Planning

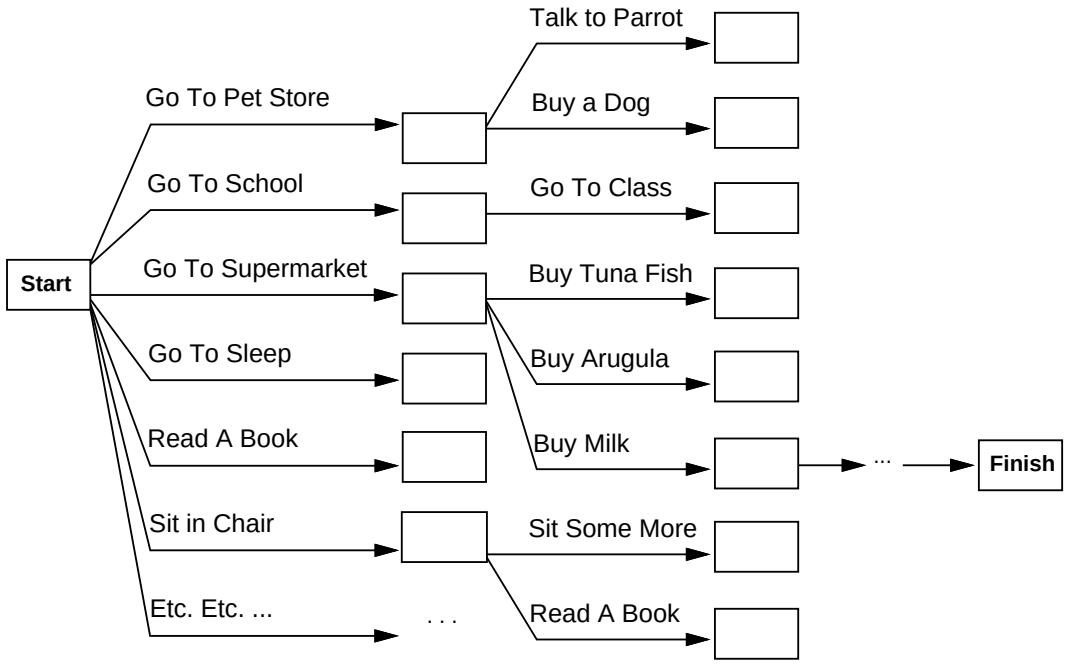
Chapter 11

Outline

- ♢ Search vs. planning
- ♢ STRIPS operators
- ♢ Partial-order planning
- ♢ Planning graphs

Search vs. planning

Consider the task *get milk, bananas, and a cordless drill*
Standard search algorithms seem to fail miserably:



Inefficient Generate and Test

Generate: Apply MANY different operators (often repeatedly)

Test: Use heuristic function (h) to assess states.

Informed Search is Still Pretty Dumb

- **Informed Search** ONLY means that you have a good heuristic (h) = estimate of distance(state, goal)
- It does NOT mean that you have knowledge that allows you to predict the effect of an operator on a state.
- Trial + Error: You have to APPLY the operator to the state to find the successor state.

Needed: **Explicit** knowledge about:

- WHEN operators are applicable: Preconditions.
- WHAT happens when they are applied: Effects.

Then, system can reason with preconditions and effects to:
Determine proper operators and operator sequences
Without necessarily applying the operators to states

Goals Must also be Explicit

In informed search, the goal is often only understood operationally:

- We can use it to compute h(state)
- But that's about it!
- The system has no deep understanding of the goal
- So it cannot do sophisticated reasoning to analyze states w.r.t. goals.

In general, the actions and goals need to be expressed DECLARATIVELY, not just PROCEDURALLY.

1. Procedural - the machine can EXECUTE it.
2. Declarative - the AI system can REASON with it, often in many different ways.

Similar to the difference between a data file and a database.

Search vs. planning contd.

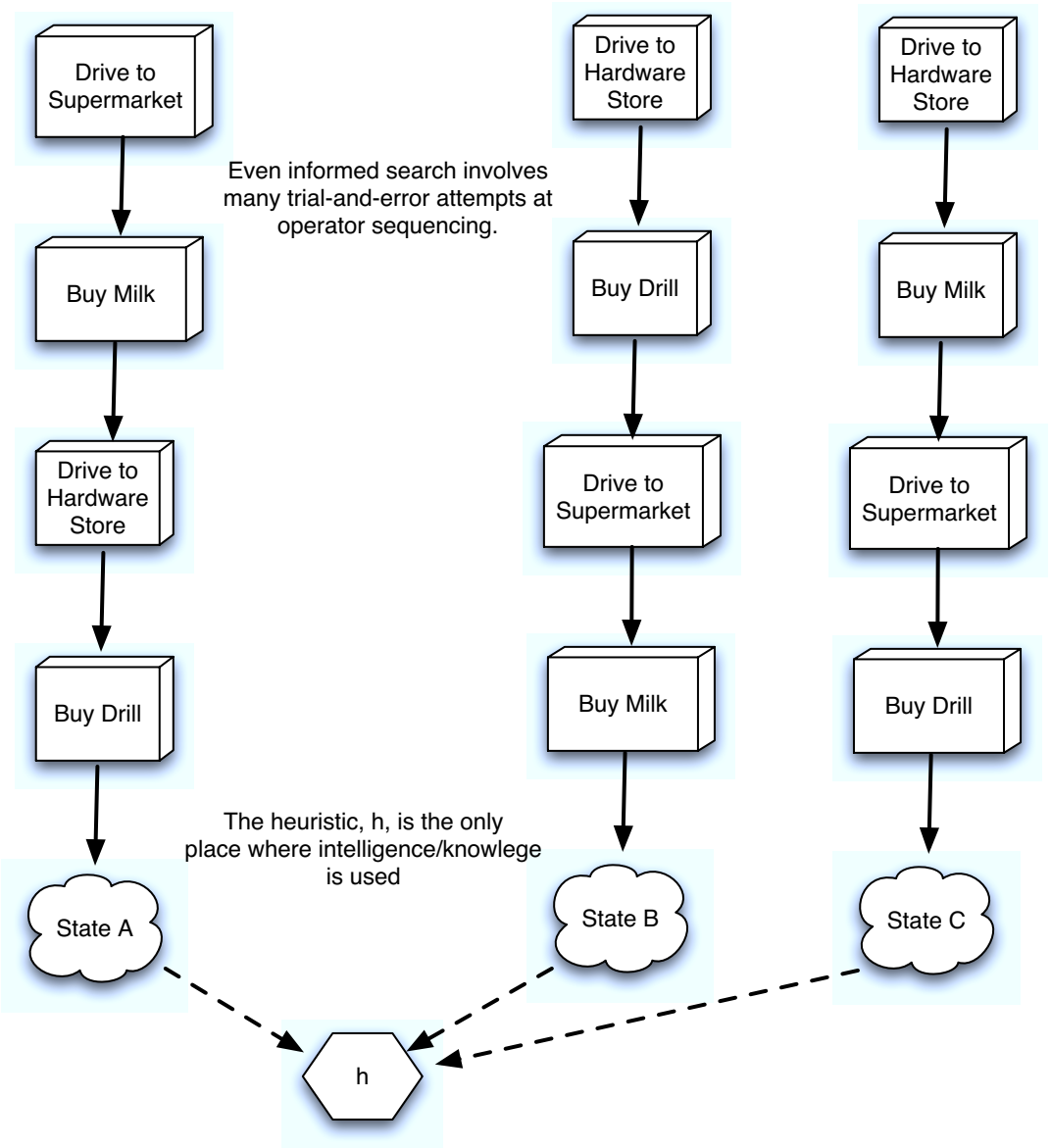
Planning systems do the following:

1) open up action and goal representation to allow selection
2) divide-and-conquer by subgoaling
3) relax requirement for sequential construction of solutions

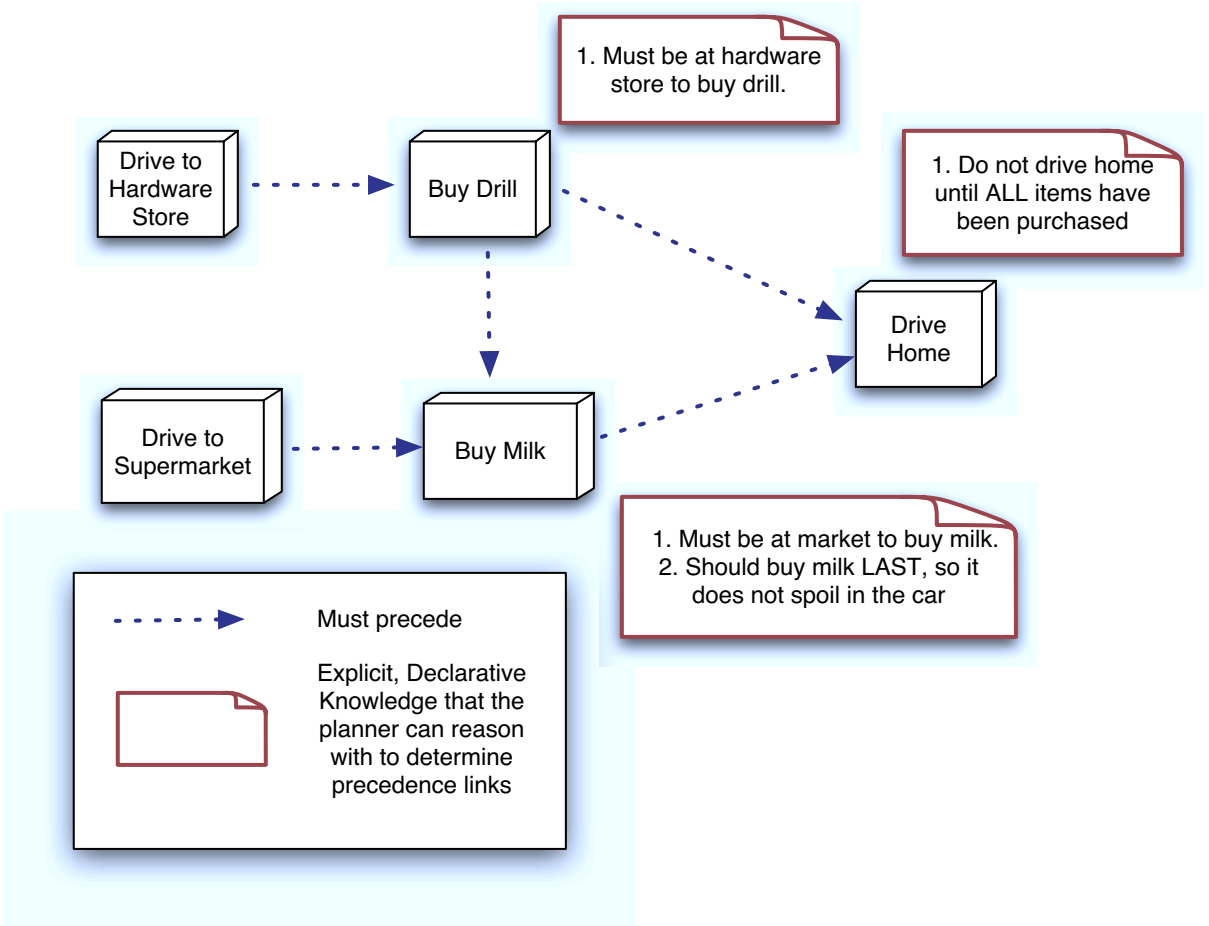
	Search	**Planning**
States	Lisp data structures	Logical sentences
Actions	Lisp code	Preconditions/outcomes
Goal	Lisp code	Logical sentence (conjunction)
Plan	Sequence from S_0	Constraints on actions

States, Actions + Goals are often procedural in search, but declarative in planning.

Search and Linear Operator Sequences



Planning and Intelligent Partial-Ordering



Classic Planning Environments

1. Fully Observable - we see everything that matters
2. Deterministic - the effects of actions are known, exactly.
3. Static - no changes happen to environment other than those caused by agent actions
4. Discrete - Changes in time and space occur in quantum amounts

Ch. 11 methods assume a classic environment
Ch. 12 methods handle real-world situations, where many of the classic assumptions cannot be assumed.

Representations for Planning

Making operators/actions and goals explicit.

- Operators: Breaking into preconditions, action, and effects.
- Goals: Breaking into subgoals

Preconditions: $off(motor) \wedge in(key, ignition) \wedge on(foot, clutch)$
Action: turn(key)
Effects: $on(motor) \wedge \neg off(motor)$

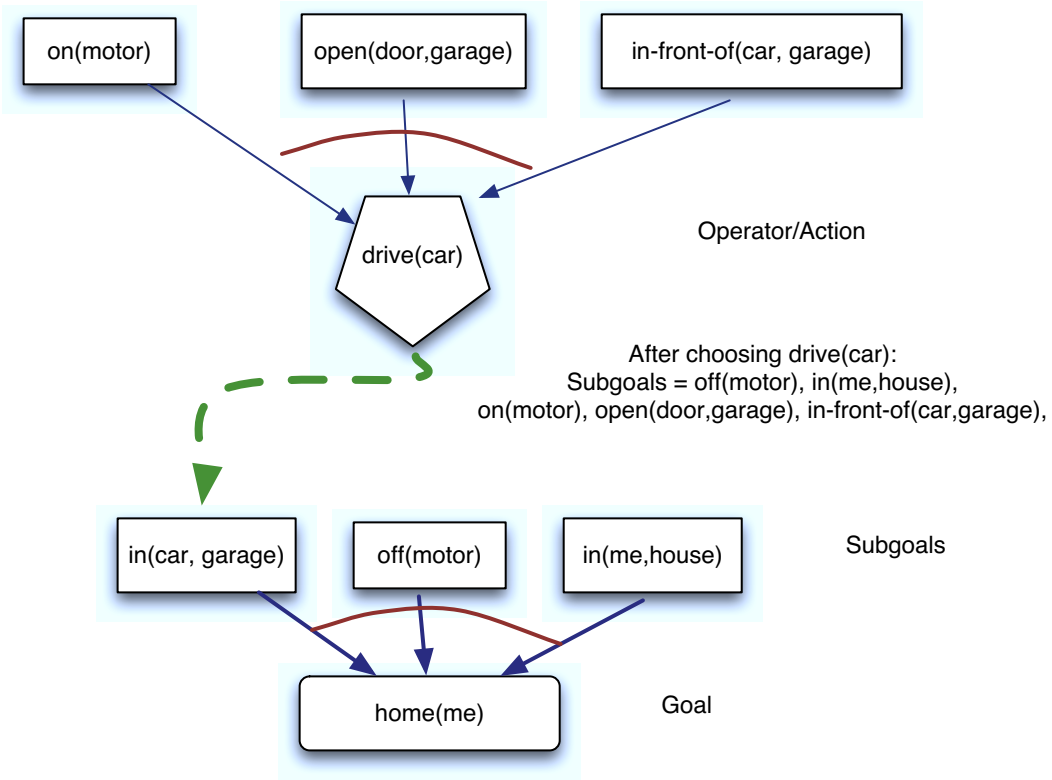
Goal: home(me)
Subgoals: $in(car, garage) \wedge off(motor) \wedge in(me, house)$

Problem Decomposition

To solve a goal, just solve all the subgoals.

A Subgoal (S) is **solved** when:

1) Either true initially, or action chosen that has S as an effect.
2) No other actions make S false after it becomes true



Representational Issues

1. Effects -vs- Add + Delete Lists
2. Can Negative Literals appear in states (or just positive ones)?
3. Unmentioned literals are:
   (a) FALSE (Closed World Assumption) - The only things that are true are those that are explicitly stated as true; all else is assumed false.
   (b) UNKNOWN (Open World Assumption) - some true things may simply not have been mentioned.
4. Frame Problem - What literals remain the same after an action?
   STRIPS assumption: All literals not explicitly mentioned in the Effects (or Add/Delete) list(s) are unchanged.

Representational Issues (2)

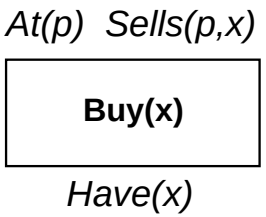
5. Goals may contain:
   (a) ONLY Conjunctions: $On(blockA, blockB) \wedge On(blockB, blockC)$
   (b) BOTH Conjunctions and Disjunctions:
   $On(blockA, blockB) \wedge (On(blockB, blockC) \vee On(blockB, , blockD))$
6. Goals may contain:
   (a) ONLY ground literals:
   $At(plane227, airport99) \vee At(plane376, airport99)$
   (b) Quantified variables:
   $\exists x \; plane(x) \wedge At(x, airport99)$
7. Conditional effects, equality tests, variable typing, etc., etc.

Expressibility - Efficiency Tradeoff:
The more you can represent, the larger the search space!!

STRIPS operators

Tidily arranged actions descriptions, restricted language

PRECONDITION: $At(p), Sells(p, x)$
ACTION: $Buy(x)$
EFFECT: $Have(x)$



[Note: this abstracts away many important details!]

Restricted language $\Rightarrow$ efficient algorithm
Precondition: conjunction of positive literals
Effect: conjunction of literals

A complete set of STRIPS operators can be translated into a set of successor-state axioms

STRIPS Examples

BlocksWorld

PRECONDITION: $Clear(x) \wedge Clear(y) \wedge On(x, z)$
ACTION: Stack(x,y)
EFFECT: $\neg Clear(y) \wedge \neg On(x, z) \wedge On(x, y) \wedge Clear(z)$
By the STRIPS Assump, Clear(x) remains true since it's not mentioned in effects.

..Alternatively...

PRECONDITION: $Clear(x) \wedge Clear(y) \wedge empty(hand)$
ACTION: Stack(x,y)
DELETE: Clear(y) , On(x,z)
ADD: Clear(z) , On(x,y)

PlaneWorld

PRECONDITION: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
ACTION: Fly(p,from,to)
EFFECT: $\neg At(p, from) \wedge At(p, to)$

Successor-State Axioms for Flying

$$FlyPrecond(p, f, to, s) \Leftrightarrow At(p, f, s) \wedge Plane(p) \wedge Airport(f) \wedge Airport(to)$$

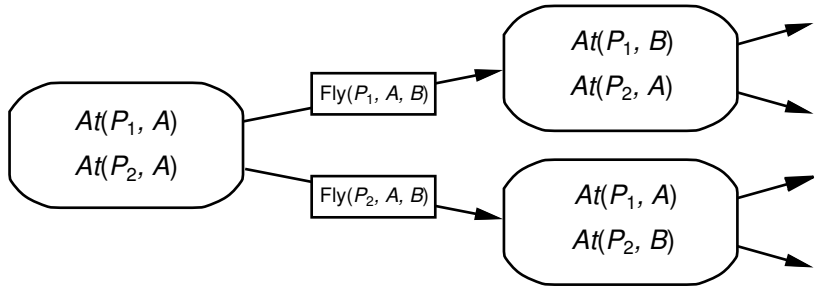
$$At(p, x, Result(a, s)) \Leftrightarrow (At(p, x, s) \wedge (a \neq Fly(p, f, x) \vee \neg FlyPrecond(p, f, x, s))) \vee (At(p, f, s) \wedge a = Fly(p, f, x) \wedge FlyPrecond(p, f, x, s))$$

Planning as State-Space Search

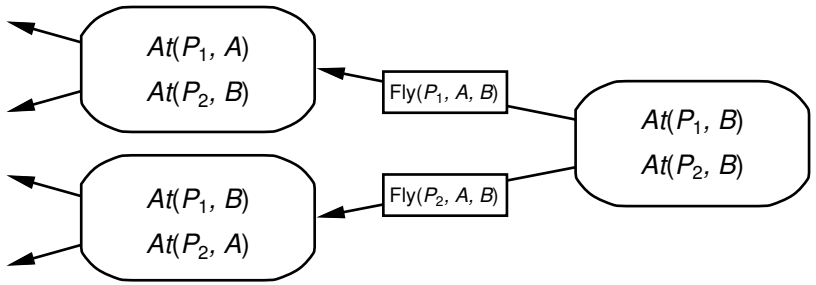
States = conjunctions of literals
Operators = planning actions
Use Forward- or Backward-Chaining Search

PRECONDITION: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
ACTION: Fly(p,from,to)
EFFECT: $\neg At(p, from) \wedge At(p, to)$

(a)



(b)



Forward State-Space Search: Progression

Initial State: Conjunction of Literals in start state
Search: Apply actions and update current state based on action effects.
End Test: Current state is superset of the goal state.

Heuristics - for both Progression and Regression

1. Subgoal Independence Assumption - Each subgoal can be solved independently.
   (a) Optimistic - when actions can clobber (negate) other subgoals
   (b) Pessimistic - when actions can solve more than 1 subgoal at once.
2. Relaxed Problem - abstract the operators
   (a) Remove all preconditions - assume all ops are always applicable.
   (b) Remove negative effects - so no action clobbers a subgoal.

Using these often involves running a planner based on these assumptions (e.g. with abstract operators and all subgoals treated separately), just to calculate h(state)!!

Backward State-Space Search: Regression

Initial State: Conjunction of Literals in goal state
Search: Apply actions **in reverse** and update current state based on pre-conditions.
End Test: Current state is a subset of the initial state.

- **Relevant action:** achieves one or more subgoals
- **Consistent action:** does not undo any subgoals.

With progresssion, it is hard to know what operators are relevant, so many unecessary ones are often applied.
With regression, only operators that achieve a subgoal are worth applying (i.e., relevant)

Regressing a state S through a relevant and consistent action A:
1) Any positive effects of A that are in S are deleted.
2) Any preconditions of A are added, unless already true in S.

Blocksworld Regression Example

PRECONDITION: $Clear(x) \wedge Clear(y) \wedge On(x, z)$
ACTION: Stack(x,y)
EFFECT: $\neg Clear(y) \wedge \neg On(x, z) \wedge On(x, y) \wedge Clear(z)$

S $= On(B, C) \wedge Clear(A)$
Regressing S through Stack(B,C) yields:
S* $= Clear(B) \wedge Clear(C) \wedge On(B, A)$

Note that Clear(B) need not be true in S for this regression to be legal, even though a) it must be true in S* and b) the STRIPS assumption entails that it does not change after Stack(B,C).

Least Commitment Planning

Least Commitment = Delaying choices as long as possible.
Make important, obvious and/or highly-constrained (action) choices early
Make remaining choices later, and only as needed.

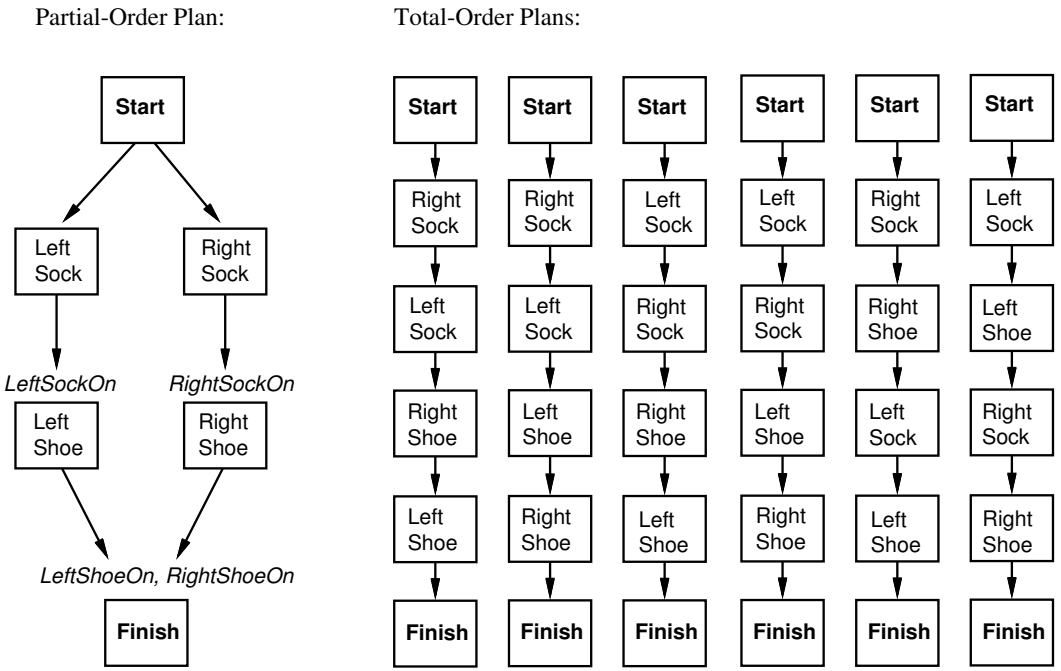
State-Space Progression and Regression are **total-order planners**:
They create linear plan sequences, one **consecutive** step at a time,
from start to goal or goal to start.

Partial-order planners

- Add actions to plans without committing to an absolute time/step.
- Deal mainly with relative constraints: *action A must precede action B*.
- Can be implemented as **search** in a space of planning states, where the planning operators are different from the real-world actions:
  - Real-world: stack(x,y), clear-hand, put-on-table(x)
  - Planner: stack(B,A) **precedes** stack(C,B); **add** stack(D,E) to plan

Partial-Order Planning

1. Generate a list of constraints among actions: partial-order plan.
2. Create one or more linear (total-order) plans that are consistent with the constraints.



Partially ordered plans

Partially ordered collection of steps with

- $Start$ step has the initial state description as its effect
- $Finish$ step has the goal description as its precondition
- causal links from outcome of one step to precondition of another
- temporal ordering between pairs of steps

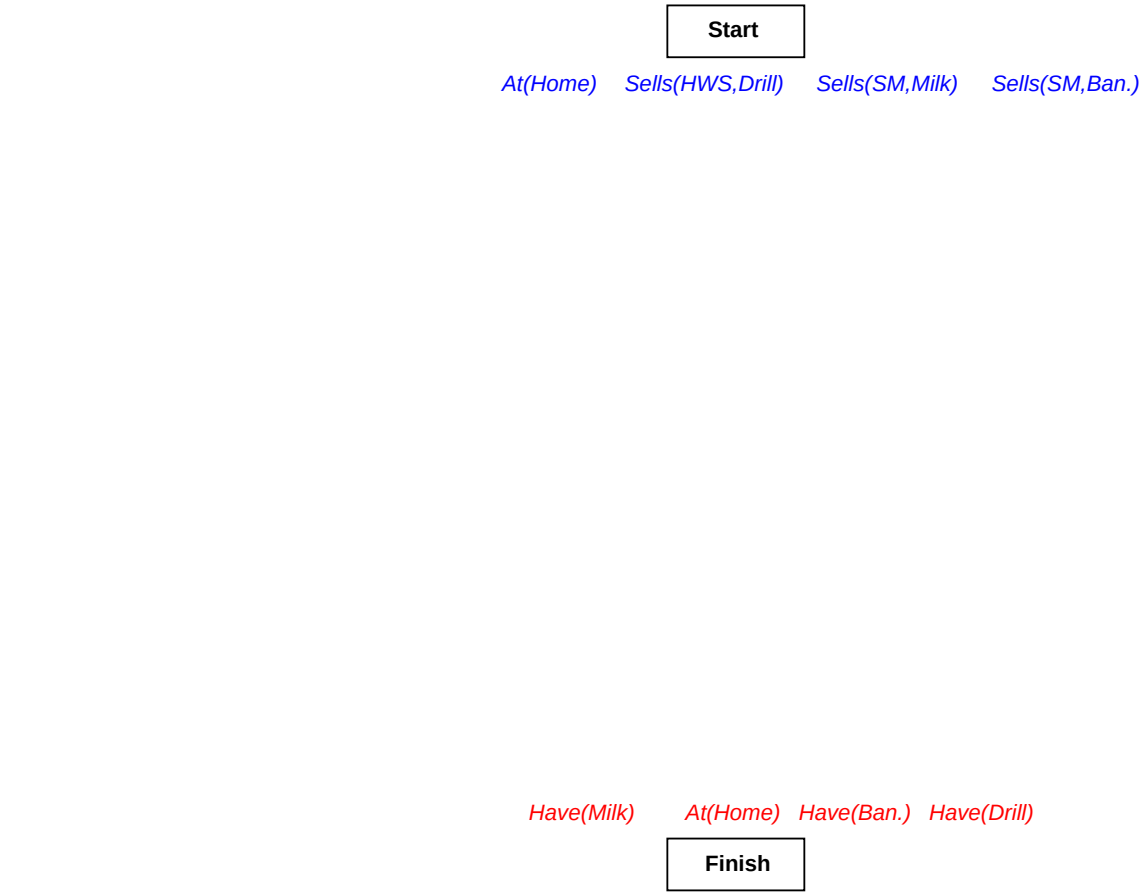
Open condition = precondition of a step not yet causally linked

A plan is complete iff every precondition is achieved

A precondition is achieved iff it is the effect of an earlier step
and no possibly intervening step undoes it

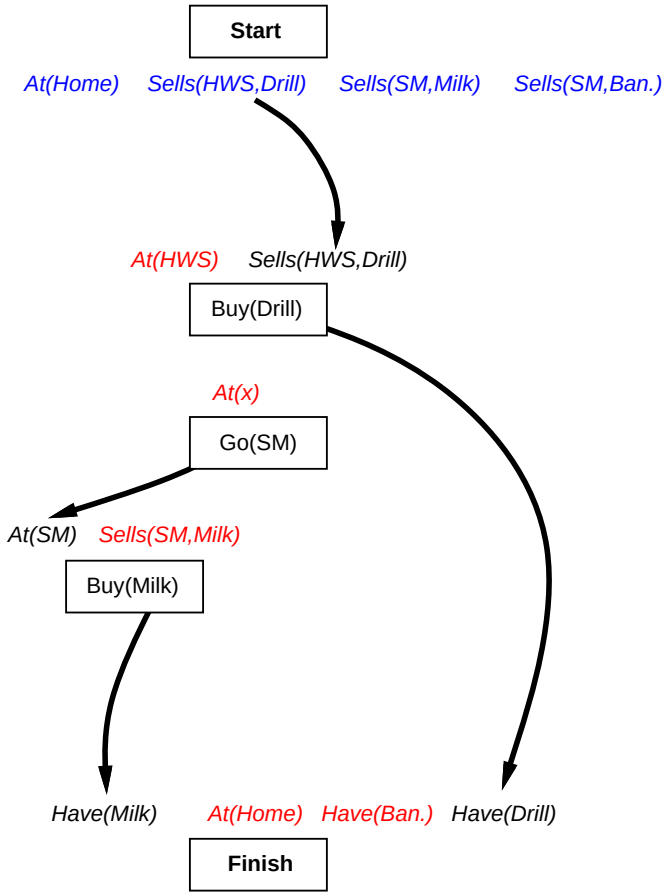
Partial-Order Shopping

Preconditions: red- open subgoals; black - satisfied subgoals
Arrows: black - causal links; green - ordering constraints: act-a $\prec$ act-b.



Partial-Order Shopping (2)

1) Add Buy(Drill) to satisfy Have(Drill) of Finish
2) Add Buy(Milk) to satisfy Have(Milk) of Finish
3) Add Go(SM) to satisfy At(SM) of Buy(Milk)
4) Add causal link from Start to precondition Sells(HWS,Drill) of Buy(Drill)



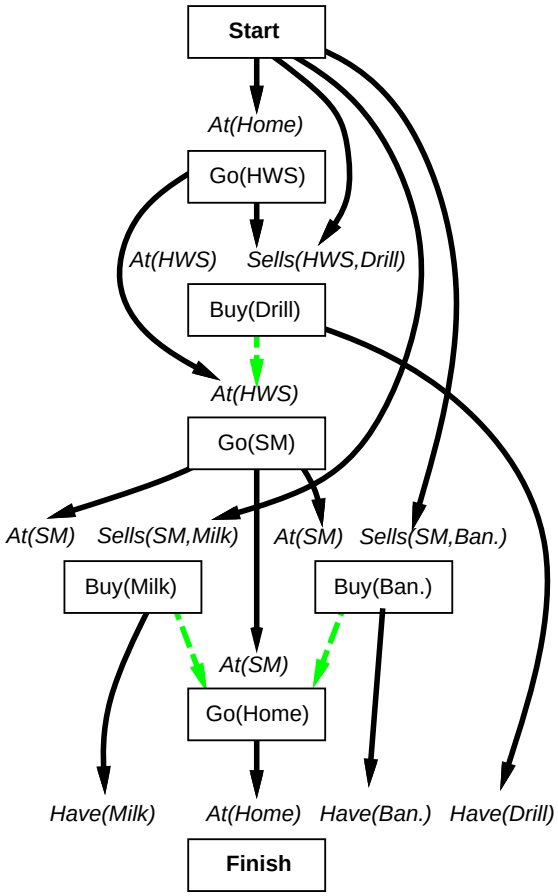
Partial-Order Shopping (3)

1. Add Buy(Bananas) to satisfy Have(Bananas) precondition of Finish.
2. Add causal links from Start to the preconditions Sells(SM,Milk) and Sells(SM,Bananas) of Buy(Milk) and Buy(Bananas), respectively.
3. Add Go(SM) to satisfy At(SM) of Buy(Bananas) and Buy(Milk).
4. Add Go(Home) to satisfy At(Home) of Finish.
5. Note that Go(Home) could conflict with Buy(Milk) and Buy(Bananas) by clobbering the At(SM) precondition. Add ordering constraints so that Go(Home) does not come between Go(SM) and Buy(Milk) and Buy(Bananas): Buy(Milk) $\prec$ Go(SM), and Buy(Bananas) $\prec$ Go(SM).

Partial-Order Shopping (4)

6. Add Go(HWS) to satisfy At(HWS) of Buy(Drill).
7. Add causal link from Start to Go(HWS), since At(Home) (the Effect of Start) satisfies the precondition: At(X) where X $\neq$ HWS.
8. Similarly, add causal link from Go(HWS) to Go(SM), since At(HWS) satisfies the precondition: At(X) where X $\neq$ SM.
9. Add a 3rd such causal link between Go(SM) and Go(Home), since Go(SM) satisfies the precondition for Go(Home).
10. Note that Go(SM) could conflict with At(HWS) precondition of Buy(Drill). Add ordering constraint so that Go(SM) does not come between Go(HWS) and Buy(Drill): Buy(Drill) $\prec$ Go(SM).

Partial-Order Shopping (5)



Partial-Order Planning (POP) Overview

Operators on partial plans:

- add a link from an existing action to an open condition
- add a step (i.e., a new action) to fulfill an open condition
- order one step wrt another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans

Backtrack if an open condition is unachievable or
if a conflict is unresolvable

POP algorithm

function POP(*initial, goal, operators*) **returns** *plan*

  plan ← MAKE-MINIMAL-PLAN(*initial, goal*)
  loop do
    if SOLUTION?(*plan*) **then return** *plan*
    $S_{need},\ c \leftarrow$ SELECT-SUBGOAL(*plan*)
    CHOOSE-OPERATOR(*plan, operators,* S_{need}, c)
    RESOLVE-THREATS(*plan*)
  end

---

function SELECT-SUBGOAL(*plan*) **returns** $S_{need},\ c$

  pick a plan step S_{need} from STEPS(*plan*)
    with a precondition c that has not been achieved
  return $S_{need},\ c$

POP algorithm (2)

procedure CHOOSE-OPERATOR($plan$, $operators$, S_{need}, c)

choose a step S_{add} from $operators$ or STEPS($plan$) that has c as an effect
if there is no such step **then fail**
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS($plan$)
add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS($plan$)
if S_{add} is a newly added step from $operators$ **then**
    add S_{add} to STEPS($plan$)
    add $Start \prec S_{add} \prec Finish$ to ORDERINGS($plan$)

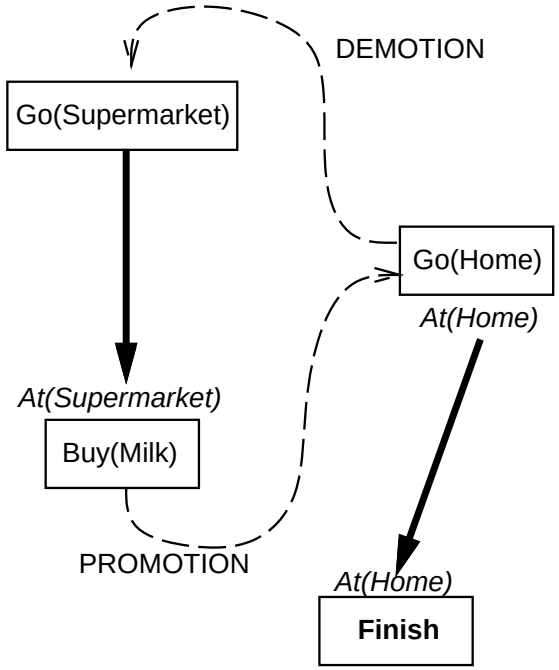
---

procedure RESOLVE-THREATS($plan$)

for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS($plan$) **do**
    choose either
        Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS($plan$)
        Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS($plan$)
    if not CONSISTENT($plan$) **then fail**
end

Subgoal Clobbering and Promotion/Demotion

A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., $Go(Home)$ clobbers $At(Supermarket)$:



Demotion: put before $Go(Supermarket)$

Promotion: put after $Buy(Milk)$

Properties of POP

Nondeterministic algorithm: backtracks at choice points on failure:

- choice of S_{add} to achieve S_{need}
- choice of demotion or promotion for clobberer
- selection of S_{need} is irrevocable

POP is sound, complete, and systematic (no repetition)

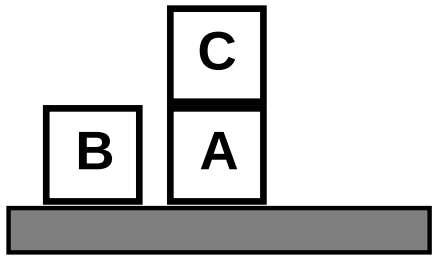
Extensions for disjunction, universals, negation, conditionals

Can be made efficient with good heuristics derived from problem description

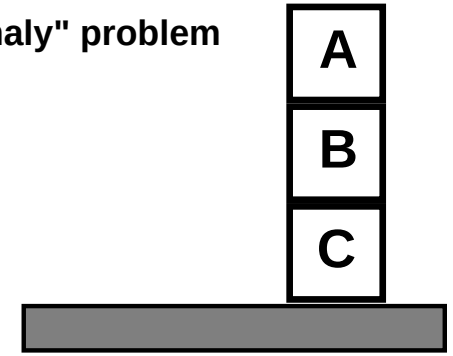
Particularly good for problems with many loosely related subgoals

Example: Blocksworld

"Sussman anomaly" problem



Start State



Goal State

Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

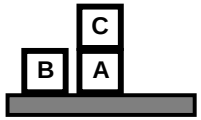
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Blocksworld (2)

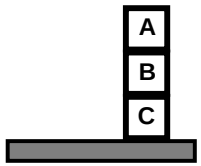
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

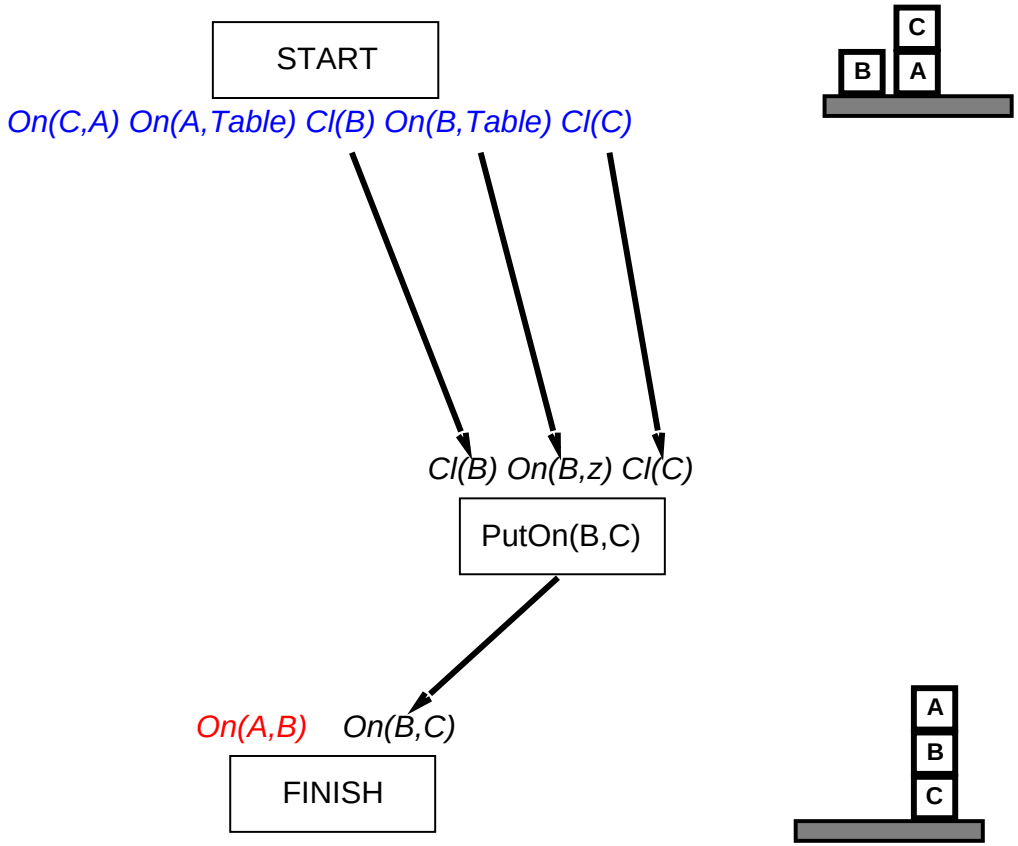


On(A,B) On(B,C)

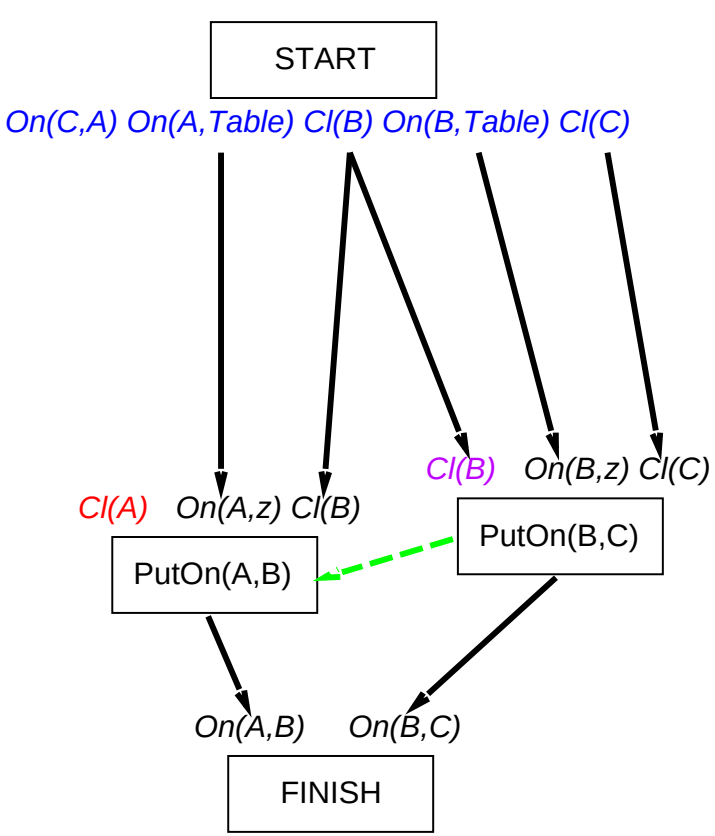
FINISH

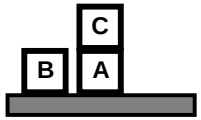


Blocksworld (3)

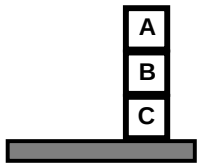


Blocksworld (4)

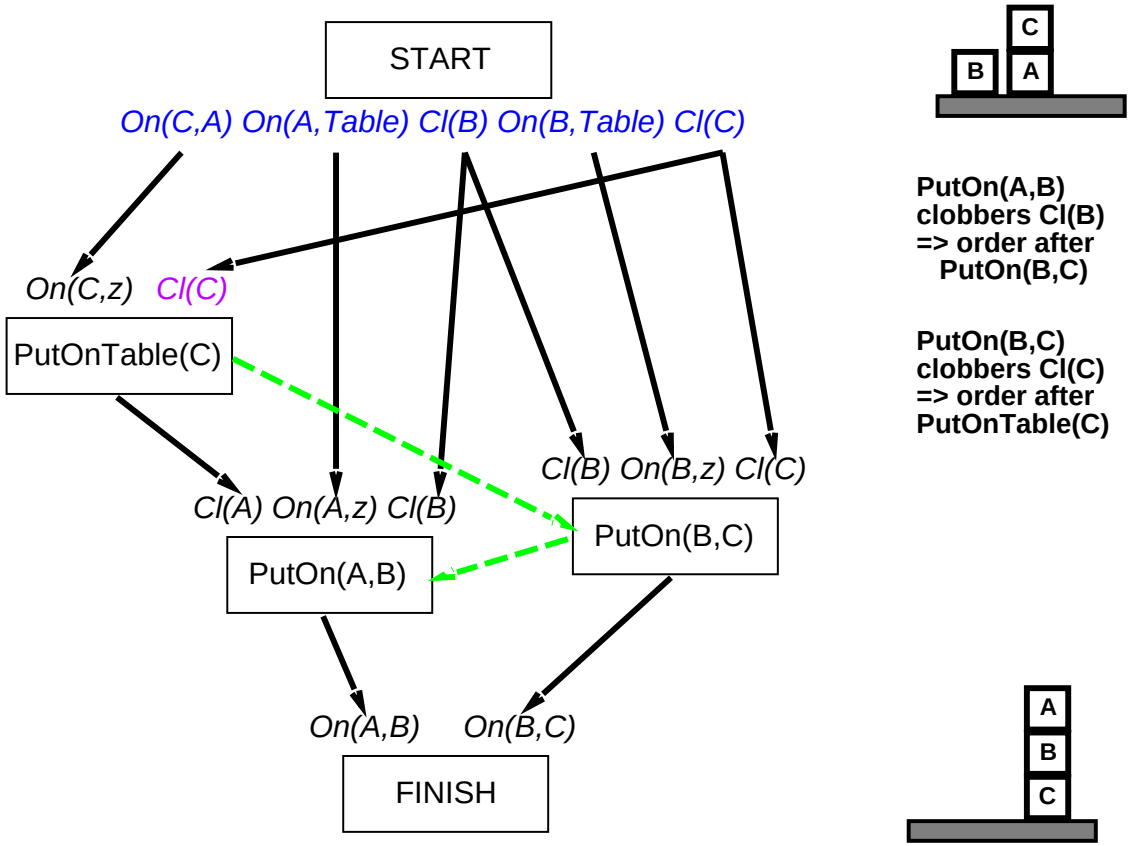




PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)



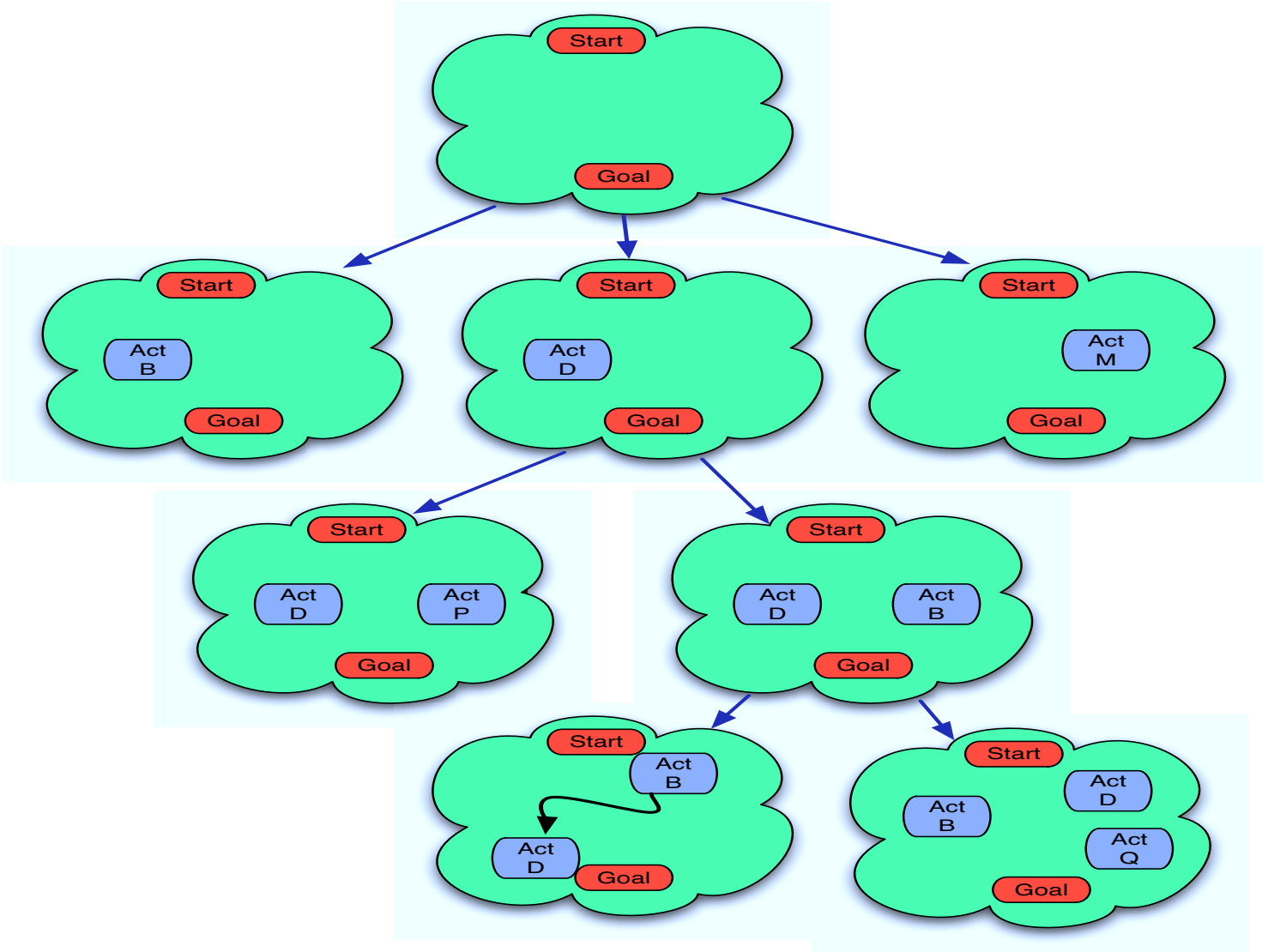
Blocksworld (5)



Final totally-ordered plan:

1. PutOnTable(C) 2. PutOn(B,C) 3. PutOn(A,B)

Partial-Order Planning as Search



Choices during POP Search

1. The planning-state to expand - use h
2. The open subgoal/pre-condition to try to satisfy.

Partial-Order Planning Search Heuristics

POP doesn't work directly with real-world states (as does total-order planning) $\rightarrow$ hard to estimate h.

Best Possibilities for h

- Number Open Preconditions
- Number Open Preconditions - Number Preconds satisfied by Start (but possibly clobbered) along the way.
  - Optimistic - when initially-true subgoals get clobbered
  - Pessimistic - when one operator can achieve multiple subgoals

Best Possibility for next open-precondition to satisfy

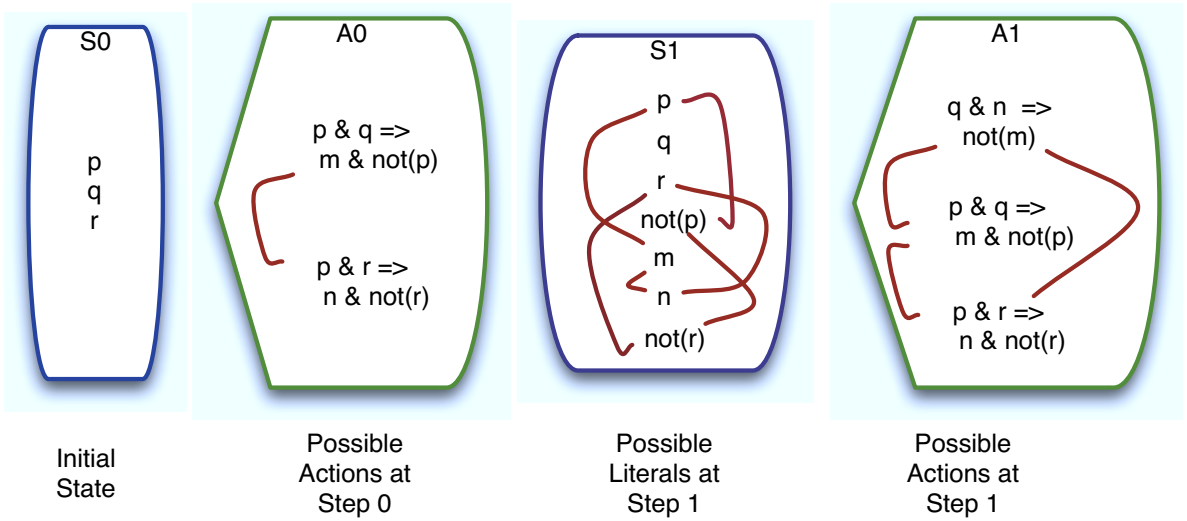
Best generator criteria = Most-constrained precondition: the one with FEWEST number of actions that can satisfy it.

Planning Graphs

Series of Levels, where the nth level contains:

1) All literals that *could* be true after nth planning steps.
2) All actions whose preconditions *could* be true after nth planning step.

ONLY work with propositional reps - no variables!



Red arcs = mutual exclusion (mutex):
Cannot be simultaneously true (literals) or independently executed (rules)
during the given planning step.

Planning-Graph Generation

1. S0 = the initial state = all initially-true literals
2. A0 = set of all actions whose preconditions are satisfied by S0 literals.
3. Record the mutually-exclusive (mutex) constraints among actions of A0.
4. S1 = set of all literals that could be true if at least one action in A0 is performed. Also, all literals in S0 are in S1 via **persistence actions**.
5. Record the mutex constraints among literals in S1.
6. A1 = set of all actions whose preconditions are satisfied by S1 literals.
7. Continue until level S(n) = S(n+1)*

*In GRAPHPLAN, halting may occur before reaching this steady-state:
If all literals in the goal state are found in S(k),
and if no pairs of these literals are mutex,
Then GraphPlan will stop generating states
and search backwards for a proper action sequence.

Criteria for Mutual Exclusion (Mutex Links)

Actions

1. **Inconsistent Effects**: The effects of one action are inconsistent with the effects of another, i.e. one asserts A and the other asserts not(A).
2. **Interference**: An effect of one action is inconsistent with a precondition of the other, i.e., one asserts not(A) and the other has A as a precondition.
3. **Competing Needs**: A precondition of one action is mutex with a precondition of the other action.

In general, two actions are NOT mutex iff they are independent and hence COULD both be executed at the same time.

Literals

1. **Inconsistent**: One literal is the negation of the other.
2. **Inconsistent Support**: Every pair of actions (that produce the two literals) is mutex. This assumes that the two literals are not effects of the same action.

Solution Extraction in Graphplan

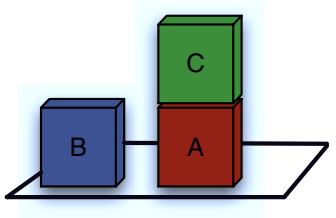
Solving Boolean CSP, where vars are actions at each A(s) level.
Find a set of in/out assignments to those variables so that all subgoals are satisfied.

1. Init state = last-generated layer, S(k) of Planning Graph. This includes all subgoals, SG(k).
2. n = k.
3. If n== 0, return C(1)..C(k) as the solution; else continue.
4. At action level A(n-1), choose a conflict-free subset of the actions that cover the subgoals in S(n). Call this covering set C(n).
   Conflict-free $\longrightarrow$ for each action pair, the actions are not mutex, nor are any of their preconditions.
5. If no C(n) is found, backtrack (i.e. n = n + 1, Go to Step 3).
6. Let SG(n-1) = Preconditions(C(n)).
7. n = n - 1.
8. Go to Step 3

Graphplan Blocksworld

S0

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

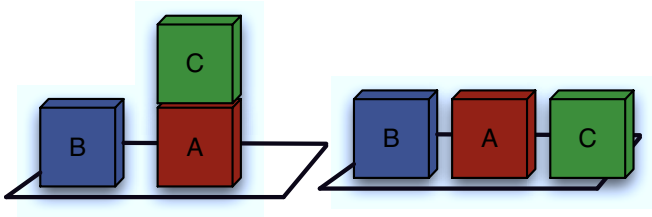


A0

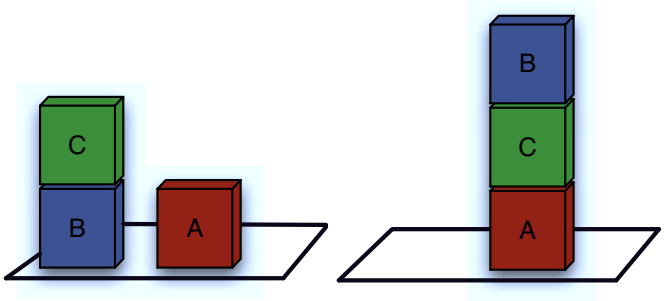
stack(c,b)	persist(on(a,table))
stack(b,c)	persist(on(c,a))
putontable(c)	persist(clear(b))
persist(on(b,table))	persist(clear(c))

S1

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



*These complete states are only for illustrative purposes. Graph planners only work with literals and rules, not complete states.



Graphplan Blocksworld (2)

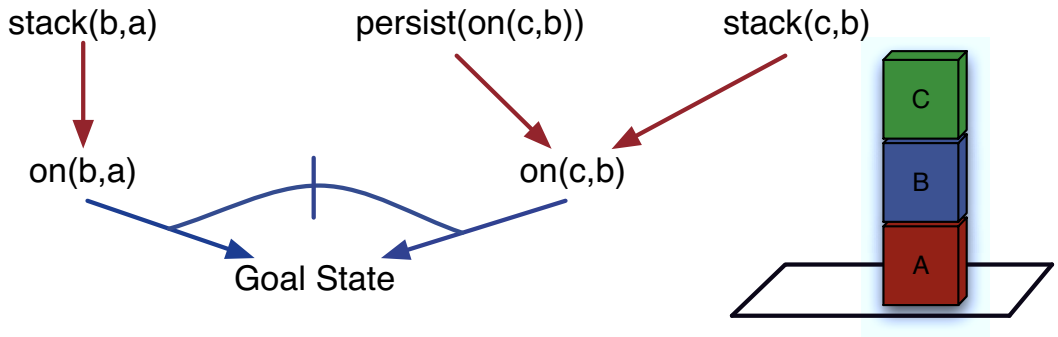
A1

stack(a,b)	persist(on(a,table))	persist(clear(a))	putontable(b)
stack(a,c)	persist(on(b,table))	persist(clear(b))	putontable(c)
stack(b,a)	persist(on(c,table))	persist(clear(c))	
stack(b,c)	persist(on(b,c))	persist(not(clear(b)))	
stack(c,a)	persist(on(c,b))	persist(not(clear(c)))	
stack(c,b)	persist(on(c,a))		

S2

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

Every literal is possible, but MANY are mutually exclusive (mutex).



Graphplan Blocksworld (3)

The two subgoals(literals) on(b,a) and on(c,b) are mutually exclusive (mutex), because each possible action pair that achieves them is mutex:

- stack(b,a) – mutex – persist(on(c,b)) by **competing needs** criteria: The precondition clear(b) of stack(b,a) is mutex with the precondition on(c,b) of persist(on(c,b)).
- stack(b,a) –mutex–stack(c,b) by **interference** criteria: The effect not(clear(b)) of stack(c,b) is the negation of the precondition clear(b) of stack(b,a).

The goal state cannot be achieved in S2, so GraphPlan needs to generate A2 and S3.

Graphplan Blocksworld (4)

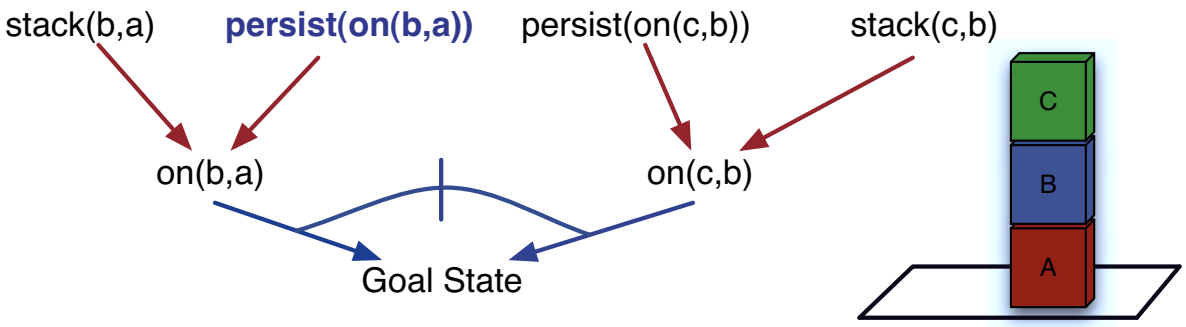
A2

stack(a,b)	persist(on(a,table))	**persist(on(a,b))**	putontable(b)
stack(a,c)	persist(on(b,table))	**persist(on(a,c))**	putontable(c)
stack(b,a)	persist(on(c,table))	**persist(on(b,a))**	persist(clear(a))
stack(b,c)	persist(on(b,c))	**persist(not(clear(a)))**	persist(clear(b))
stack(c,a)	persist(on(c,b))	persist(not(clear(b)))	persist(clear(c))
stack(c,b)	persist(on(c,a))	persist(not(clear(c)))	

S3

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

Again, every literal is possible, but FEWER are mutually exclusive than in S2, since there are MORE actions that support them.



Graphplan Blocksworld (5)

- In S3, on(b,a) and on(c,b) are NOT mutex, since at least one pair of their action supports are not mutex: persists(on(b,a)) and stack(c,b).
- This is a **necessary**, but not **sufficient** criterium for the goal to be achievable by a legal plan.
- So Graphplan stops at S3 and tries to find a plan by working backwards.
- If none is found, it continues trying by generating A3 and S4.

Graphplan Halting

1. Literals increase monotonically: Due to persistence actions, once a literal appears in S(i), it exists in S(j) $\forall j > i$
2. Actions increase monotonically: Since literals increase monotonically, the preconditions for an action never disappear once present. So more and more actions become possible in successive layers.
3. Mutexes DECREASE monotonically! Several cases depending upon type of mutex:
   (a) Inconsistent effect - property of actions only, so these don't change.
   (b) Interference - again, property of actions; no change.
   (c) Competing needs - preconditions that are mutex =¿ actions are mutex. This CAN change over time if the precondition literals become non-mutex.

Graphplan Halting (2)

Mutex changes among literals:

1. Inconsistent - this never changes.
2. Inconsistent support - this can change. Since actions are increasing monotonically, the support for literals increases monotonically. So at some point, two mutex literals may gain action supports that are not mutex. E.g., the two subgoals in the above Blocksworld example

- Since number of literals and actions are monotonic and finite, they will eventually reach a stable level.
- Since the number of mutexes is monotonically decreasing, they too will reach a stable level.
- Hence, the planning graph will eventually level off, with S(m) = S(m+1) for some m.

Planning Graphs used for Heuristics

- Heuristics for other planners. The level at which a literal first appears (**level cost**) gives a good estimate of the minimal number of actions needed to produce it.
- Hence, a 2nd planner could sum the level costs of a state's subgoals to compute h(state).
- However, this is not an *admissible* heuristic, since each level may involve several parallel actions
- So use a **serial planning graph**.

Heuristics from a Serial Planning Graph

Serial Planning Graph = planning graph which has mutex links among all pairs of non-persistent actions.

1. **Max-Level**: Max level-cost of the state's subgoals - admissible but weak.
2. **Level-Sum**: Sum of level costs of all subgoals - nonadmissible, since an action may achieve several subgoals at once, but quite useful for problems that are reasonably decomposable.
3. **Set-Level**: Level at which ALL subgoals first appear, with no mutexes between pairs. Admissible and very effective, especially in domains with a lot of subgoal interaction.

Complexity Analysis of Graph-Based Planning

Let A = Number of general actions, e.g. stack(x,y)
Let L = Number of literals (e.g. on(block-A, block-B))
Let M = f(A,L) = Number of specific actions (e.g. stack(block-A, block-B))
- e.g. f(A,L) = A*L.

Normal State-Space Search

- worst-case branching factor of M (all specific actions from each state)
- So a k-level search has complexity $O(M^k)$.

Complexity Analysis of GraphPlan

Generating the Planning Graph

- $O(L^2)$ to generate each state level; consider mutex relations between all pairs of literal. item $O((L+M)^2)$ to generate an action level: consider mutex relations between all actions (including O(L) persistence links)
- $O(k(L^2+(L+M)^2)) = O(k(L^2+M^2)) = O(kM^2)$ for a k-level search.

Extracting a Solution

- This is backward search across the Action levels, looking for groups of independent acts that satisfy the next state-levels subgoals.
- Assume a maximum action-group size of g.
- Then each search node in this space has a worst-case branching factor of:

$$\binom{L+M}{g} \approx (L+M)^g \tag{1}$$

Complexity Analysis of GraphPlan (2)

- So a k-level search has extraction complexity $O(k(L+M)^g)$

Total Graph Generation + Plan Extraction: $O(kM^2 + kM^g) = O(kM^g)$
So GraphPlan beats conventional state-space planning when $g < k$
That is quite common, since most significant problems have a larger k than g:
The number of plan steps that could be done in parallel is normally much less than the total number of steps in the plan!!