not actions.

[[clear(c),clear(2),on(c,a),clear(b),on(a,1)],
[clear(a),clear(b),on(a,1)], [on(a,b)]]

Combining means-ends planning with best-first heuristic

- **Instead, we represent nodes as pairs of the form Goals -> Action.**

```
s(Goals -> NextAction, NewGoals -> Action, 1) :-
    member(Goal, Goals), achieves(Action, Goal),
    can(Action, Condition), preserves(Action, Goals),
    regress(Goals, Action, NewGoals).
goal(Goals -> Action) :- start(State), satisfied(State,Goals).
h(Goals -> Action, H) :- start(State),
    delete_all(Goals, State, Unsatified),
    length(Unsatisfied, H).
```

Combining means-ends planning with best-first heuristic

⦿ Using the revised state-space definition

```
start([on(a,1), on(b,3), on(c,a), clear(b), clear(c), clear(2),
  clear(4)].

?- bestfirst([on(a,b), on(b,c)] -> stop, Plan).

Plan = [
  [clear(2),on(c,a),clear(c),on(b,3),clear(b),on(a,1)] ->
  move(c,a,2),
  [clear(c),on(b,3),clear(a),clear(b),on(a,1)] ->
  move(b,3,c),
  [clear(a),clear(b),on(a,1),on(b,c)] -> move(a,1,b),
  [on(a,b),on(b,c)] -> stop]
```

Uninstantiated actions and partial-order planning

⊙ Uninstantiated actions and goals

- All the goals for the planner should always be completely instantiated, and this may result in the generation of numerous irrelevant alternative moves.

```
can(move(Block,From,To),[clear(Block),clear(To),
     on(Block,From)])
  :- block(Block), object(To), ...
```

move(b,a,1), move(b,a,2), move(b,a,3), move(b,a,4), ...

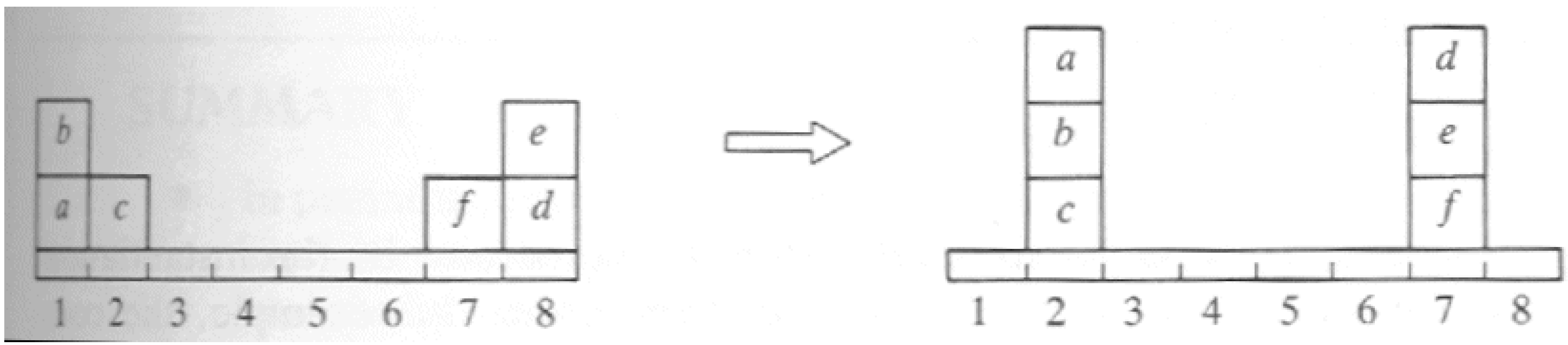
- Use uninstantiated actions and goals instead.

```
can(move(Block,From,To),[clear(Block),clear(To),
    on(Block,From)]).
```

move(Something,a,Somewhere)

Uninstantiated actions and partial-order planning

⦿ Partial-order planning



Summary

- Representing actions
- Deriving plans by means-ends analysis
- Protecting goals
- Procedural aspects and breadth-first regime
- Goal regression
- Combining means-ends planning with best-first heuristic
- Uninstantiated actions and partial-order planning