PLANNING

Ivan Bratko
University of Ljubljana

These slides are meant to be used with a Prolog system to demonstrate the examples, and the book: I. Bratko, Prolog Programming for Artificial Intelligence, 4th edn., Pearson Education 2011. The slides are not self-sufficient.

MEANS-ENDS PLANNING

- Problem of planning
  - Given:

    (1) possible actions in the world

    (2) start state of the world

    (3) goals to be achieved
  - Find:

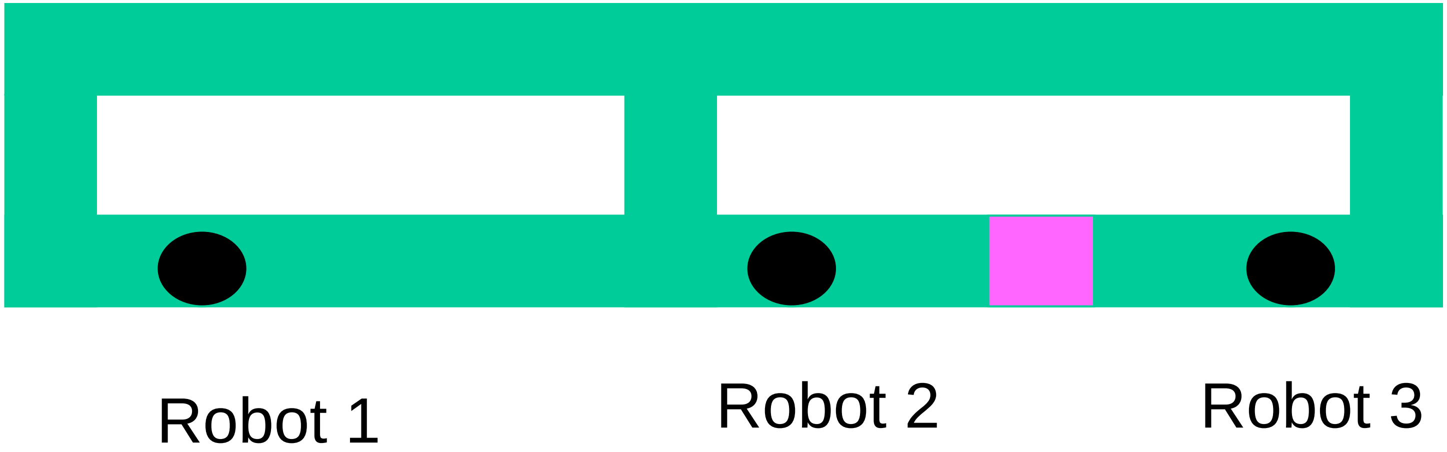
    A plan to achieve the goals

- Plan = sequence of actions, i.e. totally ordered set of actions
- Plan may also be *partially* ordered set of actions
- For a start, we consider total order planning

PLANNING BY MEANS-ENDS ANALYSIS

- Plans can be constructed by the familiar state-space search
- Alternatively, plans can be constructed through “means-ends analysis”
- In narrow sense, “planning” refers to means-ends planning
- Means-ends stands for:
  - ends ~ goals (goals of plan)
  - means ~ actions (actions the agent can perform)
- The planner reasons about what actions can possibly achieve what goals

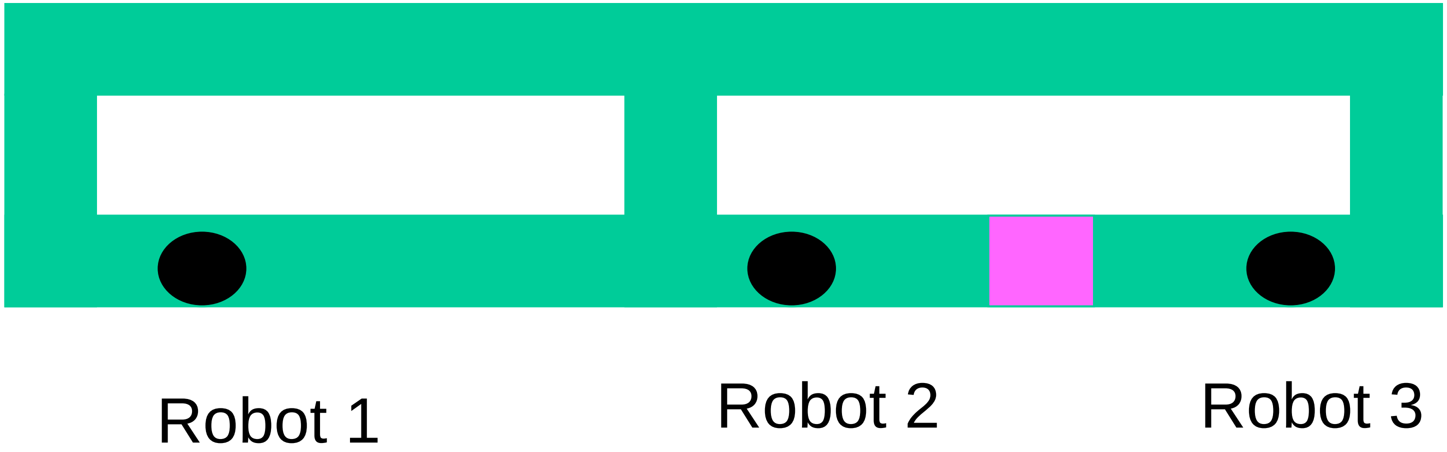
Example: mobile robots



Robots can move along green corridors

Task: Robot 1 wants to move into pink

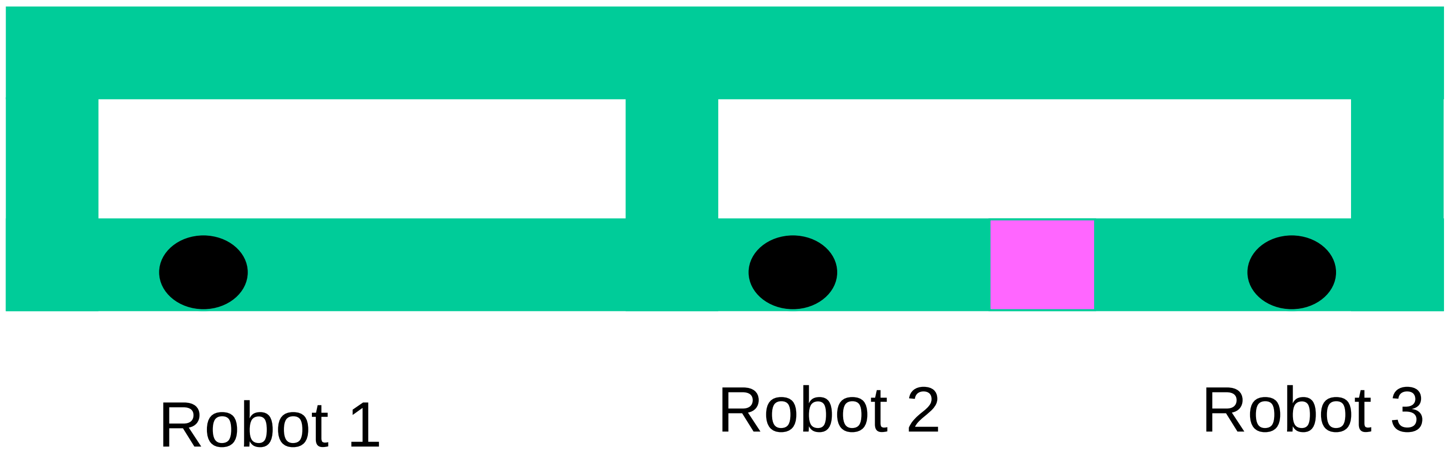
Solving with state-space



Task: Robot 1 wants to move into pink

Construct state-space graph:
states + successor relation between states

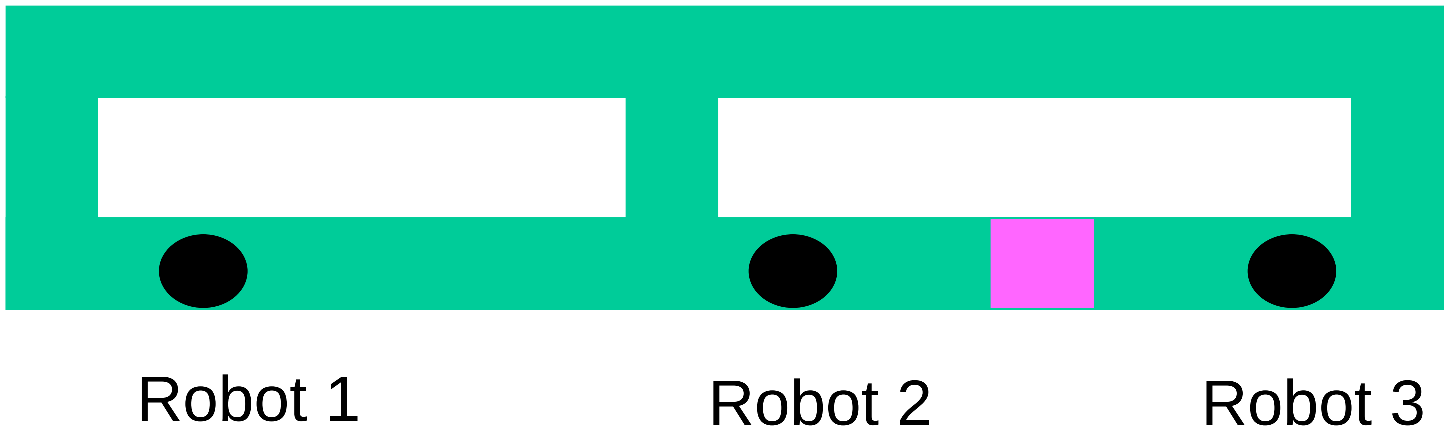
Solving by means-ends planner



Task: Robot 1 wants to move into pink

Formulate goal
Formulate actions in terms of preconditions and effects

Solving by means-ends planner



Means-ends reasoning may proceed like this:

First idea: Robot1 moves horizontally to “pink”
Next: Is this action possible?
No, action requires free path for Robot1 to pink
Next: How can I enable Robot1 move by making path free?
Now planner’s next subgoal is “Make horizontal path free”
Idea: Robot2 moves away from bottom horizontal path
Then Robot1 can move to pink, which completes the plan

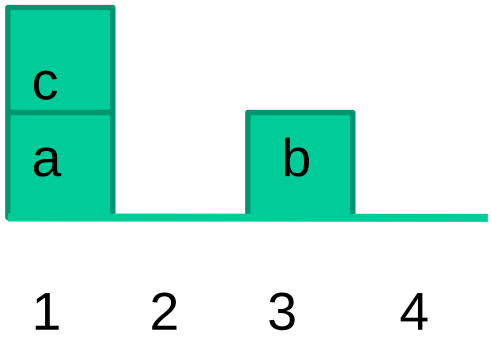
CLASSICAL PLANNING

- We consider the “classical planning” setting which assumes:
  - The world is completely observable
  - Actions’ effects are deterministic (completely predicateble, no uncertainty)
  - Any changes in the world only occur as results of agent’s actions, but not “on their own”
  - Implicit time: actions have no durations; time is only reflected in the order of actions

Representation

- How to represent a classical planning problem?
- Traditional, “STRIPS-like” representation, introduced by the STRIPS planner (Stanford Research Institute Problem Solver, 1970’s)

A BLOCKS WORLD PROBLEM



- Three blocks a, b, c; four locations 1, 2, 3, 4
- Relationships in initial state:
  on(c,a), on(a,1), on(b,3), clear(2), clear(4), clear(b), clear(c)
- Goal of plan e.g. build stack a, b, c
  Goals stated as: on(a,b), on(b,c)

Representing planning problems

- A goal:

  on(a,c)

- An action:

  move(a, b, c)

- Action's preconditions:

  clear(a), on(a,b), clear(c)

- Action's effects:

  on(a,c), clear(b), not on(a,b), not clear(c)

add (on(a,c), clear(b))

delete (not on(a,b), not clear(c))

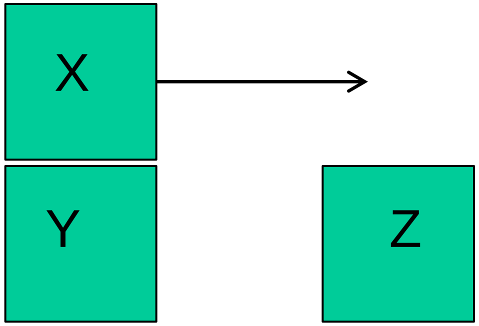
Action schema

- Represents a number of actions by using variables
- move(X, Y, Z)

  X stands for any block

  Y, Z stand for any block or location

BLOCKS WORLD:
STRIPS REPRESENTATION



Action: move(X, Y, Z)

Preconditions: on(X, Y), clear(X), clear(Y)

Add list: on(X, Z), clear(Y)

Delete list: on(X, Y), clear(Z)

BLOCKS WORLD: STRIPS-LIKE REPRESENTATION IN PROLOG

```
% can( Action, Condition): Action possible if Condition true

can( move( Block, From, To), [ clear( Block), clear( To), on( Block, From)] )
    :-
  block( Block),          % Block to be moved
  object( To),            % "To" is a block or a place
  To \== Block,           % Block cannot be moved to itself
  object( From),          % "From" is a block or a place
  From \== To,            % Move to new position
  Block \== From.         % Block not moved from itself
```

ADDS, DELETES

```
% adds( Action, Relationships): Action establishes Relationships

adds( move(X,From,To), [ on(X,To), clear(From)]).


% deletes( Action, Relationships): Action destroys Relationships

deletes( move(X,From,To), [ on(X,From), clear(To)]).
```

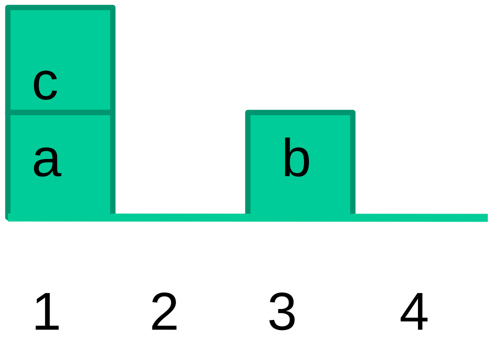
BLOCKS AND PLACES

```
object( X)  :-          % X is an objects if
  place( X)             % X is a place
  ;                     % or
  block( X).            % X is a block


% A blocks world

block( a).   block( b).   block( c).

place( 1).    place( 2).    place( 3).    place( 4).
```

A STATE IN BLOCKS WORLD

```
%  A state in the blocks world
%
%           c
%           a b
%           ====
%  place  1234

state1( [ clear(2), clear(4), clear(b), clear(c), on(a,1), on(b,3), on(c,a) ] ).
```

BLOCKS WORLD
MEANS-ENDS REASONING



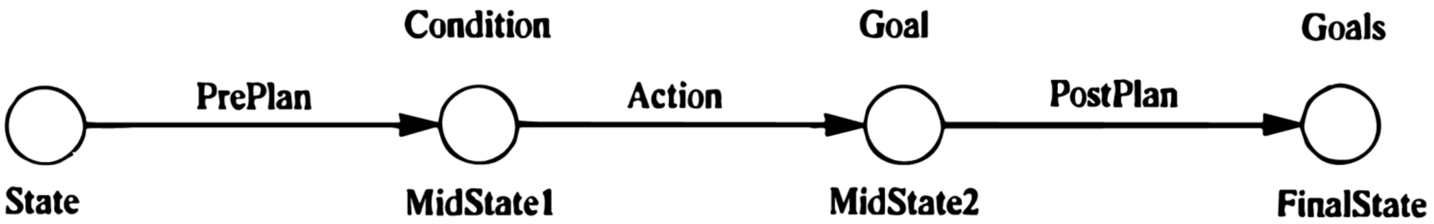
True in this state:

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

Let goal of plan be on(a,b); find a plan:

- Which action establishes on(a,b)? move(a,X,b)
  - What is the precondition COND for this move?
    - Set COND as intermediate goal, find plan to achieve COND
      - ...

MEANS-ENDS PLANNING: A FIRST IDEA



This can be easily translated into Prolog, next slide

A SIMPLE MEANS-ENDS PLANNER IN PROLOG

```
%   plan( State, Goals, Plan, FinalState)

plan( State, Goals, [], State)  :-
  satisfied( State, Goals).

plan( State, Goals, Plan, FinalState)  :-
  conc( PrePlan, [Action | PostPlan], Plan),        % Divide plan
  select( State, Goals, Goal),                       % Select a goal
  achieves( Action, Goal),                           % Relevant action
  can( Action, Condition),
  plan( State, Condition, PrePlan, MidState1),       % Enable Action
  apply( MidState1, Action, MidState2),              % Apply Action
  plan( MidState2, Goals, PostPlan, FinalState).     % Remaining goals
```

PROCEDURAL ASPECTS

```
% The way plan is decomposed into stages by conc, the
% precondition plan (PrePlan) is found in breadth-first
% fashion. However, the length of the rest of plan is not
% restricted and goals are achieved in depth-first style.

plan( State, Goals, Plan, FinalState) :-
  conc( PrePlan, [Action | PostPlan], Plan),            % Divide plan
  ...
  plan( State, Condition, PrePlan, MidState1),          % Breadth-first
  apply( MidState1, Action, MidState2),                 % Apply Action
  plan( MidState2, Goals, PostPlan, FinalState).        % Depth-first
```

PROCEDURAL ASPECTS: GENERATED PLANS CAN BE VERY AWKWARD

```
c
a  b
====
1234
```

?- start1(S), plan(S, [on(a,b), on(b,c)], P).

P = [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)]

This is far from shortest plan!

Try to explain how the planner found this

PROCEDURAL ASPECTS

- **conc(PrePlan, [Action | PostPlan], Plan)**
  enforces a strange combination of search strategies:

  1. Iterative deepening w.r.t. PrePlan
  2. Depth-first w.r.t. PostPlan

- We can force global iterative deepening by adding at front:
  conc(Plan, _, _)

A SIMPLE MEANS-ENDS PLANNER WITH ITERATIVE DEEPENING

```
%   plan( State, Goals, Plan, FinalState)

plan( State, Goals, [], State)  :-
  satisfied( State, Goals).

plan( State, Goals, Plan, FinalState)  :-
  conc( Plan, _, _),                              % Shortest plans first
  conc( PrePlan, [Action | PostPlan], Plan),      % Divide plan
  select( State, Goals, Goal),                    % Select a goal
  achieves( Action, Goal),                        % Relevant action
  can( Action, Condition),
  plan( State, Condition, PrePlan, MidState1),    % Breadth-first
  apply( MidState1, Action, MidState2),           % Apply Action
  plan( MidState2, Goals, PostPlan, FinalState).  % Breadth-first
```

```
?- start( S), plan( S, [on(a,b), on(b,c)], P).

P =
   [ move( c, a, 2),
     move( b, 3, a),
     move( b, a, c),
     move( a, 1, b) ]
```

- This is a surprise!
- This is still suboptimal, and quite mysterious!
- How can this be explained? How the second move got into the plan

PROBLEM WITH COMPLETENESS

- Even with global iterative deepening, our planner still has problems.
- E.g. it finds a four step plan for our example blocks task
- Why??? Incompleteness!
- Problem: *locality*; sometimes referred to as ‘linearity’

Planner keeps working myopicly on just one goal, and only when this is achieved, it starts working on a second goal. So it may fail to consider at all some useful actions

GOAL REGRESSION

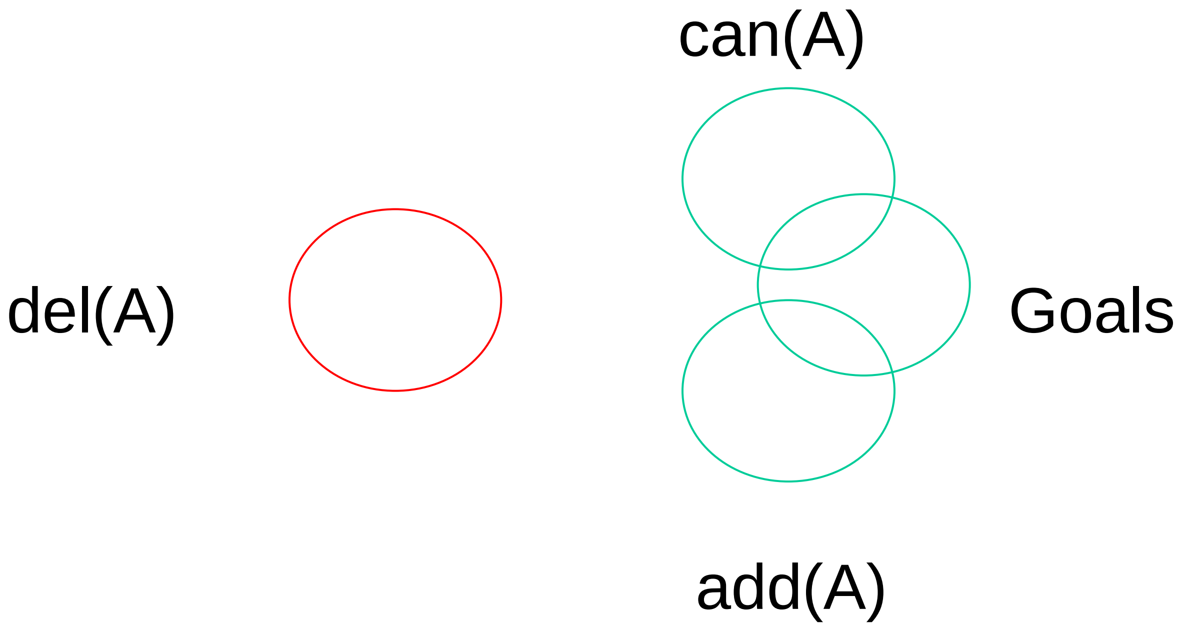
- Goal regression overcomes incompleteness; it achieves global planning
- Main mechanism: “Regressing Goals through Action”

RegressedGoals ——→ Goals

Action

- Given Goals and Actions, find RegressedGoals
- That is: what has to be true before Action so that Goals are true after Action?

GOAL REGRESSION



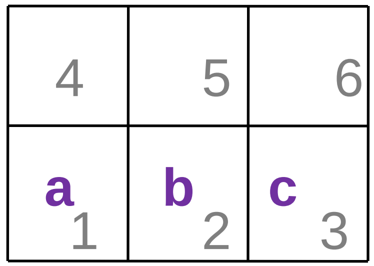
RegressedGoals = Goals + can(A) - add(A)

Goals and del(A) must be disjoint

GOAL REGRESSION ENABLES GLOBAL PLANNING

- It makes the planner consider *all* relevant actions at any point of planning

EXAMPLE: ROBOTS MOVING IN RECTANGULAR GRID



Robots a, b, c, cells 1, ..., 6

Goal: at(a,3)

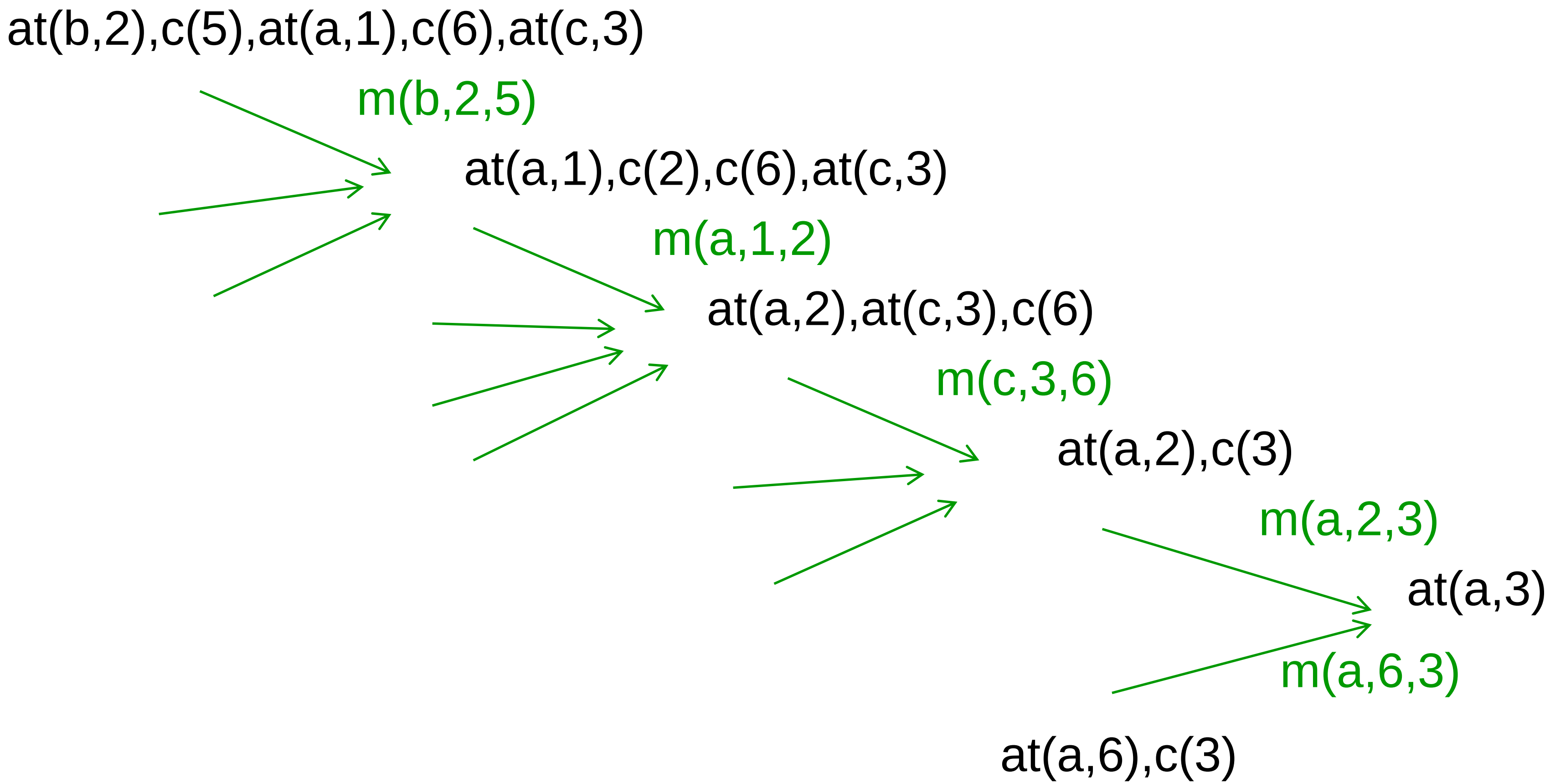
Plan: m(b,2,5), m(a,1,2), m(c,3,6), m(a,2,3)

DOMAIN DEFINITION

```
% m(R,A,B): robot R moves from cell A to cell B

can( m(R,A,B), [ at(R,A), c(B)])  :-
  robot(R), adjacent(A,B).

adds( m(R,A,B), [ at(R,B), c(A)]).

deletes( m(R,A,B), [ at(R,A), c(B)]).

adjacent( 1, 2).  adjacent( 2, 1).  adjacent( 1, 4).
...
```

FINDING PLAN FOR at(a,3)

Start state: at(a,1),at(b,2),at(c,3),c(4),c(5),c(6)



EXERCISE

- Demonstrate that plan of length 3 for achieving on(a,b) and on(b,c) in blocks world from our usual start state can be generated by the goal regression mechanism

A means-ends planner with goal regression in Prolog

```
%   plan( State, Goals, Plan)

plan( State, Goals, [ ])  :-
  satisfied( State, Goals).                 % Goals true in State
```

PLANNER WITH GOAL REGR. CTD.

```
%   plan( State, Goals, Plan)

plan( State, Goals, [ ])  :-
  satisfied( State, Goals).                % Goals true in State

plan( State, Goals, Plan)  :-
  conc( PrePlan, [Action], Plan),          % Enforce breadth-first effect
  select( State, Goals, Goal),             % Select a goal
  achieves( Action, Goal),
  can( Action, Condition),                 % Ensure Action contains no variables
  preserves( Action, Goals),               % Protect Goals
  regress( Goals, Action, RegressedGoals), % Regress Goals
  plan( State, RegressedGoals, PrePlan).
```

PLANNER WITH GOAL REGR. CTD.

```
preserves( Action, Goals)  :-         % Action does not destroy Goals
  deletes( Action, Relations),
  \+ ( member( Goal, Relations),
       member( Goal, Goals) ).
```

PLANNER WITH GOAL REGR. CTD.

```
regress( Goals, Action, RegressedGoals)  :-
        % Regress Goals through Action
    adds( Action, NewRelations),
    delete_all( Goals, NewRelations, RestGoals),
    can( Action, Condition),
    addnew( Condition, RestGoals, RegressedGoals).
                % Add precondition, check if RegressedGoals impossible
                % For example: on(a,b) and clear(b) is impossible
```

DOMAIN KNOWLEDGE

- At which places in this program domain-specific knowledge can be used?

- select(State, Goals, Goal)

  Which goal next (Last)?

- achieves(Action Goal)

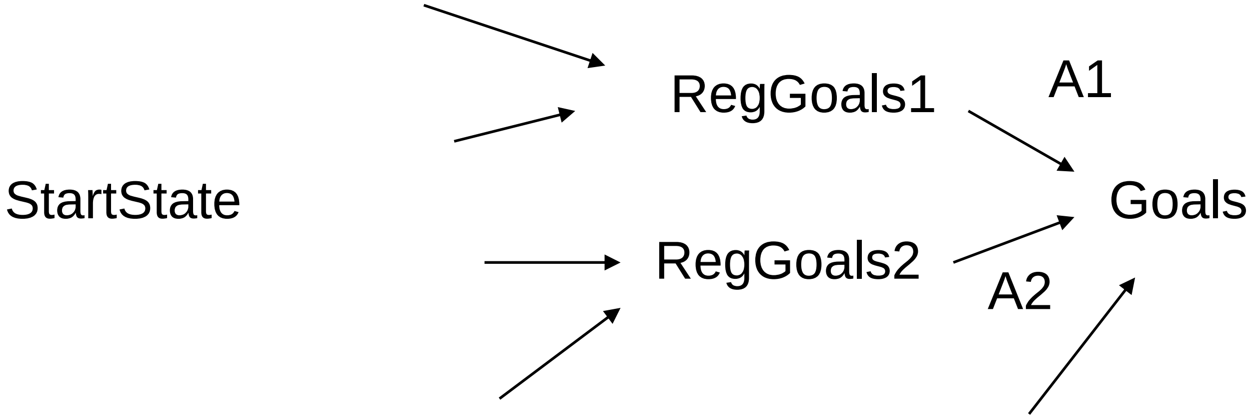
  Which action among those that achieve Goal

- impossible(Goal, Goals)

  Avoid impossible tasks

- Heuristic function h in state-space goal-regression planner?

STATE SPACE FOR PLANNING WITH GOAL REGRESSION



Begin with Goals, search towards StartState

SEARCHING SPACE OF SETS OF GOALS

- What are states in this “state space”? Sets of goals
- What is the goal condition? Goals in StartState
- Can we search with A*
- What could be a heuristic function?
- Maybe: h = | Goals – StartState |
- For the blocks world, does this h satisfy the admissibility condition from the admissibility theorem?

State space representation of means-ends planning with goal regression in Prolog

```
:- op( 300, xfy, ->).

s( Goals -> NextAction, NewGoals -> Action, 1)  :-
                                             % All costs are 1
  member( Goal, Goals),
  achieves( Action, Goal),                   % Action relevant to Goals
  can( Action, Condition),
  preserves( Action, Goals),
  regress( Goals, Action, NewGoals).
```

Goal state and heuristic

```
goal( Goals -> Action) :-
  start( State),                        % User-defined initial situation
  satisfied( State, Goals).             % Goals true in initial situation


h( Goals -> Action, H)  :-              % Heuristic estimate
  start( State),
  delete_all( Goals, State, Unsatisfied),   % Unsatisfied goals
  length( Unsatisfied, H).                  % Number of unsatisfied goals
```

QUESTION

- Does this heuristic function for the blocks world satisfy the condition of admissibility theorem for best-first search?

UNINSTANTIATED ACTIONS

- Our planner forces complete instantiation of actions:

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

MAY LEAD TO INEFFICIENCY

For example, to achieve clear(a):

move(Something, a, Somewhere)

Precondition for this is established by:

can(move(Something, ...), Condition)

This backtracks through 10 instantiations:

move(b, a, 1)

move(b, a, 3)

....

move(c, a, 1)

MORE EFFICIENT: UNINSTANTATED VARIABLES IN GOALS AND ACTIONS

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

Now variables remain uninstantiated:

[clear(Something), clear(Somewhere), on(Something,a)]

This is satisfied immediately in initial situation by instantiation:

Something = c,   Somewhere = 2

- Uninstantiated moves and goals stand for *sets* of moves and goals

- However, complications arise

  To prevent e.g. move(c,a,c) we need:

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

TREATING different(X,Y)

- Some conditions do not depend on state of world
- They cannot be achieved by actions
- Add new clause for satisfied/2:

```
satisfied( State, [Goal | Goals])  :-
    holds( Goal),
    satisfied( Goals).
```

Handling new type of conditions

holds(different(X, Y))

(1) If X, Y do not match then true.

(2) If X==Y then fail.

(3) Otherwise postpone decision until later (maintain list of postponed conditions; one way of implementing this is with CLP - Constraint Logic Programming)

Complications with uninstantiated actions

- Consider

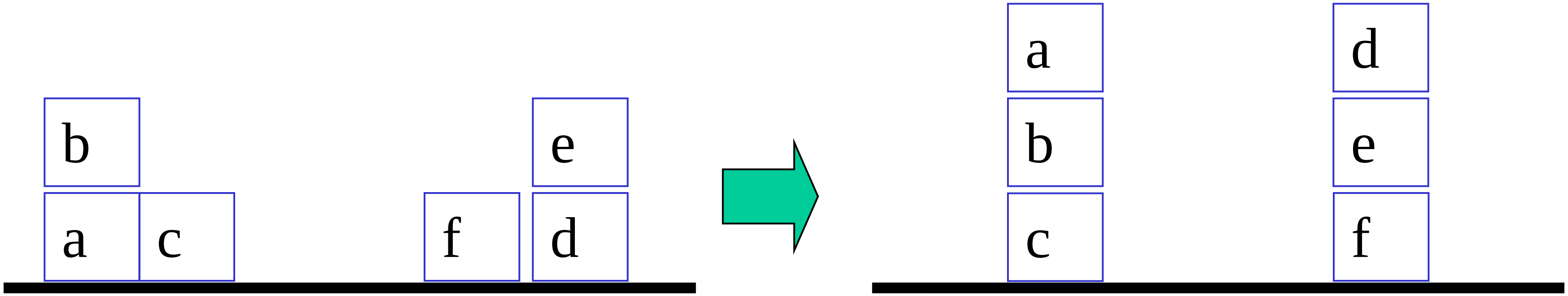
    move(a, 1, X)

- Does this delete clear(b)?
- Two alternatives:

(1) Yes if X=b

(2) No if different(X, b)

PARTIAL ORDER PLANNING



- The left group of three blocks can be solved independently of the right group
- This gives rise to partially ordered plan

PARTIAL ORDER PLAN

move(b, a, c) ⟶ move(a, table, b)

move(c, d, f) ⟶ move(d, table, c)

- The only ordering constraints are:

  move(b,a,c) is before move(a,table,b), and

  move(c,d,f) is before move(d,table,c)

- The execution of the top two actions can be interleaved in any order with bottom two actions; they can even be executed in parallel (e.g. by two robots)

PARTIAL ORDER PLANNING and NONLINEAR PLANNING

- Partial order planning is sometimes (problematically) called “nonlinear planning”
- May lead to ambiguity: nonlinear w.r.t. actions or goals
- Standard abbreviation: POP

POP ALGORITHM OUTLINE

- Search space of possible partial order plans (POP)
- Start plan is { Start, Finish}
- Start and Finish are virtual actions:
  - effect of Start is start state of the world
  - precondition of Finish is goals of plan

- Plan looks like this:

Start : StartState ⟶ ..... ⟶ Goals : Finish

PARTIAL ORDER PLAN

- Each POP consists of:
  - set of actions $\{A_i, A_j, ...\}$
  - set of ordering constraints e.g. $A_i < A_j$ (A_i before A_j)
  - set of *causal links*
- Causal links are of form

  causes(A_i, P, A_j)

  read as: A_i achieves P for A_j
- Example causal link:

  causes(move(c, a, 2), clear(a), move(a, 1, b))

CAUSAL LINKS AND CONFLICTS

- Causal link causes(A, P, B) “protects” P in interval between A and B
- Action C *conflicts with* causes(A, P, B) if C’s effect is ~P, that is deletes(C, P)
- Such conflicts are resolved by additional ordering constraints:

  C < A   or   B < C

  This ensures that C is outside interval A..B

PLAN CONSISTENT

- A plan is *consistent* if there is no cycle in the ordering constraints and no conflict

- E.g. a plan that contains A<B and B<A contains a cycle (therefore not consistent, obviously impossible to execute!)

- Property of consistent plans:

  Every linearisation of a consistent plan is a total-order solution whose execution from the start state will achieve the goals of the plan

SUCCESSOR RELATION BETWEEN POPs

A successor of a POP Plan is obtained as follows:

- Select an open precondition P of an action B in Plan (i.e. a precondition of B not achieved by any action in Plan)
- Find an action A that achieves P
- A may be an existing action in Plan, or a new action; if new then add A to Plan and constrain: Start < A, A < Finish
- Add to Plan causal link causes(A,P,B) and constraint A < B
- Add appropriate ordering constraints to resolve all conflicts between:
  - new causal link and all existing actions, and
  - A (if new) and existing causal links

SEARCHING A SPACE OF POPs

- POP with no open precondition is a solution to our planning problem
- Some interesting questions:
  - Heuristics for this search?
  - Means-ends planning for game playing?
- Heuristic estimates can be extracted from *planning graphs*; GRAPHPLAN is an algorithm for constructing planning graphs