

PLANNING

CHAPTER 11

Outline

- ◇ Planning
- ◇ Classical planning
- ◇ Representation of planning problems
- ◇ State-space planning
- ◇ Plan-space planning
- ◇ Graph-based planning
- ◇ Planning as satisfiability

Planning

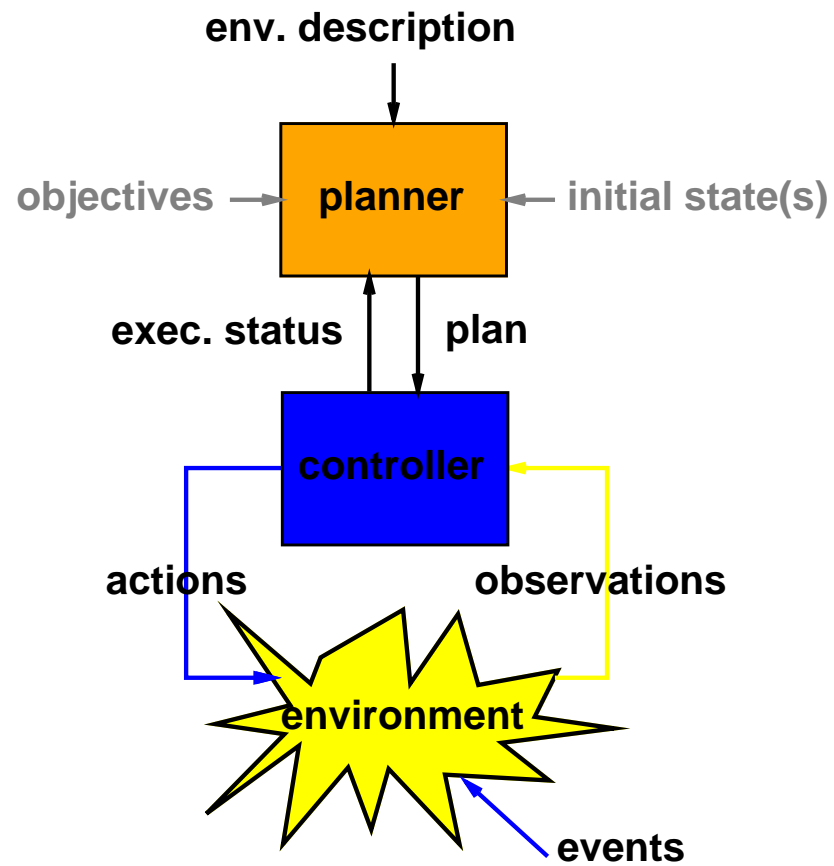
“Planning is the reasoning side of acting. It is an explicit deliberation process that chooses and organises actions, on the basis of their expected outcomes, in order to achieve some objective as best as possible.” [Ghallab et. al., 2003]

Application examples:

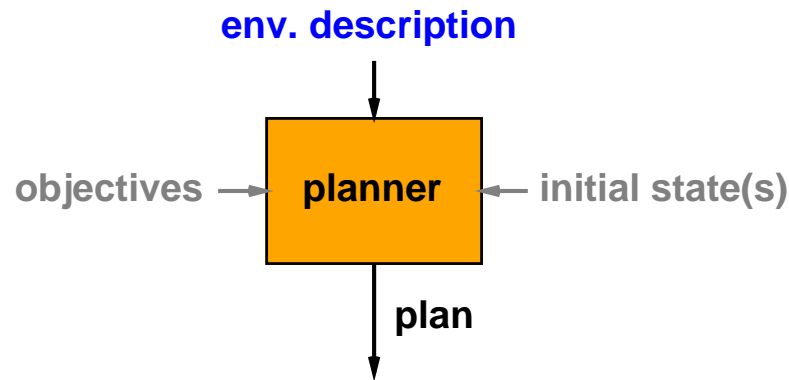
- spacecraft flying (NASA)
- Mars rover control (NASA)
- power supply restoration (EDF)
- elevator control (Shindler, Rockwell)
- sheet-metal bending (Amada)
- operations planning (DSTO-NICTA)
- bridge playing (University of Maryland)
- computer games (e.g., achieve more realistic NPC behaviour)



Planning agents



Domain-independent planning

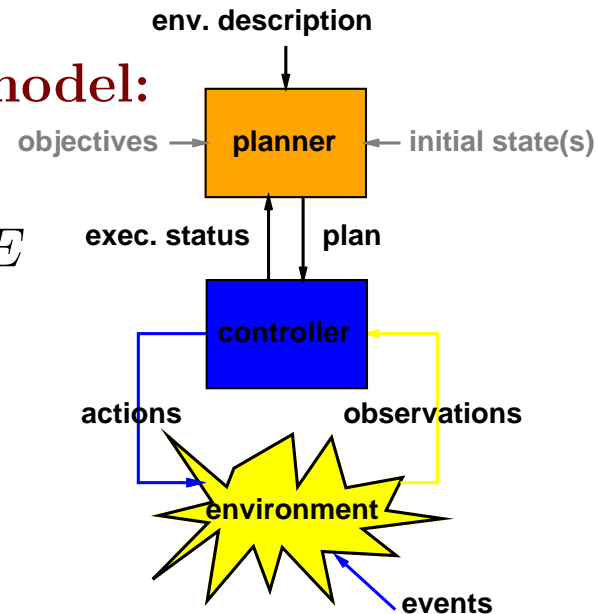


We often seek to build domain independent planners taking both the environment description and the problem description (initial state, objectives) as input. These work for an entire class of **planning models**.

Planning models

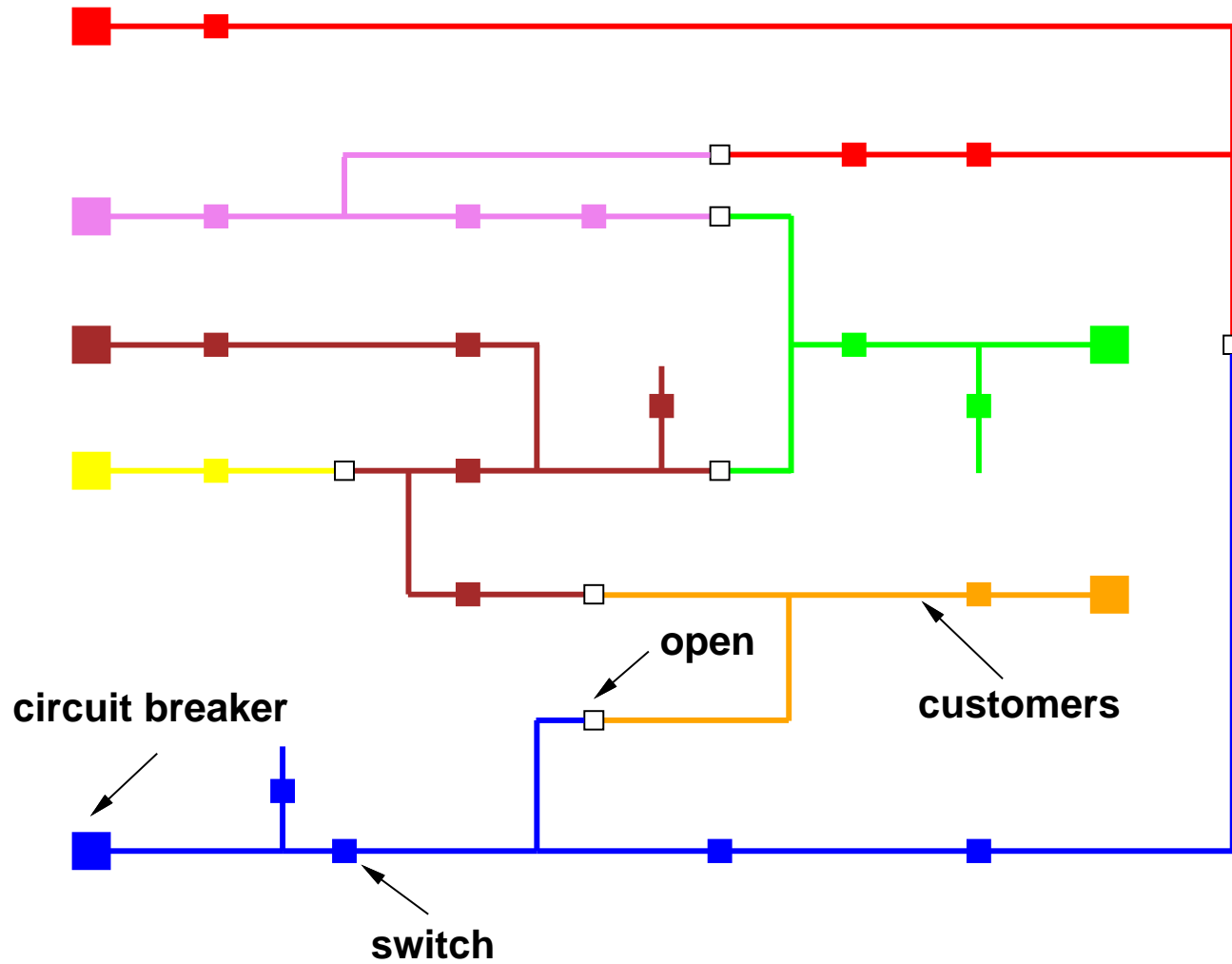
- **Typical ingredients of a planning model:**

- a set of states S
- a set of events E , a set of actions $A \subseteq E$
- a transition function $\gamma : S \times E \mapsto 2^S$
- a set of observations O
- an observation function $\lambda : S \mapsto 2^O$
- a set of possible initial states $S_I \subseteq S$
- conditions defining acceptable execution sequences
- a cost function on acceptable execution sequences

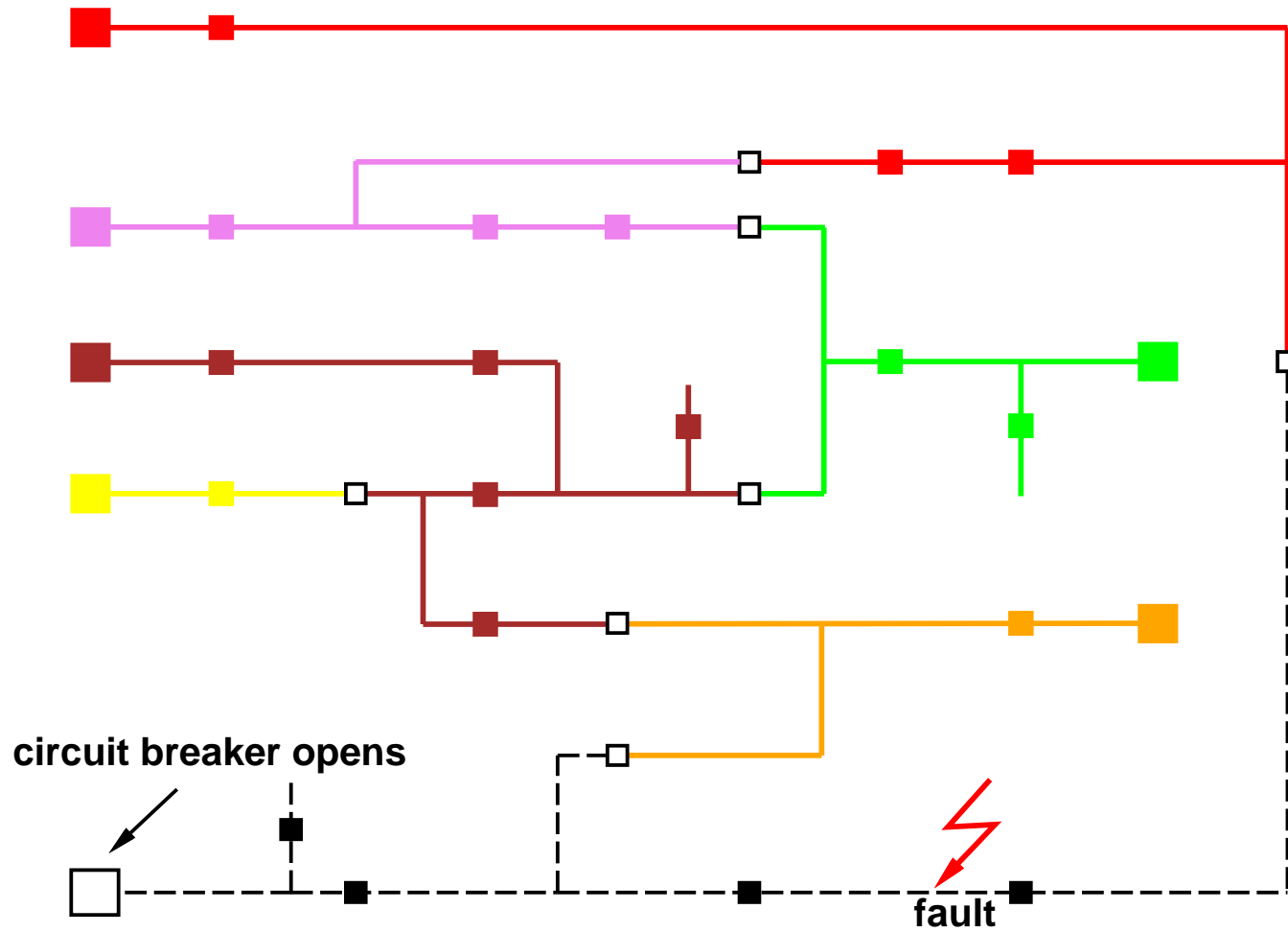


- **The problem** is to control the environment by performing actions so as to allow only acceptable execution sequences, of minimum cost
- **A solution** is a plan which can be (partially or totally ordered) set of actions, a policy $S \mapsto A$ or $O \mapsto A$, an automaton ...

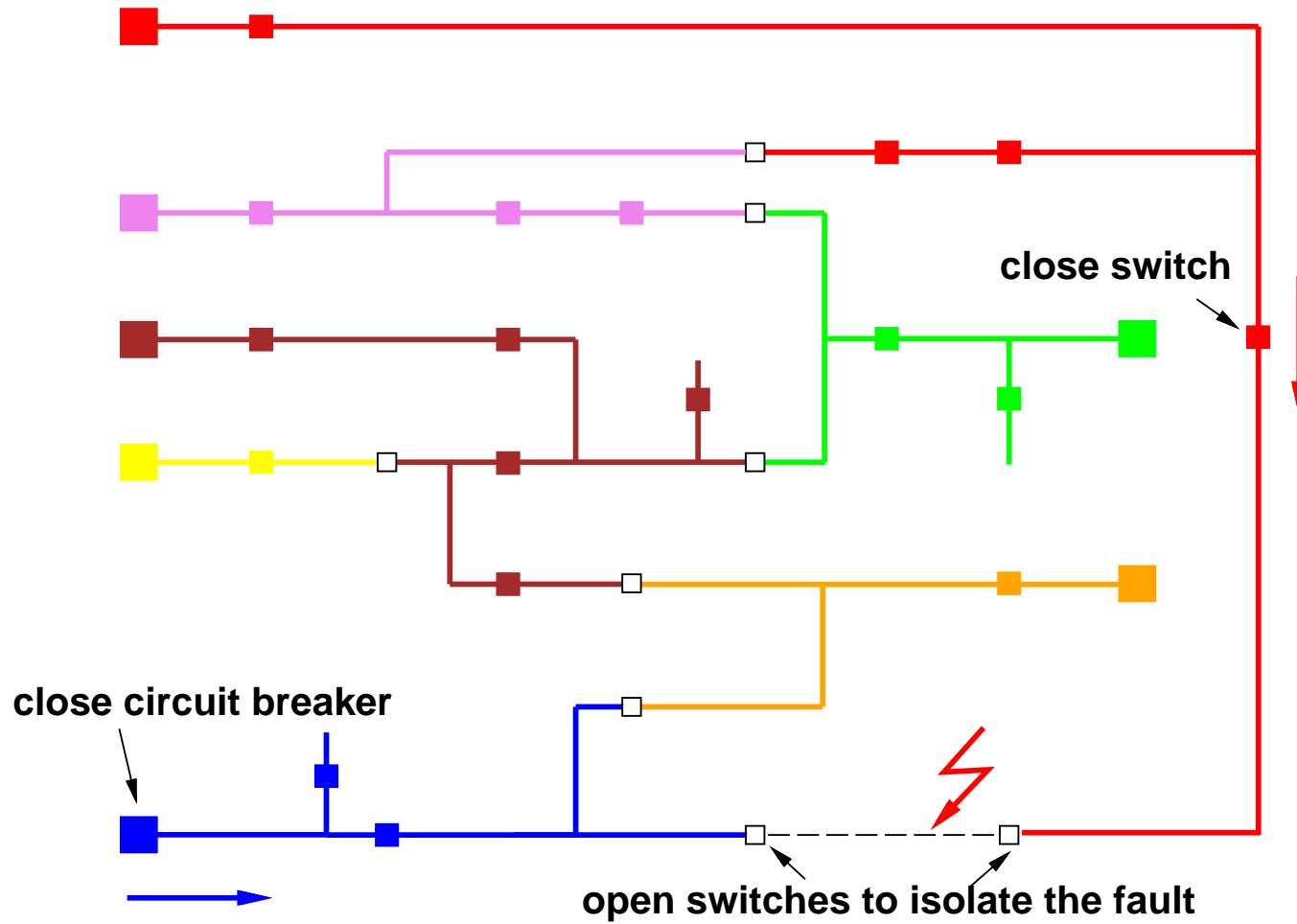
Example: power supply restoration



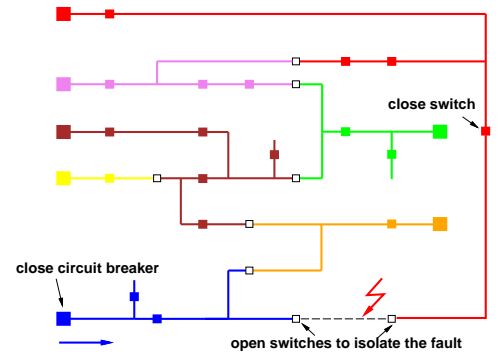
Example: power supply restoration



Example: power supply restoration



Example: power distribution systems



states??: connections, status faulty/non-faulty of the lines, positions open/closed of the switches and circuit-breakers, power consumed on each line, capacity of circuit-breakers and lines

observations??: given by fault, position, and power sensors (unreliable)

events??: faults, open/close a switch or a circuit-breaker (unreliable)

acceptable sequences??: do not violate capacity constraints, eventually re-supply non-faulty lines every time faults occur

cost??: accounts for distance to the nominal configuration, power margins

Classical planning assumptions

- **finite:** S, E and O are finite
- **static, single agent:** $E = A$
- **deterministic:** $S_I = \{s_0\}$ and $|\gamma(s, a)| = 1$ whenever a is executable in s
- **fully observable:** $O = S$ and $\lambda(s) = \{s\}$
- **implicit time:** no durations, instantaneous actions
- **reachability goals:** acceptable sequences end in a goal state in S_G
- **sequential:** solution is a sequence of actions
- **cost function:** length of the sequence
- **off-line planning:** planner does not know execution status

Classical planning model

Classical planning model sums up to:

- a finite set of states S
- a finite set of actions A
- a transition function $\gamma : S \times A \mapsto S$
- an initial state s_0
- a set S_G of goal states

Classical planning problem

Classical planning problem sums up to:

given (S, A, γ, s_0, S_G) , find a sequence of actions $\langle a_1, a_2, \dots, a_n \rangle$, $a_i \in A$, leading the environment from the initial state s_0 to a goal state in S_G , at minimum cost. That is, the sequence of actions must induce a sequence of state transitions:

$$s_1 = \gamma(s_0, a_1)$$

$$s_2 = \gamma(s_1, a_2)$$

...

$$s_n = \gamma(s_{n-1}, a_n) \in S_G$$

(and n is the smallest integer such that such a sequence exists)

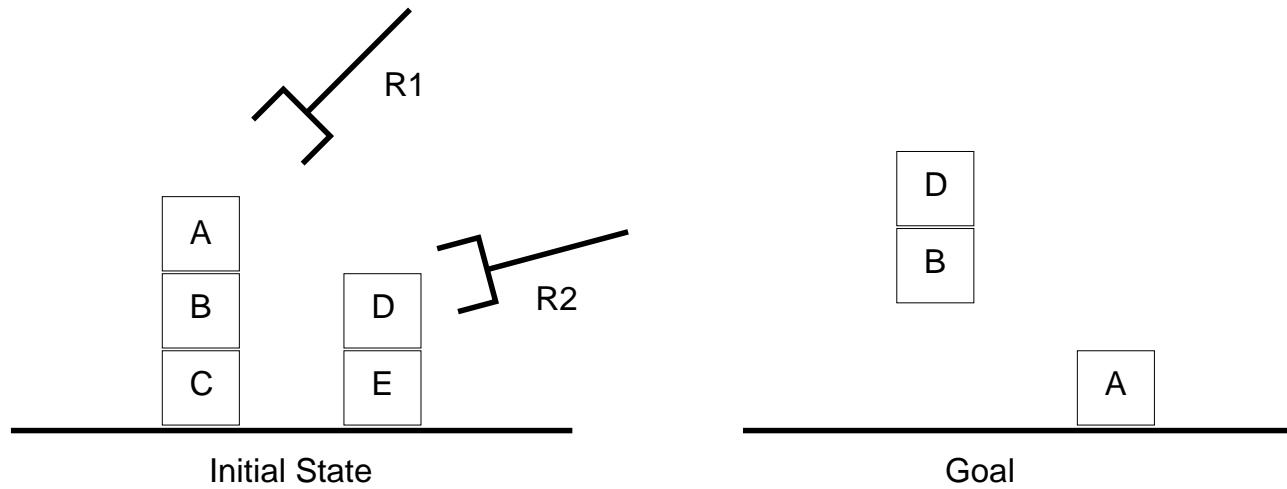
Classical plans

linear plan (or sequence): totally ordered set $\langle a_1, \dots, a_n \rangle$, $a_i \in A$, such that $\gamma(\dots \gamma(\gamma(s_0, a_1), a_2), \dots, a_n) \in S_G$. Produced by state-space planning approaches.

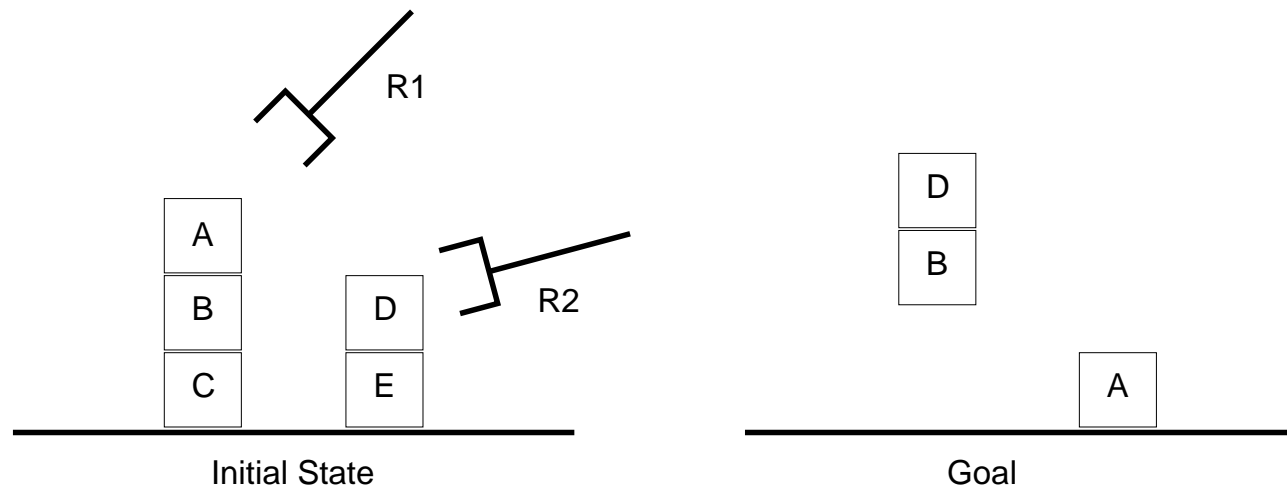
non-linear plan: partially ordered set $\langle \{a_1, \dots, a_n\}, < \rangle$, $a_i \in A$, such that each linearisation is a totally ordered plan. Produced by plan-state planning approaches.

parallel plan: sequence of parallel actions $\langle \{a_{1,1}, \dots, a_{1,l(1)}\}, \dots, \{a_{1,n}, \dots, a_{1,l(n)}\} \rangle$, $a_{i,j} \in A$, special case of partially ordered plan. Produced by graph-based planning approaches.

Example: blocks world



Example: blocks world



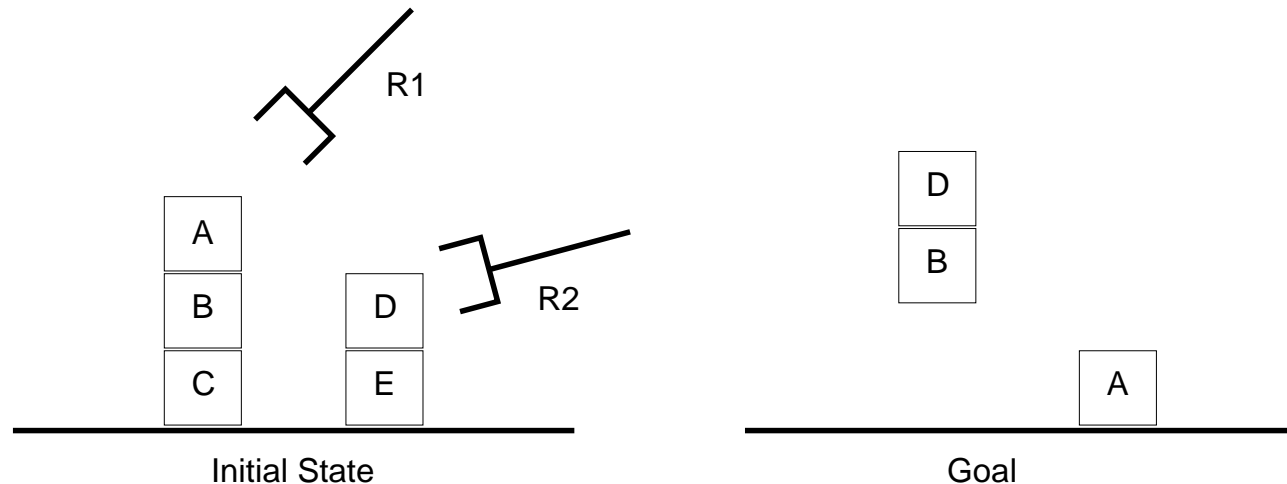
states??: configurations of n blocks, i.e., the data of the object on which each given block is, if any, and the block held by each given robot, if any.

actions??: robot **picks up** clear block from table, robot **puts down** held block onto the table, robot **unstacks** clear block from top of another block, robot **stacks** held block on top of another clear block.

initial state??: given configuration

goal??: given (possibly partial) configuration

Example: blocks world



linear plan??:

$\langle \text{unstack}(\text{R1}, \text{A}, \text{B}), \text{unstack}(\text{R2}, \text{D}, \text{E}), \text{putdown}(\text{R1}, \text{A}), \text{stack}(\text{R2}, \text{D}, \text{B}) \rangle$

non-linear plan??:

$\langle \{ \text{unstack}(\text{R1}, \text{A}, \text{B}), \text{unstack}(\text{R2}, \text{D}, \text{E}), \text{putdown}(\text{R1}, \text{A}), \text{stack}(\text{R2}, \text{D}, \text{B}) \},$
 $\{ \text{unstack}(\text{R1}, \text{A}, \text{B}) < \text{putdown}(\text{R1}, \text{A}), \text{unstack}(\text{R2}, \text{D}, \text{E}) < \text{stack}(\text{R2}, \text{D}, \text{B}),$
 $\text{unstack}(\text{R1}, \text{A}, \text{B}) < \text{stack}(\text{R2}, \text{D}, \text{B}) \} \rangle$

parallel plan??:

$\langle \{ \text{unstack}(\text{R1}, \text{A}, \text{B}), \text{unstack}(\text{R2}, \text{D}, \text{E}) \}, \{ \text{putdown}(\text{R1}, \text{A}), \text{stack}(\text{R2}, \text{D}, \text{B}) \} \rangle$

Classical planning agents

We have already seen examples of classical planning agents:

- search-based problem solving agents
- situation-calculus based logical agents

These do not scale up very well to large planning problems – problems with a large branching factor would require human-supplied heuristics, defeating the goal of building domain-independent planners.

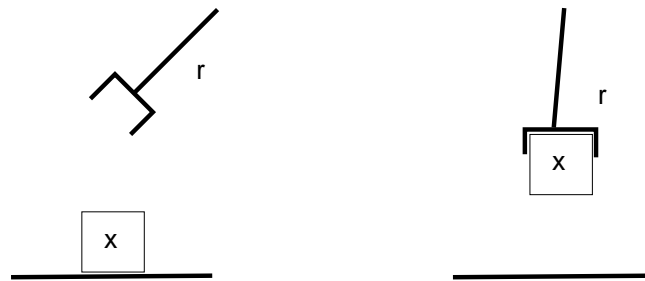
Here we look at more specific techniques that scale better. These rely on adequate **representations** of the planning problem. These representations enable the exploitation of the logical structure of the problem.

The STRIPS Representation

- Use first-order logic to represent states:
 - logical language (predicates, connectives, variables, quantifiers)
 - a property of states (a set $S' \subseteq S$) is represented by a formula
$$\forall x(\text{block}(x) \rightarrow \text{ontable}(x) \vee \exists r \text{ holding}(r, x))$$
 - a state $s \in S$ is represented by a set of ground atoms under the closed world assumption
$$\{\text{on}(A, B), \text{clear}(A), \text{ontable}(B), \text{holding}(R1, C)\}$$
 - ground atoms (e.g., $\text{on}(A, B)$) are obtained from domain predicates (e.g., $\text{on}(x, y)$) by instantiating their parameters with actual objects

The STRIPS Representation

- Use operators with logical pre-post conditions to represent actions:
 - operator o has a name and parameters
 $\text{pickup}(r, x)$
 - precondition $\text{PRE}(o)$ is a set of positive literals that must be true for the action to be applicable
 $\{\text{ontable}(x), \text{clear}(x), \text{handempty}(r)\}$
 - effect (postcondition) $\text{EFF}(o)$: literals true in the resulting state
 $\{\text{holding}(r, x), \neg\text{ontable}(x), \neg\text{clear}(x), \neg\text{handempty}(r)\}$
 - the effect is split into add effect $\text{EFF}^+(o) = \{\text{holding}(r, x)\}$ and delete effect $\text{EFF}^-(o) = \{\text{ontable}(x), \text{clear}(x), \text{handempty}(r)\}$
 - an action $a \in A$ is represented by an instance of an operator



The STRIPS Representation

- Use first-order logic to represent states
- Use operators with logical pre-post conditions to represent actions:
- Use the STRIPS rule to represent the transition relation γ :

$$\gamma(s, a) = \begin{cases} s \setminus \text{EFF}^-(a) \cup \text{EFF}^+(a) & \text{if } \text{PRE}(a) \subseteq s \\ \text{undefined} & \text{otherwise (action not executable)} \end{cases}$$

simple approach to the **frame problem**

- Example:

– $s = \{\text{on}(A, B), \text{clear}(A), \text{ontable}(B), \text{holding}(R1, C)\}$

– $a = \text{putdown}(R1, C)$

operator $\text{putdown}(r, x)$

precondition $\{\text{holding}(r, x)\}$

effect $\{\text{ontable}(x), \text{clear}(x), \text{handempty}(r), \neg\text{holding}(r, x)\}$

– $\gamma(s, a) = \{\text{on}(A, B), \text{clear}(A), \text{ontable}(B), \text{ontable}(C), \text{clear}(C), \text{handempty}(R1)\}$

The ADL Representation

STRIPS	ADL
Only positive literals in states closed world assumption unmentioned literals are false {Poor, Unknown}	Positive and negative literals in states open world assumption unmentioned literals are unknown { \neg Rich, \neg Famous}
Effect {P, \neg Q} means add P delete Q	Effect {P, \neg Q} means add P and \neg Q, delete Q and \neg P
Only positive literals in prec. & goals {Rich, Famous}	Prec. & goals are arbitrary formulae $\exists l \text{ At}(T1, l) \vee \text{At}(T2, l)$
Effects are sets (conjunctions)	Conditional & univ. quantified effects when $C : E$ when P forall $x \ Q(x)$ E takes place only when C is true
No support for equality and types	Equality predicate ($x=y$) built in Variables may have types: pickup($r : \text{robot}, x : \text{block}$)

PDDL (Planning Domain Definition Language) supports STRIPS and ADL

PDDL - Example

```
(define (domain elevator)
  (:requirements :adl)
  (:types passenger floor)

  (:predicates
    (origin ?p - passenger ?f - floor)
    (destin ?p - passenger ?f - floor)
    (boarded ?p - passenger)
    (served ?p - passenger)
    (lift-at ?f - floor))

  (:operator go
    :parameters (?f1 ?f2 - floor)
    :precondition (lift-at ?f1)
    :effect (and (lift-at ?f2) (not (lift-at ?f1))
      (forall (?p - passenger)
        (when (and (boarded ?p) (destin ?p ?f2))
          (and (not (boarded ?p)) (served ?p))))
      (forall (?p - passenger)
        (when (and (origin ?p ?f2) (not (served ?p)))
          (boarded ?p)))))))
```

PDDL - Example

```
(define (problem simple)
  (:domain elevator)
  (:objects p0 p1 p2 p3 p4 p5 - passenger
            f0 f1 f2 f3 f4 f5 f6 f7 - floor)
  (:init
    (origin p0 f2) (destin p0 f4)
    (origin p1 f2) (destin p1 f6)
    (origin p2 f1) (destin p2 f4)
    (origin p3 f7) (destin p3 f2)
    (origin p4 f5) (destin p4 f3)
    (origin p5 f6) (destin p5 f7)
    (lift-at f0))

  (:goal (forall (?p - passenger) (served ?p))))

(go f0 f1) (go f1 f2) (go f2 f4) (go f4 f5)
(go f5 f6) (go f6 f7) (go f7 f3) (go f3 f2)
```

STATE-SPACE PLANNING

CHAPTER 11

Outline

- ◇ State-space planning
- ◇ Progression planning (forward search)
- ◇ Search control rules
- ◇ Heuristics
- ◇ Regression planning (backward search)
- ◇ Lifting

State-space planning

- Planning procedures are often search procedures
- They differ by the search space they consider
- State-space planning explores the most obvious search space:
 - Nodes correspond to states of the world
 - Actions define successor states
 - Plans are paths through the space
- Search space can be explored in many ways:
 - forward, backward
 - using a variety of strategies (breadth-first, depth-first, best-first, ...)
 - using a variety of heuristics
 - The STRIPS representation enables an efficient exploration

Progression planning (forward search)

function FORWARD-SEARCH(O, s_0, g) **returns** an action sequence, or failure

$s \leftarrow s_0$

$\pi \leftarrow \langle \rangle$

loop do

if s satisfies g **then return** π

$E \leftarrow \{a \mid a \text{ is a ground instance of an operator in } O$
 such that $\text{PRE}(a)$ is satisfied in $s\}$

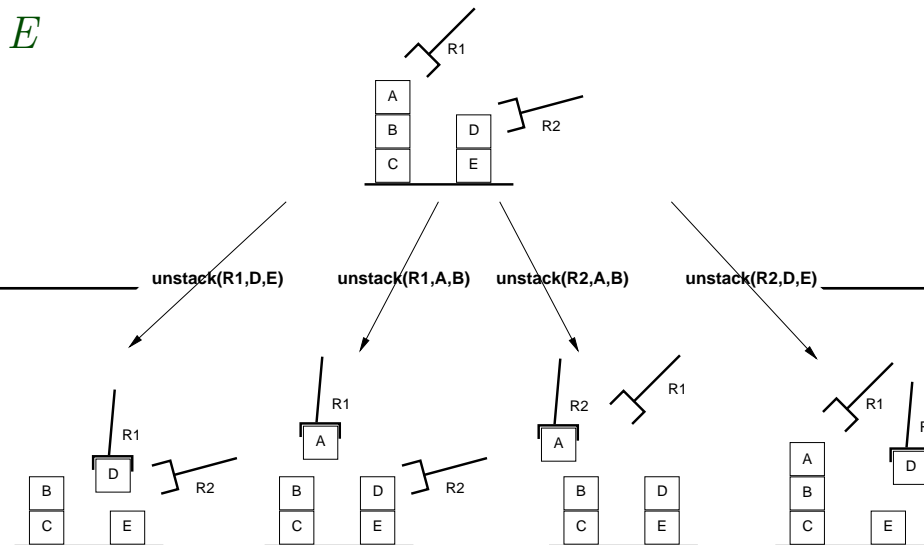
if $E = \{\}$ **then return** failure

choose an action $a \in E$

$s \leftarrow \gamma(s, a)$

$\pi \leftarrow \pi.a$

end



Progression planning (forward search)

function FORWARD-SEARCH(O, s_0, g) **returns** an action sequence, or failure

$s \leftarrow s_0$

$\pi \leftarrow \langle \rangle$

loop do

if $g \subseteq s$ **then return** π

$E \leftarrow \{a \mid a \text{ is a ground instance of an operator in } O$
 such that $\text{PRE}(a) \subseteq s\}$

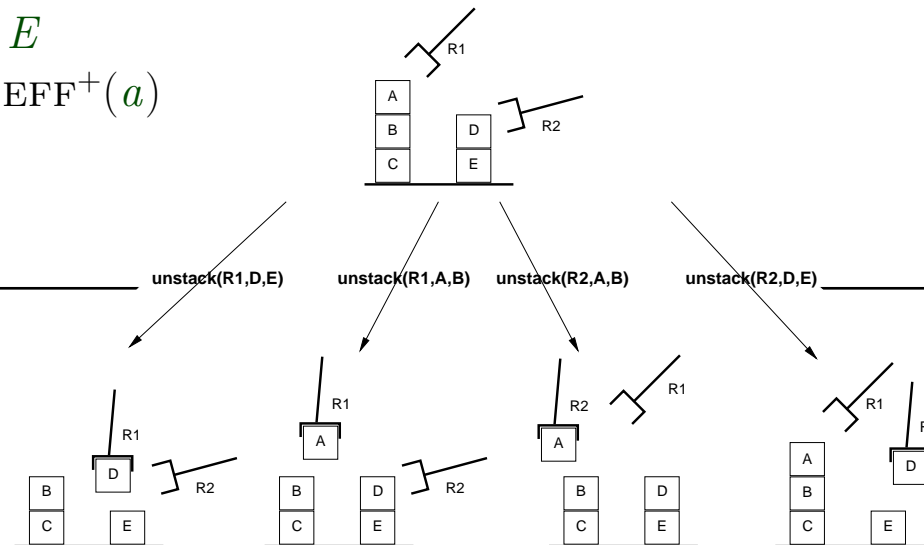
if $E = \{\}$ **then return** failure

choose an action $a \in E$

$s \leftarrow (s \setminus \text{EFF}^-(a)) \cup \text{EFF}^+(a)$

$\pi \leftarrow \pi.a$

end



Properties of FORWARD-SEARCH

FORWARD-SEARCH can be used in conjunction with any search strategy to implement **choose**, breadth-first search, depth-first search, iterative-deepening, greedy search, A*, IDA*, ...

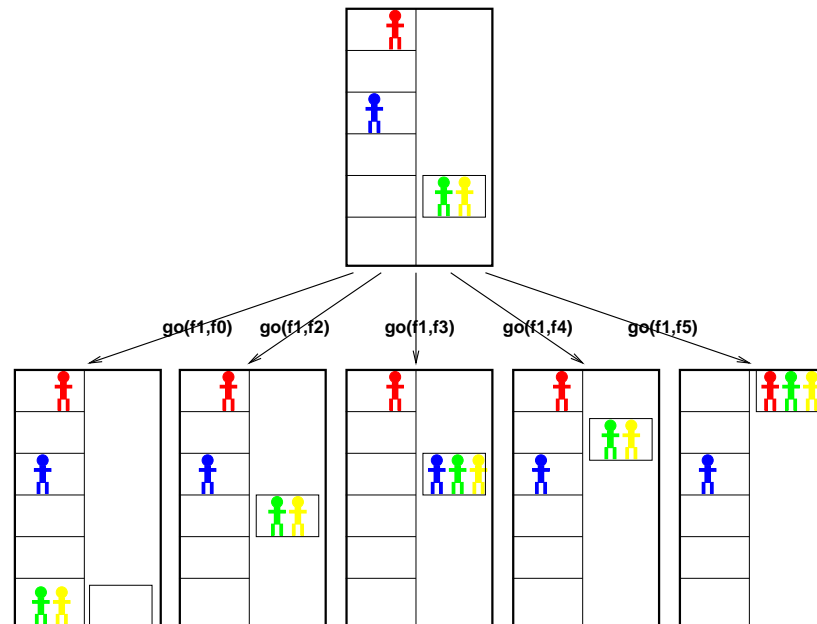
FORWARD-SEARCH is **sound**: any plan returned is guaranteed to be a solution to the problem.

FORWARD-SEARCH is **complete**: provided the underlying search strategy is complete, it will always return a solution to the problem if there is one.

For instance, when used with breadth-first search it will be complete, when used with depth-first search it will be complete if the state space is finite – in general, we need to detect and forbid loops.

Branching factor in FORWARD-SEARCH

FORWARD-SEARCH can have a large branching factor



It wastes a lot of time trying **irrelevant** actions

How do we cope with this?:

domain-specific: search control, heuristics, hierarchical task networks

domain-independent: heuristics automatically generated from the STRIPS problem description

Search control knowledge

Knowledge about what constitutes a promising plan in the domain

Constraints on execution sequences induced by a plan

Elegantly expressed in temporal logic (FOL + $\{\square, \bigcirc\}$):

- Every stop leads a passenger to board or be served:

$$\square((\forall p \text{ served}(p)) \vee (\exists p (\neg \text{served}(p) \wedge \bigcirc \text{served}(p)) \vee (\neg \text{boarded}(p) \wedge \bigcirc \text{boarded}(p))))$$

- Once the elevator starts going down, it cannot go up:

$$\square((\exists f_1 f_2 \text{ above}(f_1, f_2) \wedge \text{liftat}(f_1) \wedge \bigcirc \text{liftat}(f_2)) \rightarrow \bigcirc(\square(\exists f_1 f_2 \text{ above}(f_1, f_2) \wedge \text{liftat}(f_1) \wedge \bigcirc \text{liftat}(f_2))))$$

Incremental pruning of plan prefixes violating the control knowledge

Used in TLPlan [Bacchus & Kabanza, AIJ 2000]

Automatically generated heuristics

Heuristics $\Delta(s, g)$ estimate the minimum cost (nb actions) from s to g

Relax the problem by ignoring the negative effects EFF^- :

- $\Delta(s, \{p\}) = 0$ if $p \in s$
- $\Delta(s, \{p\}) = \infty$ if $\forall a \in A, p \notin \text{EFF}^+(a)$
- $\Delta(s, \{p\}) = \min_{a \in A} \{1 + \Delta(s, \text{PRE}(a)) \mid p \in \text{EFF}^+(a)\}$

Further relax the problem by ignoring interactions between subgoals:

- admissible: distance to a set is the max of distances:
$$\Delta(s, g) = \max_{p \in g} \Delta(s, \{p\})$$
- non-admissible: distance to a set is the sum of distances.
Assumes independence:
$$\Delta(s, g) = \sum_{p \in g} \Delta(s, \{p\})$$
- admissible: distance to a set is the max of distances to each pair.
Similar heuristic used in FF [Hoffmann & Nebel, AIJ 2000]

Regression planning (backward search)

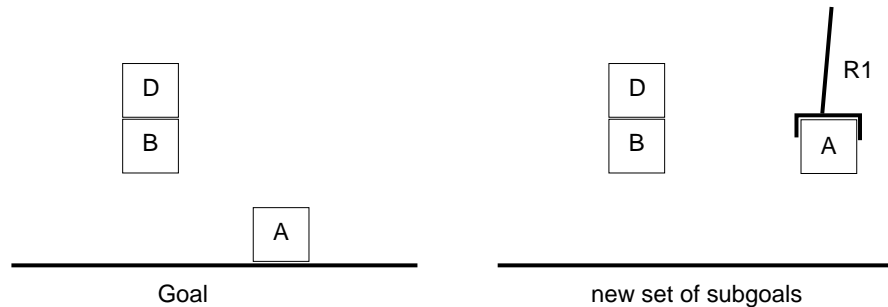
For forward search, we started at the initial state and computed state transitions, leading to a new state $\gamma(s, a)$

For backward search, we start at the goal and compute inverse state transitions, leading to a new set of **subgoals** $\gamma^{-1}(g, a)$

What do we really mean by $\gamma^{-1}(g, a)$?? First we need to define **relevance**

Inverse state transitions - relevance

- An action a is **relevant** for goal g if:
 - it makes at least one of g 's propositions true: $g \cap \text{EFF}^+(a) \neq \{\}$
 - it does not make any of g 's proposition false: $g \cap \text{EFF}^-(a) = \{\}$
- If a is relevant for g then: $\gamma^{-1}(g, a) = (g \setminus \text{EFF}^+(a)) \cup \text{PRE}(a)$



- **Example:**

- $g = \{\text{on}(D, B), \text{ontable}(A), \text{clear}(D), \text{clear}(A)\}$
- $a = \text{putdown}(R1, A)$
 - operator $\text{putdown}(r, x)$
 - precondition $\{\text{holding}(r, x)\}$
 - effect $\{\text{ontable}(x), \text{clear}(x), \text{handempty}(r), \neg\text{holding}(r, x)\}$
- $\gamma^{-1}(g, a) = \{\text{on}(D, B), \text{clear}(D), \text{holding}(R1, A)\}$

Regression planning (backward search)

function BACKWARD-SEARCH(O, s_0, g) **returns** an action sequence, or failure

$\pi \leftarrow \langle \rangle$

loop do

if s_0 satisfies g **then return** π

$E \leftarrow \{a \mid a \text{ is a ground instance of an operator in } O$
 such that a is relevant for $g\}$

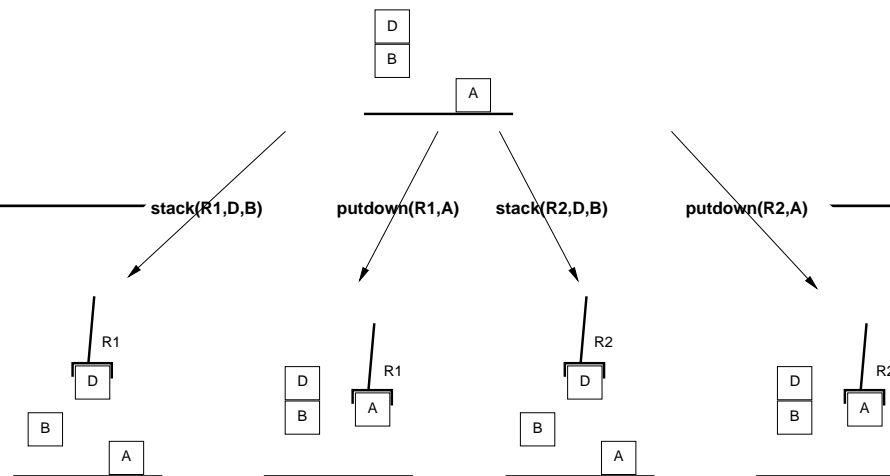
if $E = \{\}$ **then return** failure

choose an action $a \in E$

$g \leftarrow \gamma^{-1}(g, a)$

$\pi \leftarrow a.\pi$

end



Regression planning (backward search)

function BACKWARD-SEARCH(O, s_0, g) **returns** an action sequence, or failure

$\pi \leftarrow \langle \rangle$

loop do

if $g \subseteq s_0$ **then return** π

$E \leftarrow \{a \mid a \text{ is a ground instance of an operator in } O$

such that $g \cap \text{EFF}^+(a) \neq \{\}$ and $g \cap \text{EFF}^-(a) = \{\}$

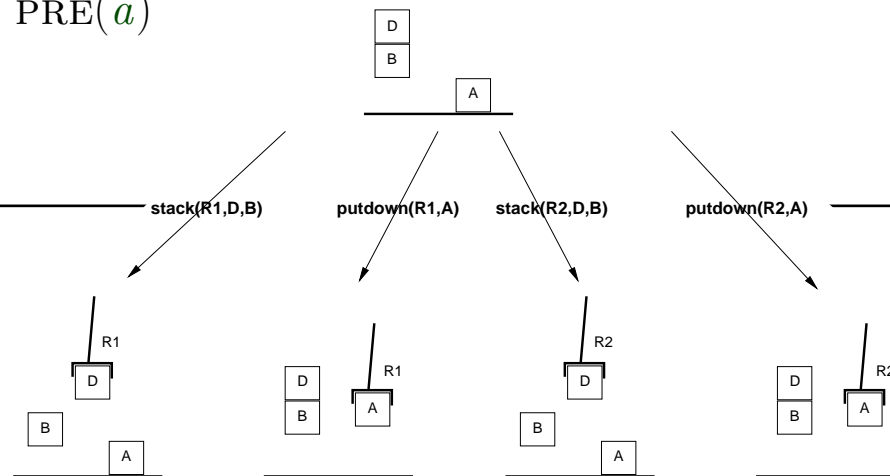
if $E = \{\}$ **then return** failure

choose an action $a \in E$

$g \leftarrow (g \setminus \text{EFF}^+(a)) \cup \text{PRE}(a)$

$\pi \leftarrow a.\pi$

end



Properties of BACKWARD-SEARCH

BACKWARD-SEARCH can be used in conjunction with any search strategy to implement **choose**, breadth-first search, depth-first search, iterative-deepening, greedy search, A*, IDA*, ...

BACKWARD-SEARCH is **sound**: any plan returned is guaranteed to be a solution to the problem.

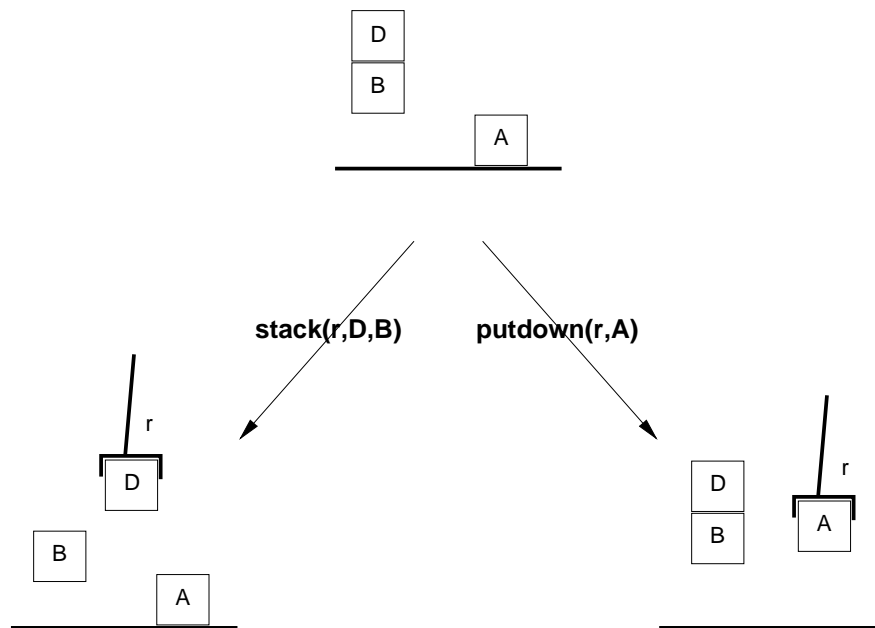
BACKWARD-SEARCH is **complete**: provided the underlying search strategy is complete, it will always return a solution to the problem if there is one.

For instance, when used with breadth-first search it will be complete, when used with depth-first search it will be complete if the state space is finite – in general, need to detect and forbid loops, by checking that **no previous subgoal is a subset of the current one**.

Lifting

We can substantially reduce the branching factor if we only **partially instantiate** the operators.

For instance, in the Blocks World, we may not need to distinguish between using robot hand R1 and robot hand R2. Just any hand will do:



Lifted backward search

More complicated because we have to keep track of the substitutions performed.

function LIFTED-BACKWARD-SEARCH(O, s_0, g) **returns** an action sequence, or failure

$\pi \leftarrow \langle \rangle$

loop do

if s_0 satisfies g **then return** π

$E \leftarrow \{(o, \sigma) \mid \begin{array}{l} o \text{ is a standardisation of an operator in } O \text{ relevant for } g \\ \sigma \text{ is an mgu for an atom of } g \text{ and an atom of } \text{EFF}(o) \\ \text{causing the relevance} \end{array}\}$

if $E = \{ \}$ **then return** failure

choose a pair $(o, \sigma) \in E$

$g \leftarrow \gamma^{-1}(\sigma(g), \sigma(o))$

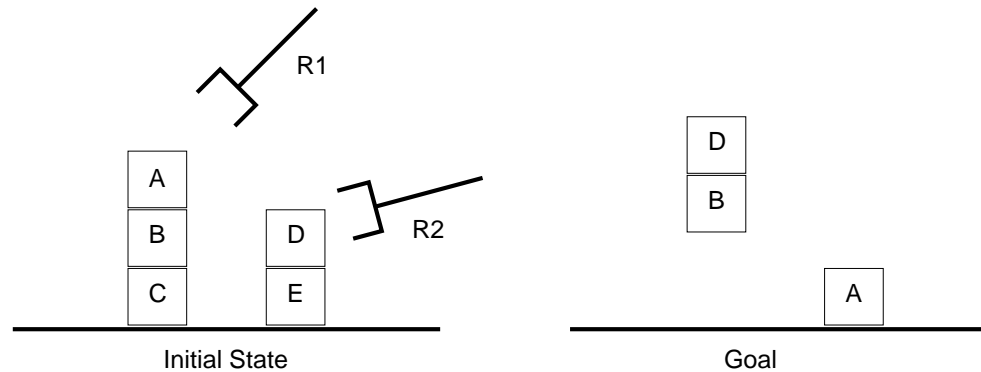
$\pi \leftarrow \sigma(o). \sigma(\pi)$

end

Trace example

relevance: $g \cap \text{EFF}^+(a) \neq \{\}$, $g \cap \text{EFF}^-(a) = \{\}$

inverse transition: $\gamma^{-1}(g, a) = (g \setminus \text{EFF}^+(a)) \cup \text{PRE}(a)$



1. $g \leftarrow \{\text{on}(D, B), \text{clear}(D), \text{ontable}(A), \text{clear}(A)\}$
 $\pi \leftarrow \langle \rangle$

$o \leftarrow \text{stack}(r, x, y),$

$\sigma \leftarrow \{x \leftarrow D, y \leftarrow B\}$

PRE : $\{\text{holding}(r, D), \text{clear}(B)\}$

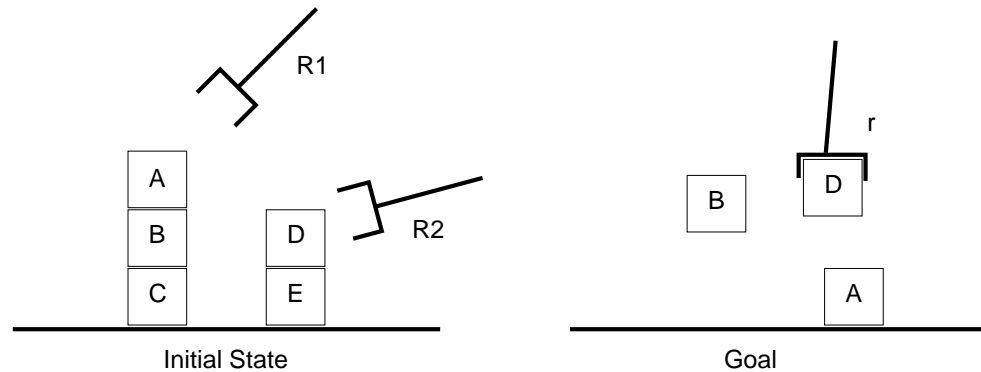
EFF : $\{\text{on}(D, B), \text{handempty}(r), \text{clear}(D),$
 $\neg\text{holding}(r, D), \neg\text{clear}(B)\}$

2. $g \leftarrow \{\text{holding}(r, D), \text{clear}(B), \text{ontable}(A), \text{clear}(A)\}$
 $\pi \leftarrow \langle \text{stack}(r, D, B) \rangle$

Trace example

relevance: $g \cap \text{EFF}^+(a) \neq \{\}$, $g \cap \text{EFF}^-(a) = \{\}$

inverse transition: $\gamma^{-1}(g, a) = (g \setminus \text{EFF}^+(a)) \cup \text{PRE}(a)$



2. $g \leftarrow \{\text{holding}(r, D), \text{clear}(B), \text{ontable}(A), \text{clear}(A)\}$
 $\pi \leftarrow \langle \text{stack}(r, D, B) \rangle$

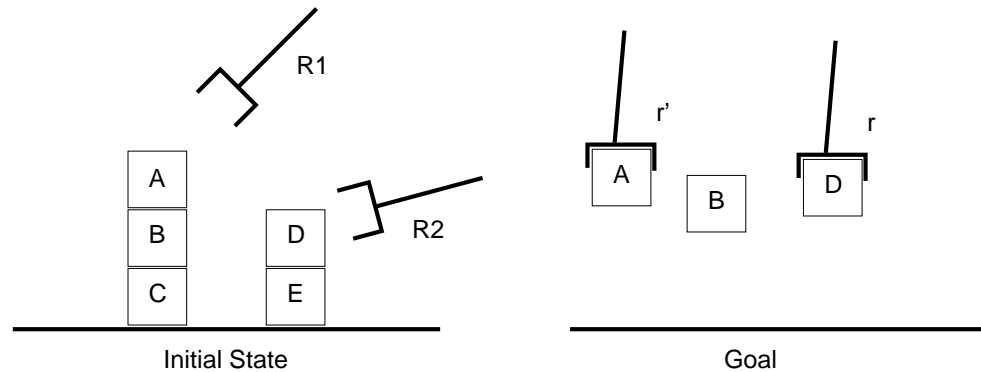
$o \leftarrow \text{putdown}(r', x), \quad \text{PRE} : \{\text{holding}(r', A)\}$
 $\sigma \leftarrow \{x \leftarrow A\} \quad \text{EFF} : \{\text{ontable}(A), \text{handempty}(r'), \text{clear}(A),$
 $\quad \neg \text{holding}(r', A)\}$

3. $g \leftarrow \{\text{holding}(r, D), \text{clear}(B), \text{holding}(r', A)\}$
 $\pi \leftarrow \langle \text{putdown}(r', A), \text{stack}(r, D, B) \rangle$

Trace example

relevance: $g \cap \text{EFF}^+(a) \neq \{\}$, $g \cap \text{EFF}^-(a) = \{\}$

inverse transition: $\gamma^{-1}(g, a) = (g \setminus \text{EFF}^+(a)) \cup \text{PRE}(a)$



3. $g \leftarrow \{\text{holding}(r, D), \text{clear}(B), \text{holding}(r', A)\}$
 $\pi \leftarrow \langle \text{putdown}(r', A), \text{stack}(r, D, B) \rangle$

$o \leftarrow \text{unstack}(r'', x, y),$

$\sigma \leftarrow \{r'' \leftarrow r, x \leftarrow D, y \neq B\}$

PRE : $\{\text{on}(D, y), \text{clear}(D), \text{handempty}(r)\}$

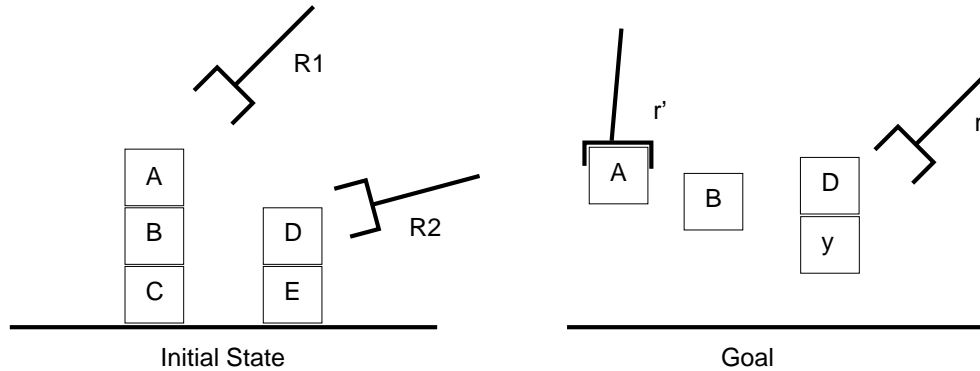
EFF : $\{\text{holding}(r, D), \text{clear}(y),$
 $\neg \text{handempty}(r), \neg \text{on}(D, y), \neg \text{clear}(D)\}$

4. $g \leftarrow \{\text{on}(D, y), \text{clear}(D), \text{handempty}(r), \text{clear}(B), \text{holding}(r', A), y \neq B\}$
 $\pi \leftarrow \langle \text{unstack}(r, D, y), \text{putdown}(r', A), \text{stack}(r, D, B) \rangle$ with $y \neq B$

Trace example

relevance: $g \cap \text{EFF}^+(a) \neq \{\}$, $g \cap \text{EFF}^-(a) = \{\}$

inverse transition: $\gamma^{-1}(g, a) = (g \setminus \text{EFF}^+(a)) \cup \text{PRE}(a)$



4. $g \leftarrow \{\text{on}(D, y), \text{clear}(D), \text{handempty}(r), \text{clear}(B), \text{holding}(r', A), y \neq B\}$
 $\pi \leftarrow \langle \text{unstack}(r, D, y), \text{putdown}(r', A), \text{stack}(r, D, B) \rangle$ with $y \neq B$

$o \leftarrow \text{unstack}(r'', x, y'),$
 $\sigma \leftarrow \{r'' \leftarrow r', x \leftarrow A, y' \leftarrow B, r' \neq r\}$

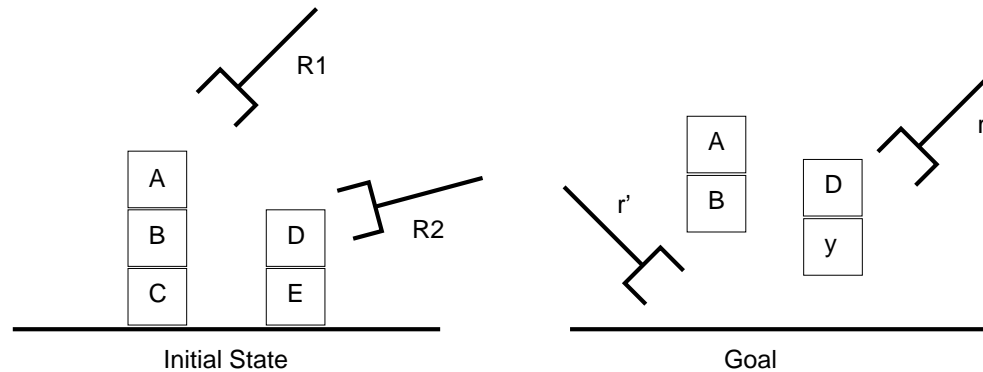
PRE : $\{\text{on}(A, B), \text{clear}(A), \text{handempty}(r')\}$
 EFF : $\{\text{holding}(r', A), \text{clear}(B),$
 $\neg \text{handempty}(r'), \neg \text{on}(A, B), \neg \text{clear}(A)\}$

5. $g \leftarrow \{\text{on}(D, y), \text{clear}(D), \text{handempty}(r), \text{on}(A, B), \text{clear}(A), \text{handempty}(r'), y \neq B, r \neq r'\}$
 $\pi \leftarrow \langle \text{unstack}(r', A, B), \text{unstack}(r, D, y), \text{putdown}(r', A), \text{stack}(r, D, B) \rangle$ with $y \neq B, r \neq r'$

Trace example

relevance: $g \cap \text{EFF}^+(a) \neq \{\}$, $g \cap \text{EFF}^-(a) = \{\}$

inverse transition: $\gamma^{-1}(g, a) = (g \setminus \text{EFF}^+(a)) \cup \text{PRE}(a)$



5. $g \leftarrow \{\text{on}(D, y), \text{clear}(D), \text{handempty}(r), \text{on}(A, B), \text{clear}(A), \text{handempty}(r'), y \neq B, r \neq r'\}$
 $\pi \leftarrow \langle \text{unstack}(r', A, B), \text{unstack}(r, D, y), \text{putdown}(r', A), \text{stack}(r, D, E) \rangle$ with $y \neq B, r \neq r'$
- $s = \{\text{on}(D, E), \text{clear}(D), \text{handempty}(R1), \text{on}(A, B), \text{clear}(A), \text{handempty}(R2), E \neq B, R1 \neq R2\}$
- $\pi \leftarrow \langle \text{unstack}(R2, A, B), \text{unstack}(R1, D, y), \text{putdown}(R2, A), \text{stack}(R1, D, E) \rangle$

Properties of BACKWARD-SEARCH

LIFTED-BACKWARD-SEARCH can be used in conjunction with any search strategy to implement **choose**, breadth-first search, depth-first search, iterative-deepening, greedy search, A*, IDA*, ...

LIFTED-BACKWARD-SEARCH is **sound**: any plan returned is guaranteed to be a solution to the problem.

LIFTED-BACKWARD-SEARCH is **complete**: provided the underlying search strategy is complete, it will always return a solution to the problem if there is one.

For instance, when used with breadth-first search it will be complete, when used with depth-first search it will be complete if the state space is finite – in general, we need to detect and forbid loops, by checking that **no previous subgoal unifies with a subset of the current one**.

PLAN-SPACE PLANNING

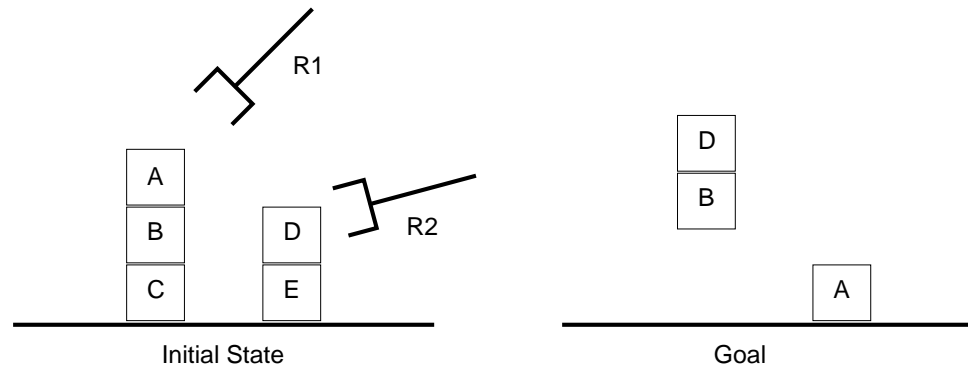
CHAPTER 11

Outline

- ◇ Motivation
- ◇ Partial plans
- ◇ Open preconditions
- ◇ Threats
- ◇ Plan-space planning algorithm
- ◇ Example

Motivation

Part of the ordering in an action sequence is not related to causality:



sequence:

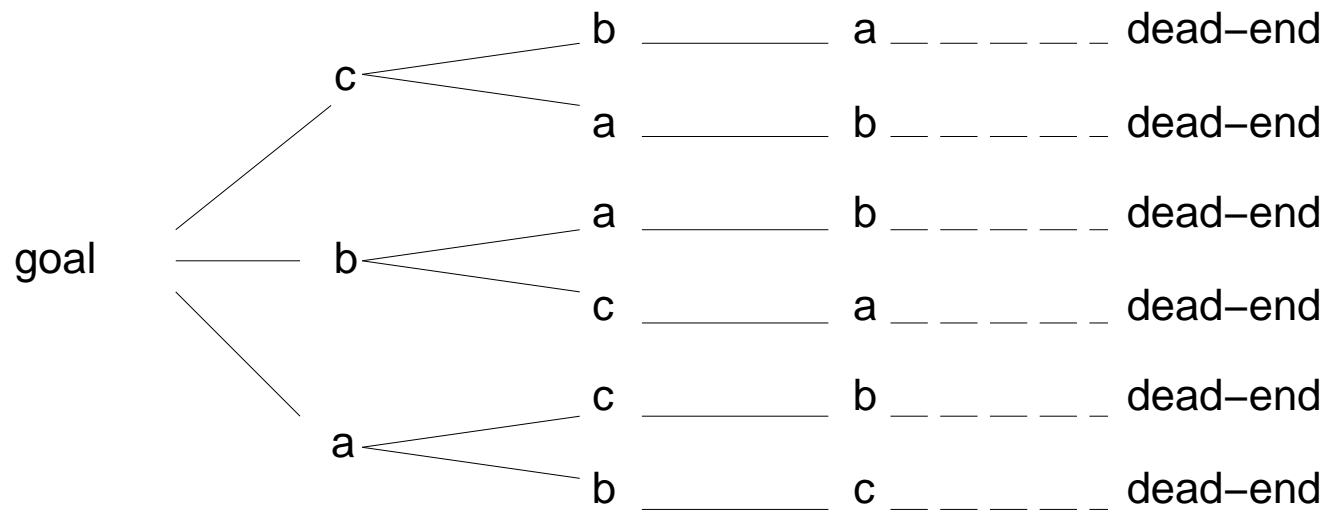
$\langle \text{unstack}(R1, A, B), \text{unstack}(R2, D, E), \text{putdown}(R1, A), \text{stack}(R2, D, B) \rangle$

causality only dictates: $\text{unstack}(R1, A, B) < \text{putdown}(R1, A)$,
 $\text{unstack}(R2, D, E) < \text{stack}(R2, D, B)$, and $\text{unstack}(R1, A, B) < \text{stack}(R2, D, B)$

State-space search produces inflexible plans.

Motivation

State-space search wastes time examining many different orderings of the same set of actions:

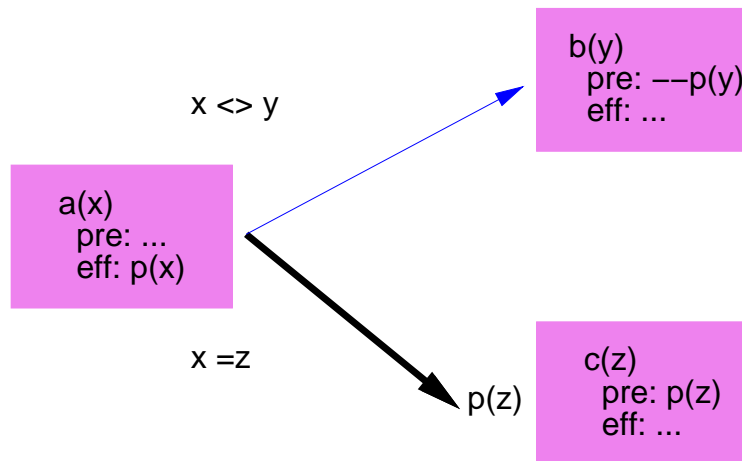


Least-commitment strategy: don't commit to orderings, instantiations, etc, until necessary.

Plan space search: basic idea

Plan-space search builds a **partial plan**:

- multiset O of operators $\{o_1, \dots, o_n\}$
- set $<$ of ordering constraints $o_i < o_j$ (with transitivity built in)
- set B of binding constraints $x = y, x \neq y, x \in D, x \notin D$, substitutions
- set L of causal links $o_i \xrightarrow{p} o_j$ stating that (effect p) of o_i establishes precondition p of o_j , with $o_i < o_j$ and binding constraints in B for parameters of o_i and o_j appearing in p



Plan-space search: basic idea

Nodes are **partial plans**

- initial node is $(O : \{\text{start}, \text{end}\}, < : \{\text{start} < \text{end}\}, B : \{\}, L : \{\})$
with $\text{EFF}(\text{start}) = s_0$ and $\text{PRE}(\text{end}) = g$.

Arcs are plan **refinement operations**

- each operation add elements to O , $<$, B , L to resolve a **flaw** in the plan

Search through the plan space until a partial plan is found which has no **flaw**:

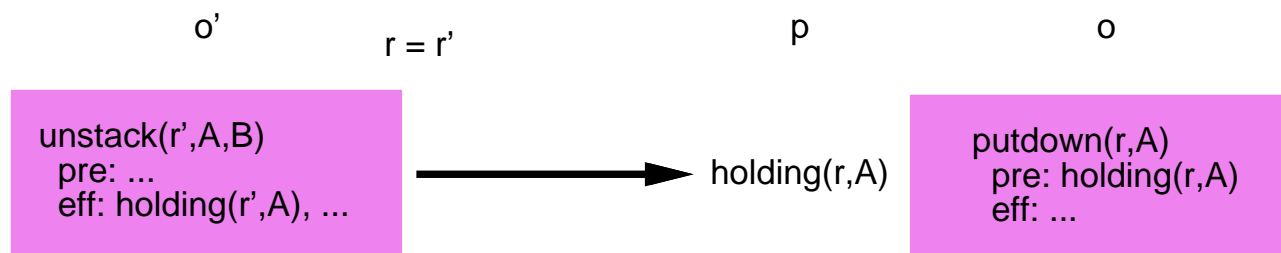
- no **open precondition**: all preconditions of all operators in O are established by causal links in L
- no **threat** (each linearisation is safe): for every causal link $o_i \xrightarrow{p} o_j$, every o_k with $\text{EFF}^-(o_k)$ unifable with p is such that $o_k < o_i$ or $o_j < o_k$
- $<$ and B are consistent

How to resolve an open precondition flaw?

Flaw: an operator o in the plan has a precondition p which is not established

Resolving the flaw:

1. find an operator o' (either already in the plan or insert it) which can be used to establish p , i.e. o' can be ordered before o and one of its effects can unify with p
2. add to B binding constraints to unify the effect of o' with p
3. add to L the causal link $o' \xrightarrow{p} o$ (and the ordering constraint $o' < o$).

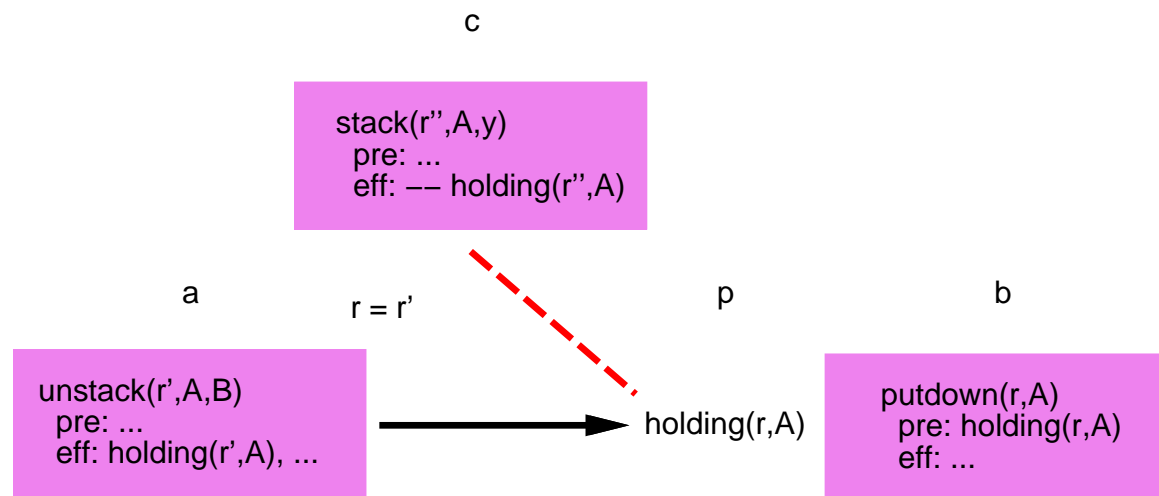


How to resolve a threat flaw?

Flaw: An operator a establishes a condition p for operator b , but another operator c is capable of deleting p before b gets to use it

Resolving the flaw - 3 possibilities:

1. order c after b
2. order c before a
3. add a binding constraint preventing c to delete p



Plan-space planning algorithm

function PLAN-SPACE-PLANNING(π) **returns** a plan, or failure

$F \leftarrow \text{OPEN-PRECONDITIONS}(\pi) \cup \text{THREATS}(\pi)$

if $F = \{\}$ **then return** π

select a flaw $f \in F$

$R \leftarrow \text{RESOLVE}(f, \pi)$

if $R = \{\}$ **then return** failure

choose a resolver $r \in R$

$\pi' \leftarrow \text{REFINE}(r, \pi)$

return PLAN-SPACE-PLANNING(π')

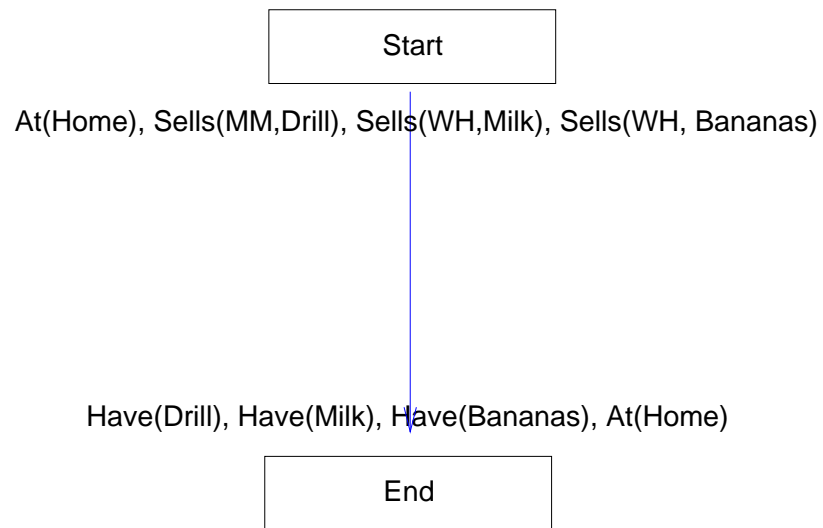
PLAN-SPACE-PLANNING is sound and complete

Example

- operator Start
 - precondition: $\{ \}$
 - effect: $\{ \text{At}(\text{Home}), \text{Sells}(\text{MM}, \text{Drill}), \text{Sells}(\text{WH}, \text{Milk}), \text{Sells}(\text{WH}, \text{Bananas}) \}$
- operator End
 - precondition: $\{ \text{At}(\text{Home}), \text{Have}(\text{Drill}), \text{Have}(\text{Milk}), \text{Have}(\text{Bananas}) \}$
 - effect: $\{ \}$
- operator $\text{Go}(l, l')$
 - precondition: $\{ \text{At}(l) \}$
 - effect: $\{ \text{At}(l'), \neg \text{At}(l) \}$
- operator $\text{Buy}(i, s)$
 - precondition: $\{ \text{At}(s), \text{Sells}(s, i) \}$
 - effect: $\{ \text{Have}(i) \}$

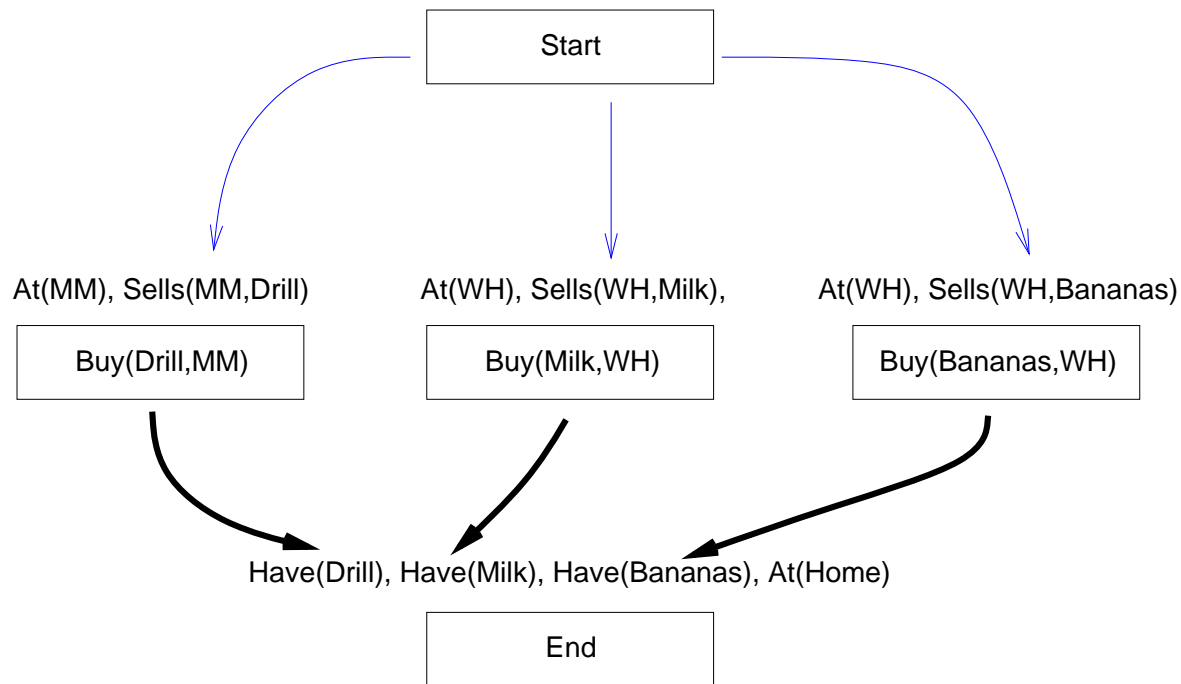
Example

Initial Plan



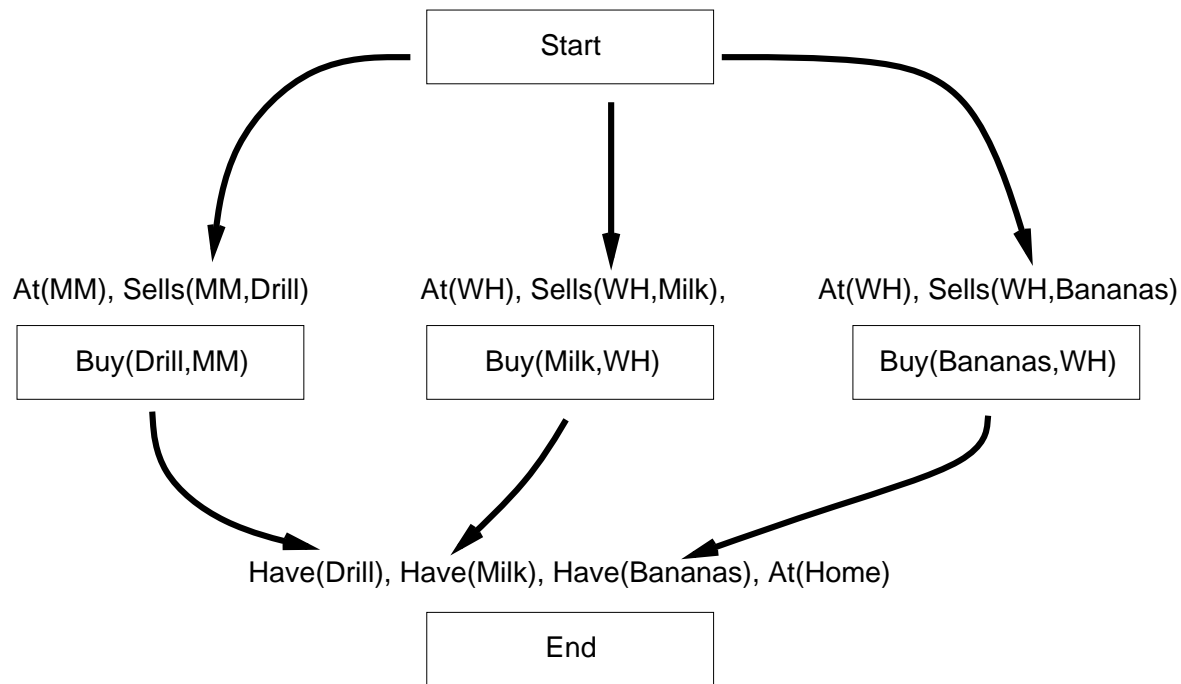
Example

The only possible ways to establish the “Have” preconditions



Example

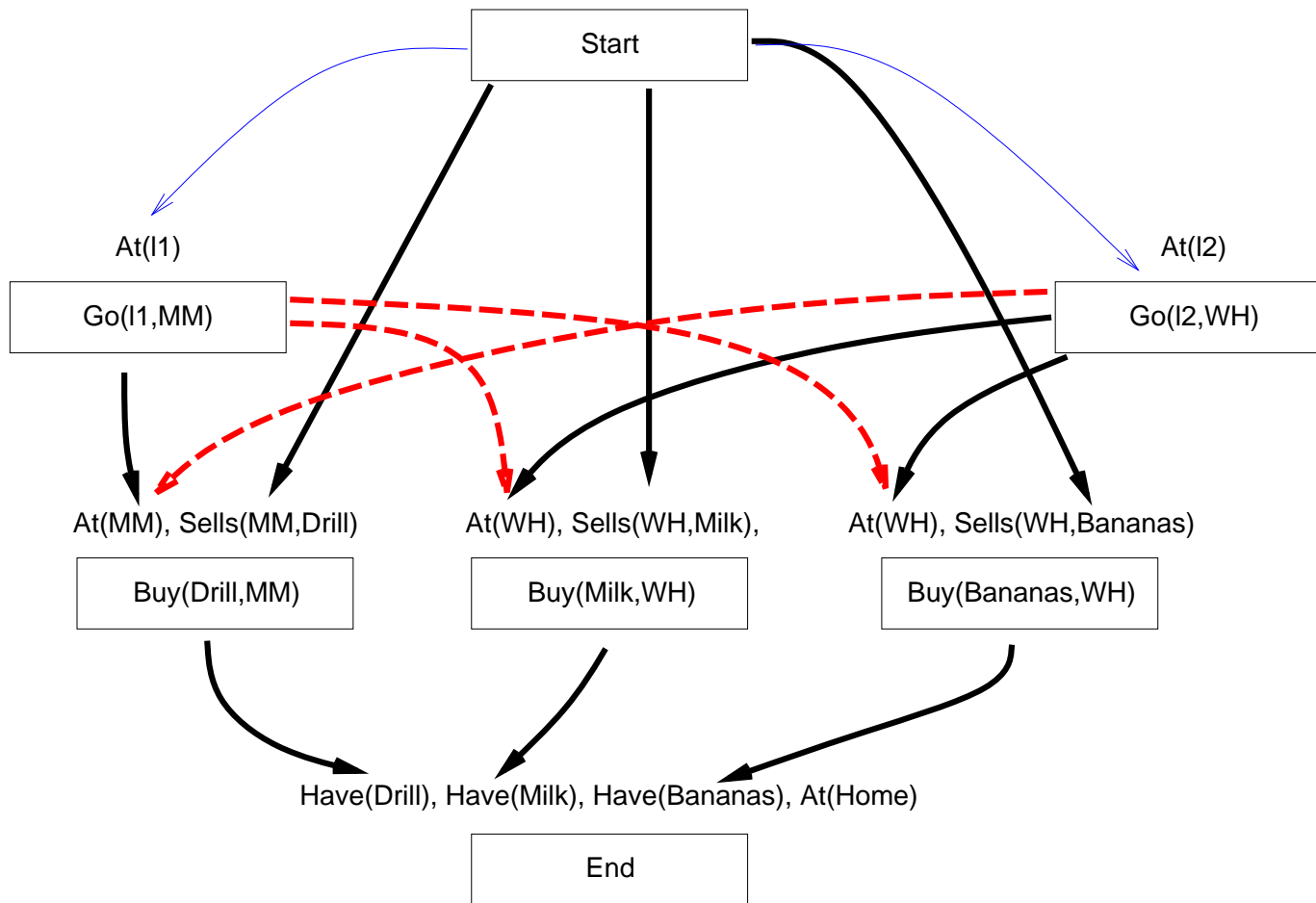
The only possible ways to establish the “Sells” preconditions



Example

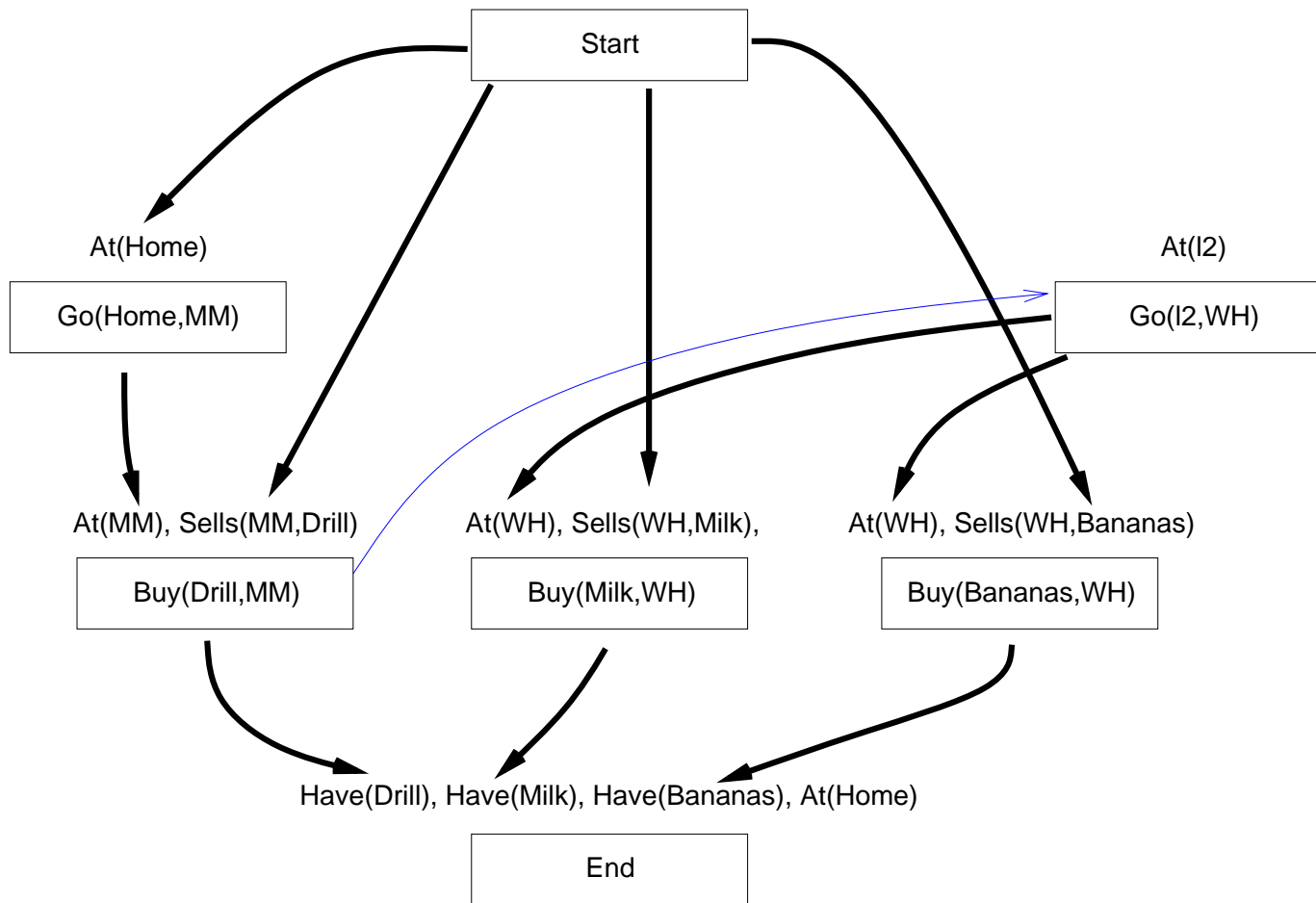
The only ways to establish $At(MM)$ and $At(WH)$.

Note the threats



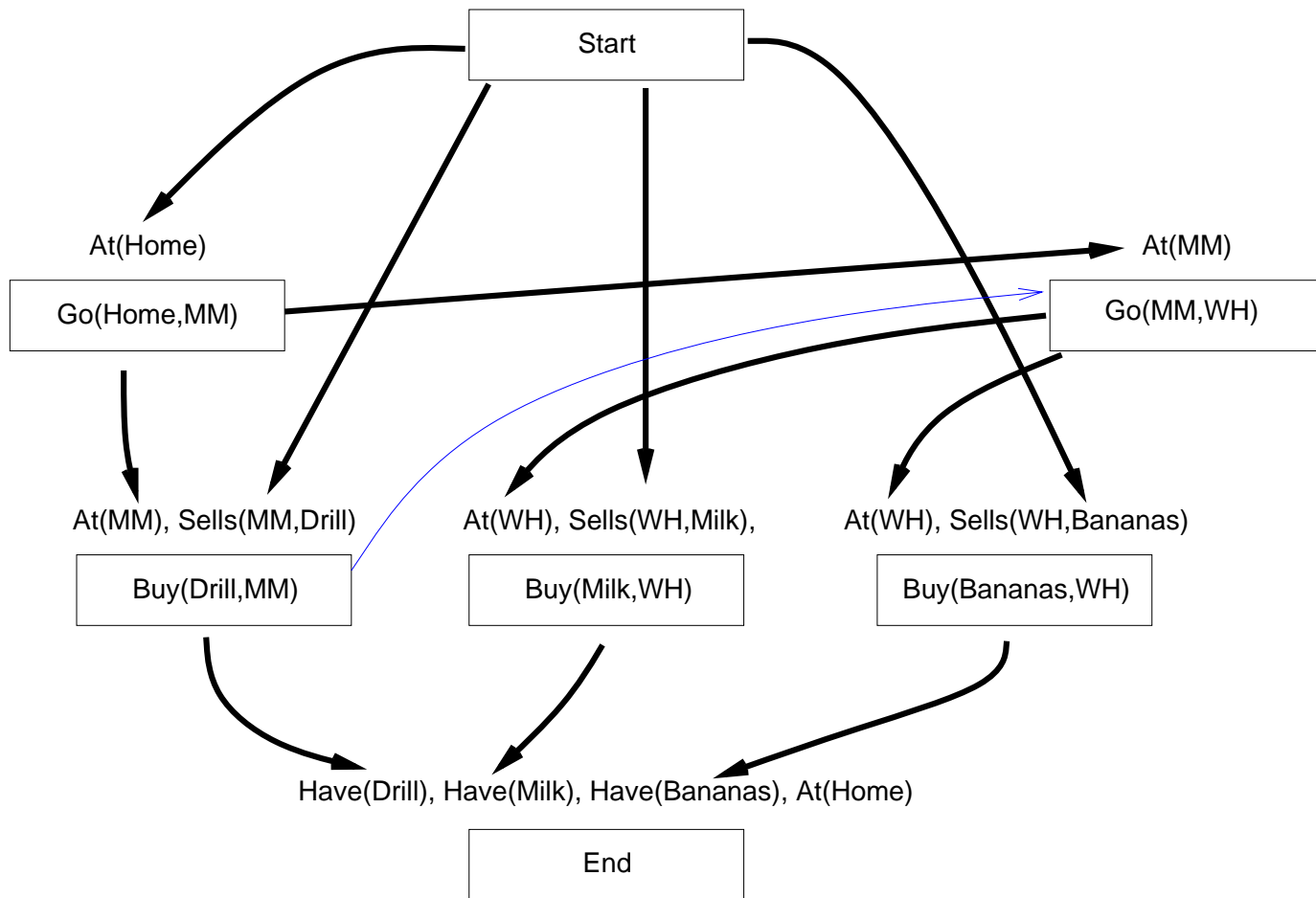
Example

Add binding constraint $l1 = \text{Home}$ and causal link to establish $\text{At}(l1)$



Example

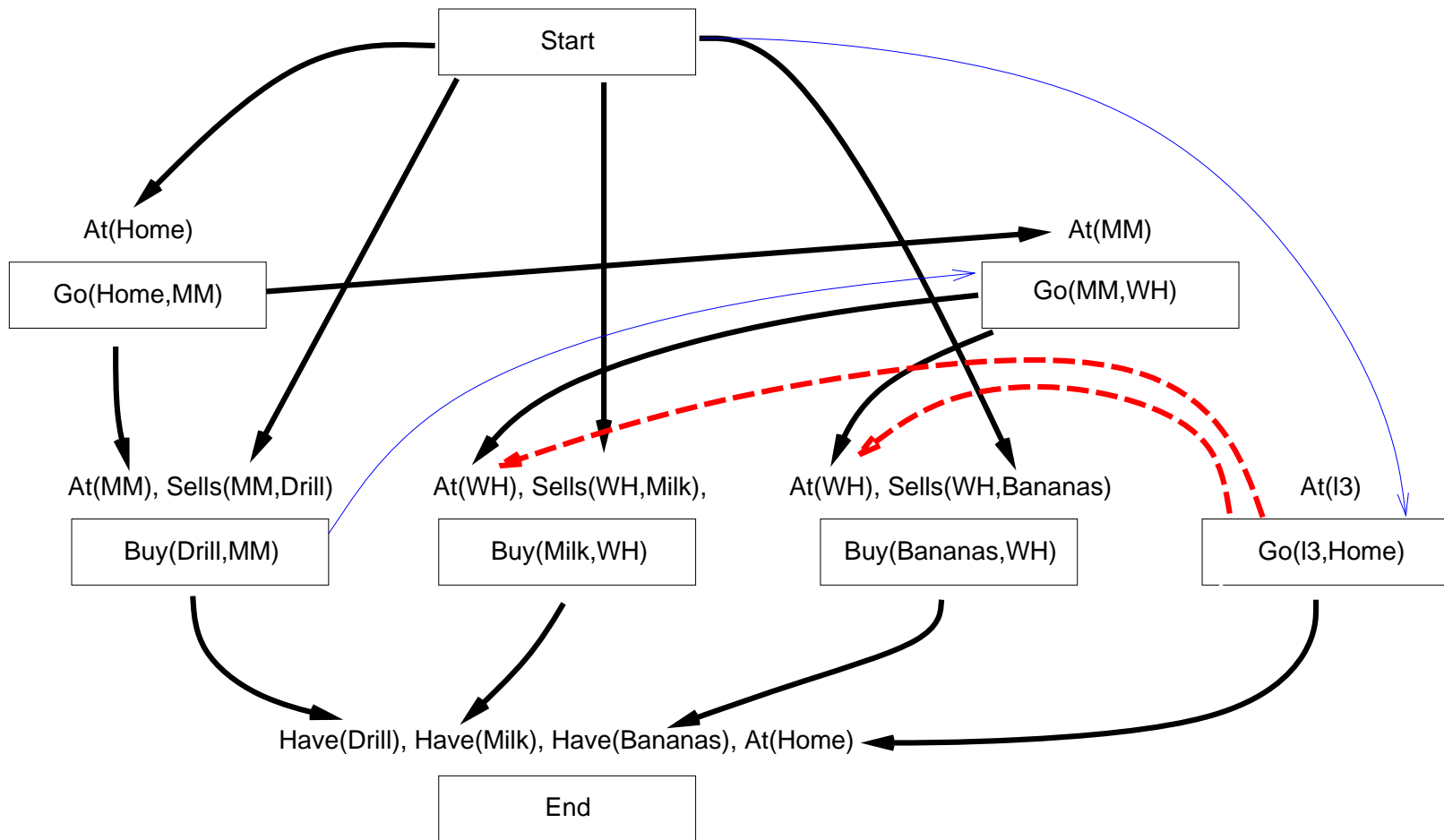
Add binding constraint $l2 = MM$ and causal link to establish $At(l2)$



Example

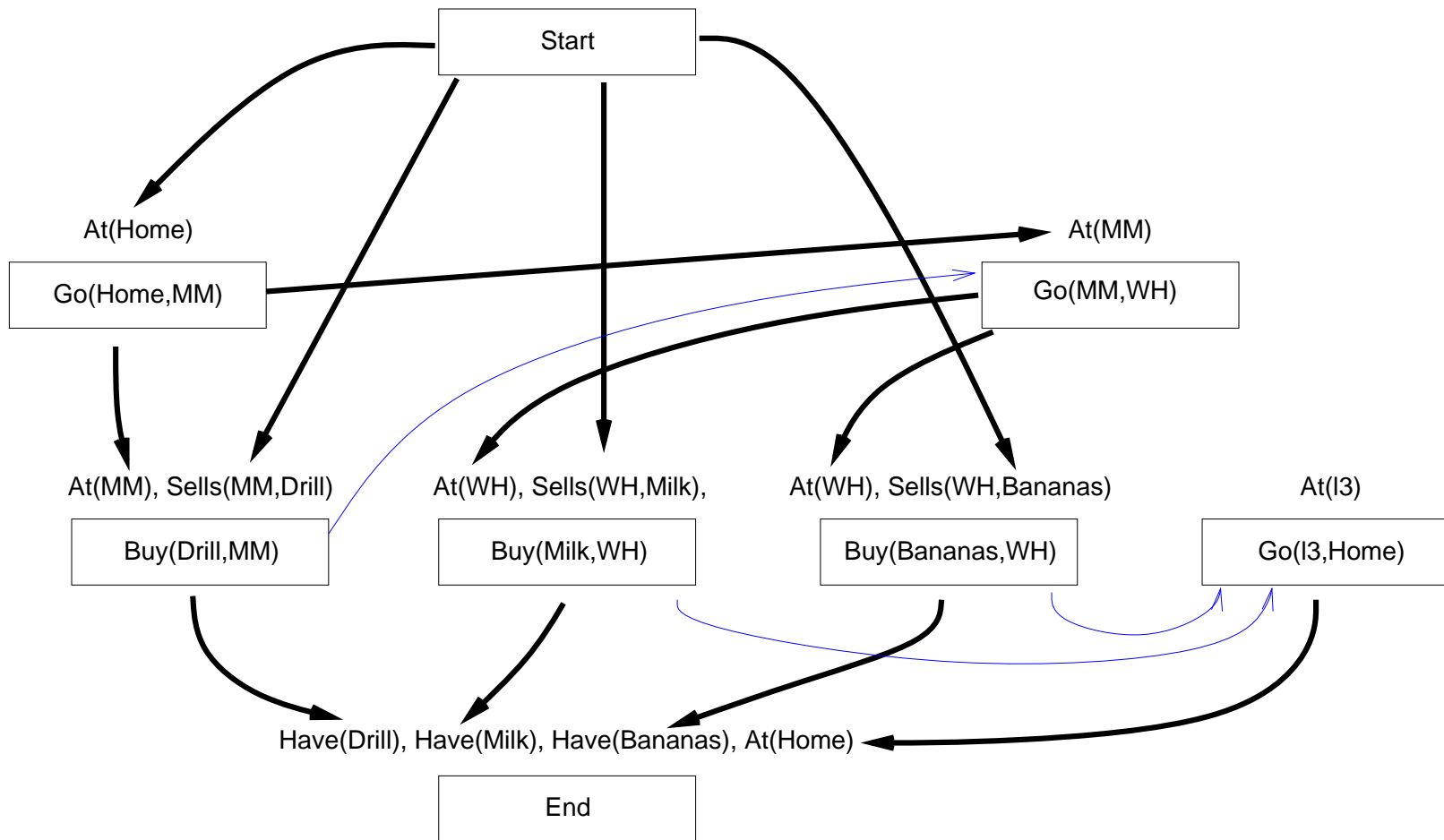
Establish $At(Home)$ for end.

Note the threats.



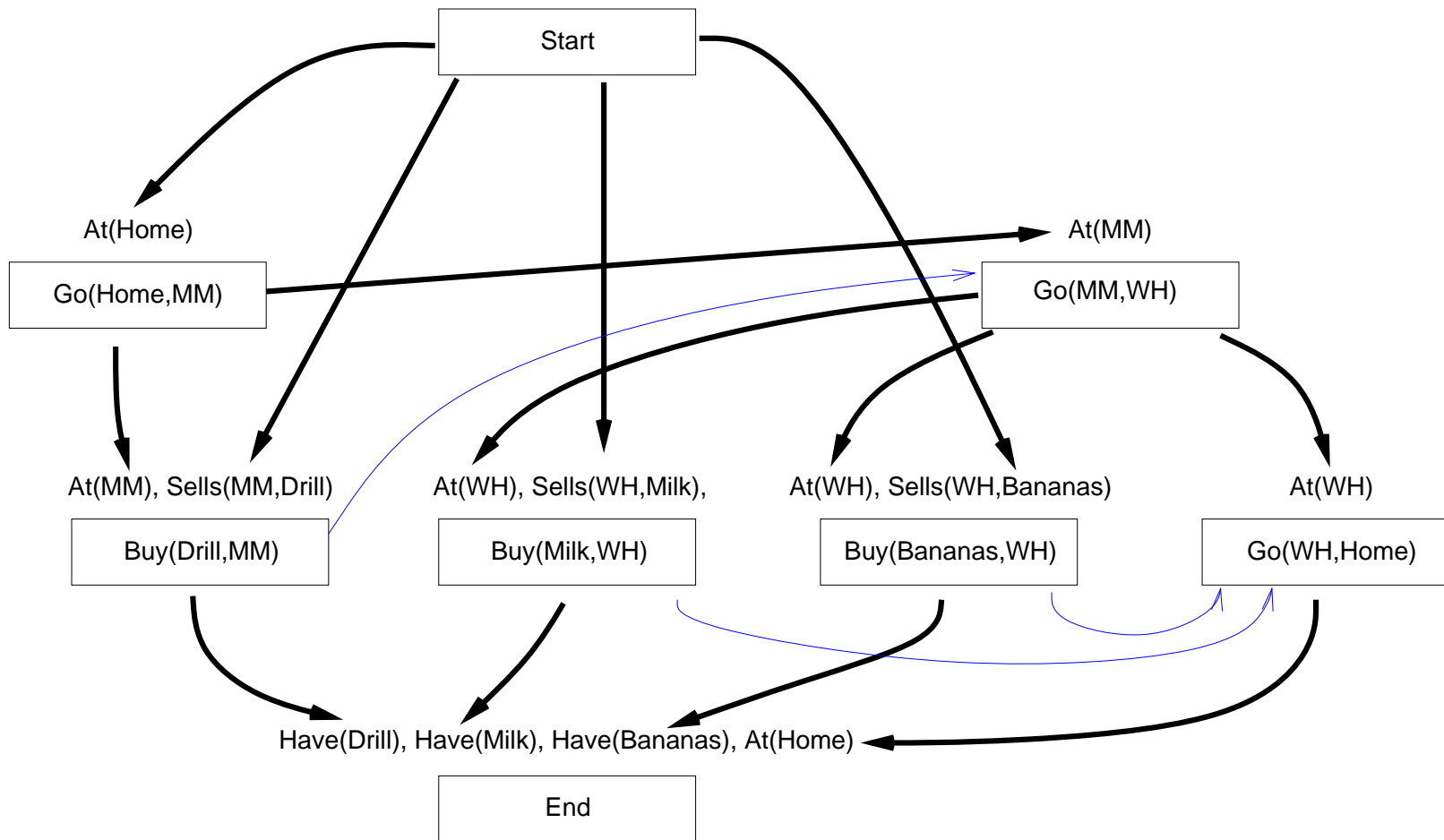
Example

Order Go(Home) after Buy(Milk) and Buy(Banana) to remove the threats.



Example

Add binding constraint $l3 = WH$ and causal link to establish $At(l3)$.
The plan is flawless.



PLANNING-GRAPH TECHNIQUES

CHAPTER 11

Outline

- ◇ Motivation
- ◇ Planning graph
- ◇ Mutual exclusion
- ◇ Plan extraction
- ◇ Example

Motivation

Produce flexible plans. Recall the various types of classical plans:

- **linear plan** (or sequence): totally ordered set $\langle a_1, \dots, a_n \rangle$, $a_i \in A$, such that $\gamma(\dots \gamma(\gamma(s_0, a_1), a_2), \dots, a_n) \in S_G$. Produced by state-space planning approaches.
- **non-linear plan**: partially ordered set $\langle \{a_1, \dots, a_n\}, < \rangle$, $a_i \in A$, such that each linearisation is a totally ordered plan. Produced by plan-state planning approaches.
- **parallel plan**: sequence of parallel actions $\langle \{a_{1,1}, \dots, a_{1,l(1)}\}, \dots, \{a_{1,n}, \dots, a_{1,l(n)}\} \rangle$, $a_{i,j} \in A$, special case of partially ordered plan. Produced by graph-based planning approaches.

Parallel plans are a useful intermediate between linear and non-linear plans.

Motivation

Reduce the branching factor. One way to do this is:

1. Do a fast approximate reachability analysis, i.e., determine whether there is any chance that the goal might be reachable in k steps, and if so, which actions might be useful.
2. Restrict the search to those actions.
3. If the search is unsuccessful, increase k .

Graphplan

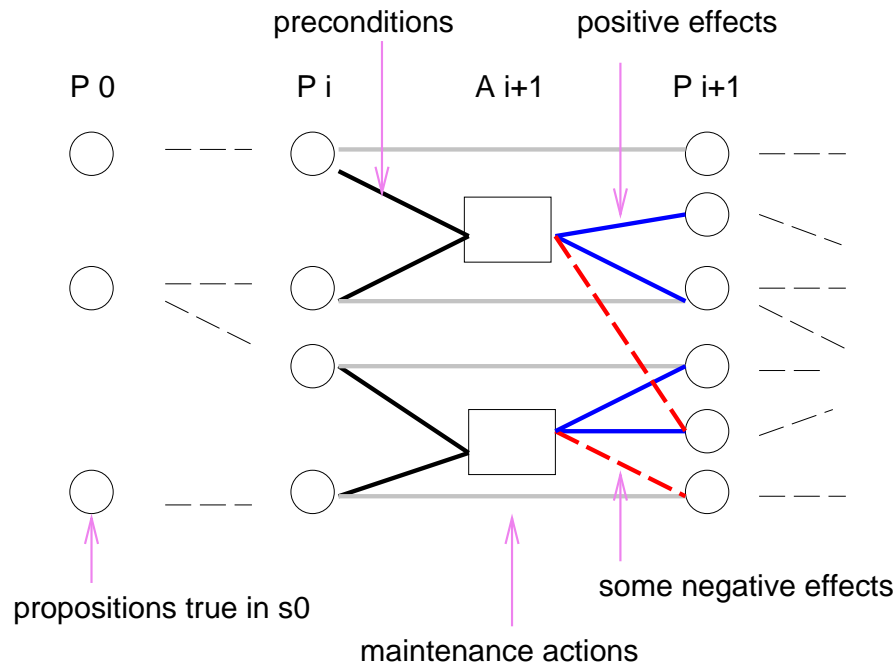
This is how Graphplan [Blum & Furst, IJCAI 1995] implements this idea:

1. Build a **planning graph** which can be viewed as a relaxation of the state space over k steps (note: a step includes several parallel actions). This can be done in polynomial time.
2. The planning graph gives us a **necessary** but insufficient condition for when the goal is reachable in k steps.
3. Attempt to **extract** a parallel plan from the graph using a form of backward search through the graph.
4. If the extraction is unsuccessful, k is incremented, the graph extended, and a new extraction performed, and so on, until a plan is found or we determine that the problem is unsolvable.

The planning graph

Alternating layers of propositions and actions, $P_0, A_1, P_1, \dots, A_i, P_i, \dots, A_k, P_k$:

- $P_0 = s_0$
- A_{i+1} contains the actions that might occur at time step $i + 1$. Their preconditions must belong to P_i . We include maintenance actions (prec p , eff p) for each proposition $p \in P_i$ (represents what happens if no action in the final plan affects p).
- P_{i+1} contains the propositions that are **positive** effects of actions in A_{i+1}

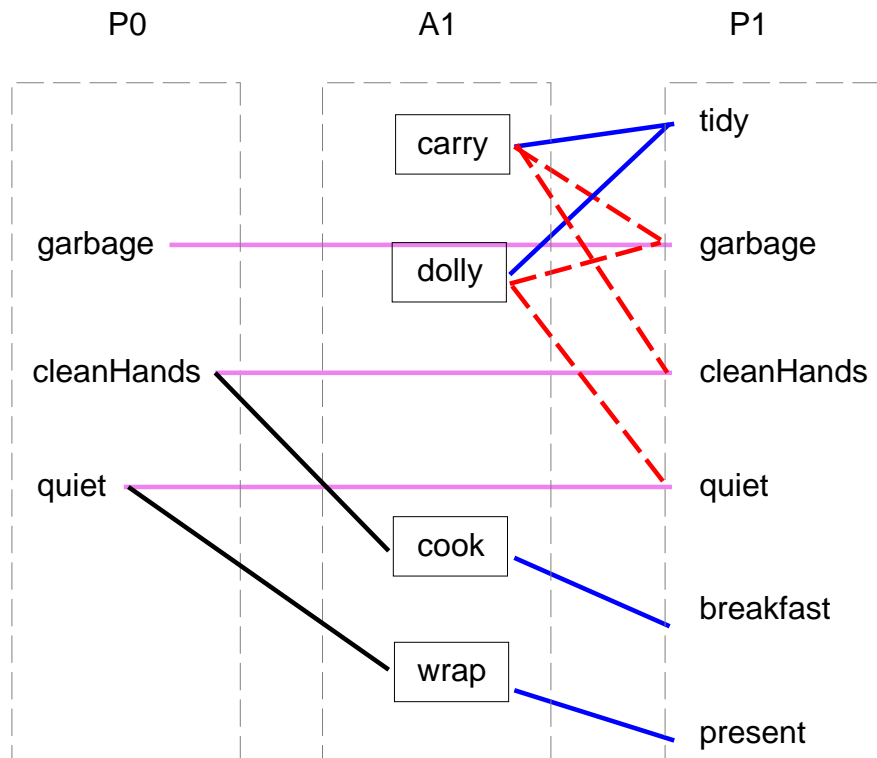


The planning graph: example

Suppose you want to make surprise for your sweetheart who is asleep

Operator	Precondition	Effect
cook()	{cleanHands}	{breakfast}
wrap()	{quiet}	{present}
carry()	{}	{tidy, ¬garbage, ¬cleanHands}
dolly()	{}	{tidy, ¬garbage, ¬quiet}

$s_0 = \{\text{garbage, cleanHands, quiet}\}$
 $g = \{\text{breakfast, present, tidy}\}$



Mutual exclusion

The planning graph records information about pairs of actions which cannot happen in parallel and pairs of propositions which cannot be simultaneously true. These are called **mutex**.

The notion underlying the mutex relation of the graph is called independence.

Two actions are **independent** when executing them in any order (or in parallel) is possible and yields the same result. We must avoid:

- **interference**: one action deletes a precondition of the other (one of the two orderings is not be executable)
- **inconsistence**: one action deletes a positive effect of the other (the two orderings yield different results)

Mutual exclusion

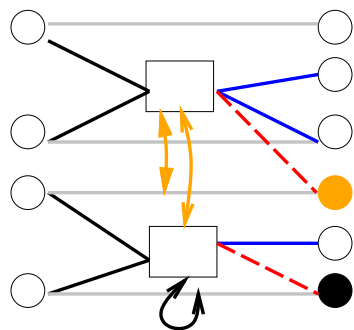
Two actions at the same level of the graph are mutex if they:

- **interfere**: one deletes a precondition of the other
- **are inconsistent**: one deletes a positive effect of the other
- **have competing needs**: they have mutually exclusive preconditions

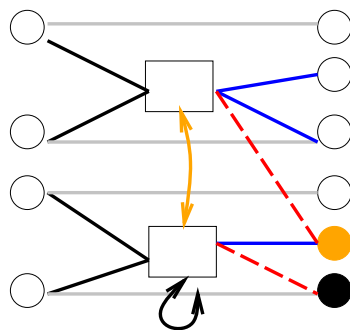
Two propositions at the same level are mutex if they:

- **have inconsistent support**: all ways of achieving both are pairwise mutex

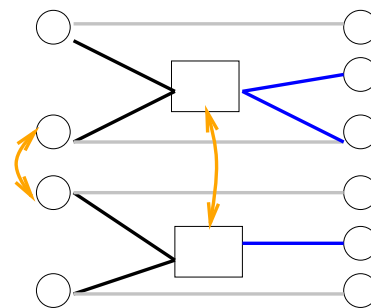
An action appear in A_{i+1} iff its preconditions appear & are mutex-free in P_i .



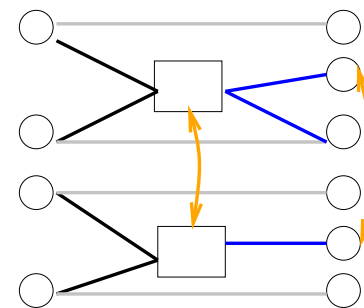
interference



inconsistence



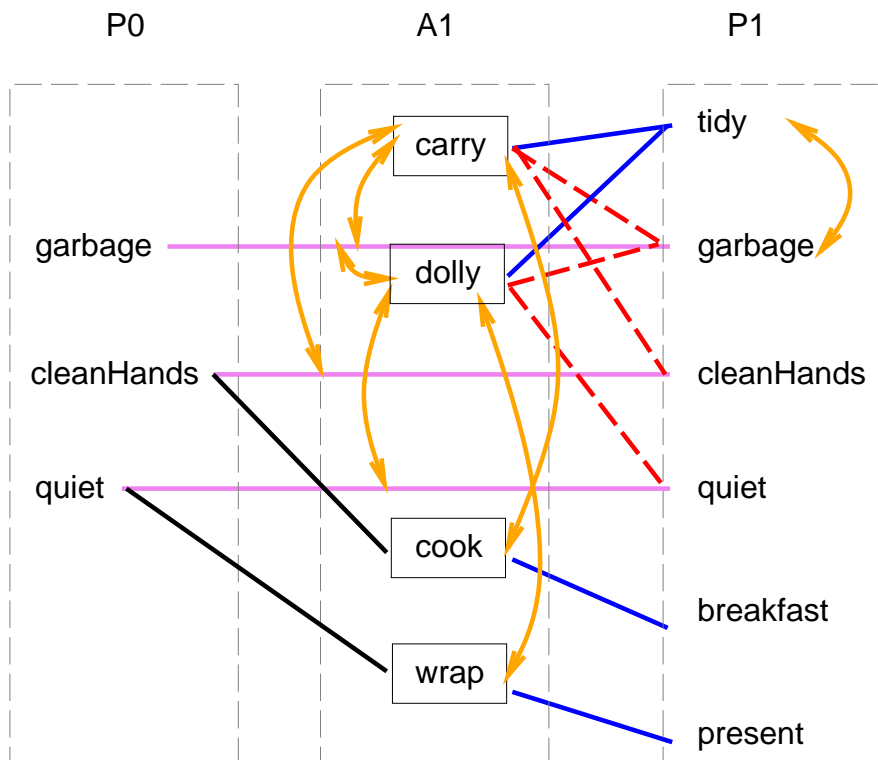
competing needs



inconsistent support

Mutual exclusion: example

- carry & dolly are mutex with the maintenance action for garbage (interference+inconsistence)
- carry is mutex with the maintenance action for cleanHands (interference+inconsistence)
- dolly is mutex with the maintenance action for quiet (interference+inconsistence)
- carry and cook are mutex, dolly and wrap are mutex (interference)
- tidy is mutex with garbage (inconsistent support)



Graph formal definition (if it helps you)

The set of actions A includes maintenance actions m_p with

$$\text{PRE}(m_p) = \text{EFF}^+(m_p) = \{p\}$$

The graph alternates layers of propositions and action nodes $P_0, A_1, P_1, A_2, \dots, A_k, P_k$ and records mutex pairs μA_i and μP_i at each layer, such that:

- $P_0 = s_0$
- $\mu P_0 = \{ \}$
- $A_{i+1} = \{a \in A \mid \text{PRE}(a) \subseteq P_i \text{ and } \forall \{p, p'\} \in \mu P_i \{p, p'\} \not\subseteq \text{PRE}(a)\}$
- $\mu A_{i+1} = \{ \{a, a'\} \subseteq A_{i+1} \mid \text{EFF}^-(a) \cap (\text{PRE}(a') \cup \text{EFF}^+(a')) \neq \{ \} \text{ or } \exists \{p, p'\} \in \mu P_i \text{ s.t. } p \in \text{PRE}(a) \text{ and } p' \in \text{PRE}(a') \}$
- $P_{i+1} = \bigcup_{a \in A_{i+1}} \text{EFF}^+(a)$
- $\mu P_{i+1} = \{ \{p, p'\} \subseteq P_{i+1} \mid \forall a, a' \in A_{i+1} \text{ s.t. } p \in \text{EFF}^+(a) \text{ and } p' \in \text{EFF}^+(a'), \{a, a'\} \in \mu A_{i+1} \}$

Properties of the graph

Necessary condition for plan existence:

If the goal propositions are present and mutex-free at some level P_k

$$g \subseteq P_k \text{ and } \forall \{p, q\} \subseteq g \{p, q\} \notin \mu P_k$$

then a k step parallel plan achieving the goal **might** exist.

The graph has a fixpoint n such that for all $i \geq n$:

$$P_i = P_n, \mu P_i = \mu P_n, A_i = A_n, \text{ and } \mu A_i = \mu A_n$$

The size of the fixpoint graph is polynomial in that of the planning problem.

Graphplan: build the graph up until the necessary condition is reached; try extracting a plan from the graph, if this fails, extend the graph over one more level; repeat until successful or the fixpoint is reached (failure).

Plan extraction

Backward search, from the goal to the initial state.

Works layer by layer:

- Select an open precondition at the current layer, and choose an action producing it. The action must not be mutex with any of the parallel actions already chosen for that layer.
- When there is no more open precondition at that layer, work on achieving, at the previous layer, the preconditions of the chosen actions.

Plan extraction

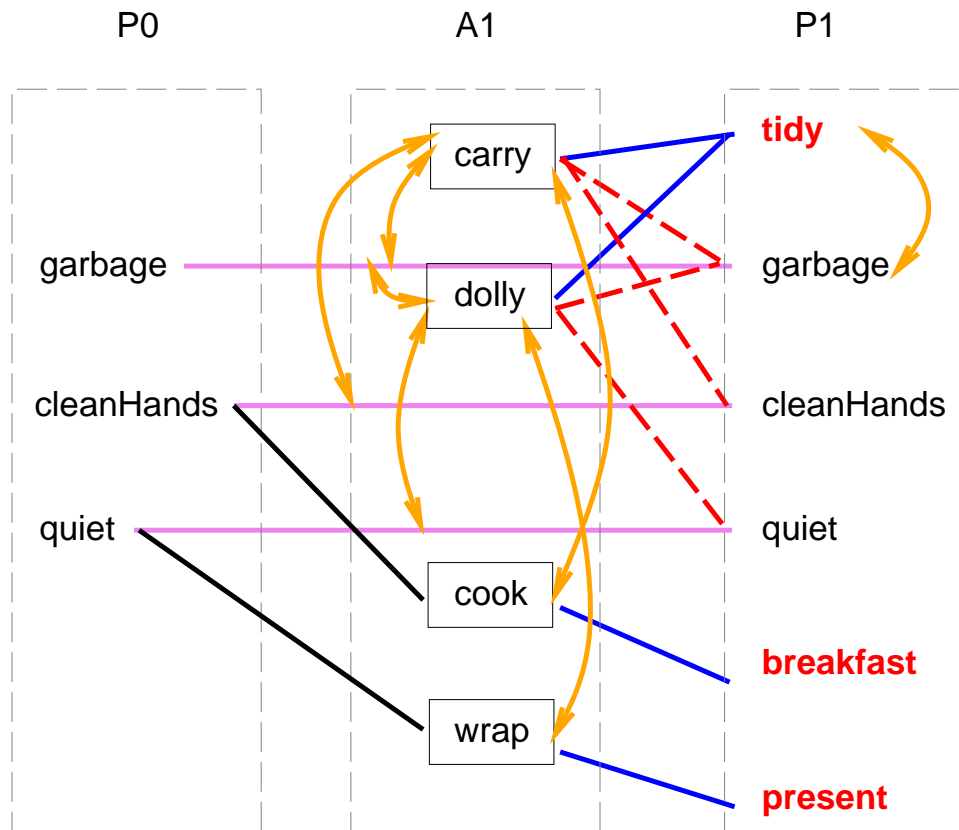
```
function EXTRACT( $i, g_i, \pi_i$ ) returns a parallel plan, or failure
  if  $i = 0$  then return  $\langle \rangle$ 
  if  $g_i \neq \{ \}$  then
    select any  $p \in g_i$ 
     $E \leftarrow \{ a \in A_i \mid p \in \text{EFF}^+(a) \text{ and } \forall b \in \pi_i \{ a, b \} \notin \mu A_i \}$ 
    if  $E = \{ \}$  then return failure
    choose  $a \in E$ 
    return EXTRACT( $i, g_i \setminus \text{EFF}^+(a), \pi_i \cup \{ a \}$ )
  else
     $\pi \leftarrow \text{EXTRACT}(i - 1, \cup_{a \in \pi_i} \text{PRE}(a), \{ \})$ 
    if  $\pi = \text{failure}$  then return failure
    return  $\pi.\pi_i$ 
  end
```

call: EXTRACT($k, g, \{ \}$) where k is the last layer in the graph

Graphplan is sound and complete.

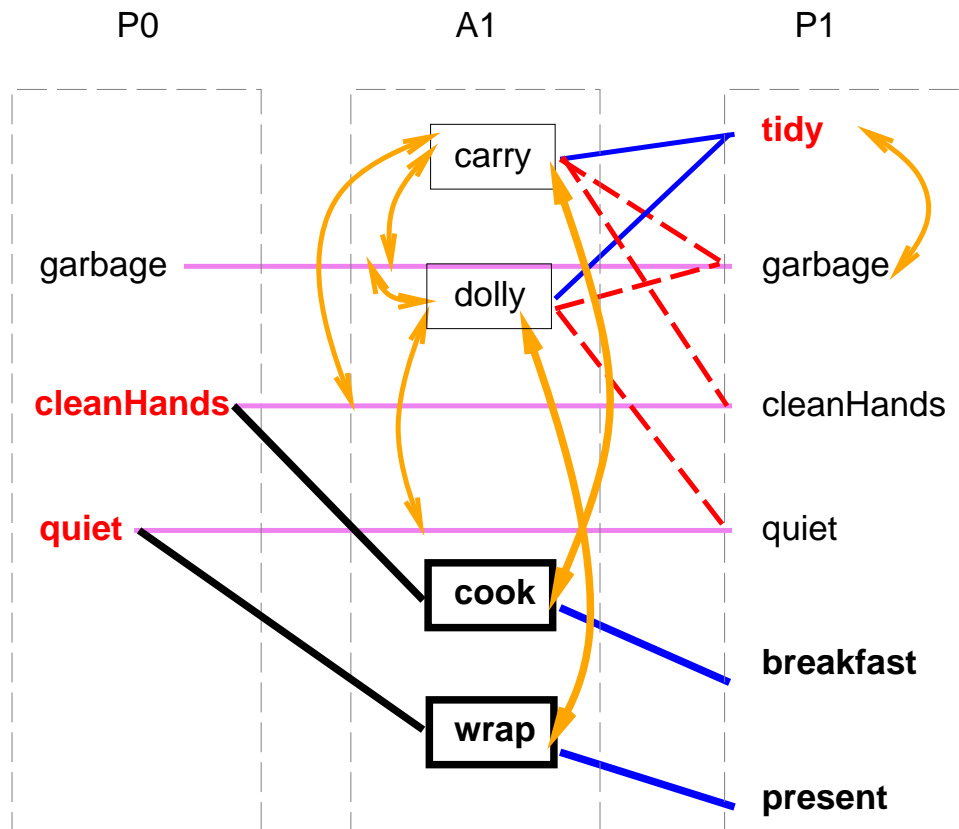
Example

The necessary condition for plan existence is satisfied at level 1 so we can attempt extraction.



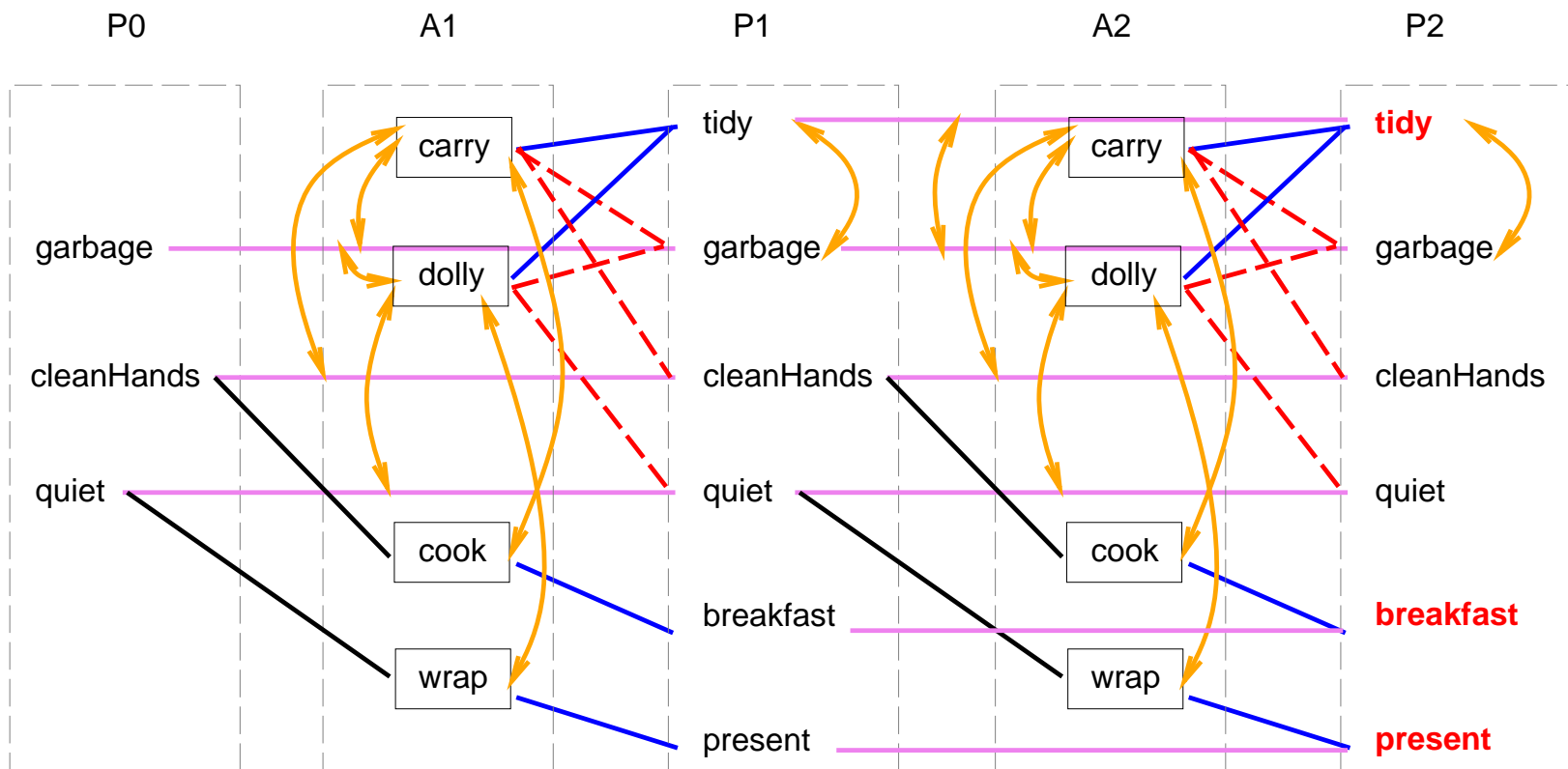
Example

At level 1, we can use cook to produce breakfast, wrap to produce present, but then we cannot achieve tidy because the actions producing it (carry and dolly) are mutex with either cook or wrap. So extraction fails.



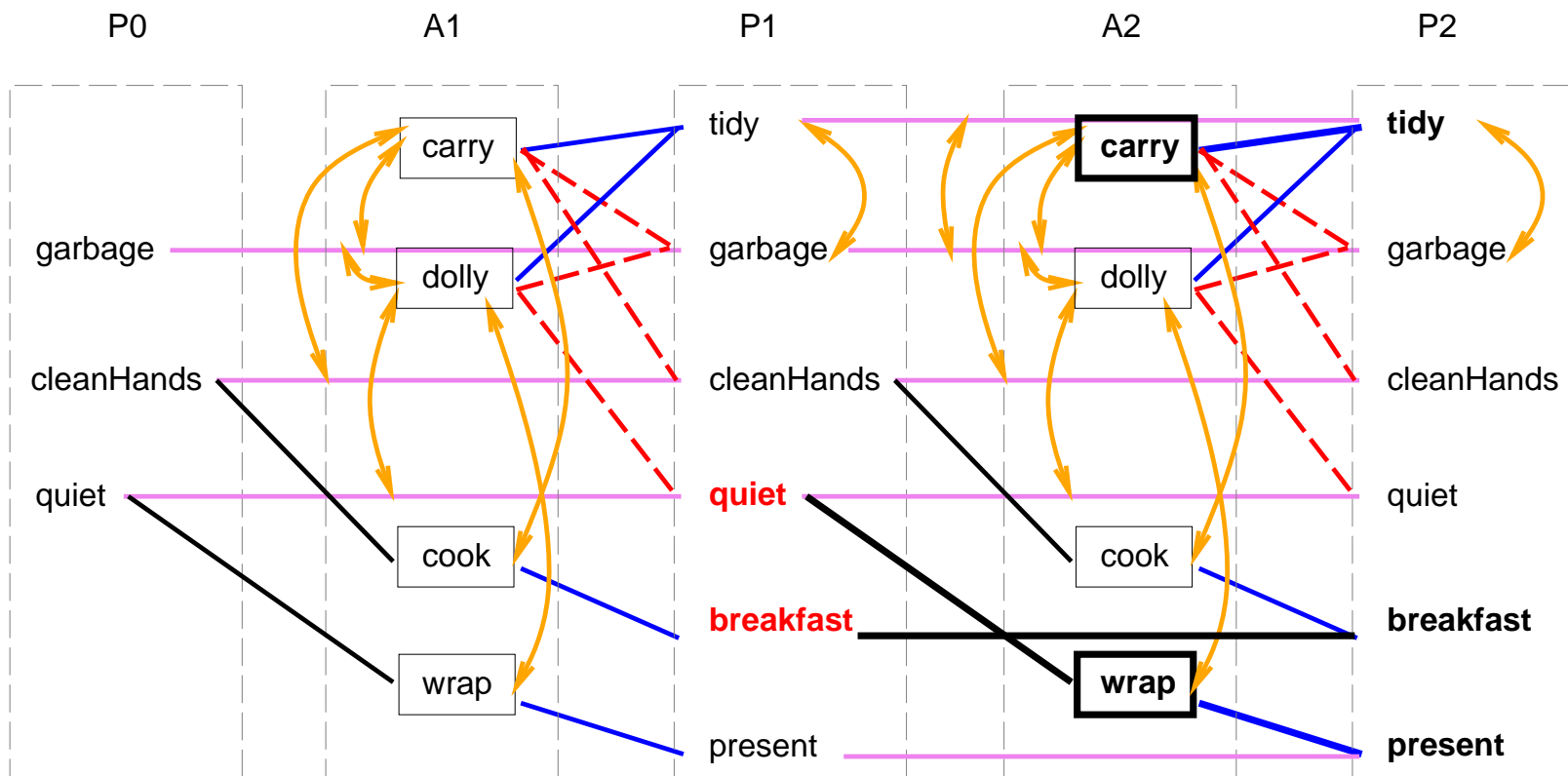
Example

And we extend the graph of one level. Note the apparition of new maintenance actions (for tidy, breakfast, and present), and of a new mutex between the tidy and garbage maintenance actions.



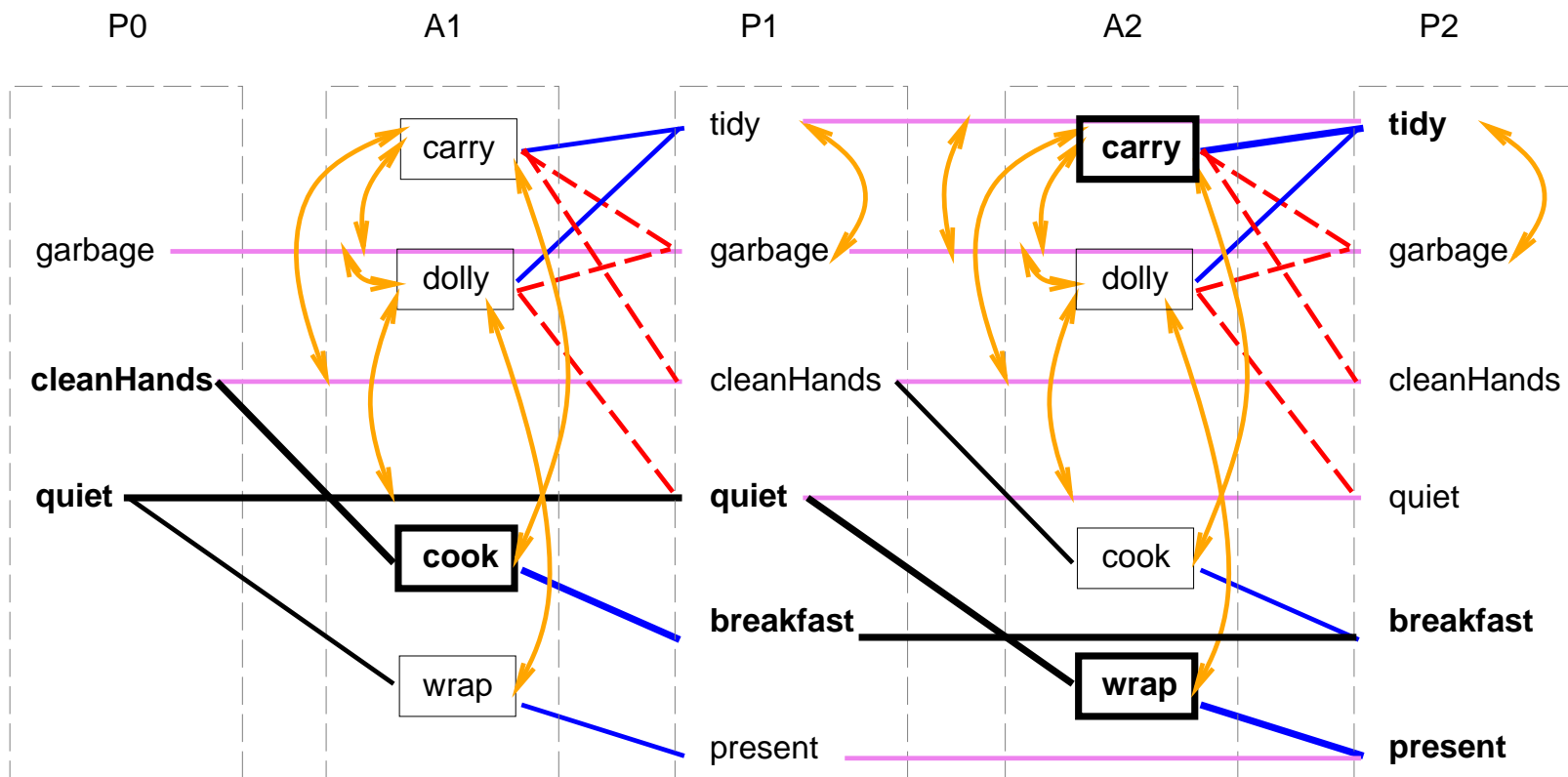
Example

There are 3 possibilities to achieve tidy (maintenance, carry, or dolly), 2 to achieve breakfast (maintenance or cook), and 2 to achieve present (maintenance or wrap), with several combinations being okay. One of them is carry, wrap, and maintenance of breakfast.



Example

There is only one possibility to achieve quiet and breakfast at level 1. This yields a solution whose parallel length is 2.



...

PLANNING AS SATISFIABILITY

CHAPTER 11.5 OF TEXTBOOK
CONTENTS ADAPTED FROM DANA NAU

Outline

- ◇ Motivation
- ◇ Overall Approach
- ◇ Mapping from planning to CNF
- ◇ Example

Big Picture

- Convert planning problem into CNF formula.
- Use any SAT solver to find a solution to the CNF problem.
- Convert CNF solution back into a plan.

Motivation

- No need for planning-specific search algorithms. These often require sophisticated heuristics.
- Take advantage of progress in SAT technology. There is lots of research on solving algorithms.
- It is possible/practical to obtain parallel plans that are optimal as number of time steps (not necessarily as number of actions).

Bounded planning problem

A bounded planning problem is a pair (P, n) :

- P is a planning problem and n is a positive integer.
- Any solution to P of length n is a solution for (P, n) . Here “length” can mean either total number of actions or total number of time steps (makespan). This depends on the particular mapping from planning to CNF.

We use a **ground** representation of the problem. I.e., all atoms and all actions have all parameters instantiated with constant objects.

Overall Approach

Planning algorithm

- Do iterative deepening like in Graphplan:
 - For $n = 0, 1, 2, \dots, N$
 - * Encode (P, n) into a satisfiability problem (i.e., formula) Φ .
 - * Feed Φ to a SAT solver.
 - * If Φ is satisfiable, then convert a solution (model) into a plan and return it.

Encoding planning problems into CNF

- We need to map the problem (P, n) into a CNF formula Φ .
- I.e., we need to define the variables and the clauses of Φ .
- Several encodings are possible.
- E.g., Graphplan based approaches are among the most popular and successful.

Variables in CNF formulation

Boolean variables

- For each ground atom l and for each time step $t \leq n$, define a new boolean variable l_t .
Example: $\text{holding}(\text{blue-arm}, A)_0$ is a boolean variable that tells whether the robot arm *blue-arm* is holding block A at time 0.
- For each ground action a and for each time step $t \leq n - 1$, define a new boolean variable a_t .
Example: $\text{putdown}(\text{blue-arm}, A)_2$ is a boolean variable that tells whether the corresponding action is selected at time step 2.

Clauses in CNF formulation

Formula Φ is a **conjunction** of several types of clauses:

- Describe the initial state.
- Describe the goal.
- Describe the connections between each action and its preconditions and effects.
- **Frame axioms**: Describe what does not change when an action is applied.
- **Exclusion axioms**: Describe how actions exclude each other at a given time $t \leq n - 1$.

Details on clauses

- Formula describing the initial state:

$$\bigwedge_{l \in s_0} l_0 \wedge \bigwedge_{l \notin s_0} \neg l_0$$

- Formula describing the goal (this example allows both positive and negative literals in the goal condition):

$$\bigwedge_{l \in g^+} l_n \wedge \bigwedge_{l \in g^-} \neg l_n$$

- For every ground action a , describe what changes a would make if it were applied at time step i :

$$a_i \Rightarrow \bigwedge_{p \in \text{Precond}(a)} p_i \wedge \bigwedge_{e \in \text{Effects}(a)} e_{i+1}$$

Frame axioms

- State what does not change between steps i and $i + 1$.
- Several ways to write these
- One way: explanatory frame axioms
 - One axiom for each literal l
 - If l changes between steps i and $i + 1$, then an action selected at step i must be responsible for this:

$$(\neg l_i \wedge l_{i+1} \Rightarrow \bigvee_{a \in A | l \in \text{Effects}^+(a)} a_i) \wedge$$

$$(l_i \wedge \neg l_{i+1} \Rightarrow \bigvee_{a \in A | l \in \text{Effects}^-(a)} a_i)$$

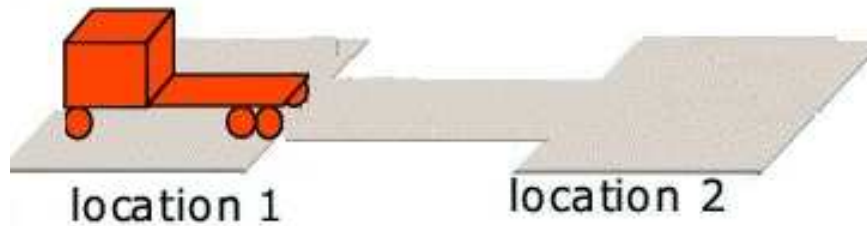
Exclusion axioms

- Add a bunch of clauses in the form $\neg a_i \vee \neg b_i$, where a and b are actions.
- If linear plans are sought, add such clauses for all combinations i, a, b with $i \leq n - 1$ and $a \neq b$. I.e., two actions a and b cannot be applied at the same time i .
- If parallel plans are allowed, restrict such clauses only to conflicting actions a and b . E.g., in a Graphplan based encoding, for each mutex (a, b) at time step (graph level) i , add a clause $(\neg a_i \vee \neg b_i)$.
The same can be done for atom mutexes.

Example

Planning domain:

- One robot r_1
- Two adjacent locations l_1 and l_2
- One operator move



Example

Encode (P, n) where $n = 1$:

- Initial state: $\text{at}(r_1, l_1)$
Encoding: $\text{at}(r_1, l_1)_0 \wedge \neg \text{at}(r_1, l_2)_0$
- Goal: $\text{at}(r_1, l_2)$
Encoding: $\text{at}(r_1, l_2)_1$
- Actions, frame axioms and exclusion axioms: see next slides

Example

Operator:

$\text{move}(?r, ?l, ?l')$

precond: $\text{at}(?r, ?l)$

effect: $\text{at}(?r, ?l') \wedge \neg\text{at}(?r, ?l)$

Encoding:

$\text{move}(r_1, l_1, l_2)_0 \Rightarrow \text{at}(r_1, l_1)_0 \wedge \text{at}(r_1, l_2)_1 \wedge \neg\text{at}(r_1, l_1)_1$

$\text{move}(r_1, l_2, l_1)_0 \Rightarrow \text{at}(r_1, l_2)_0 \wedge \text{at}(r_1, l_1)_1 \wedge \neg\text{at}(r_1, l_2)_1$

$\text{move}(r_1, l_1, l_1)_0 \Rightarrow \text{at}(r_1, l_1)_0 \wedge \text{at}(r_1, l_1)_1 \wedge \neg\text{at}(r_1, l_1)_1$

$\text{move}(r_1, l_2, l_2)_0 \Rightarrow \text{at}(r_1, l_2)_0 \wedge \text{at}(r_1, l_2)_1 \wedge \neg\text{at}(r_1, l_2)_1$

The last two clauses correspond to trivial actions where the start is the same as the destination.

Example

Exclusion axioms:

$(\text{move}(r_1, l_2, l_1), \text{move}(r_1, l_2, l_1))$

We ignore the trivial move actions where the start is the same as the destination.

Encoding:

$\neg \text{move}(r_1, l_2, l_1)_0 \vee \neg \text{move}(r_1, l_1, l_2)_0$

Example

Explanatory frame actions:

$$\neg \text{at}(r_1, l_1)_0 \wedge \text{at}(r_1, l_1)_1 \Rightarrow \text{move}(r_1, l_2, l_1)_0$$

$$\neg \text{at}(r_1, l_2)_0 \wedge \text{at}(r_1, l_2)_1 \Rightarrow \text{move}(r_1, l_1, l_2)_0$$

$$\text{at}(r_1, l_1)_0 \wedge \neg \text{at}(r_1, l_1)_1 \Rightarrow \text{move}(r_1, l_1, l_2)_0$$

$$\text{at}(r_1, l_2)_0 \wedge \neg \text{at}(r_1, l_2)_1 \Rightarrow \text{move}(r_1, l_2, l_1)_0$$

Extracting a Plan

- Suppose we find an assignment that satisfies Φ . This means that P has a solution with n time steps.
- Consider each variable a_i that corresponds to an action and that is assigned the true value. The action a will be a plan step executed at time i .
- In the example, $\text{move}(r_1, l_1, l_2)_0 = \text{true}$ and all other variables generated by actions are false. Hence $\text{move}(r_1, l_1, l_2)$ is a solution.

Summary of classical planning

The **STRIPS representation** enables planning algorithms to exploit the logical structure of the problem. ADL is a useful extension of STRIPS.

State-space planning produces totally-ordered plans by a forward or backward search in the state space. This requires domain-independent heuristics or domain-specific control rules to be efficient.

Plan-space planning produces partially-ordered plans. This approach does not commit to orderings or bindings unless necessary. It searches the space of partial plans, refining the plan at each step to remove flaws.

Summary of classical planning

Graph-based planning produces parallel plans. A planning graph is a relaxation of the state space, which gives us a necessary condition for the existence of a parallel plan of a given length. If one really exist, it can be extracted by backward search through the graph.

Planning as SAT maps a planning problem into CNF and uses a standard SAT solver to find a solution.

Current planning research extends classical planning to handle time, uncertainty, partial observability, complex cost measures, extended goals. . .