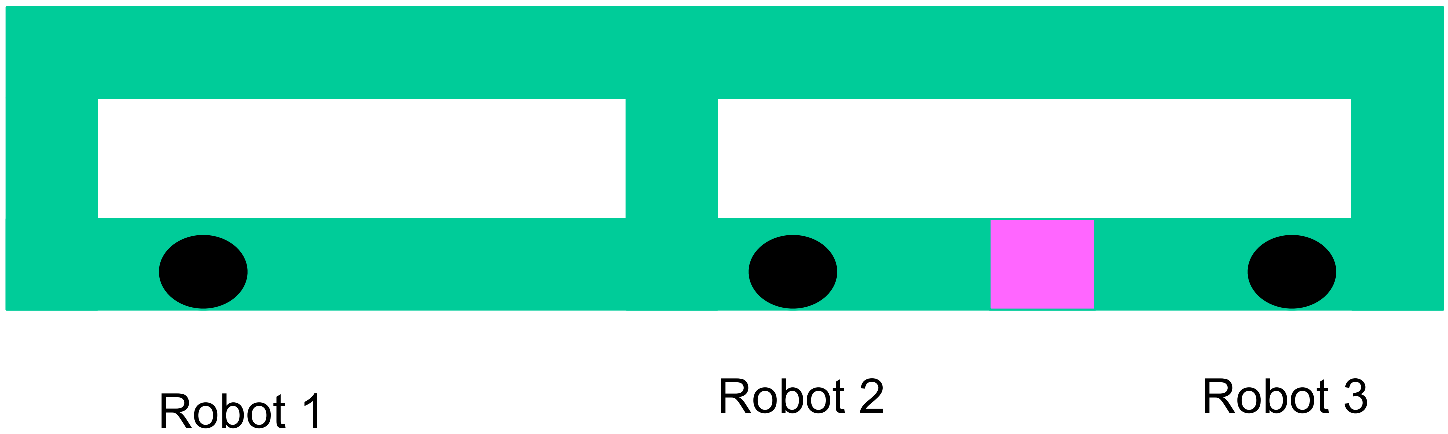
PLANNING

Ivan Bratko

Acknowledgement: Some of these slides were adapted from D. Nau's course on planning

Example: mobile robots

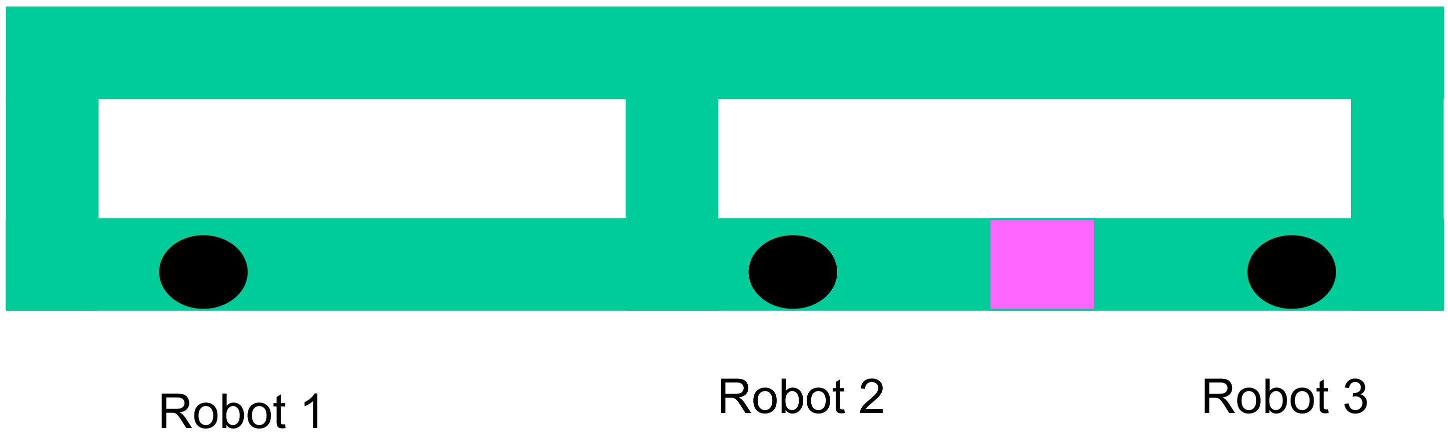


Task: Robot 1 wants to move into pink

How can plan be found with state-space search?

Means-ends planning avoids irrelevant actions

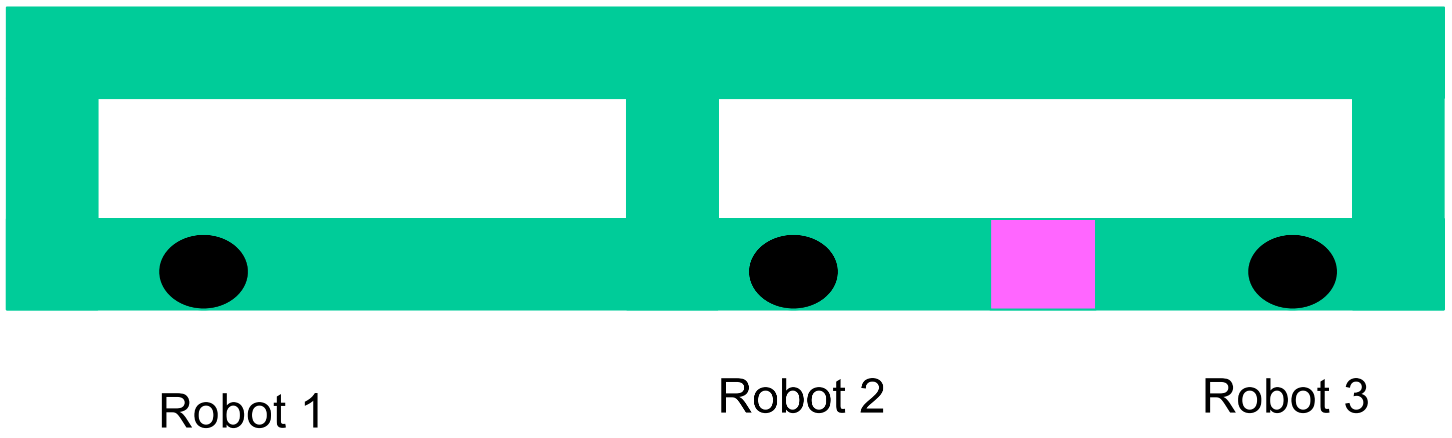
Solving with state-space



Task: Robot 1 wants to move into pink

Construct state-space search graph: states + successors

Solving by means-ends planner



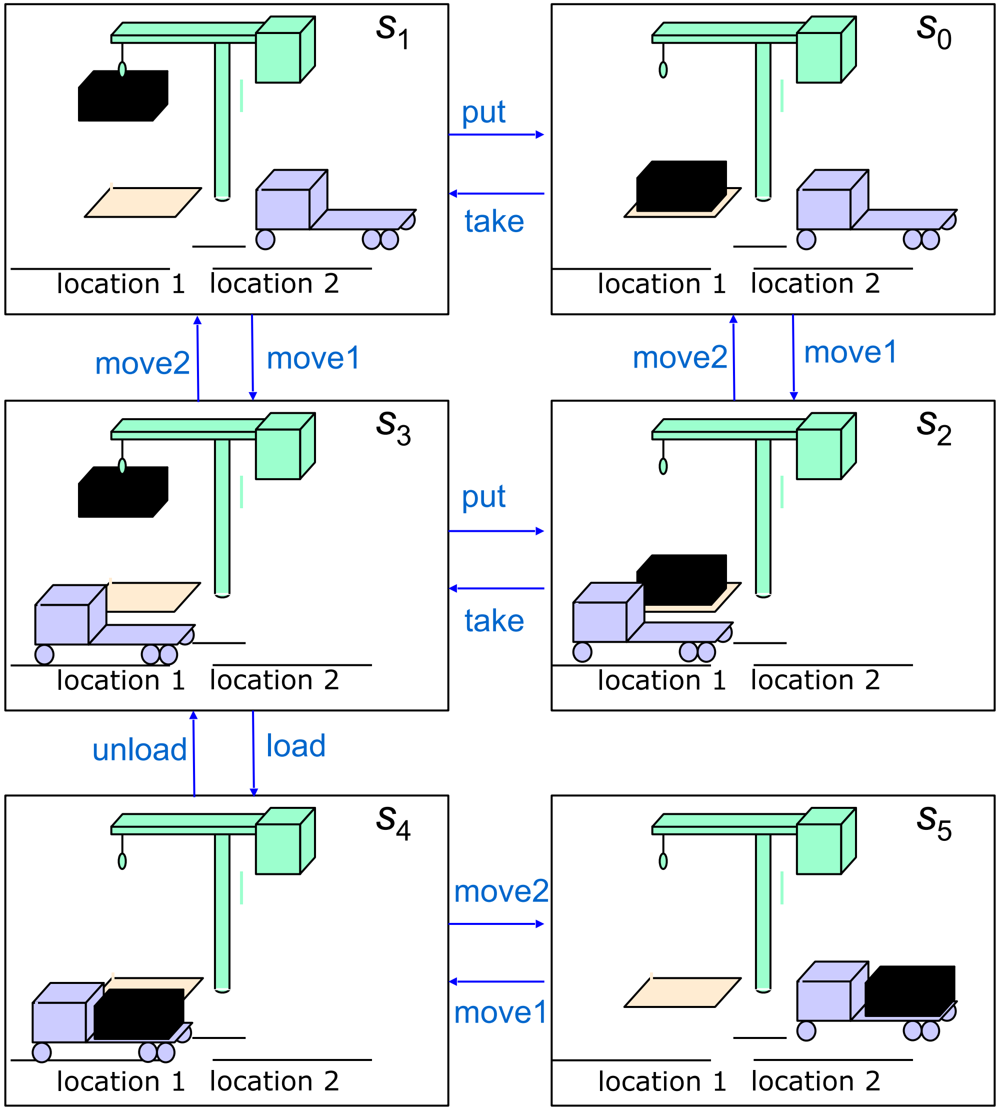
Task: Robot 1 wants to move into pink

Formulate goal
Formulate actions in terms of preconditions and effects

Conceptual Model: Example

State transition system
$\Sigma = (S,A,E,\gamma)$

- $S = \{s_1, s_2, \ldots\}$ is a set of *states*
- $A = \{a_1, a_2, \ldots\}$ is a set of (controllable) *actions*
- $E = \{e_1, e_2, \ldots\}$ is a set of (uncontrollable) *events*
- gamma: $S \times (A \cup E) \rightarrow 2^S$ is a *state-transition function*



From Nau

Restrictive Assumptions

- Classical Planning:
  - A0) Finite set of states S - no new objects in the domain
  - A1) Full observability (or no need of observations)
  - A2) Deterministic: $S \times (A \cup E) \rightarrow S$ - no uncertainty
  - A3) Static S: E is empty - no uncontrollable events
  - A4) Restricted goals: - set of final desired states
  - A5) Sequential Plans: - linearly ordered seqs
  - A6) Implicit time - no durations, no time constraints

$$S_g \subseteq S$$

$$\langle a_{1,} \ldots, a_n \rangle$$

From Nau

Relaxing the Assumptions

- Beyond Classical Planning
- Motivations:
  - A0: infinite number of states: new objects, numbers
  - A1: partial observability: non observable variables
  - A2: nondeterminism: unpredictable action outcomes
  - A2: uncontrollable events: not everything is controllable
  - A4: extended goals: conditions on the whole execution path
  - A5: beyond sequential plans: react to unpredictable outcomes
  - A6: explicit time: durations and deadlines

From Nau

Relaxing the Assumptions

- Beyond Classical Planning
- Issues:
  - A0: infinite number of states: undecidability
  - A1: partial observability: search in the set of sets of states
  - A2: nondeterminism: search with multiple paths
  - A2: uncontrollable events: environment or other agents
  - A4: extended goals: complexity of goals as temporal formulas
  - A5: beyond sequential plans: the problem is close to synthesis
  - A6: explicit time: need for a different representation

From Nau

Classical Planning Problems

- Suppose we impose *all* of the restrictive assumptions
- Then a planning problem is $P = (\Sigma, s_0, S_g)$
  - Σ is a finite state-transition system
    - each node is a state
    - each edge is a state transition
  - s_0 is the initial state
  - S_g is a set of goal states
- The objective is to find a sequence of state transitions that goes from s to a state in G

From Nau

Discussion

- Classical planning is a *very* restricted problem
  - Most real-world planning problems have properties that it doesn't capture

- However
  - Most AI planning research has focused on classical planning
    - e.g., in the AIPS-1998 and AIPS-2000 planning competitions, all of the problems were classical ones
  - Even in research on other kinds of planning, many of the concepts are based on those of classical planning
- How to represent a classical planning problem?

From Nau

Representing planning problems

- Goals:

  on(a,c)

- Actions:

  move(a, b, c)

- Action preconditions:

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

- Action effects:

  add *delete*

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

Action schemas

- Represents a number of actions by using variables

- move(X, Y, Z)

    X, Y, Z stand for any block

Problem language STRIPS

- STRIPS language, traditional representation
- STRIPS makes a number of simplifying assumptions like:
  - no variables in goals
  - unmentioned literals assumed false (c.w.a.)
  - positive literals in states only
  - effects are conjunctions of literals

ADL, Action Description Language

ADL removes some of the STRIPS assumptions, for example:

STRIPS	ADL
States: + literals only	
on(a,b), clear(a)	on(a,b), clear(a), ~clear(b)
Effects: + literals only	
Add clear(b), Delete clear(c)	Add clear(b) and ~clear(c), Delete ~clear(b) and clear(c)
Goals: no variables	
on(a,c), clear(a)	Exists X: on(X,c), clear(X)

A MOBILE ROBOTS WORLD

```
% Planning problem for moving robots in a space of places.
% Space is defined as a directed graph by predicate link(Place1,Place2).

can( move(Robot,Place1,Place2), [at(Robot,Place1),clear(Place2)])  :-
  link( Place1, Place2).        % Premik mozen samo v dani smeri!

adds( move(Robot,Place1,Place2), [at(Robot,Place2), clear(Place1)]).

deletes( move(Robot,Place1,Place2), [at(Robot,Place1), clear(Place2)]).
```

MOBILE ROBOTS WORLD CTD.

```
% A robot space; places are a, b, c, and d

link(a,b).  link(b,c).  link(c,d).  link(c,a).  link(d,a).


% A state with three robots r1, r2, r3 in this space

state0( [ at(r1,a), at(r2,b), at(r3,d), clear(c)]).
```

BLOCKS WORLD

```
% 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)]).
```

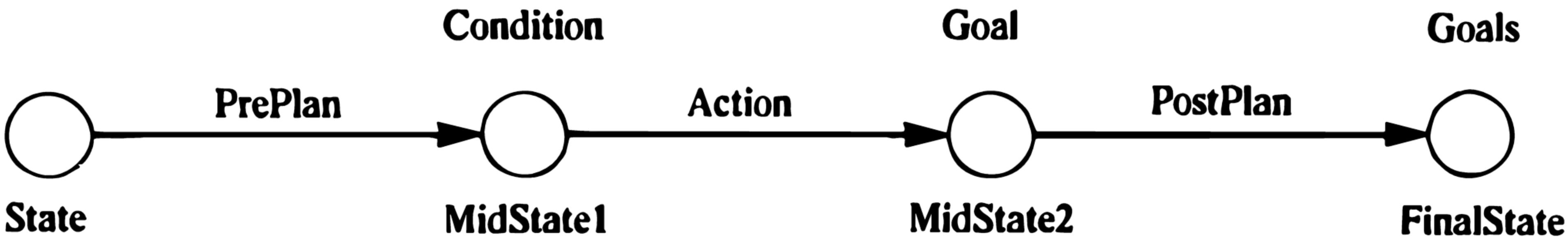
BLOCKS AND PLACES

```
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) ] ).
```

MEANS-ENDS PLANNING



A SIMPLE MEANS-ENDS PLANNER

```
%   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
```

% For definition of select, achieves, apply see Bratko, Prolog Programming for AI, 2nd ed., Chapter 17

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
  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

```
?- start( 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)]
```

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

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

The plan found by the planner is:

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

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)]

This plan containing seven steps is not exactly the most elegant! The shortest possible plan for this task only needs three steps. Let us analyze why our planner needs so many.

Planner pursues different goals at different stages of planning

move(b, 3, c)	to achieve goal **on(b,c)**
move(b, c, 3)	to achieve **clear(c)** to enable next move
move(c, a, 2)	to achieve **clear(a)** to enable **move(a,1,b)**
move(a, 1, b)	to achieve goal **on(a, b)**
move(a, b, 1)	to achieve **clear(b)** to enable **move(b,3,c)**
move(b, 3, c)	to achieve goal **on(b,c)** (again)
move(a, 1, b)	to achieve goal **on(a,b)** (again)

Planner sometimes destroys goals that have already been achieved. The planner achieved **on(b, c)**, but then destroyed it immediately when it started to work on the other goal **on(a, b)**. Then it attempted the goal **on(b, c)** again. This was re-achieved in two moves, but **on(a, b)** was destroyed in the meantime. Luckily, **on(a, b)** was then re-achieved without destroying **on(b, c)** again.

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, _, _)

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’

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

produces the plan:

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

Two questions:

1. what reasoning led the planner to construct the funny plan above?
2. why did the planner not find the optimal plan in which the mysterious **move(b, 3, a)** is not included?

OBSERVATIONS

First question: How did plan come about?

The last move, **move(a, 1, b)**, achieves the goal **on(a, b)**.
The first three moves achieve the precondition for **move(a, 1, b)**, in particular the condition **clear(a)**.
The third move clears *a*, and part of the precondition for the third move is **on(b, a)**.
This is achieved by the second move, **move(b, 3, a)**.
The first move clears *a* to enable the second move.

Second question:
Why after **move(c, a, 2)** did the planner not immediately consider **move(b, 3, c)**, which leads to the optimal plan?

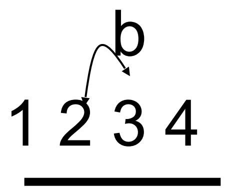
The reason is that the planner was working on the goal **on(a, b)** all the time.
The action **move(b, 3, c)** is completely superfluous to this goal and hence not tried.
Our four-step plan achieves **on(a, b)** and, *by chance*, also **on(b, c)**.
So **on(b, c)** is a result of pure luck and not of any “conscious” effort by the planner.

GOAL PROTECTION

- Try this:

?- start(S), plan(S, [clear(2), clear(3)], Plan).

- Planner repeats achieving goals and destroying them:



GOAL PROTECTION

```
plan( InitialState, Goals, Plan, FinalState)  :-
  plan( InitialState, Goals, [ ], Plan, FinalState).

%   plan( InitialState, Goals, ProtectedGoals, Plan, FinalState):
%     Goals true in FinalState, ProtectedGoals never destroyed by Plan

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

GOAL PROTECTION, CTD.

```
plan( State, Goals, Protected, Plan, FinalState)  :-
  conc( PrePlan, [Action | PostPlan], Plan),     % Divide plan
  select( State, Goals, Goal),                    % Select an unsatisfied goal
  achieves( Action, Goal),
  can( Action, Condition),
  preserves( Action, Protected),                  % Do not destroy protected goals
  plan( State, Condition, Protected, PrePlan, MidState1),
  apply( MidState1, Action, MidState2),
  plan( MidState2, Goals, [Goal | Protected], PostPlan, FinalState).
```

GOAL PROTECTION, CTD.

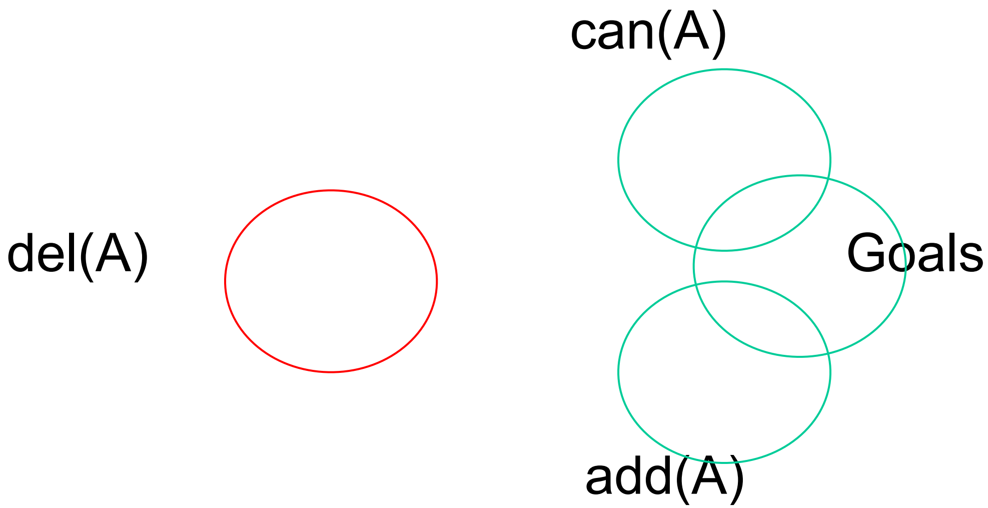
```
% preserves( Action, Goals): Action does not destroy any
    one of Goals

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

GOAL REGRESSION

- Idea to achieve global planning
- “Regressing Goals through Action”

$$\text{RegressedGoals} \xrightarrow[\text{Action}]{} \text{Goals}$$

GOAL REGRESSION



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

Goals and del(A) are disjoint

A means-ends planner with goal regression

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

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

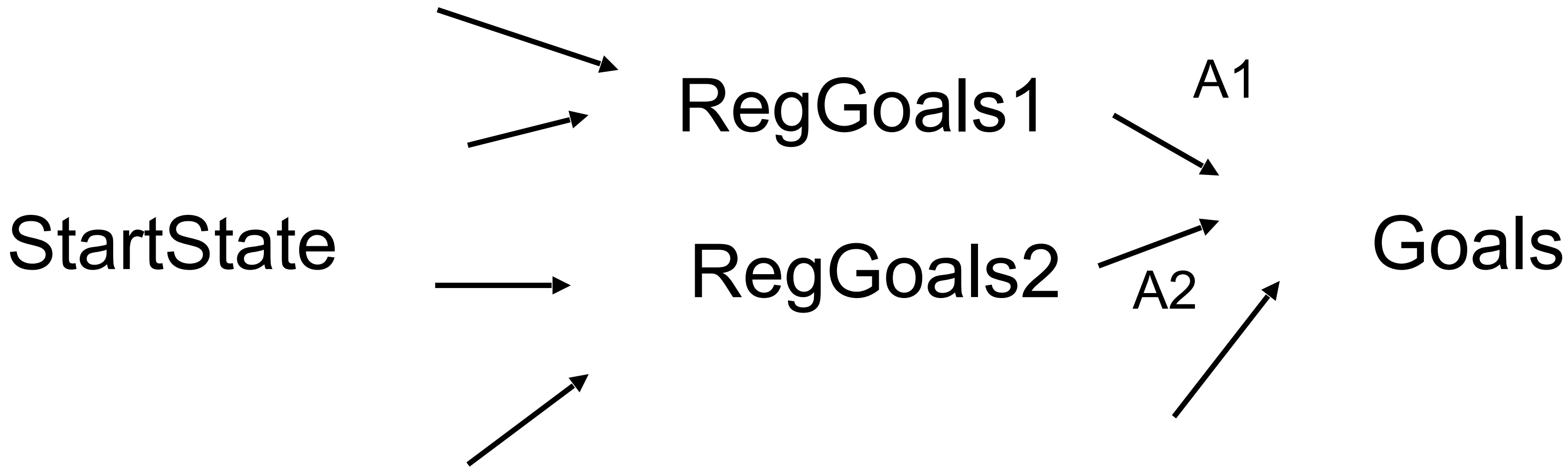
PLANNER WITH GOAL REGR. CTD.

```
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).
```

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)
- impossible(Goal, Goals)
  Avoid impossible tasks
- Heuristic function h in state-space goal-regression planner

STATE SPACE FOR PLANNING WITH GOAL REGRESSION



State space representation of means-ends planning with goal regression

```
:- 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:

Something = c,   Somewhere = 2

MORE EFFICIENT: UNINSTANTATED VARIABLES IN GOALS AND ACTIONS

- 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; cf. CLP)

Complications with uninstantiated actions

- Consider

  move(a, 1, X)

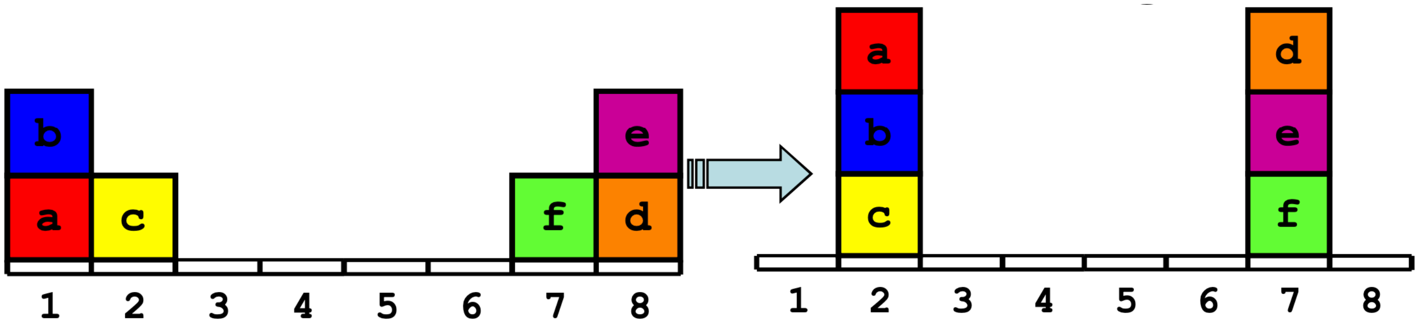
- Does this delete clear(b)?

- Two alternatives:

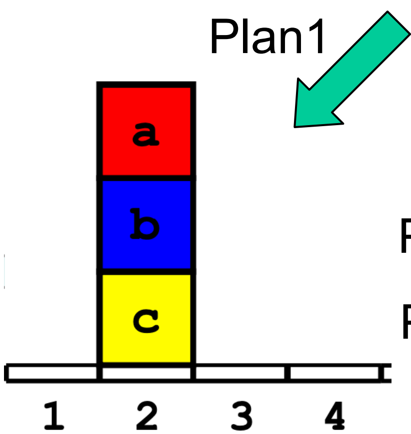
  (1) Yes if X=b

  (2) No if different(X, b)

PARTIAL ORDER PLANNING

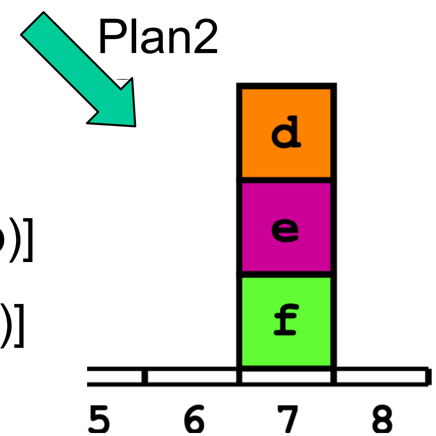


Linear Plan = [move(b, a, c), move(e, d, f), move(d, 8, e), move(a, 1, b)]



Plan1 = [move(b, a, c), move(a, 1, b)]

Plan2 = [move(e, d, f), move(d, 8, e)]



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

POP ALGORITHM OUTLINE

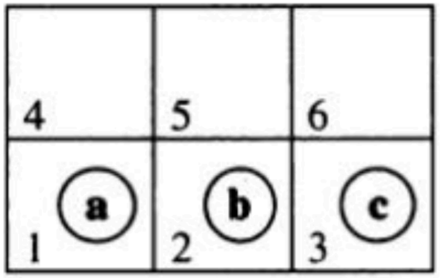
- Search space of possible partial order plans (POP)
- Start plan is { Start, Finish}
- Start and Finish are virtual actions:
  - effect of Start is start state of the world
  - precond. of Finish is goals of plan

true :: Start:: StartState → ..... → Goals :: Finish

PARTIAL ORDER PLAN

Each POP is:

- 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 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

GRAPHPLAN METHOD

- Uses "planning graphs"
- A planning graph enables the computation of planning heuristics
- A planning graph consists of levels, each level contains literals and actions
- These are roughly:
  - Literals that could be true at this level
  - Actions that that could have their preconditions satisfied
- Planning graphs only work for propositional representation (no variables in actions and literals)

CONSTRUCTIONG A PLANNING GRAPH

- Start with level S0 (conditions true in start state)
- Next level, A0: actions that have their preconditions satisfied in previous level
- Also include virtual “persistence” actions (that just preserve literals of previous level)

- Result is a *planning graph* where state levels S_i and action levels A_i are interleaved
- Ai contains all actions that are executable in S_i, and *mutex* constraints between actions
- Si contains all literals that could result from any possible choice of actions in A_{i-1} plus mutex constraints between literals

MUTEX CONSTRAINTS

- Mutual exclusion constraints = “mutex” constraints.
- E.g. move(a,1,b) and move(a,1,b) are mutex
- E.g. clear(a) and ~clear(a) are mutex

MUTEX RELATION BETWEEN ACTIONS

- Mutex relation holds between two actions at A and B the same level if:
  - A negates an effect of the other (“inconsistent effects”)
  - An effect A is the negation of a precondition of B (“interference”)
  - A precondition of A is mutex with a precondition of B (“competing needs”)

MUTEX RELATION BETWEEN LITERALS

- Mutex relation holds between two literals L1 and L2 of the same level if:
  - L1 is negation of L2, or
  - Each possible pair of actions that could achieve is mutually exclusive

USES OF PLANNING GRAPHS

- A planning graph enables the computation of heuristic estimates of how many time steps (levels) will at least be needed to solve the problem (length of plan)

- A plan may be extracted directly from a planning graph by the GRAPHPLAN algorithm

GRAPHPLAN algorithm

- Given:
  - PROBLEM (a planning problem)
  - GOALS = goals of PROBLEM

```
GRAPH := initial_planning_graph( PROBLEM)
SOLUTION = false; SOLUTION_IMPOSSIBLE := false;
 While SOLUTION = false and SOLUTION_IMPOSSIBLE = false do
    if GOALS all non-mutex in last level of GRAPH
         then SOLUTION := extract_solution( GRAPH, GOALS)
         else SOLUTION_IMPOSSIBLE := no_solution( GRAPH);
     if SOLUTION_IMPOSSIBLE = false
         then GRAPH := expand_graph( GRAPH, PROBLEM)
```

EXTRACTING PLAN FROM GRAPH

- One possibility:
  - view plan extraction problem as a binary CSP problem (variables correspond to actions in graph, domains are {in_plan, not_in_plan}
- Another possibility:
  - View plan extraction as a state-space problem
  - States correspond to subsets of (non-exclusive) literals at a level
  - Start at last level, end at 0-level
  - Move from higher to lower levels by a kind of goal regression (regressing all literals)

PLANNING SAFE PATHS

- A problem often considered related to planning:
  Find *safe paths* for robot among obstacles
  (obstacle avoidance)

- Usual approach is:

  1. Transform the problem:
     - shrink moving object to a point
     - correspondingly enlarge obstacles
  2. Construct visibility graph
  3. Search visibility graph with, say, A*