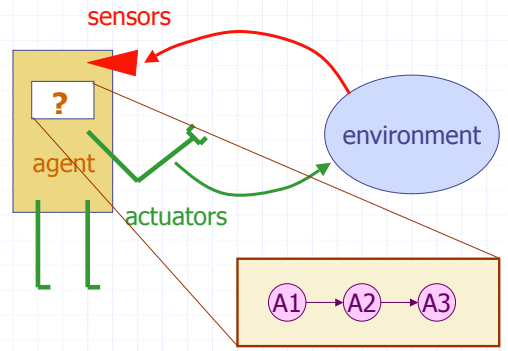
Planning

Russell and Norvig: Chapter 11

CMSC421 – Fall 2003

based on material from Jim Blythe, JC Latombe, Marie desJardins and Daphne Koller

Planning Agent



Planning problem

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,

  compute a plan, which is
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of subgoals to be achieved

Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Search often proceeds through *plan space* rather than *state space* (though there are also state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Goal of Planning

- • Suppose that the goal is HAVE(MILK).
- • From some initial state where HAVE(MILK) is not satisfied, the successor function must be repeatedly applied to eventually generate a state where HAVE(MILK) is satisfied.
- • An explicit representation of the possible actions and their effects would help the problem solver select the relevant actions

Otherwise, in the real world an agent would be overwhelmed by irrelevant actions

Goal of Planning

Suppose that the goal is HAVE(MILK)∧HAVE(BOOK)

Without an explicit representation of the goal, the problem solver cannot know that a state where HAVE(MILK) is already achieved is more promising than a state where neither HAVE(MILK) nor HAVE(BOOK) is achieved

states are domain-specific data structures, and heuristics must be supplied for each new problem

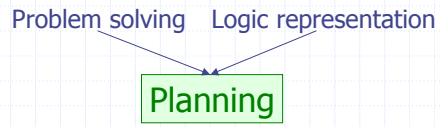
Goal of Planning

- Choose actions to achieve a certain goal
- HAVE(MILK) and HAVE(BOOK) may be achieved by two nearly independent sequences of actions
- Some difficulties with problem solving:
  - The goal may consist of several nearly independent subgoals, but there is no way for the problem solver to know it

Representations in Planning

Planning opens up the black-boxes by using logic to represent:

- Actions
- States
- Goals



Major approaches

Planning rapidly changing subfield of AI

In biannual competition at AI Planning Systems Conference:
- four years ago, best planner did plan space search using SAT solver
- three years ago, the best planner did regression search
- last year, best planner did forward state space search with an inadmissable heuristic function

- Hierarchical decomposition (HTN planning)

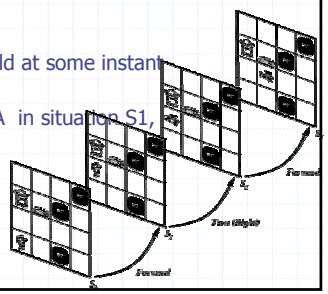
cmsc722: Planning, taught by Prof. Nau

Situation Calculus Planning

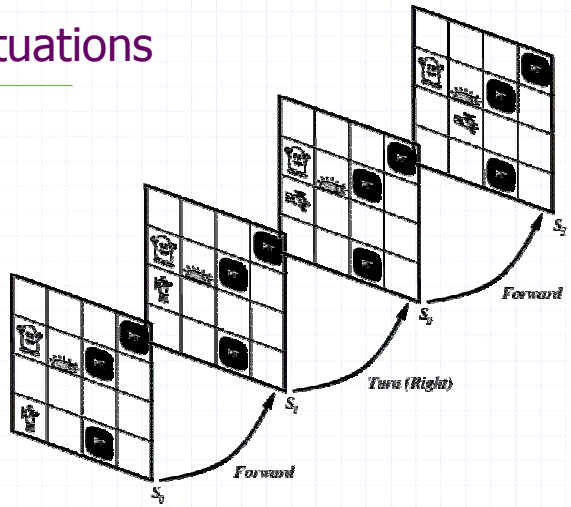
- Formulate planning problem in FOL
- Use theorem prover to find proof (aka plan)

Representing change

- Representing change in the world in logic can be tricky.
- One way is just to change the KB
  - Add and delete sentences from the KB to reflect changes
  - How do we remember the past, or reason about changes?
- **Situation calculus** is another way
- A **situation** is a snapshot of the world at some instant in time
- When the agent performs an action A in situation S1, the result is a new situation S2.



Situations



Situation calculus

- A **situation** is a snapshot of the world at an interval of time during which nothing changes
- Every true or false statement is made with respect to a particular situation.
  - Add **situation variables** to every predicate.
  - at(hunter,1,1) becomes at(hunter,1,1,s0): at(hunter,1,1) is true in situation (i.e., state) s0.
- Add a new function, **result(a,s),** that maps a situation s into a new situation as a result of performing action a. For example, result(forward, s) is a function that returns the successor state (situation) to s
- Example: The action agent-walks-to-location-y could be represented by
  - $(\forall x)(\forall y)(\forall s)$ (at(Agent,x,s) ^ ~onbox(s)) -> at(Agent,y,result(walk(y),s))

Situation calculus planning

- **Initial state**: a logical sentence about (situation) S_0
  At(Home, S_0) ^ ~Have(Milk, S_0) ^ ~ Have(Bananas, S_0) ^ ~Have(Drill, S_0)
- **Goal state**:
  $(\exists s)$ At(Home,s) ^ Have(Milk,s) ^ Have(Bananas,s) ^ Have(Drill,s)
- **Operators** are descriptions of actions:
  $\forall$(a,s) Have(Milk,Result(a,s)) <=> ((a=Buy(Milk) ^ At(Grocery,s)) $\vee$ (Have(Milk, s) ^ a~=Drop(Milk)))
- Result(a,s) names the situation resulting from executing action a in situation s.
- Action sequences are also useful: Result'(l,s) is the result of executing the list of actions (l) starting in s:
  $(\forall s)$ Result'([],s) = s
  $(\forall a,p,s)$ Result'([a|p]s) = Result'(p,Result(a,s))

Situation calculus planning II

- A solution is thus a plan that when applied to the initial state yields a situation satisfying the goal query:
  At(Home,Result'(p,S_0))
  ^ Have(Milk,Result'(p,S_0))
  ^ Have(Bananas,Result'(p,S_0))
  ^ Have(Drill,Result'(p,S_0))
- Thus we would expect a plan (i.e., variable assignment through unification) such as:
  p = [Go(Grocery), Buy(Milk), Buy(Bananas), Go(HardwareStore), Buy(Drill), Go(Home)]

SC planning: analysis

- This is fine in theory, but remember that problem solving (search) is exponential in the worst case
- Also, resolution theorem proving only finds *a* proof (plan), not necessarily a good plan
- Another important issue: **the Frame Problem**

The Frame Problem

- In SC, need not only axioms to describe what changes in each situation, but also need axioms to describe what stays the same (can do this using successor-state axioms)
- Qualification problem: difficulty in specifying all the conditions that must hold in order for an action to work
- Ramification problem: difficulty in specifying all of the effects that will hold after an action is taken

So…

- we restrict the language and use a special-purpose algorithm (a planner) rather than general theorem prover

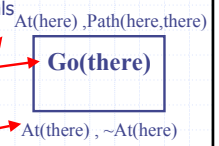
Basic representations for planning

- Classic approach first used in **STRIPS** planner circa 1970
- States represented as a conjunction of ground literals
  - at(Home)
- Goals are conjunctions of literals, but may have existentially quantified variables
  - at(?x) ^ have(Milk) ^ have(bananas) …
- Do not need to fully specify state
  - Non-specified either don't-care or assumed false
  - Represent many cases in small storage
  - Often only represent changes in state rather than entire situation
- Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it

Operator/action representation

- Operators contain three components:
  - **Action description**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied
- Example:

  Op[Action: Go(there),
  Precond: At(here) ^ Path(here,there),
  Effect: At(there) ^ ~At(here)]



- All variables are universally quantified
- Situation variables are implicit
  - preconditions must be true in the state immediately before operator is applied; effects are true immediately after

Blocks world

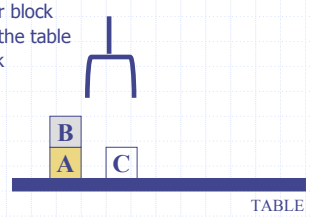
The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand.

Some domain constraints:

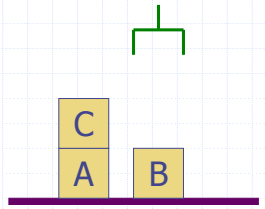
- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

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

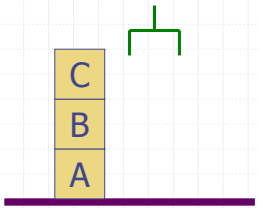


State Representation



Conjunction of propositions:
BLOCK(A), BLOCK(B), BLOCK(C), ON(A,TABLE), ON(B,TABLE), ON(C,A), CLEAR(B), CLEAR(C), HANDEMPTY

Goal Representation



Conjunction of propositions:
ON(A,TABLE), ON(B,A), ON(C,B)

The goal G is achieved in a state S if all the propositions in G are also in S

Action Representation

Unstack(x,y)

- P = HANDEMPTY, BLOCK(x), BLOCK(y), CLEAR(x), ON(x,y)
- E = ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x), ¬ ON(x,y), CLEAR(y)

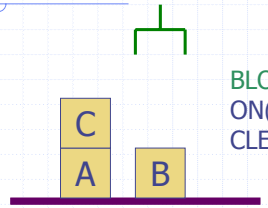
Effect: list of literals

Precondition: conjunction of propositions

Means: Remove HANDEMPTY from state

Means: Add HOLDING(x) to state

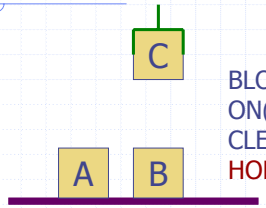
Example



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ON(C,A),
CLEAR(B), CLEAR(C), HANDEMPTY

Unstack(C,A)
- P = HANDEMPTY, BLOCK(C), BLOCK(A), CLEAR(C), ON(C,A)
- E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C), ¬ ON(C,A), CLEAR(A)

Example



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ~~ON(C,A)~~,
CLEAR(B), ~~CLEAR(C)~~, ~~HANDEMPTY~~,
HOLDING(C), CLEAR(A)

Unstack(C,A)
- P = HANDEMPTY, BLOCK(C), BLOCK(A), CLEAR(C), ON(C,A)
- E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C), ¬ ON(C,A), CLEAR(A)

Action Representation

Unstack(x,y)
- P = HANDEMPTY, BLOCK(x), BLOCK(y), CLEAR(x), ON(x,y)
- E = ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x), ¬ ON(x,y), CLEAR(y)

Stack(x,y)
- P = HOLDING(x), BLOCK(x), BLOCK(y), CLEAR(y)
- E = ON(x,y), ¬CLEAR(y), ¬HOLDING(x), CLEAR(x), HANDEMPTY

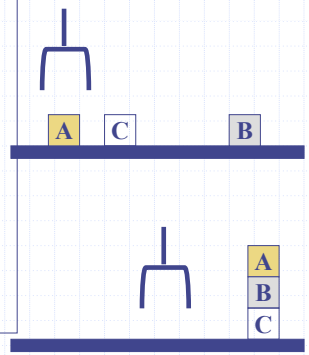
Pickup(x)
- P = HANDEMPTY, BLOCK(x), CLEAR(x), ON(x,TABLE)
- E = ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x), ¬ON(x,TABLE)

PutDown(x)
- P = HOLDING(x)
- E = ON(x,TABLE), ¬HOLDING(x), CLEAR(x), HANDEMPTY

Typical BW planning problem

Initial state:
- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:
- on(b,c)
- on(a,b)
- ontable(c)

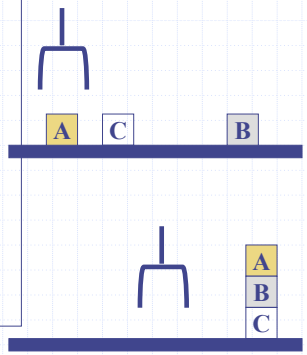


A plan:
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Another BW planning problem

Initial state:
- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

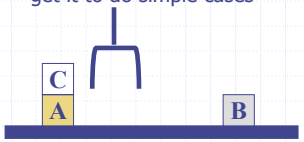
Goal:
- on(a,b)
- on(b,c)
- ontable(c)



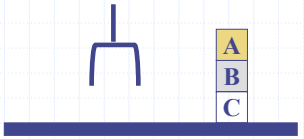
A plan:
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Goal interaction

- Simple planning algorithms assume that the goals to be achieved are independent
  - Each can be solved separately and then the solutions concatenated
- This planning problem, called the "Sussman Anomaly," is the classic example of the goal interaction problem:
  - Solving on(A,B) first (by doing unstack(C,A), stack(A,B) will be undone when solving the second goal on(B,C) (by doing unstack(A,B), stack(B,C)).
  - Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS could not handle this, although minor modifications can get it to do simple cases

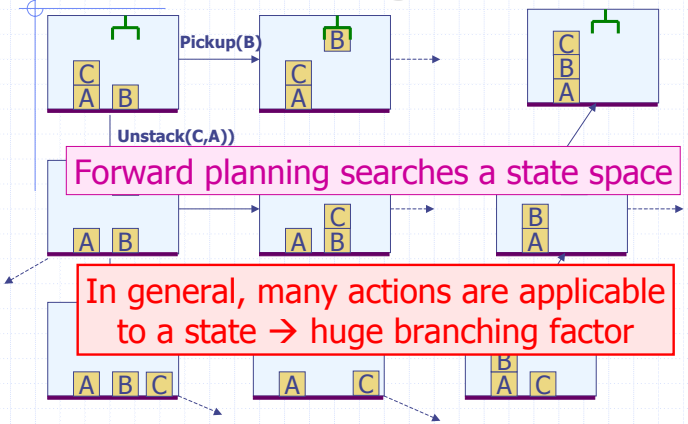


Initial state



Goal state

Forward Planning

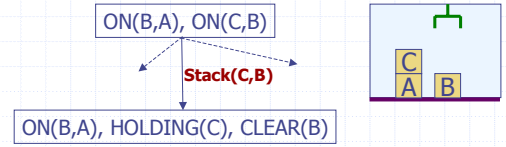


Relevant Action

- An action is relevant to a goal if one of its effects matched a goal proposition
- Stack(B,A)
  - P = HOLDING(B), BLOCK(A), CLEAR(A)
  - E = ON(B,A), ¬CLEAR(A), ¬HOLDING(B), CLEAR(B), HANDEMPTY

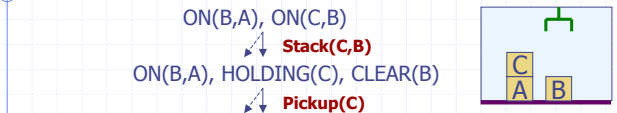
  is relevant to ON(B,A), ON(C,B)

Backward Chaining



In general, there are much fewer actions relevant to a goal than there are actions applicable → smaller branching factor than with forward planning

Backward Chaining



Backward planning searches a goal space

Pickup(B)

CLEAR(C), ON(C,TABLE), CLEAR(A), HANDEMPTY, CLEAR(B), ON(B,TABLE)

Putdown(C)

CLEAR(A), CLEAR(B), ON(B,TABLE), HOLDING(C)

Unstack(C,A)

CLEAR(B), ON(B,TABLE), CLEAR(C), HANDEMPTY, ON(C,A)

State-space planning

- We initially have a space of situations (where you are, what you have, etc.)
- The plan is a solution found by "searching" through the situations to get to the goal
- A **progression planner** searches forward from initial state to goal state
- A **regression planner** searches backward from the goal
  - Ideally this leads to reduced branching –you are only considering things that are relevant to the goal

Plan-space planning

- An alternative is to **search through the space of *plans***, rather than situations.
- Start from a **partial plan** which is expanded and refined until a complete plan that solves the problem is generated.
- **Refinement operators** add constraints to the partial plan and modification operators for other changes.
- We can still use STRIPS-style operators:

  Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)
  Op(ACTION: RightSock, EFFECT: RightSockOn)
  Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)
  Op(ACTION: LeftSock, EFFECT: leftSockOn)

  could result in a partial plan of

  [RightShoe, LeftShoe]

Partial-order planning

- A **linear planner** builds a plan as a **totally ordered sequence** of plan steps
- A **non-linear planner (aka partial-order planner)** builds up a plan as a set of steps with some temporal constraints
  - constraints of the form S1<S2 if step S1 must comes before S2.
- One **refines** a partially ordered plan (POP) by either:
  - **adding a new plan step**, or
  - **adding a new constraint** to the steps already in the plan.
- A POP can be **linearized** (converted to a totally ordered plan) by topological sorting

Least commitment

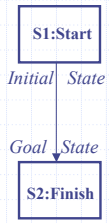
- Non-linear planners embody the principle of **least commitment**
  - only choose actions, orderings, and variable bindings that are absolutely necessary, leaving other decisions till later
  - avoids early commitment to decisions that don't really matter
- A linear planner always chooses to add a plan step in a particular place in the sequence
- A non-linear planner chooses to add a step and possibly some temporal constraints

Non-linear plan

- A non-linear plan consists of
  - (1) A set of **steps** $\{S_1, S_2, S_3, S_4 \ldots\}$
    - Each step has an operator description, preconditions and post-conditions
  - (2) A set of **causal links** $\{ \ldots (S_i,C,S_j) \ldots\}$
    - Meaning a purpose of step S_i is to achieve precondition C of step S_j
  - (3) A set of **ordering constraints** $\{ \ldots S_i<S_j \ldots \}$
    - if step S_i must come before step S_j
- A non-linear plan is **complete** iff
  - Every step mentioned in (2) and (3) is in (1)
  - If S_j has prerequisite C, then there exists a causal link in (2) of the form (S_i,C,S_j) for some S_i
  - If (S_i,C,S_j) is in (2) and step S_k is in (1), and S_k threatens (S_i,C,S_j) (makes C false), then (3) contains either $S_k<S_i$ or $S_j>S_k$

The initial plan

Every plan starts the same way



Trivial example

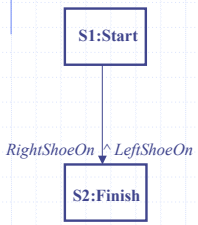
Operators:

Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)

Op(ACTION: RightSock, EFFECT: RightSockOn)

Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)

Op(ACTION: LeftSock, EFFECT: leftSockOn)

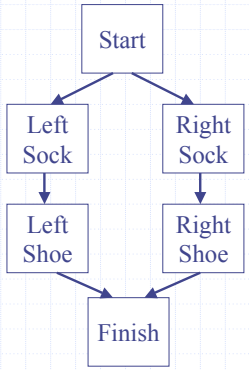


Steps: {S1:[Op(Action:Start)],
S2:[Op(Action:Finish,
Pre: RightShoeOn^LeftShoeOn)]}

Links: {}

Orderings: {S1<S2}

Solution

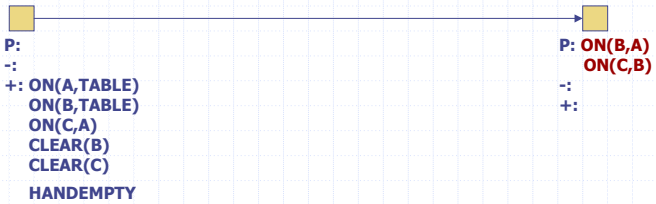


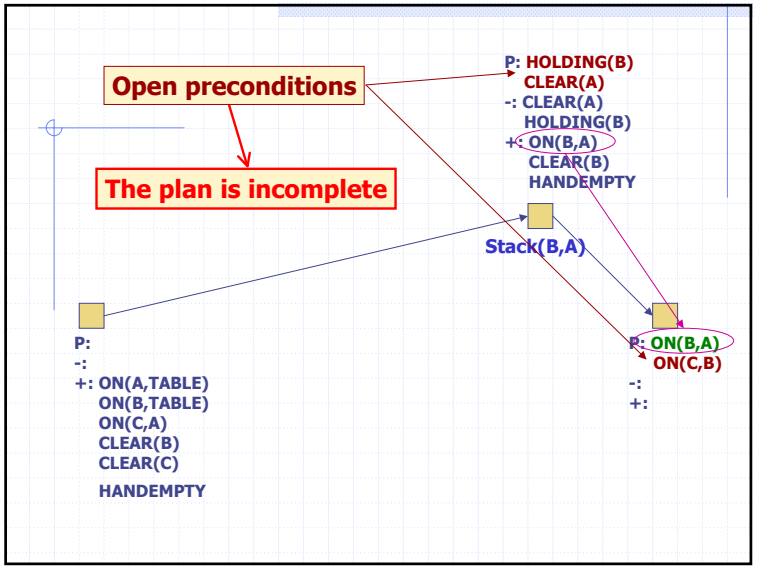
Nonlinear Planning

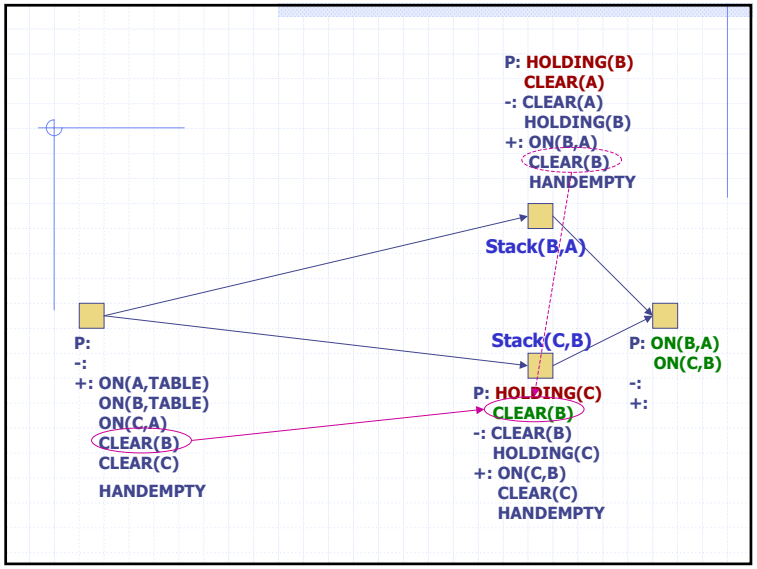
- Both forward and backward planning methods commit to a total ordering on the selected actions
- Drawb[illegible]ge of pos[illegible]ng subgoals
- Nonlinear planning: No unnecessary ordering among subgoals

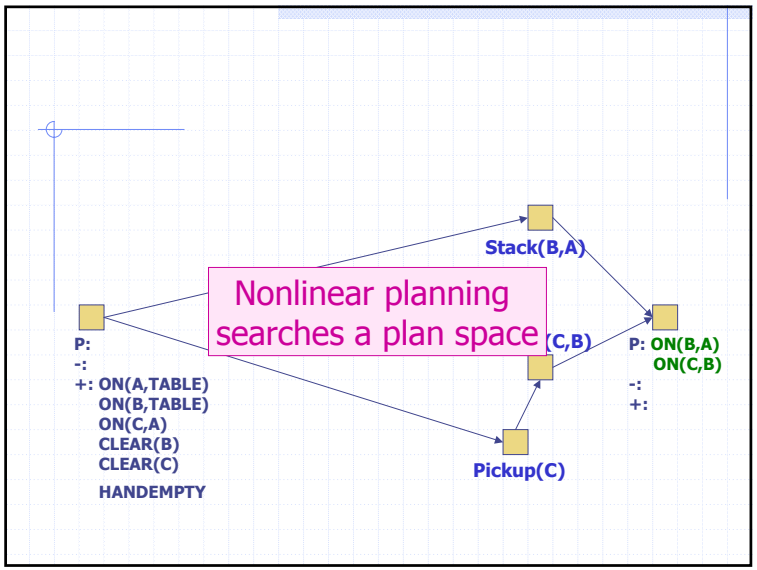
Form of least commitment

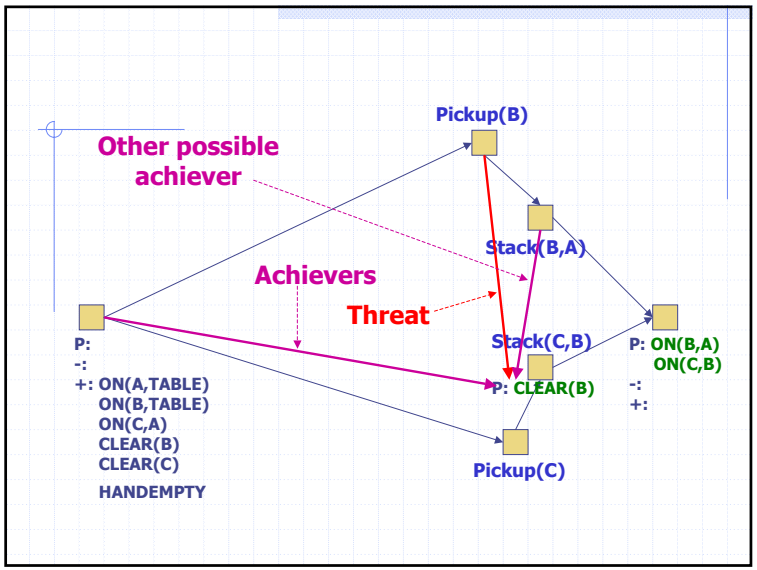
Nonlinear Planning

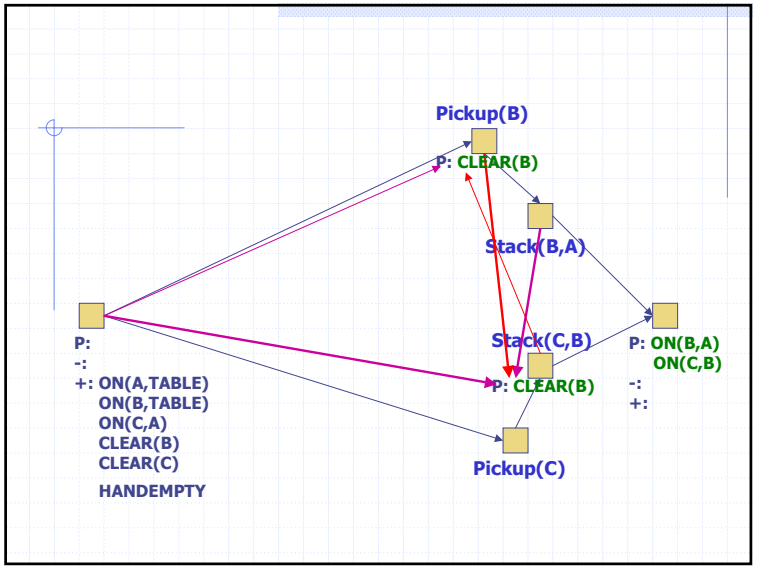


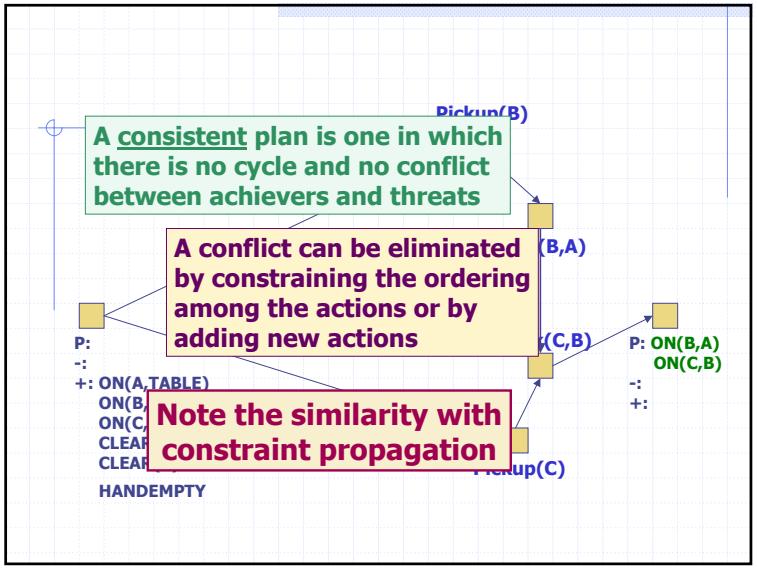

Open preconditions
The plan is incomplete
P: HOLDING(B)
CLEAR(A)
-: CLEAR(A)
HOLDING(B)
+: ON(B,A)
CLEAR(B)
HANDEMPTY
Stack(B,A)
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY
P: ON(B,A)
ON(C,B)
-:
+:

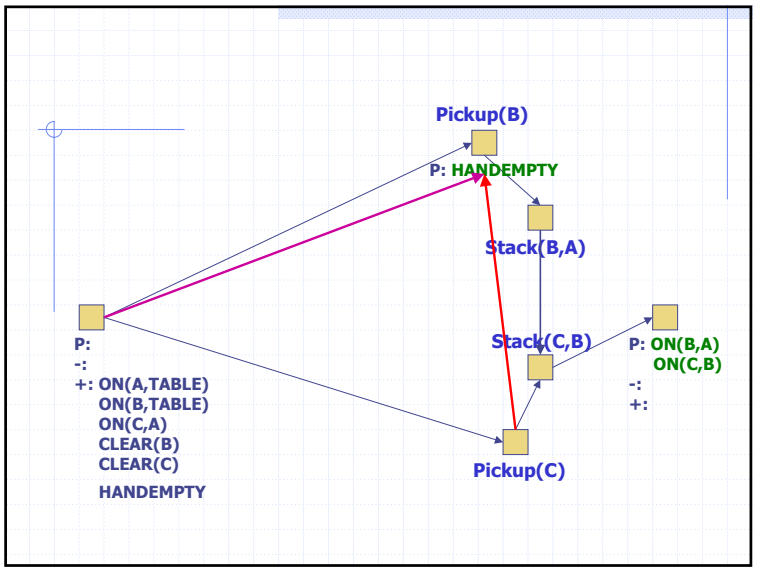

P: HOLDING(B)
CLEAR(A)
-: CLEAR(A)
HOLDING(B)
+: ON(B,A)
CLEAR(B)
HANDEMPTY
Stack(B,A)
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY
Stack(C,B)
P: HOLDING(C)
CLEAR(B)
-: CLEAR(B)
HOLDING(C)
+: ON(C,B)
CLEAR(C)
HANDEMPTY
P: ON(B,A)
ON(C,B)
-:
+:

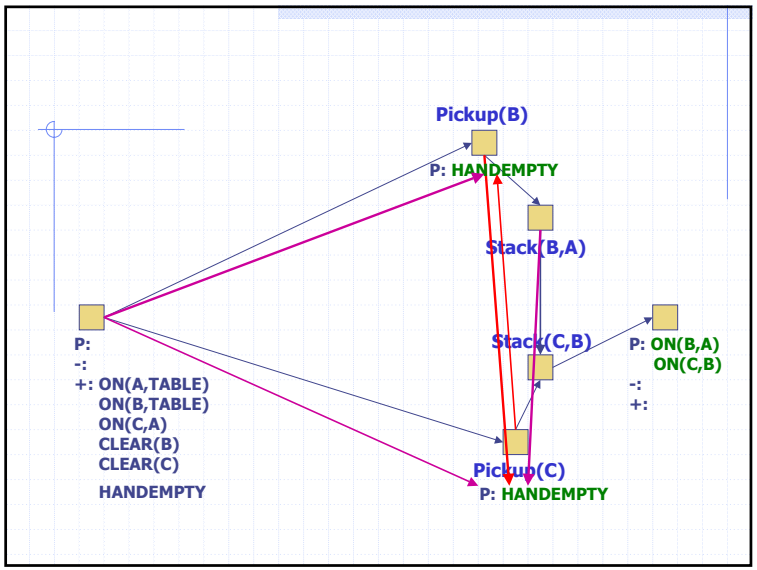

Nonlinear planning
searches a plan space
Stack(B,A)
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY
Pickup(C)
P: ON(B,A)
ON(C,B)
-:
+:

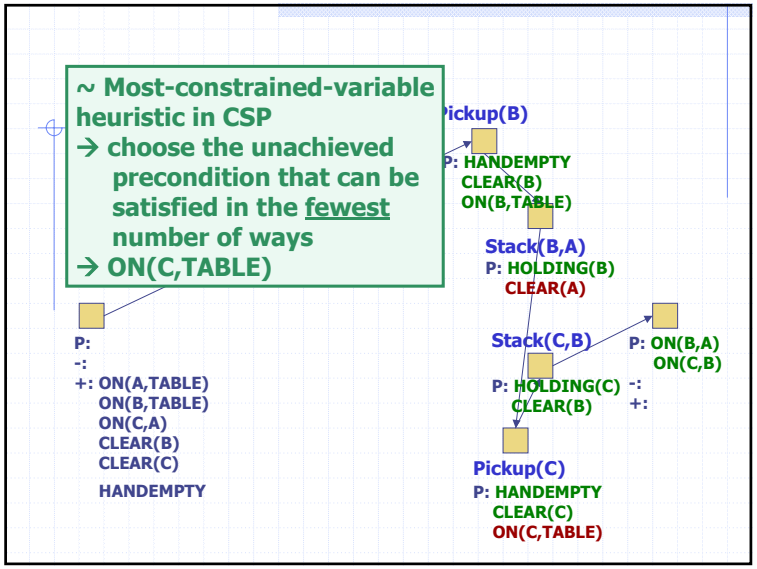

Other possible
achiever
Pickup(B)
Stack(B,A)
Achievers
Threat
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY
Stack(C,B)
P: CLEAR(B)
Pickup(C)
P: ON(B,A)
ON(C,B)
-:
+:

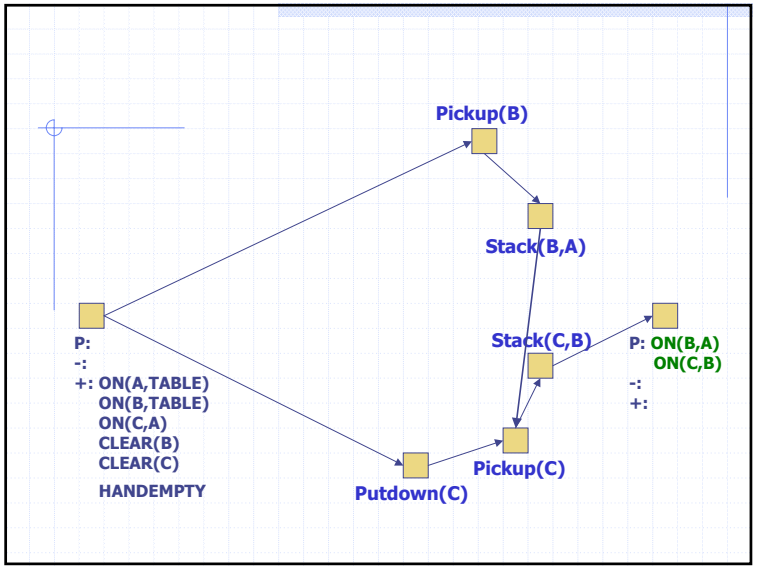

Pickup(B)
P: CLEAR(B)
Stack(B,A)
Stack(C,B)
P: CLEAR(B)
P: ON(B,A)
ON(C,B)
-:
+:
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY
Pickup(C)

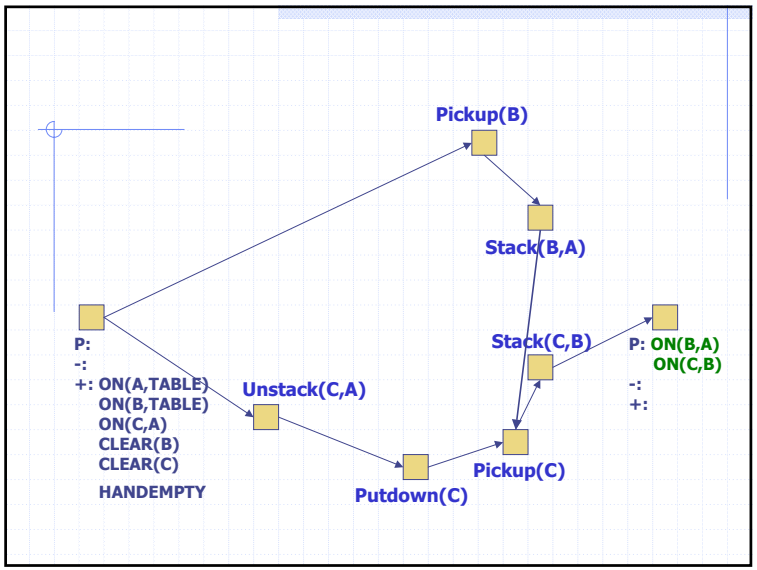

A consistent plan is one in which there is no cycle and no conflict between achievers and threats
A conflict can be eliminated by constraining the ordering among the actions or by adding new actions
Note the similarity with constraint propagation
P: ON(B,A)
ON(C,B)
-:
+:
HANDEMPTY

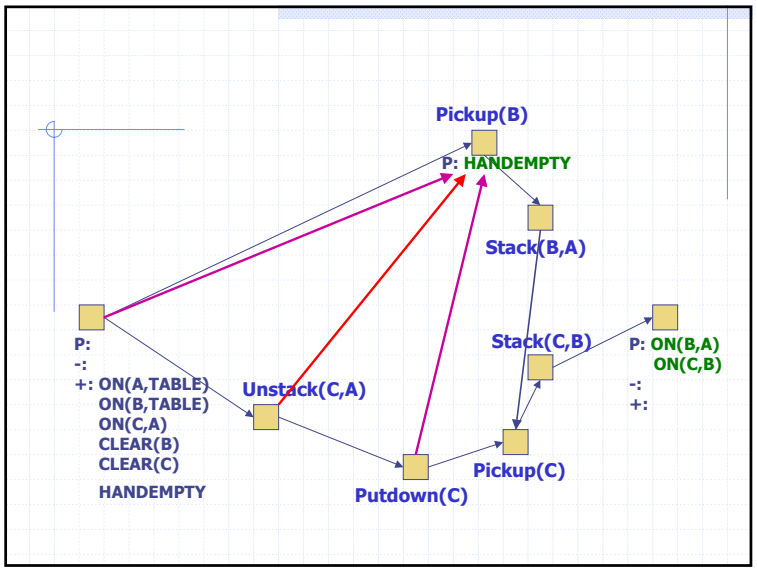

Pickup(B)
P: HANDEMPTY
Stack(B,A)
Stack(C,B)
P: ON(B,A)
ON(C,B)
-:
+:
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY
Pickup(C)

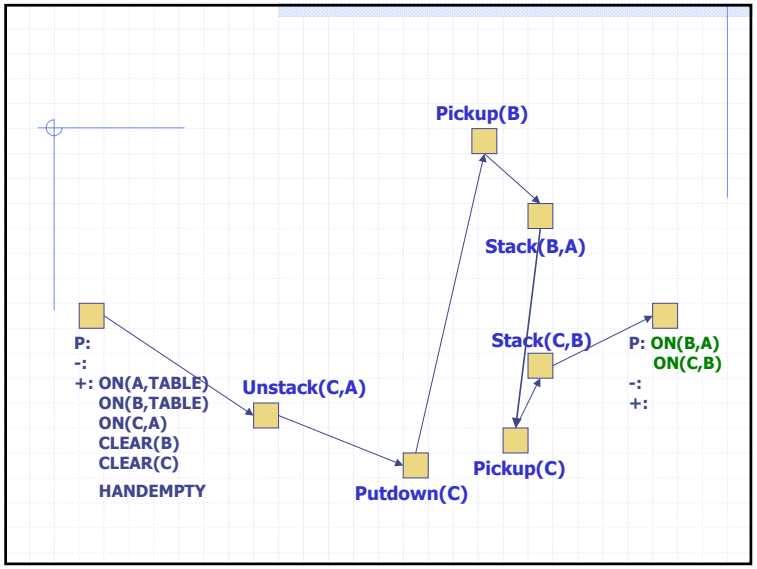

Pickup(B)
P: HANDEMPTY
Stack(B,A)
Stack(C,B)
P: ON(B,A)
ON(C,B)
-:
+:
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY
Pickup(C)
P: HANDEMPTY

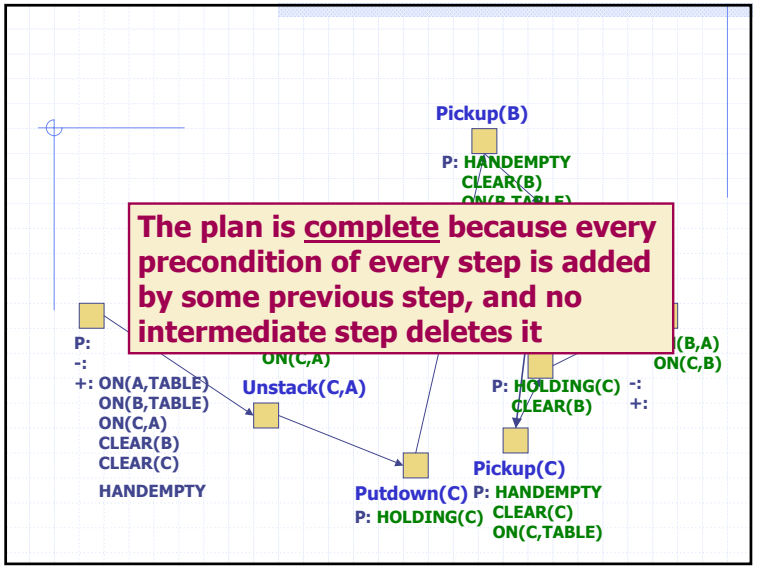

~ Most-constrained-variable heuristic in CSP
→ choose the unachieved precondition that can be satisfied in the fewest number of ways
→ ON(C,TABLE)
Pickup(B)
P: HANDEMPTY
CLEAR(B)
ON(B,TABLE)
Stack(B,A)
P: HOLDING(B)
CLEAR(A)
Stack(C,B)
P: HOLDING(C)
CLEAR(B)
Pickup(C)
P: HANDEMPTY
CLEAR(C)
ON(C,TABLE)
P: ON(B,A)
ON(C,B)
-:
+:
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY


Pickup(B)
Stack(B,A)
Stack(C,B)
Pickup(C)
Putdown(C)
P: ON(B,A)
ON(C,B)
-:
+:
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY


Pickup(B)
Stack(B,A)
Stack(C,B)
Unstack(C,A)
Pickup(C)
Putdown(C)
P: ON(B,A)
ON(C,B)
-:
+:
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY


Pickup(B)
P: HANDEMPTY
Stack(B,A)
Stack(C,B)
Unstack(C,A)
Pickup(C)
Putdown(C)
P: ON(B,A)
ON(C,B)
-:
+:
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY





More Planning Algorithms

- GraphPlan
- Hierarchical Planning

Planning graph

- Directed, leveled graph with alternating layers of nodes
- Odd layers represent candidate propositions that might hold at time I
- Even layers represent candidate actions that we might possibly execute at time I, including maintain actions
- We can only execute one real action at any step, but the data structure keeps track of which actions are possibilities

Basic idea

- Construct a graph that encodes constraints on possible plans
- Use this “planning graph” to constrain search for a valid plan
- Planning graph can be built for each problem in relatively short time

Problem handled by GraphPlan

- STRIPS operators: conjunctive preconditions, no conditional or universal effects, no negations
- Finds “shortest” plans (by some definition)
- Sound, complete and will terminate with failure if there is no plan.

Planning graph

- Nodes divided into “levels”, arcs go from one level to the next
- For each time period, a state level and an action level
- Arcs represent preconditions, adds and deletes

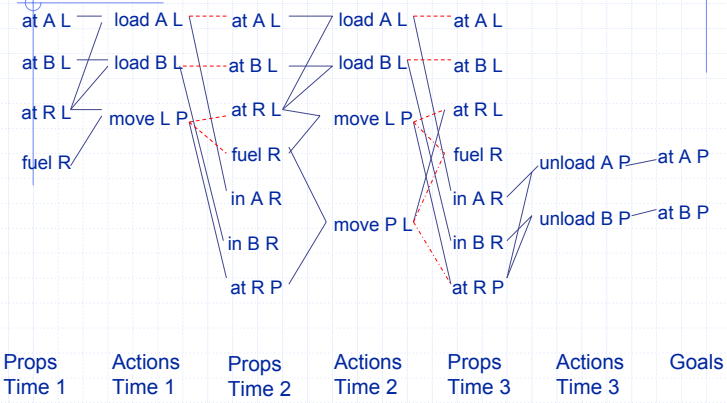
What actions and what literals?

- Add an action in level Ai if all its preconditions are present in level Pi
- Add a precondition in level Pi if it is the effect of some action in level Ai-1 (*including no-ops*)
- Level P1 has all the literals from the initial state

Simple NASA domain, ☺

- Literals:
  - at X Y — X is at location Y
  - fuel R — rocket has fuel
  - in X R — X is in the rocket
- Actions:
  - load X L — load X onto rocket at location L (X and R must be at L)
  - unload X L — unload X from rocket at location L (R must be at L, X must be in L)
  - move X Y — move rocket from X to Y (R must be at L and have fuel)
- Graph representation:
  - Solid black lines: preconditions/effects
  - Dotted red lines: negated preconditions/effects

Example planning graph

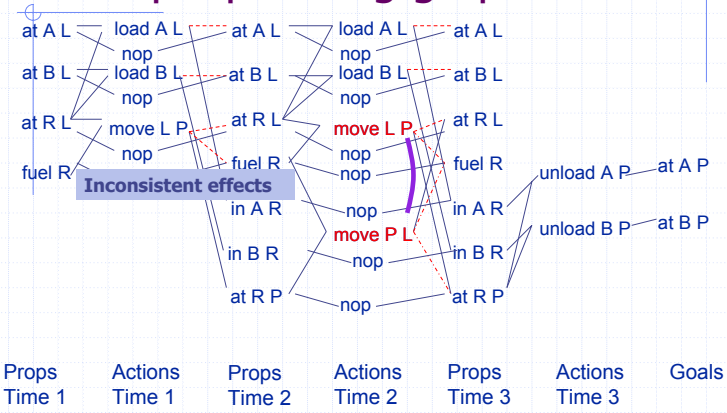


Not all actions at possible at time 2 and 3 are shown

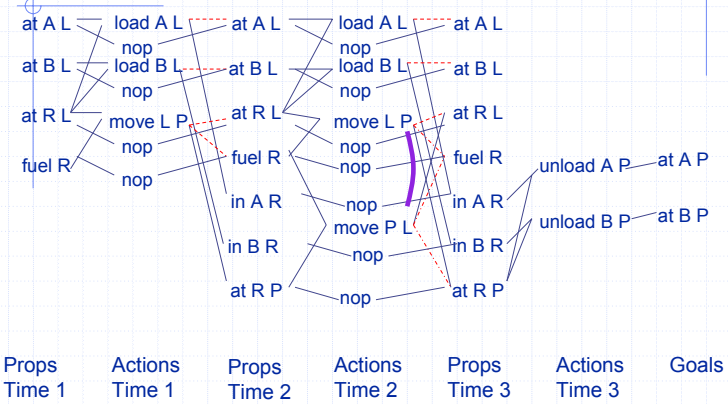
Exclusion relations (mutexes)

- Two actions (or literals) are mutually exclusive at some stage if no valid plan could contain both.
- Can quickly find and mark *some* mutexes:
  - **Inconsistent Effects**: one action negates an effect of the other
  - **Interference**: One of the effects of one action is the negation of the precondition of the other
  - **Competing needs**: If some precond of one action is mutex with a precond of the other action
  - **Inconsistent support**: Two propositions are mutex if all ways of creating them both are mutex

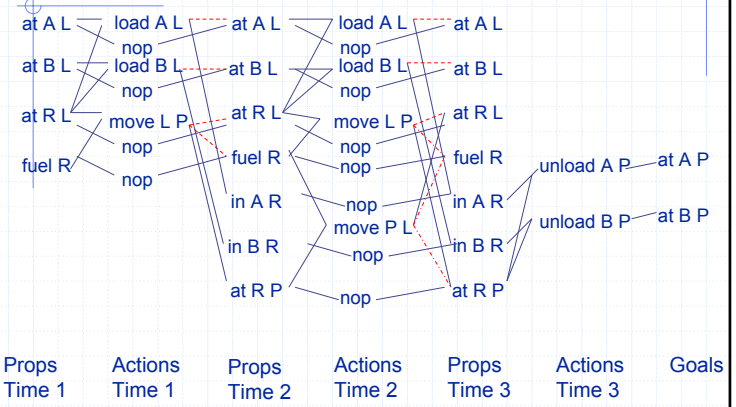
Example planning graph



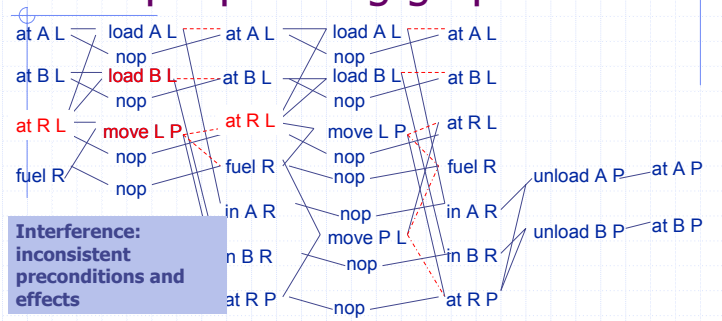
Example planning graph



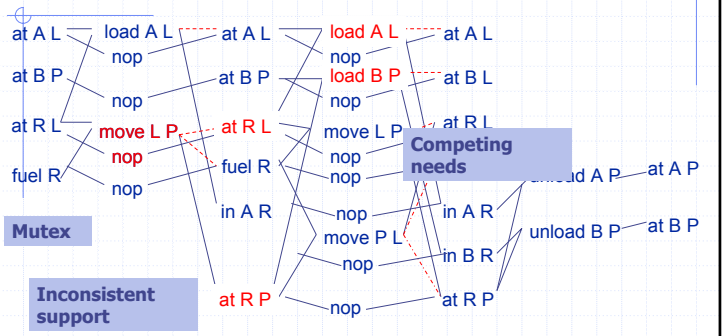
Example planning graph



Example planning graph



A slightly different planning graph...



Mutex: Summary

- Actions A,B are ***mutex at level i*** if no valid plan could possibly contain both at i:
  - **inconsistent effects**: A's effects negate one of B's effects
  - **interference:** A's effects delete one of B's preconditions, or vice-versa
  - **competing needs**: A & B have mutex preconditions
- Propositions P,Q are ***mutex at level i*** if no valid plan could possibly contain both at i
  - **inconsistent support:** If at i, all ways to achieve P exclude all ways to achieve Q

GraphPlan algorithm (without termination)

- Grow the planning graph (PG) until all goals are reachable and not mutex. (If PG levels off first, fail)
- Search the PG for a valid plan
- If none found, add a level to the PG and try again

Valid plans

- A valid plan is a planning graph where:
- Actions at the same level don't interfere (delete each other's preconditions or add effects)
- Each action's preconditions are made true by the plan
- Goals are satisfied

Extending the planning graph

- Action level: For each possible instantiation of each operator (including no-ops), add if all of its preconditions are in the previous level and no two are mutex.
- Proposition level: Add all effects of actions in previous level (including no-ops).
  Mark pairs of propositions mutex if all ways to create one are mutex of all ways to create the other

Creating the planning graph is usually fast

- Theorem 1:
  The size of the t-level PG and the time to create it are polynomial in t, n (= number of objects),
  m (= number of operators) and p (= propositions in the initial state)

Searching for a plan

- Backward chain on the planning graph
- Complete all goals at one level before going back:
- At each level, pick a subset of actions to achieve the goals that are not mutex. Their preconditions become the goals at the earlier level.
- Build subset by picking each goal and choosing an action to add. Use one already selected if possible. Do forward checking on remaining goals.

Summary: GraphPlan

- Generate all actions at level 2i+1 that have all of their preconditions at level 2i and no two are mutex
- Mark as mutex all action-maintain pairs that are incompatible
- Mark as mutex all action-action pairs that have competing needs
- Generate all propostions at level 2i+2 that are the effect of some action at level 2i+1
- Mark as mutex all pairs of propostions that can only be generated by mutex action pairs

SATplan

- Formulate the planning problem as a CSP
- Assume that the plan has k actions
- Create a binary variable for each possible action a:
  - Action(a,i) (TRUE if action a is used at step i)
- Create variables for each proposition that can hold at different points in time:
  - Proposition(p,i) (TRUE if proposition p holds at step i)

Constraints

- Only one action can be executed at each time step (XOR constraints)
- Constraints describing effects of actions
- Persistence: if an action does not change a proposition p, then p's value remains unchanged
- A proposition is true at step i only if some action (possibly a maintain action) made it true
- Constraints for initial state and goal state

Now apply our favorite CSP solver!

Applications of Planning

- Military operations

Most applied systems use extended representation languages, nonlinear planning techniques, and domain-specific heuristics

- Command sequences for satellite

Summary: classical planning

- Representations in planning
- Representation of action:
  preconditions + effects
- Approaches:
  - State-space search
  - Plan-space search
  - Constraint-based search