Foundations of Artificial Intelligence
Chapter 11: **Planning**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it
https://disi.unitn.it/rseba/DIDATTICA/fai_2026/

Teaching assistant: Nicola Debole, nicola.debole@unitn.it,

M.S. Course “Artificial Intelligence Systems”, academic year 2026-2027

Last update: Monday 7th September, 2026, 15:25

Outline

Outline

Outline

Automated Planning (aka “Planning”)

Automated Planning

Synthesize a sequence of actions (plan) to be performed by an agent leading from an initial state of the world to a set of target states (goal)

- Planning is both:
  - an application per se
  - a common activity in many applications
    (e.g. design & manufacturing, scheduling, robotics,...)
- Similar to problem-solving agents (Ch.03), with factored/structured representation of states
- “Classical” Planning (this chapter):
  fully observable, deterministic, static environments with single agents

Automated Planning [cont.]

Automated Planning

- Given:
  - an initial state
  - a set of actions you can perform
  - a (set of) state(s) to achieve (goal)
- Find:
  - a plan: a partially- or totally-ordered set of actions needed to achieve the goal from the initial state

Decidability and Complexity

- PlanSAT: the question of whether there exists any plan that solves a planning problem
  - decidable for classical planning
  - with function symbols, the number of states becomes infinite
    $\Longrightarrow$ undecidable
  - in PSPACE
    - harder than NP, no polynomial-size witness (e.g., Tower of Hanoi)
- Bounded PlanSAT: the question of whether there exists any plan of a given length k or less
  - can be used for optimal-length plan
  - decidable for classical planning
  - decidable even in the presence of function symbols
  - in PSPACE, NP for many problems of interest

Outline

A Language for Planning: PDDL

Planning Domain Definition Language (PDDL)

- A state is a conjunction of fluents: ground, function-less atoms
  - ex: $Poor \wedge Unknown$, $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - examples of non-fluents:
    $At(x, y)$ (non ground), $\neg Poor$ (negated), $At(Father(Fred), Sydney)$ (not function-less)
  - closed-world assumption: all non-mentioned fluents are false
  - unique-name assumption: distinct names refer to distinct objects
- Actions are described by a set of action schemata
  - concise description: describe which fluent change

  $\Longrightarrow$ the other fluents implicitly maintain their values
- Action Schema: consists in action name, a list of variables in the schema, the precondition, the effect (aka postcondition)
  - precondition and effect are conjunctions of literals
    (positive or negated atomic sentences)
  - lifted representation: variables implicitly universally quantified
- Can be instantiated into (ground) actions

PDDL: Example

- Action schema:
  $Action(Fly(p, from, to),$
  $PRECOND : Plane(p) \wedge Airport(from) \wedge Airport(to) \wedge At(p, from)$
  $EFFECT \ : \neg At(p, from) \wedge At(p, to))$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND : Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge At(P_1, SFO)$
  $EFFECT \ : \neg At(P_1, SFO) \wedge At(P_1, JFK))$

A Language for Planning: PDDL [cont.]

- Precondition: must hold to ensure the action can be executed
  - defines the states in which the action can be executed
  - action is applicable in state s if the preconditions are satisfied by s
- Effect: represent the effects of the action on the world
  - defines the result of executing the action
- Add list (ADD(a)): (the fluents in) the positive literals in the action's effects
  - ex: $\{At(p, to)\}$
- Delete list (DEL(a)): (the fluents in) the negative literals in the action's effects
  - ex: $\{At(p, from)\}$
- Result of action a in state s: RESULT(s,a)$\stackrel{\text{def}}{=}$(s$\setminus$DEL(a) $\cup$ ADD(a))
  - start from s
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex: $Fly(P_1, SFO, JFK) \Longrightarrow$ remove $At(P_1, SFO)$, add $At(P_1, JFK)$

PDDL: Example [cont.]

- Action schema:
  $Action(Fly(p, from, to),$
  $PRECOND : Plane(p) \wedge Airport(from) \wedge Airport(to) \wedge At(p, from)$
  $EFFECT \ : \neg At(p, from) \wedge At(p, to))$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND : Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge At(P_1, SFO)$
  $EFFECT \ : \neg At(P_1, SFO) \wedge At(P_1, JFK))$
- $s : At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge ...$

$\Longrightarrow$ $s' : At(P_1, JFK) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge ...$

Sometimes we want to propositionalize a PDDL problem:
replace each action schema with a set of ground actions.

- Ex: $...At\_P_1\_SFO \wedge Plane\_P_1 \wedge Airport\_SFO \wedge Airport\_JFK)...$

A Language for Planning: PDDL [cont.]

Time in PDDL

- Fluents do not explicitly refer to time
- Times and states are implicit in the action schemata:
  - the precondition always refers to time t
  - the effect to time t+1.

PDDL Problem

- PDDL problem: a planning domain, an initial state and a goal
  - a planning domain is a set of action schemata
  - the initial state is a conjunction of ground atoms (positive literals)
    - closed-world assumption: any not-mentioned atoms are false
  - the goal is a conjunction of literals (positive or negative)
    - may contain variables, which are implicitly existentially quantified
    - a goal g may represent a set of states (the set of states entailing g)
- Ex: goal: $At(p, SFO) \wedge Plane(p)$:
  - variable “p” implicitly means “for some plane p”
  - the state $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge ...$ entails g

A Language for Planning: PDDL [cont.]

Planning as a search problem

Contains all components of a search problem

- an initial state
- an ACTIONS function
- a RESULT function
- and a goal test

Example: Air Cargo Transport

$$
\begin{aligned}
&Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \\
&\quad \land Airport(JFK) \land Airport(SFO)) \\
&Goal(At(C_1, JFK) \land At(C_2, SFO)) \\
&Action(Load(c, p, a), \\
&\quad \text{PRECOND: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } \neg At(c, a) \land In(c, p)) \\
&Action(Unload(c, p, a), \\
&\quad \text{PRECOND: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } At(c, a) \land \neg In(c, p)) \\
&Action(Fly(p, from, to), \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

One solution: $[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK), Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)]$

Example: Spare Tire Problem

$$
\begin{array}{l}
Init(Tire(Flat) \ \wedge \ Tire(Spare) \ \wedge \ At(Flat, Axle) \ \wedge \ At(Spare, Trunk)) \\
Goal(At(Spare, Axle)) \\
Action(Remove(obj, loc), \\
\quad \text{PRECOND: } At(obj, loc) \\
\quad \text{EFFECT: } \neg \, At(obj, loc) \ \wedge \ At(obj, Ground)) \\
Action(PutOn(t, \ Axle), \\
\quad \text{PRECOND: } Tire(t) \ \wedge \ At(t, Ground) \ \wedge \ \neg \, At(Flat, Axle) \\
\quad \text{EFFECT: } \neg \, At(t, Ground) \ \wedge \ At(t, Axle)) \\
Action(LeaveOvernight, \\
\quad \text{PRECOND:} \\
\quad \text{EFFECT: } \neg \, At(Spare, Ground) \ \wedge \ \neg \, At(Spare, Axle) \ \wedge \ \neg \, At(Spare, Trunk) \\
\qquad \wedge \neg \, At(Flat, Ground) \ \wedge \ \neg \, At(Flat, Axle) \ \wedge \ \neg \, At(Flat, \ Trunk))
\end{array}
$$

(We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.)

One solution: [*Remove*(*Flat*, *Axle*), *Remove*(*Spare*, *Trunk*), *PutOn*(*Spare*, *Axle*)]

Example: Blocks World

$Init(On(A, Table) \land On(B, Table) \land On(C, A)$
$\quad \land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C))$
$Goal(On(A, B) \land On(B, C))$
$Action(Move(b, x, y),$
$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land$
$\quad\quad (b \neq x) \land (b \neq y) \land (x \neq y),$
$\quad$ EFFECT: $On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$
$Action(MoveToTable(b, x),$
$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Block(b) \land (b \neq x),$
$\quad$ EFFECT: $On(b, Table) \land Clear(x) \land \neg On(b, x))$

One solution: [*MoveToTable*(*C*, *A*), *Move*(*B*, *Table*, *C*), *Move*(*A*, *Table*, *B*)]

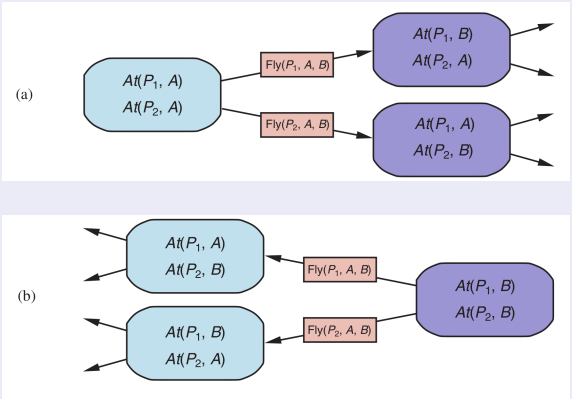
Outline

Outline

Two Main Approaches

(a) Forward search (aka progression search)
- start in the initial state
- use actions to search forward for a goal state

(b) Backward search (aka regression search)
- start from goals
- use reverse actions to search forward for the initial state



Forward Search

- Forward search (aka progression search)
  - choose actions whose preconditions are satisfied
  - add positive effects, delete negative
- Goal test: does the state satisfy the goal?
- Step cost: each action costs 1

$\Longrightarrow$ We can use any of the search algorithms from Ch. 03, 04

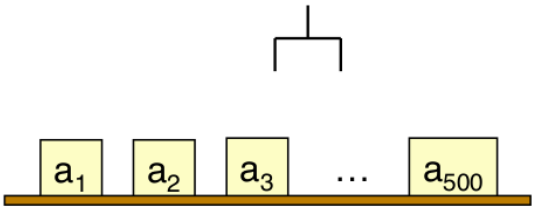
  - need keeping track of the actions used to reach the goal
- Breadth-first and best-first
  - Sound: if they return a plan, then the plan is a solution
  - Complete: if a problem has a solution, then they will return one
  - Require exponential memory wrt. solution length! $\Longrightarrow$ unpractical
- Depth-first search and greedy search
  - Sound
  - Not complete
    - may enter in infinite loops
    - (classical planning only): made complete by loop-checking
  - Require linear memory wrt. solution length

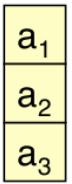
Branching Factor of Forward Search

- Planning problems can have huge state spaces
- Forward search can have a very large branching factor
  - ex: $pickup(a_1), pickup(a_2), ..., pickup(a_{500})$

$\Longrightarrow$ Forward-search can waste time trying lots of irrelevant actions

$\Longrightarrow$ Need a good heuristic to guide the search





Backward Search (aka Regression or Relevant-States)

- Predecessor (sub)goal g' of ground goal g via ground action a:
  $Pos(g') \stackrel{\text{def}}{=} (Pos(g) \setminus Add(a)) \cup Pos(Precond(a))$
  $Neg(g') \stackrel{\text{def}}{=} (Neg(g) \setminus Del(a)) \cup Neg(Precond(a))$
- Note: Both g and g' represent many states
  - irrelevant ground atoms unassigned

Example

- Consider the goal $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the ground action:
  $Action(Unload(C_1, P_1, SFO),$
  $PRECOND : In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO)$
  $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, P_1))$
- This produces the sub-goal g':
  $In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Both g' and g represent many states
  - e.g. truth value of $In(C_3, P_2)$ irrelevant

Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations

  $\Longrightarrow$ no need to produce a goal for every possible instantiation
- use the most general unifier $\Longrightarrow$ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

Example

- Consider the goal $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:
  $Action(Unload(C_1, p', SFO)$,
  $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$
  $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal g':
  $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, ...$

Backward Search [cont.]

Which action to choose?

- Relevant action: could be the last step in a plan for goal g
  - at least one of the action's effects (positive or negative) must unify with an element of the goal
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

Example

- Ex: consider the goal $At(C_1, SFO) \wedge At(C_2, JFK)$
  - $Action(Unload(C_1, p', SFO), ...)$ is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), ...)$ is not relevant
  - $Action(Load(C_2, p', JFK), ...)$ is not consistent $\Longrightarrow$ is not relevant

\+ B.S. typically keeps the branching factor lower than F.S.

\- B.S. reasons with state sets
$\Longrightarrow$ makes it harder to come up with good heuristics

- Most planners work with forward search plus heuristics

Outline

Heuristics for (Forward-Search) Planning

A^* for Planning

- Recall: A^* is a best-first algorithm which
  - uses an evaluation function f(s) = g(s) + h(s),
  - g(s): (exact) cost to reach s
  - h(s): admissible (optimistic) heuristics
    (never overestimates the distance to the goal)
- A technique for admissible heuristics: problem relaxation

  $\Longrightarrow$ h(s): the exact cost of a solution to the relaxed problem
- Forms of problem relaxation exploiting problem structure
  - Add arcs to the search graph $\Longrightarrow$ make it easier to search
    - ignore-preconditions heuristics
    - ignore-delete-lists heuristics
  - Clustering nodes (aka state abstraction) $\Longrightarrow$ reduce search space
    - ignore less-relevant fluents

Ignore (some) Preconditions Heuristics

- Ignore all preconditions drops all preconditions from actions
  - every action is applicable in any state
  - any single goal literal can be satisfied in one step
    (or there is no solution)
  - fast, but over-optimistic
- Ignore some selected (less relevant) preconditions
  - relevance based on heuristics or domain-depended criteria

Ignore-Preconditions Heuristics: Example

Sliding tiles

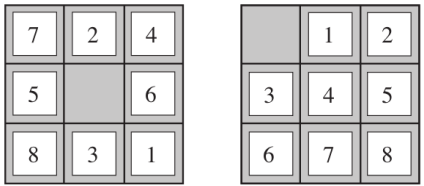
$Action(Slide(t, s_1, s_2),$
$PRECOND : Tile(t) \wedge Blank(s_2) \wedge On(t, s_1) \wedge Adjacent(s_1, s_2)$
$EFFECT : On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2))$

- Remove the preconditions $Blank(s_2) \wedge Adjacent(s_1, s_2)$

  $\Longrightarrow$ we get the number-of-misplaced-tiles heuristics

- Remove the precondition $Blank(s_2)$

  $\Longrightarrow$ we get the Manhattan-distance heuristics



(Courtesy of Maria Simi, UniPI)

Ignore Delete-list Heuristics

- Assumption: goals & preconditions contain only positive literals
  - reasonable in many domains
- Idea: Remove the delete lists from all actions
  - No action will ever undo the effect of actions

  $\Longrightarrow$ once a fluent is made true by one action, it remains true forever

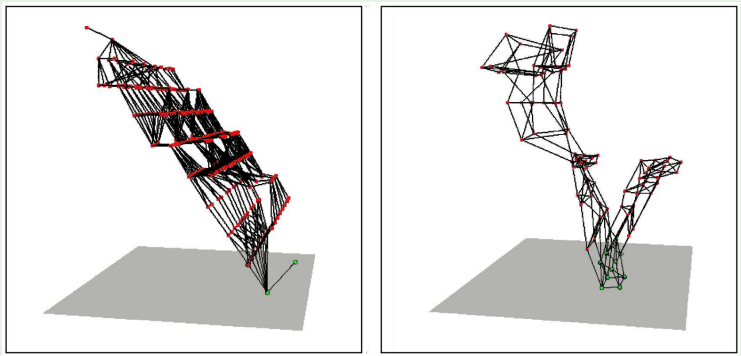
  $\Longrightarrow$ there is a monotonic progress towards the goal
- Still NP-hard to find the optimal solution of the relaxed problem
  - can be approximated in polynomial time, with hill-climbing
- Can be very effective for some problems

Ignore Delete-list Heuristics: Example (Hoffmann'05)

- Planning state spaces with ignore-delete-lists heuristic
  - height above the bottom plane is the heuristic score of a state
  - states on the bottom plane are goals

$\Longrightarrow$ No local minima, non dead-ends, non backtracking

$\Longrightarrow$ Search for the goal is straightforward for hill-climbing



Exercise

Consider the formalization of the sliding tiles, and relax it by applying the Delete-list Heuristics

- Consider one instance and find one solution of the relaxed problem.
- compare the number of steps performed against the value returned by the Manhattan-distance heuristic

State Abstractions

- Many-to-one mapping from states in the ground/original representation of the problem to a more abstract representation
  - drastically reduces the number of states
- Common strategy: ignore some (less-relevant) fluents
  - drop k fluents $\Longrightarrow$ reduce search space by 2^k factors
  - relevance based on (heuristic) evaluation or domain knowledge

- Air cargo problem: 10 airports, 50 planes, 200 pieces of cargo
  $\Longrightarrow$ $10^{50} \cdot (50 + 10)^{200} \approx 10^{405}$ states (*)
- Consider particular problem in that domain
  - all pieces are at 5 airports $\Longrightarrow$ only 5 pieces considered
  - all pieces at a given airport have the same destination $\Longrightarrow$ only 5 planes considered
- Abstraction: drop all "At" and "In" fluents except for these involving one plane and one piece at each of the the 5 airports
  $\Longrightarrow$ $10^5 \cdot (5 + 10)^5 \approx 10^{11}$ states (*)
  - abstract solution shorter than ground solutions $\Longrightarrow$ admissible
  - abstract solution easy to extend: add Load and Unload actions for the other pieces

(*) wrong in AIMA III Ed, corrected in later editions

Other Strategies for Planning

Other strategies to define heuristics

- Problem decomposition
  - “divide & conquer” problem into subproblem
  - solve subproblems independently
- Using a data structure called “planning graph” (next section)

Outline

Outline

Generalities

Planning Graph

- A data structure which is a rich source of information:
  - can be used to give better heuristic estimates h(s)
  - can drive an algorithm called Graphplan
- A polynomial-size over-approximation to the (exponential) search tree
  - can be constructed very quickly

\- cannot answer definitively if goal g is reachable from initial state

\+ may discover that the goal is not reachable

\+ can estimate the most-optimistic step # to reach g

$\Longrightarrow$ it can be used to derive an admissible heuristic $h(s)$

Deals with ground states and actions only

Planning Graph: Definition

- A directed graph, built forward and organized into levels
  - level S_0: contain each ground fluent that holds in the initial state (including negative literals by close-word assumption)
  - level A_0: contains each ground action with preconditions inS_0 (i.e. applicable in S_0)
  - ...
  - level A_i: contains all ground actions with preconditions in S_i
  - level S_{i+1}: all the effects of all the actions in A_i
    - each S_i may contain both P_j and $\neg P_j$

  until $S_N = S_{N-1}$ ("leveled off").
- Contains persistence actions (aka maintenance actions, no-ops)
  - say that a literal l persists if no action negates it
- Mutual exclusion links (mutex) connect
  - incompatible pairs of actions
  - incompatible pairs of literals

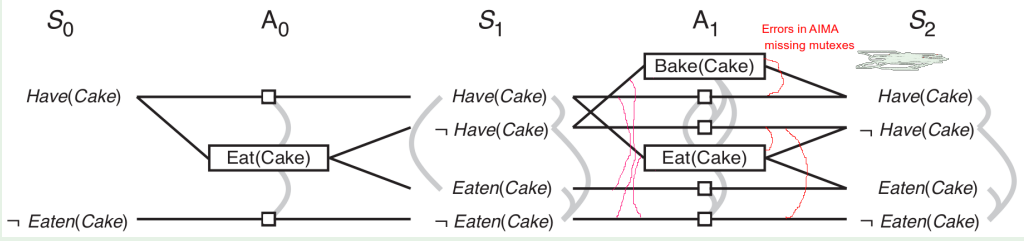
- Literals and actions increase monotonically with index i
- Mutexes decrease monotonically with index i

Planning Graph: Example

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake))$

You would like to eat your cake and still have a cake. Fortunately, you can bake a new one.

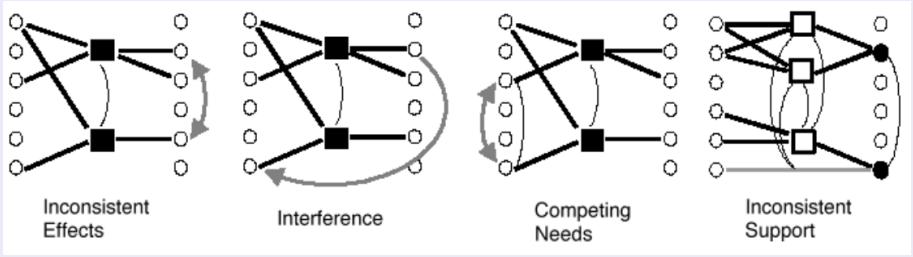
Rectangles indicate actions
Small squares persistence actions (**no-ops**)
Straight lines indicate preconditions and effects
Mutex links are shown as curved gray lines



Mutex Computation

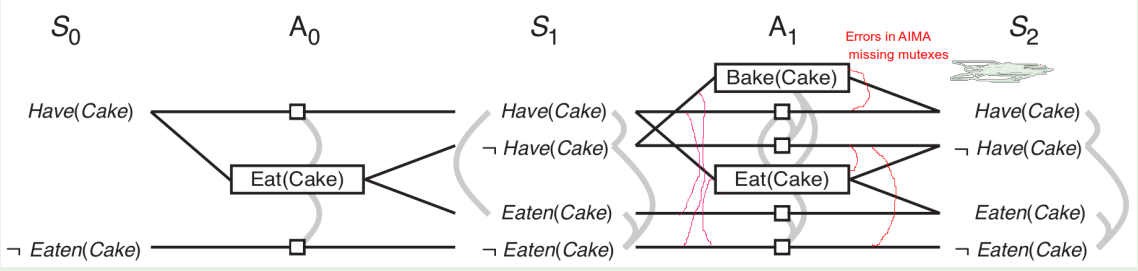
- Two actions at the same action-level have a mutex relation if
  - Inconsistent effects: an effect of one negates an effect of the other
  - Interference: one deletes a precondition of the other
  - Inconsistent preconditions (aka competing needs): they have mutually exclusive preconditions
- Otherwise they don't interfere with each other

  $\Longrightarrow$ both may appear in a solution plan
- Two literals at the same state-level have a mutex relation if
  - inconsistent support: one is the negation of the other
  - all ways of achieving them (including no-ops) are pairwise mutex



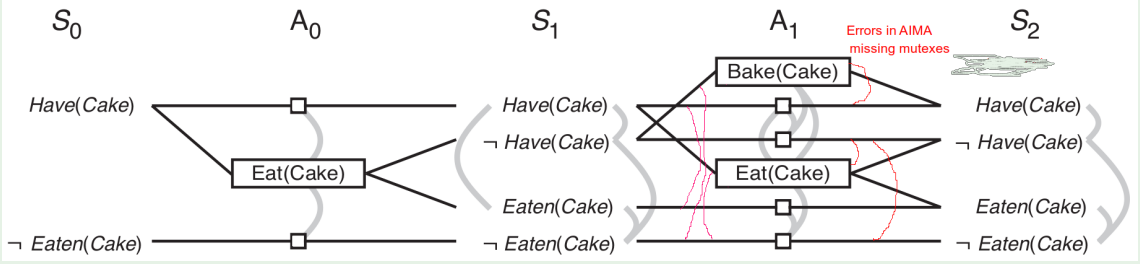
Mutex Computation: Example

- Two actions at the same action-level have a mutex relation if
  - Inconsistent effects: an effect of one negates an effect of the other
    ex: persistence of *Have*(*Cake*), *Eat*(*Cake*) have competing effects
    ex: *Bake*(*Cake*), *Eat*(*Cake*) have competing effects
  - Interference: one deletes a precondition of the other
    ex: *Eat*(*Cake*) interferes with the persistence of *Have*(*Cake*)
  - Inconsistent preconditions (aka competing needs): they have mutually exclusive preconditions
    ex: *Bake*(*Cake*) and *Eat*(*Cake*)



Mutex Computation: Example [cont.]

- Two literals at the same state-level have a mutex relation if
  - inconsistent support: one is the negation of the other
    ex.: *Have*(*Cake*), ¬*Have*(*Cake*)
  - all ways of achieving them are pairwise mutex
    ex.: (S_1): *Have*(*Cake*) in mutex with *Eaten*(*Cake*)
    because persistence of *Have*(*Cake*), *Eat*(*Cake*) are mutex



(© S. Russell & P. Norwig, AIMA)

Building of the Planning Graph

Create initial layer S_0:

1. insert into S_0 all literals in the initial state

Repeat for increasing values of $i = 0, 1, 2, ...$:

Create action layer A_i:

1. for each action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i
2. for every literal in S_i, enter a no-op action node into A_i
3. add mutexes between the newly-constructed action nodes

Create state layer S_{i+1}:

1. for each action node a in A_i,
   - add to S_{i+1} the fluents in his Add list, linking them to a
   - add to S_{i+1} the negated fluents in his Del list, linking them to a
2. for every "no-op" action node a in A_i,
   - add the corresponding literal to S_{i+1}
   - link it to a
3. add mutexes between literal nodes in S_{i+1}

... until $S_{i+1} = S_i$ (aka “graph leveled off”) or bound reached (if any)

Planning Graphs: Properties

- Literals and actions increase monotonically and are finite
  $\Longrightarrow$ we eventually reach a level where they stabilize
- Mutexes decrease monotonically (and cannot become less than zero)
  $\Longrightarrow$ they too eventually must level off

$\Longrightarrow$ When we reach this stable state, if one of the goal literals is missing or is mutex with another goal literal, then it will remain so
$\Longrightarrow$ we can stop

Planning Graphs: Complexity

- A planning graph is polynomial in the size of the problem:
  - a graph with n levels, a actions, l literals, has size $O(n(a+l)^2)$
  - time complexity is also $O(n(a+l)^2)$

$\Longrightarrow$ The process of constructing the planning graph is very fast
  - does not require choosing among actions

Outline

Planning Graphs for Heuristic Estimation

Information provided by Planning Graphs

- Each level S_i represents a set of possible belief states
  - two literals connected by a mutex belong to different belief state
- A literal not appearing in the final level of the graph cannot be achieved by any plan
  $\Longrightarrow$ if a goal literal is not in the final level, the problem is unsolvable
- The level S_j a literal l appears first is never greater than the level it can be achieved in a plan
  - j is called the level cost of literal l
- the level cost of a literal g_i in the graph constructed starting from state s, is an estimate of the cost to achieve it from s (i.e. h(g))
  - this estimate is admissible
  - ex: from s_0 Have(cake) has cost 0 and Eaten(cake) has cost 1
- Planning graph admits several actions per level
  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)
  $\Longrightarrow$ better estimate

Planning Graphs for Heuristic Estimation [cont.]

Estimating the heuristic cost of a conjunction of goal literals

- Max-level heuristic: the maximum level cost of the sub-goals
  - admissible
- Level-sum heuristic: the sum of the level costs of the goals
  - inadmissible only if goals are independent,
  - it may work well in practice
- Set-level heuristic: the level at which all goal literals appear together, without pairwise mutexes
  - admissible, more accurate

Outline

The Graphplan Algorithm

- A strategy for extracting a plan from the planning graph
- Repeatedly adds a level to a planning graph (EXPAND-GRAPH)
- If all the goal literals occur in last level and are non-mutex
  - search for a plan that solves the problem (EXTRACT-SOLUTION)
  - if that fails, add all $\langle$*failed goal*, *level*$\rangle$ pairs as nogoods, expand another level and try again
- If both graph and nogoods have both leveled off then return failure
- Depends on EXPAND-GRAPH & EXTRACT-SOLUTION

function GRAPHPLAN(*problem*) **returns** solution or failure

graph $\leftarrow$ INITIAL-PLANNING-GRAPH(*problem*)
goals $\leftarrow$ CONJUNCTS(*problem*.GOAL)
nogoods $\leftarrow$ an empty hash table
for $t = 0$ **to** ∞ **do**
  if *goals* all non-mutex in S_t of *graph* **then**
    solution $\leftarrow$ EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)
    if *solution* $\neq$ *failure* **then return** *solution*
  if *graph* and *nogoods* have both leveled off **then return** *failure*
  graph $\leftarrow$ EXPAND-GRAPH(*graph*, *problem*)

typo in AIMA book

[Recall] Example: Spare Tire Problem

$$
\begin{aligned}
&Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))\\
&Goal(At(Spare, Axle))\\
&Action(Remove(obj, loc),\\
&\quad \text{PRECOND: } At(obj, loc)\\
&\quad \text{EFFECT: } \neg At(obj, loc) \land At(obj, Ground))\\
&Action(PutOn(t, Axle),\\
&\quad \text{PRECOND: } Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle)\\
&\quad \text{EFFECT: } \neg At(t, Ground) \land At(t, Axle))\\
&Action(LeaveOvernight,\\
&\quad \text{PRECOND:}\\
&\quad \text{EFFECT: } \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)\\
&\qquad\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))
\end{aligned}
$$

(We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.)

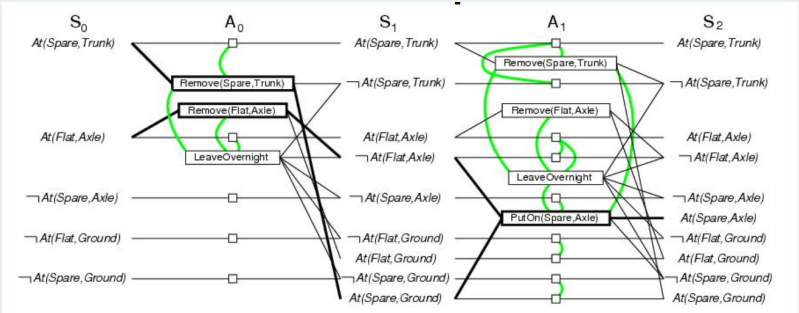
One solution: [*Remove*(*Flat*, *Axle*), *Remove*(*Spare*, *Trunk*), *PutOn*(*Spare*, *Axle*)]

Graphplan: Example

Spare Tire problem

- Initial plan 5 literals from initial state and the Closed-World-Assumption literals (S_0).
  - fixed literals (e.g. $Tire(Flat)$) ignored here
  - irrelevant literals ignored here
- Goal $At(Spare, Axle)$ not present in S_0

  $\Longrightarrow$ no need to call EXTRACT-SOLUTION
- Graph and nogoods not leveled off $\Longrightarrow$ invoke EXPAND-GRAPH



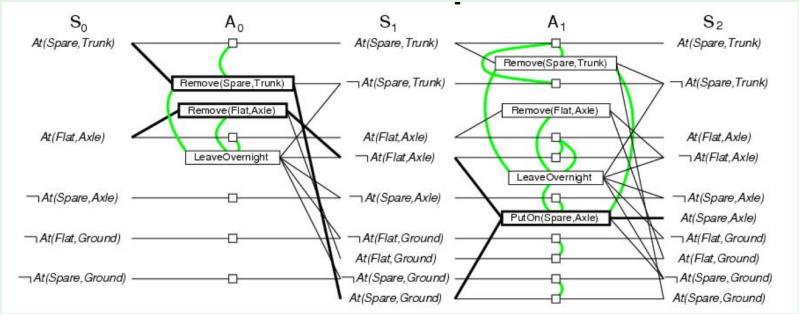
(© S. Russell & P. Norwig, AIMA) (inter-fluent mutexes omitted for readability)

Graphplan: Example [cont.]

Spare Tire problem

- Invoke EXPAND-GRAPH
  - add actions A_0, persistence actions and mutexes
  - add fluents S_1 and mutexes
- Goal $At(Spare, Axle)$ not present in S_1

  $\Longrightarrow$ no need to call EXTRACT-SOLUTION
- Graph and nogoods not leveled off $\Longrightarrow$ invoke EXPAND-GRAPH

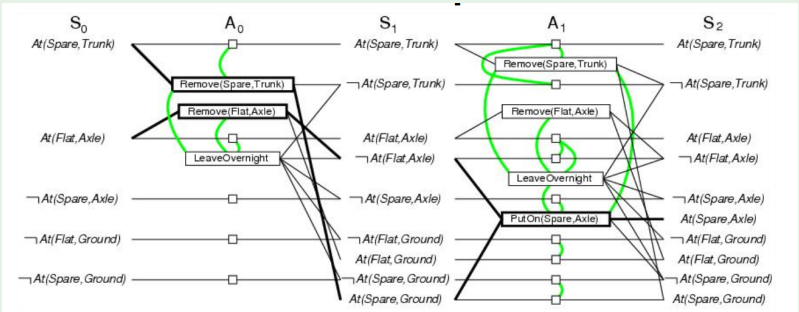


(© S. Russell & P. Norwig, AIMA) (inter-fluent mutexes omitted for readability)

Graphplan: Example [cont.]

Spare Tire problem

- Invoke EXPAND-GRAPH
  - add actions A_1, persistence actions and mutexes
  - add fluents S_2 and mutexes
- Goal $At(Spare, Axle)$ present in S_2
  - call EXTRACT-SOLUTION
- Solution found!



(© S. Russell & P. Norwig, AIMA) (inter-fluent mutexes omitted for readability)

Termination of Graphplan

- Theorem: If the graph and the no-goods have both leveled off, and no solution is found we can safely terminate with failure
- Intuition (proof sketch):
  - Literals and actions increase monotonically and are finite
    $\Longrightarrow$ we eventually reach a level where they stabilize
  - Mutexes decrease monotonically (and cannot become less than zero)
    $\Longrightarrow$ they eventually must level off
  - No-goods decrease monotonically (and cannot become less than zero)
    $\Longrightarrow$ they too eventually must level off

  $\Longrightarrow$ When we reach this stable state, if one of the goal literals is missing or is mutex with another goal literal, then it will remain so
  $\Longrightarrow$ we can stop

Remark

Nogoods may decrease for many levels after literals, actions and mutexes have leveled off

Exercise

- Consider the following variant of the Spare Tire problem:
  add $At(Flat, Trunk)$ to the goal
- Write the (non-serialized) planning graph
- Extract a plan from the graph
- Do the same with the serialized planning graph

The Graphplan Algorithm [cont.]

Graphplan "family" of algorithms, depending on approach used in EXTRACT-SOLUTION(...)

About EXTRACT-SOLUTION(...)

- Can be formulated as an (incremental) SAT problem or a CSP problem
  - one proposition for each ground action and fluent
  - clauses represent preconditions, effects, no-ops and mutexes
- Can be formulated as a backward search problem
- Planning problem restricted to planning graph
  - mutexes found by EXPAND-GRAPH prune paths in the search tree

  $\Longrightarrow$ much faster than unrestricted planning
- (if planning graph not serialized) may produce partial order plans

  $\Longrightarrow$ may be later serialized into a total-order plan

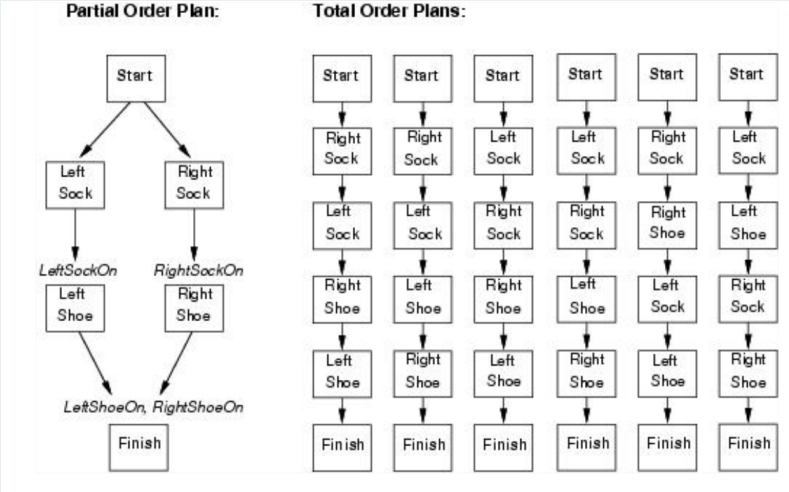
Partial-Order Plans

Partial-Order vs. Total-Order Plans

- Total-order plans: strictly linear sequences of actions
  - disregards the fact that some action are mutually independent
- Partial-order plans: set of precedence constraints between action pairs
  - form a directed acyclic graph
  - longest path to goal may be much shorter than total-order plan
  - easily converted into (possibly many) distinct total-order plans
    (any possible interleaving of independent actions)

Partial-Order Plans: Example

Socks & Shoes Examples



(Courtesy of Michela Milano, UniBO)

Exercise

- Socks & Shoes example:
  1. Formalize the Socks & Shoes example in PDDL
  2. Write the non-serialized planning graph
  3. Compute the level cost for every fluent
  4. Choose some states, compute h(s) using the three heuristics
  5. Extract a plan from the graph in (2)
  6. Compare h(s) with the level they occur in the plan
  7. Write the serialized planning graph
  8. Repeat steps (3)-(6) with the serialized graph
- Do same steps (1)-(8) for the Air Cargo Transport example

Outline

Outline

Planning as SAT Solving

- Encode bounded planning problem into a propositional formula

$\Longrightarrow$ Solve it by (incremental) calls to a SAT solver

- A model for the formula (if any) is a plan of length t
- Many variants in the encoding
- Extremely efficient with many problems of interest

function SATPLAN($init$, $transition$, $goal$, $T_{\max}$) **returns** solution or failure
  inputs: $init$, $transition$, $goal$, constitute a description of the problem
    $T_{\max}$, an upper limit for plan length

  for $t = 0$ **to** $T_{\max}$ **do**
    $cnf \leftarrow$ TRANSLATE-TO-SAT($init$, $transition$, $goal$, t)
    $model \leftarrow$ SAT-SOLVER(cnf)
    if $model$ is not null **then**
      return EXTRACT-SOLUTION($model$)
  return $failure$

Planning as SAT Solving [cont.]

- TRANSLATE-TO-SAT(INIT,TRANSITION, GOAL, T):
  - ground fluents & actions at each step are propositionalized
    - ex: $\langle At(P_1, SFO), 3\rangle \Longrightarrow At\_P_1\_SFO\_3$
    - ex: $\langle Fly(P_1, SFO, JFK), 3\rangle \Longrightarrow Fly\_P_1\_SFO\_JFK\_3$
  - returns propositional formula: $Init^0 \wedge (\bigwedge_{i=1}^{t-1} Transition^{i,i+1}) \wedge Goal^t$
- $Init^0$ and $Goal^t$: conjunctions of literals at step 0 and t resp.
  - ex: $Init^0$: $At\_P_1\_SFO\_0 \wedge At\_P_2\_JFK\_0$ $(\wedge \neg At\_P_1\_JFK \wedge \neg At\_P_2\_SFO \wedge ...)$
  - ex: $Goal^3$: $At\_P_1\_JFK\_3 \wedge At\_P_2\_SFO\_3$
- $Transition^{i,i+1}$: encodes transition from steps i to $i+1$
  - Actions: $Action^i \rightarrow (Precond^i \wedge Effects^{i+1})$
    ex: $Fly\_P_1\_SFO\_JFK\_2 \rightarrow (At\_P_1\_SFO\_2 \wedge At\_P_1\_JFK\_3)$
  - No-Ops: for each fluent F and step i:
    $$F^{i+1} \leftrightarrow \bigvee_k ActionCausingF_k^i \vee (F^i \wedge \bigwedge_j \neg ActionCausingNotF_j^i)$$
  - Mutex constraints: $\neg Action_1^i \vee \neg Action_2^i$
    ex: $\neg Fly\_P_1\_SFO\_JFK\_2 \vee \neg Fly\_P_1\_SFO\_Newark\_2$
  - If serialized: add mutex between each pair of actions at each step

Exercise

Consider the socks & shoes example

- Translate it into SAT for t=0,1,2
  - non serialized
  - no need to propositionalize: treat ground atoms as propositions
  - no need to CNF-ize here (human beings don't like CNFs)
- Find a model for the formula
- Convert it back to a plan

Outline

Planning with Time, Schedules and Resources

- Planning so far: choice of actions
- Real world: Planning with time/schedules
  - actions occur at certain moments in time
  - actions have a beginning and an end
  - actions have a duration

  $\Longrightarrow$ Scheduling
- Real world: Planning with resources
  - actions may require resources
  - ex: limited number of staff, planes, hoists, ...
- Preconditions and effects can include
  - logical inferences
  - numeric computations
  - interactions with other software packages
- Approach “plan first, schedule later”:
  - planning phase: build a (partial) plan, regardless action durations
  - scheduling phase: add other info to the plan, s.t. to meet resource and deadline constrains

Planning with Time & Resources: Example

Planning Phase

$$
\begin{aligned}
&\mathit{Init}(\mathit{Chassis}(C1) \land \mathit{Chassis}(C2) \land \mathit{Engine}(E1, C1, 30) \land \\
&\quad \mathit{Engine}(E2, C2, 60) \land \mathit{Wheels}(W1, C1, 30) \land \mathit{Wheels}(W2, C2, 15)) \\
&\mathit{Goal}(\mathit{Done}(C1) \land \mathit{Done}(C2)) \\
&\mathit{Action}(\mathit{AddEngine}(e, c, d) \\
&\quad \mathit{PRECOND} : \mathit{Engine}(e, c, d) \land \mathit{Chassis}(c) \land \neg \mathit{EngineIn}(c) \\
&\quad \mathit{EFFECT} : \mathit{EngineIn}(c) \land \mathit{Duration}(d) \\
&\quad \mathit{Consume} : \mathit{LugNuts}(20), \mathit{Use} : \mathit{EngineHoists}(1)) \\
&\mathit{Action}(\mathit{AddWheels}(w, c, d) \\
&\quad \mathit{PRECOND} : \mathit{Wheels}(w, c, d) \land \mathit{Chassis}(c) \\
&\quad \mathit{EFFECT} : \mathit{WheelsOn}(c) \land \mathit{Duration}(d) \\
&\quad \mathit{Consume} : \mathit{LugNuts}(20), \mathit{Use} : \mathit{WheelStations}(1)) \\
&\mathit{Action}(\mathit{Inspect}(c, 10) \\
&\quad \mathit{PRECOND} : \mathit{EngineIn}(c) \land \mathit{WheelsOn}(c) \land \mathit{Chassis}(c) \\
&\quad \mathit{EFFECT} : \mathit{Done}(c) \land \mathit{Duration}(10) \\
&\quad \mathit{Use} : \mathit{Inspectors}(1))
\end{aligned}
$$

Solution (partial plan):

$$
\left\{ \begin{array}{l}
\mathit{AddEngine}(E1, C1, 30) \prec \mathit{AddWheels}(W1, C1, 30) \prec \mathit{Inspect}(C1, 10); \\
\mathit{AddEngine}(E2, C2, 60) \prec \mathit{AddWheels}(W2, C2, 15) \prec \mathit{Inspect}(C2, 10)
\end{array} \right\}
$$

Job-Shop Scheduling

- Problem:
  - complete a set of jobs,
  - a job consists of a collection of actions with ordering constraints (a partial plan)
  - an action has a duration and is subject to resource constraints
  - resource constraints specify
    - the type of resource (e.g., bolts, wrenches, or pilots),
    - the number of that resource required
    - if the resource is consumable (e.g., bolts) or reusable (e.g. pilot)
    - resources can be produced by actions with negative consumption
- Solution (aka Schedule):
  - specify the start times for each action
  - must satisfy all the temporal ordering constraints and resource constraints
- Cost function
  - may be very complicate (e.g. non-linear constraints)
  - we assume is the total duration of the plan (makespan)

$\Longrightarrow$ Determine a schedule that minimizes the makespan, respecting all temporal and resource constraints

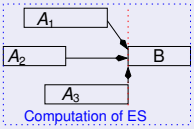
Solving Scheduling Problems

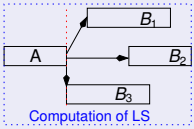
Critical-Path Method

- A path is a ordered sequence of actions from Start to Finish
- The critical path is the path with maximum total duration
  - delaying the start of any action on it slows down the whole plan

  $\Longrightarrow$ determines the duration of the entire plan
  - shortening other paths does not shorten the plan as a whole
- Actions have a window of time in which they can be started: $[ES, LS]$
  - ES: earliest possible start time
  - LS: latest possible start time
  - LS-ES: slack of the action
- LS & ES for all actions can be computed recursively:

$$
\begin{aligned}
ES(Start) &= 0 \\
ES(B) &= max_{\{A_i \mid A_i \prec B\}}(ES(A_i) + Duration(A_i)) \\
LS(Finish) &= ES(Finish) \\
LS(A) &= min_{\{B_j \mid A \prec B_j\}}(LS(B_j) - Duration(A))
\end{aligned}
$$





- Action A_i in the critical path are s.t. $ES(A_i) = LS(A_i)$
- Complexity: $O(Nb)$, N: #actions, b: max branching factor

Planning with Time & Resources: Example [cont.]

Scheduling Phase (resources not considered)

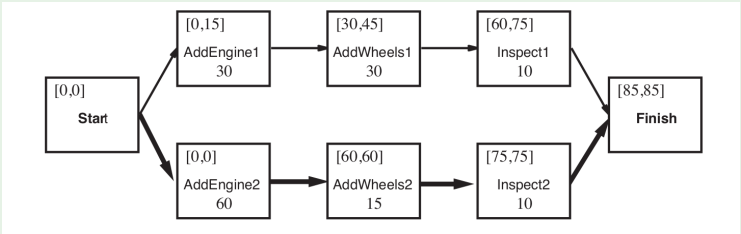
$$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},\\ \{AddEngine2 \prec AddWheels2 \prec Inspect2\})$$

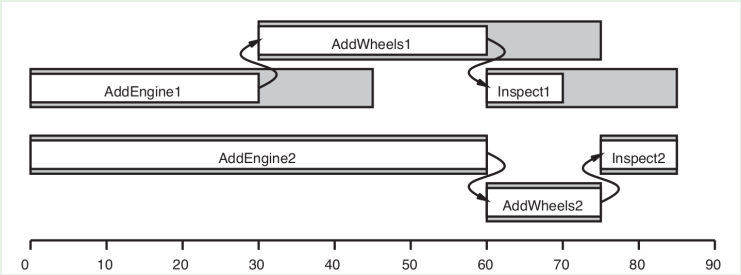
$$Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))$$

$$Action(AddEngine1, \textsc{Duration}{:}30,\\ \textsc{Use}{:}EngineHoists(1))$$
$$Action(AddEngine2, \textsc{Duration}{:}60,\\ \textsc{Use}{:}EngineHoists(1))$$
$$Action(AddWheels1, \textsc{Duration}{:}30,\\ \textsc{Consume}{:}LugNuts(20), \textsc{Use}{:}WheelStations(1))$$
$$Action(AddWheels2, \textsc{Duration}{:}15,\\ \textsc{Consume}{:}LugNuts(20), \textsc{Use}{:}WheelStations(1))$$
$$Action(Inspect_i, \textsc{Duration}{:}10,\\ \textsc{Use}{:}Inspectors(1))$$

Planning with Time & Resources: Example [cont.]

Scheduling Phase (resources not considered)





Adding Resources

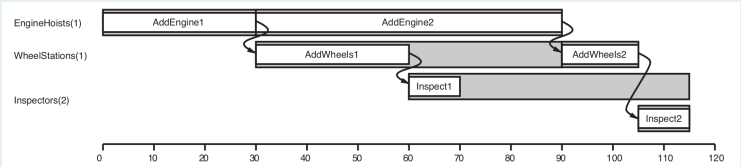
- Critical-path problems (without resources) computationally easy:
  - conjunction of linear inequalities on the start and end times:
    ex: $(ES_2 \geq ES_1 + duration_1) \wedge (ES_3 \geq ES_2 + duration_2) \wedge ...$

  $\Longrightarrow$ Polynomial: $O(Nb)$, N: number of actions; b: maximum branching factor in/out of an action
- Reusable resources: R(k) (ex: Use: EngineHoists(1))
  - k units of resource are required by the action.
  - availability is a pre-requisite before the action can be performed.
- Adding resources makes problems much harder
  - "cannot overlap" constraint is disjunction of linear inequalities
    ex: $((ES_2 \geq ES_1 + duration_1) \vee (ES_1 \geq ES_2 + duration_2)) \wedge ...$

  $\Longrightarrow$ NP-hard
- Various techniques:
  - branch-and-bound, simulated annealing, tabu search, ...
  - reduction to constraint optimization problems
  - reduction to optimization modulo theories (combined SAT+LP)
- Integrate planning and scheduling

Planning with Time & Resources: Example [cont.]

Scheduling Phase (considering resource contrains)



(© S. Russell & P. Norwig, AIMA)

- left-hand margin lists the three reusable resources
- two possible schedules: which assembly uses the hoist first
- shortest-duration solution, which takes 115 minutes

Exercise

- Consider the previous example
  - find another solution
  - draw the diagram
  - check its length and compare it with that in the previous slide

Outline

Outline

Generalities [also recall Ch.04]

- Assumptions so far:
  - the environment is deterministic
  - the environment is fully observable
  - the environment is static
  - the agent knows the effects of each action

$\Longrightarrow$ The agent does not need perception:

- can calculate which state results from any sequence of actions
- always knows which state it is in

- In the real world, the environment may be uncertain
  - partially observable and/or nondeterministic environment
  - incorrect information (differences between world and model)

$\Longrightarrow$ If one of the above assumptions does not hold, use percepts

- the agent's future actions will depend on future percepts
- the future percepts cannot be determined in advance

- Use percepts:
  - perceive the changes in the world
  - act accordingly
  - adapt plan when necessary

Handling Indeterminacy

- Sensorless planning (aka conformant planning):
  find plan that achieves goal in all possible circumstances (if any)
  - regardless of initial state and action effects
  - for environments with no observations
- Conditional planning (aka contingency planning):
  construct conditional plan with different branches for possible contingencies
  - for partially-observable and nondeterministic environments
- Execution monitoring and replanning:
  while constructing plan, judge whether plan requires revision
  - for partially-known or evolving environments

Differences wrt. general Search (Ch.04)

- planners deal with factored representations rather than atomic
- different representation of actions and observation
- different representation of belief states

Handling Indeterminacy [cont.]

Open-World vs. Closed-World Assumption

- Classical Planning based on Closed-World Assumption (CWA)
  - states contain only positive fluents
  - we assume that every fluent not mentioned in a state is false
- Sensorless & Partially-observable Planning based on Open-World Assumption (OWA)
  - states contain both positive and negative fluents
  - if a fluent does not appear in the state, its value is unknown

Belief States

- A belief state is represented by a logical formula (not an explicitly-enumerated set of states)

$\Longrightarrow$ The belief state corresponds exactly to the set of possible worlds that satisfy the formula representing it

- The unknown information can be retrieved via sensing actions (aka percept actions) added to the plan

A Case Study

The table & chair painting problem

Given a chair and a table, the goal is to have them of the same color.
Initially we have two cans of paint, but the colors of the paint and of the furniture are unknown.
Only the table is initially in the agent's field of view

Remark:

"... of the same color" does not specify which color
$\Longrightarrow$ "color" implicitly existentially quantified.

A Case Study [cont.]

The table & chair painting problem [cont.]

- Initial state: $Init(Object(Table) \wedge Object(Chair) \wedge Can(C1) \wedge Can(C2) \wedge InView(Table))$
- Goal: $Goal(Color(Chair, c) \wedge Color(Table, c))$
  - recall: in goal, variable c existentially quantified
- Actions:

  $Action(RemoveLid(can),$
  $\quad Precond : Can(can)$
  $\quad Effect : Open(can))$
  $Action(Paint(x, can),$
  $\quad Precond : Object(x) \wedge Can(can) \wedge Color(can, c) \wedge Open(can)$
  $\quad Effect : Color(x, c))$

  c is implicitly universally quantified, and is not part of action's variable list
  (partially observable only: we might not know the color)
- Add an action causing objects to come into view (one at a time):

  $Action(LookAt(x),$
  $\quad Precond : InView(y) \wedge (x \neq y)$
  $\quad Effect : InView(x) \wedge \neg InView(y))$

Personal note: I'd use distinct predicates for the color of an object and of the paint in a can

Observability and Percept Schemata

- Partially-Observable Problems:
  need to reason about percepts obtained during action

$\Longrightarrow$ Augment PDDL with percept schemata $Percept(\langle fluent \rangle, Precond : \langle fluents \rangle))$
for each fluent. Ex:

- $Percept(Color(x, c),$
  $Precond : Object(x) \wedge InView(x))$
  "if an object is in view, then the agent will perceive its color"

$\Longrightarrow$ perception will acquire the truth value of $Color(x, c)$, for every x, c

- $Percept(Color(can, c),$
  $Precond : Can(can) \wedge InView(can) \wedge Open(can))$
  "if an open can is in view, then the agent perceives the color of the paint in the can"

$\Longrightarrow$ perception will acquire the truth value of $Color(can, c)$, for every can, c

- Fully-Observable Problems:

$\Longrightarrow$ Percept schemata with no preconditions for each fluent. Ex:

- $Percept(Color(x, c))$

- Sensorless Agent: no percept schema

Handling Indeterminacy [cont.]

- Sensorless planning (aka conformant planning):
  find plan that achieves goal in all possible circumstances (if any)
  - regardless of initial state and action effects
  - for environments with no observations
  - ex: “Open any can of paint and apply it to both chair and table”
- Conditional planning (aka contingency planning):
  construct conditional plan with different branches for possible contingencies
  - for partially-observable and nondeterministic environments
  - ex: “Sense color of table and chair;
    if they are the same, then finish, else sense can paint;
    if color(can) =color(furniture) then apply color to other piece;
    else apply color to both”
- Execution monitoring and replanning:
  while constructing plan, judge whether plan requires revision
  - for partially-known or evolving environments
  - ex: Same as conditional, and can fix errors

Outline

[Recall from Ch.04]: Search with No Observation

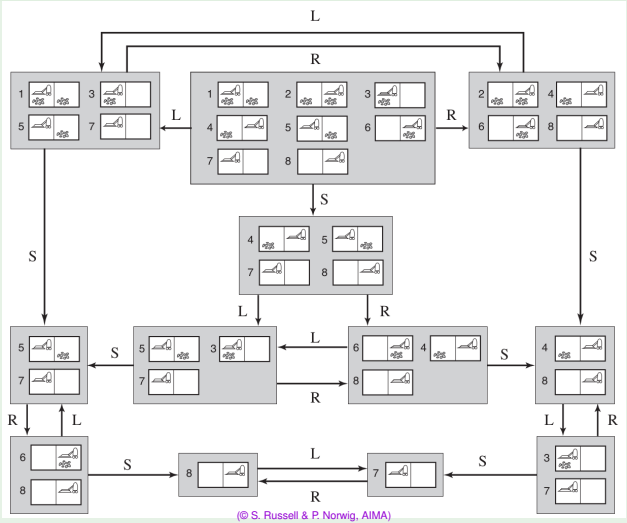
Search with No Observation

- aka Sensorless Search or Conformant Search
- Idea: To solve sensorless problems, the agent searches in the space of belief states rather than in that of physical states
  - fully observable, because the agent knows its own belief space
  - solutions are always sequences of actions (no contingency plan), because percepts are always empty and thus predictable
- Main drawback: 2^N candidate states rather than N

[Recall from Ch.04]: Belief-State Problem Formulation

Example: Sensorless Vacuum Cleaner: Belief State Space

(note: self-loops are omitted)



(© S. Russell & P. Norwig, AIMA)

$\Longrightarrow$ [Left,Suck,Right,Suck] contingent plan

Sensorless Planning

- Main idea [see ch.04]: see a sensorless planning problem as a belief-state planning problem
- Main differences:
  - planners deal with factored representations rather than atomic
  - physical transition model is a collection of action schemata
  - the belief state represented by a logical formula instead of an explicitly-enumerated set of states
- Open-World Assumption $\Longrightarrow$ a belief state corresponds to the set of possible worlds that satisfy the formula representing it
- All belief states (implicitly) include unchanging facts (invariants)
  ex: $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$
- Initial belief state includes facts that are part of the agent's domain knowledge
  - Ex: "objects and cans have colors"
    $\forall x.\exists c.\ Color(x, c) \Longrightarrow$ (Skolemization) $\Longrightarrow b_0 : Color(x, C(x))$ ($C(x)$: the color of x)

Sensorless Planning [cont.]

- Actions(b): In belief state b, it is possible to apply every action a s.t. $b \models Precond(a)$
  - e.g., $RemoveLid(Can_1)$ applicable in b_0 since $Can(C_1)$ true in b_0
- $Result(b, a)$ is computed:
  - start from b
  - set to false any atom that appears in $Del(a)$ (after unification), drop its positive occurrence if present
  - set to true any atom that appears in $Add(a)$ (after unification), drop its negative occurrence if present

  i.e., conjoin Effects(a) to b (dropping their negation from the conjunction)

Property

If the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals

- with n fluents, any belief state can be compactly represented by a conjunction of size O(n)

$\Longrightarrow$ much simplifies complexity of belief-state reasoning

Sensorless Planning: Example

$Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$ are left implicit
$InView(...)$ are ignored because the agent is sensorless.

- Start from $b_0 : Color(x, C(x))$
- Apply $RemoveLid(Can_1)$ in b_0 and obtain:
  $b_1 : Color(x, C(x)) \wedge Open(Can_1)$
- Apply $Paint(Chair, Can_1)$ in b_1 using $\{x/Chair, can/Can_1, c/C(Can_1)\}$:
  $b_2 : Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1))$
- Apply $Paint(Table, Can_1)$ in b_2 using $\{x/Table, can/Can_1, c/C(Can_1)\}$:
  $b_3 : Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)) \wedge Color(Table, C(Can_1))$
- b_3 Satisfies the goal: $b_3 \models Color(Table, c) \wedge Color(Chair, c)$

$\Longrightarrow$ $[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)]$ valid conformant plan

Exercise

- Provide a novel formalization of the above problem with distinct predicates for the color of an object and for the color the paint in a can
  - find step-by-step a plan with the new formalization

Outline

[Recall from Ch.4]: Searching with Nondeterministic Actions

Generalized notion of transition model

- RESULTS(S,A) returns a set of possible outcomes states
  - Ex: RESULTS(1,SUCK)={5,7}, RESULTS(5,SUCK)={1,5}, ...
- A solution is a contingency plan (aka conditional plan, strategy)
  - contains nested conditions on future percepts (if-then-else, case-switch, ...)
  - Ex: from state 1 we can act the following contingency plan:
    [SUCK, IF STATE = 5 THEN [RIGHT, SUCK] ELSE [ ]]
- Can cause loops (see later)

[Recall from Ch.4]: Searching with Nondeterministic Actions [cont.]

And-Or Search Trees

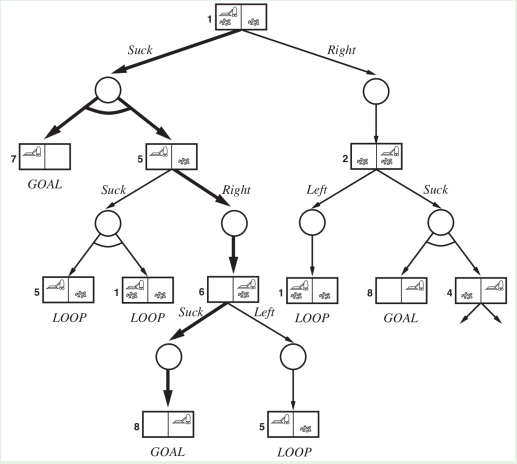
- In a deterministic environment, branching on agent's choices
  - $\Longrightarrow$ OR nodes, hence OR search trees
    - OR nodes correspond to states
- In a nondeterministic environment, branching also on environment's choice of outcome for each action
  - the agent has to handle all such outcomes
  - $\Longrightarrow$ AND nodes, hence AND-OR search trees
    - AND nodes correspond to actions
    - leaf nodes are goal, dead-end or loop OR nodes
- A solution for an AND-OR search problem is a subtree s.t.:
  - has a goal node at every leaf
  - specifies one action at each of its OR nodes
  - includes all outcome branches at each of its AND nodes

OR tree: AND-OR tree with 1 outcome each AND node (determinism)

[Recall from Ch.4]: And-Or Search Trees: Example

(Part of) And-Or Search Tree for Erratic Vacuum Cleaner Example.

Solution for [SUCK, IF STATE = 5 THEN [RIGHT, SUCK] ELSE [ ]]



[Recall from Ch.4]: AND-OR Search

Recursive Depth-First (Tree-based) AND-OR Search

function AND-OR-GRAPH-SEARCH(*problem*) **returns** *a conditional plan, or failure*
  OR-SEARCH(*problem*.INITIAL-STATE, *problem*, [ ])

---

function OR-SEARCH(*state*, *problem*, *path*) **returns** *a conditional plan, or failure*
  if *problem*.GOAL-TEST(*state*) **then return** the empty plan
  if *state* is on *path* **then return** *failure*
  for each *action* **in** *problem*.ACTIONS(*state*) **do**
    plan ← AND-SEARCH(RESULTS(*state*, *action*), *problem*, [*state* | *path*])
    if *plan* ≠ *failure* **then return** [*action* | *plan*]
  return *failure*

---

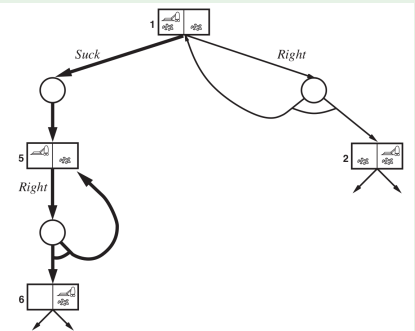
function AND-SEARCH(*states*, *problem*, *path*) **returns** *a conditional plan, or failure*
  for each s_i **in** *states* **do**
    $plan_i$ ← OR-SEARCH(s_i, *problem*, *path*)
    if $plan_i$ = *failure* **then return** *failure*
  return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** . . . **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

Note: nested if-then-else can be rewritten as case-switch

[Recall from Ch.4]: Cyclic Solution: Example

Example: Slippery Vacuum Cleaner

- Movement actions may fail: e.g., $Results(1, Right) = \{1, 2\}$
- A cyclic solution
- Use labels: [Suck, L1 : Right, if State = 5 then L1 else Suck]
- Use cycles: [Suck, While State = 5 do Right, Suck]



Contingent Planning

- Contingent Planning: generation of plans with conditional branching based on percepts
  - appropriate for partial observability, non-determinism, or both
- Main differences:
  - planners deal with factored representations rather than atomic
  - physical transition model is a collection of action schemata
  - the belief state represented by a logical formula instead of an explicitly-enumerated set of states
  - sets of belief states represented as disjunctions of logical formulas representing belief states
- When executing a contingent plan, the agent:
  - maintains its belief state as a logical formula
  - evaluate each branch condition:
    - if the belief state entails the condition formula, then proceed with the “then” branch
    - if the belief state entails the negation of the condition formula, then proceed with the “else” branch
- Note: The planning algorithm must guarantee that the agent never ends in a belief state where the condition’s truth value is unknown

Computing $Result(a, b)$ with Conditional Steps

Three steps (aka prediction-observation-update)

1. Prediction: (same as for sensorless): $\hat{b} = b \setminus Del(a) \cup Add(a)$ // $\hat{b} = b \wedge Effects(a)$
2. Observation prediction: determines the set of percepts that could be observed in the predicted belief state $P \stackrel{\text{def}}{=} PossiblePercepts(\hat{b}) \stackrel{\text{def}}{=} \{p \mid \hat{b} \models Precond(p)\}$
3. Update: $Result(b, a) = \hat{b} \wedge \bigwedge_{p \in P} b_p$, s.t.:
   - if p has one percept schema, $Percept(p, Precond : c)$, s.t. $\hat{b} \models c$, then $b_p \stackrel{\text{def}}{=} p \wedge c$
   - if p has k percept schemata, $Percept(p, Precond : c_i)$, s.t. $\hat{b} \models c_i$, $i = 1..k$, then $b_p \stackrel{\text{def}}{=} \bigvee_{i=1}^{k} (p \wedge c_i)$

$\Longrightarrow$ $Result(b, a)$ CNF formula, not simply conjunction of literals (cubes)
$\Longrightarrow$ much harder to deal with
$\Longrightarrow$ often (over)approximations used to guantantee b_i cube

Contingent Planning: Example

- Possible contingent plan for previous problem described below
  - variables in the plan to be considered existentially quantified
  - ex (2^{nd} row): "if there exists some color c that is the color of the table and the chair, then do nothing" (goal reached)
- "Color(Table,c)", "Color(Chair,c)' and "Color(Can,c)" percepts
  $\Longrightarrow$ must be matched against percept schemata

$$
\begin{array}{l}
[LookAt(Table), LookAt(Chair),\\
\quad \textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp\\
\quad\quad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1), RemoveLid(Can_2), LookAt(Can_2),\\
\quad\quad\quad \textbf{if } Color(Table, c) \wedge Color(can, c) \textbf{ then } Paint(Chair, can)\\
\quad\quad\quad \textbf{else if } Color(Chair, c) \wedge Color(can, c) \textbf{ then } Paint(Table, can)\\
\quad\quad\quad \textbf{else } [Paint(Chair, Can_1), Paint(Table, Can_1)]]]
\end{array}
$$

Is the above plan (from AIMA book) correct? First "LookAt(Table)" wrong

Contingent Planning: Example [cont.]

Invariants $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2) \wedge Color(x, C(x))$ omitted: "..."
Initial: $... \wedge InView(Table)$
Goal: $Goal(Color(Chair, c) \wedge Color(Table, c))$
Initial percepts: $p_0 \stackrel{\text{def}}{=} Color(Table, C(Table))$

- Compute plan P_0 from $b_0 \stackrel{\text{def}}{=} ... \wedge InView(Table) \wedge Color(Table, C(Table))$:
  - Or-Search$(b_0, ...)$: Goal test: $b_0 \not\models Goal$
    Actions: $LookAt(Chair), RemoveLid(C_1), RemoveLid(C_2)$
    Assume Or-Search (DFS) picks $a_0 \stackrel{\text{def}}{=} LookAt(Chair)$. Compute Result$(b_0, a_0)$:

$\Longrightarrow$ Prediction: $\hat{b}_1 \stackrel{\text{def}}{=} b_0 \setminus \{InView(Table)\} \cup \{InView(Chair)\}$

$\Longrightarrow$ Observation Prediction: Possible Percepts: $p_1 \stackrel{\text{def}}{=} Color(Chair, C(Chair))$

$\Longrightarrow$ Update: $b_1 = \text{Result}(b_0, a_0) = \hat{b}_1 \wedge Color(Chair, C(Chair))$

- And-Search: b_1 partitioned in two belief sub-states
  $b_{11} \stackrel{\text{def}}{=} b_1 \wedge (Color(Table, c) \wedge Color(Chair, c))$ – i.e., $C(Chair) = C(Table)$
  $b_{12} \stackrel{\text{def}}{=} b_1 \wedge \neg(Color(Table, c) \wedge Color(Chair, c))$ – i.e., $C(Chair) \neq C(Table)$

$\Longrightarrow$ Compute recursively plan P_{11} from b_{11} and plan P_{12} from b_{12}
return plan $[LookAt(Chair) | \textbf{if } (Color(Table, c) \wedge Color(Chair, c)) \textbf{ then } P_{11} \textbf{ else } P_{12}]$

Contingent Planning: Example [cont.]

(Invariants $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2) \wedge Color(x, C(x))$ omitted: "...")
Initial: $... \wedge InView(Table)$
Goal: $Goal(Color(Chair, c) \wedge Color(Table, c))$
Initial percepts: $p_0 \stackrel{\text{def}}{=} Color(Table, c) \Longrightarrow Color(Table, C(Table))$

- Compute plan P_0 from $b_0 \stackrel{\text{def}}{=} ... \wedge InView(Table) \wedge Color(Table, C(Table))$:
  (...) return plan $[LookAt(Chair) | \textbf{if } (Color(Table, c) \wedge Color(Table, c)) \textbf{ then } P_{11} \textbf{ else } P_{12}]$
- Compute plan P_{11} from $b_{11} \stackrel{\text{def}}{=} b_1 \wedge (Color(Table, c) \wedge Color(Table, c))$
  - Goal test: $b_{11} \models Goal \Longrightarrow$ return $P_2 = NoOp$
- Compute plan P_{12} from $b_{12} \stackrel{\text{def}}{=} b_1 \wedge \neg(Color(Table, c) \wedge Color(Table, c))$:
  - Goal test: $b_{12} \not\models Goal$
  - Actions: $RemoveLid(C_1), RemoveLid(C_2)$
  - (...)

Exercises

- Try to draw an execution of the conditional plan in previous slide against an imaginary physical state of the world of your choice
  - track step by step the belief states, the logical inferences, the actions performed