Planning (some slides from Tom Lenaerts)

- The Planning problem
- Planning with State-space search
- Partial-order planning
- Planning graphs
- Planning with propositional logic
- Analysis of planning approaches

What we have so far

- Can TELL KB about new percepts about the world
- KB maintains model of the current world state
- Can ASK KB about any fact that can be inferred from KB

How can we use these components to build a planning agent,

i.e., an agent that constructs plans that can achieve its goals, and that then executes these plans?

Remember: Problem-Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT( $p$) returns an action
    inputs: $p$, a percept
    static: $s$, an action sequence, initially empty
            $state$, some description of the current world state
            $g$, a goal, initially null
            $problem$, a problem formulation

    $state$ ← UPDATE-STATE($state$, $p$)
    if $s$ is empty then
        $g$ ← FORMULATE-GOAL($state$)
        $problem$ ← FORMULATE-PROBLEM($state$, $g$)
        $s$ ← SEARCH( $problem$)
    $action$ ← RECOMMENDATION($s$, $state$)
    $s$ ← REMAINDER($s$, $state$)
    return $action$
```

This is *offline* problem-solving – requires solution before any move.

In the real world, there is a penalty for doing nothing!!

Online problem-solving involves acting w/o complete knowledge of the problem and environment

What is Planning

- Generate sequences of actions to perform tasks and achieve objectives.
  - States, actions and goals
- Search for solution over abstract space of plans.
- Assists humans in practical applications
  - design and manufacturing
  - military operations
  - games
  - space exploration

Difficulty of real world problems

- Assume a problem-solving agent using some search method …
  - Which actions are relevant?
    - Exhaustive search vs. backward search
  - What is a good heuristic function?
    - Good estimate of the cost of the state?
    - Problem-dependent vs Problem-independent
  - How to decompose the problem?
    - Most real-world problems are ***nearly*** decomposable.

Plan

We formally define a plan as a data structure consisting of:

- Set of plan steps (each is an operator for the problem)
- Set of step ordering constraints

  e.g., A Ⅱ B means "A must be done before B"

- Set of variable binding constraints

  e.g., v = x where v variable and x constant or other variable

- Set of causal links

  e.g., $A \xrightarrow{c} B$ means "A achieves c for B"
  A makes "c" true, essentially enabling B (A is a requirement for B)

Simple planning agent

- Use percepts to build model of current world state
- IDEAL-PLANNER: Given a goal, algorithm generates plan of action
- STATE-DESCRIPTION: given percept, return initial state description in format required by planner
- MAKE-GOAL-QUERY: used to ask KB what next goal should be

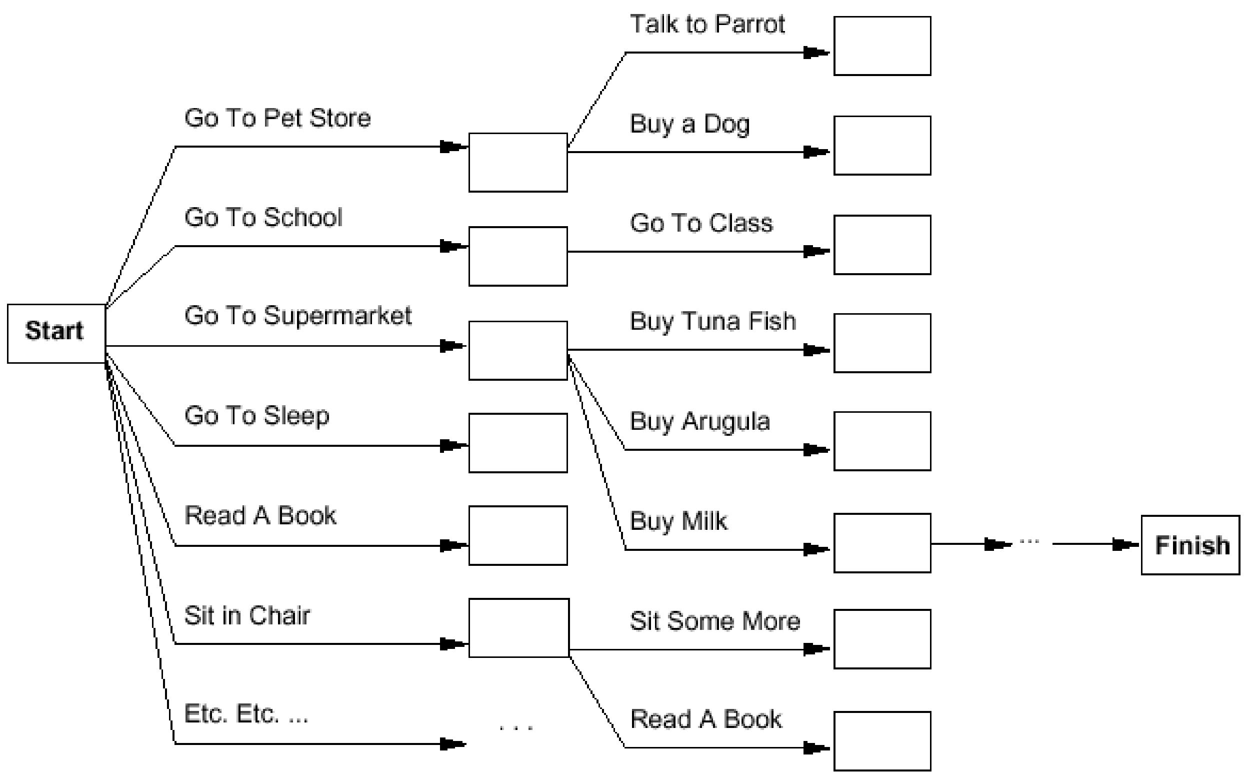
A Simple Planning Agent

```
function SIMPLE-PLANNING-AGENT(percept) returns an action
    static:           KB, a knowledge base (includes action descriptions)
                      p, a plan (initially, NoPlan)
                      t, a time counter (initially 0)
    local variables:  G, a goal
                      current, a current state description
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    current ← STATE-DESCRIPTION(KB, t)
    if p = NoPlan then
        G ← ASK(KB, MAKE-GOAL-QUERY(t))
        p ← IDEAL-PLANNER(current, G, KB)
    if p = NoPlan or p is empty then
        action ← NoOp
    else
        action ← FIRST(p)
        p ← REST(p)                 Like popping from a stack
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t ← t+1
    return action
```

Search vs. planning

Consider the task *get milk, bananas, and a cordless drill*

Standard search algorithms seem to fail miserably:



After-the-fact heuristic/goal test inadequate

Search vs. planning

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

Planning in situation calculus

$PlanResult(p, s)$ is the situation resulting from executing p in s

$$PlanResult([], s) = s$$
$$PlanResult([a|p], s) = PlanResult(p, Result(a, s))$$

Initial state $At(Home, S_0) \wedge \neg Have(Milk, S_0) \wedge \ldots$

Actions as Successor State axioms

$Have(Milk, Result(a, s)) \Leftrightarrow$
$[(a = Buy(Milk) \wedge At(Supermarket, s)) \vee (Have(Milk, s) \wedge a \neq \ldots)]$

Query

$$s = PlanResult(p, S_0) \wedge At(Home, s) \wedge Have(Milk, s) \wedge \ldots$$

Solution

$$p = [Go(Supermarket), Buy(Milk), Buy(Bananas), Go(HWS), \ldots]$$

Principal difficulty: unconstrained branching, hard to apply heuristics

Types of planners

- Situation space planner: search through possible situations

- Progression planner: start with initial state, apply operators until goal is reached (Forward Chaining)

  Problem: high branching factor!

- Regression planner: start from goal state and apply operators until start state reached (Backward Chaining)

  Why desirable? usually many more operators are applicable to initial state than to goal state.

  Difficulty: when want to achieve a conjunction of goals

Initial STRIPS algorithm: situation-space regression planner

State space vs. plan space

Standard search: node = concrete world state

Planning search: node = partial plan

Search space of plans rather than of states.

Defn: open condition is a precondition of a step not yet fulfilled

Operators on partial plans:

- add a link from an existing action to an open condition
- add a step to fulfill an open condition
- order one step wrt another

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

Operations on plans

- Refinement operators: add constraints to partial plan

- Modification operator: every other operators

Types of planners

- Partial order planner: some steps are ordered, some are not
- Total order planner: all steps ordered (thus, plan is a simple list of steps)
- Linearization: process of deriving a totally ordered plan from a partially ordered plan.

Planning languages

- What is a good language?
  - Expressive enough to describe a wide variety of problems.
  - Restrictive enough to allow efficient algorithms to operate on it.
  - Planning algorithm should be able to take advantage of the logical structure of the problem.
- STRIPS (**ST**anford **R**esearch **I**nstitute **P**roblem **S**olver) and ADL (**A**ction **D**escription **L**anguage)

Basic representation for planning

- Most widely used approach: uses STRIPS language

- states: conjunctions of function-free ground literals (I.e., predicates applied to constant symbols. Some languages allow negated literals); e.g.,

  $At(Home) \wedge Have(Milk) \wedge Have(Money)$

  $At(Home) \wedge \neg Have(Milk) \wedge \neg Have(Bananas) \wedge \neg Have(Drill) \ldots$

- goals: also conjunctions of literals; e.g.,

  $At(Home) \wedge Have(Milk) \wedge Have(Bananas) \wedge Have(Drill)$

  but some languages also allow variables (implicitly universally quant.); e.g.,

  $At(x) \wedge Sells(x, Milk)$

Planner vs. theorem prover

- Planner: ask for sequence of actions that makes goal true if executed

- Theorem prover: ask whether query sentence is true given KB

STRIPS operators

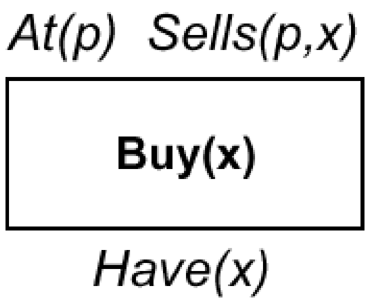
Tidily arranged actions descriptions, restricted language

Action: $Buy(x)$
Precondition: $At(p), Sells(p, x)$
Effect: $Have(x)$

[Note: this abstracts away many important details!]

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

Graphical notation:



General language features – STRIPS

- Representation of states
  - Decompose the world in logical conditions and represent a state as a *conjunction of positive literals.*
    - Propositional literals: *Poor* ∧ *Unknown*
    - FO-literals (grounded and function-free): *At(Plane1, Melbourne)* ∧ *At(Plane2, Sydney)*
  - Closed world assumption
- Representation of goals
  - Partially specified state and represented as a *conjunction of positive ground literals*
  - A goal is *satisfied* if the state contains all literals in goal.

General language features

- Representations of actions
  - Action = PRECOND + EFFECT

    Action(Fly(p,from, to),

    PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)

    EFFECT: ¬At(p,from) ∧ At(p,to))

  = action schema (p, from, to need to be instantiated)
    - Action name and parameter list
    - Precondition (conjunction of **function-free** literals)
    - Effect (conjunction of **function-free** literals and P is True and not P is False)
  - Add-list (predicates that are now true) vs delete-list (predicates that are now false) in the "Effect"

Language semantics

- How do actions affect states?
  - An action is applicable in any state that satisfies the precondition.
  - For FO action schema applicability involves a substitution θ for the variables in the PRECOND.

At(P1,JFK) $\wedge$ *At(P2,SFO)* $\wedge$ *Plane(P1)* $\wedge$ *Plane(P2)* $\wedge$ *Airport(JFK)* $\wedge$ *Airport(SFO)*

Satisfies *: At(p,from)* $\wedge$ *Plane(p)* $\wedge$ *Airport(from)* $\wedge$ *Airport(to)*

With θ *={p/P1,from/JFK,to/SFO}*

Thus the action is applicable.

Language semantics

- The result of executing action a in state s is the state s'
  - s' is same as s except
    - Any positive literal *P* in the effect of *a* is added to *s'*
    - Any negative literal *¬P* is removed from *s'*

At(P1,SFO) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)

  - STRIPS assumption: (avoids representational frame problem)

    Frame Problem → need to define a whole lot of rules that specify the things that remain the same.

    STRIPS assumes that every literal NOT mentioned explicitly in the effect remains unchanged

Expressiveness and extensions

- STRIPS is simplified
  - Important limit: function-free literals
  - Allows for propositional representation
- Function symbols lead to infinitely many states and actions
- Recent extension:Action Description language (ADL)

Action(Fly(p:Plane, from: Airport, to: Airport),
PRECOND: At(p,from) $\wedge$ *(from* $\neq$ *to)*
EFFECT: $\neg$*At(p,from)* $\wedge$ *At(p,to))*

Standardization of Planning Languages *: Planning domain definition language (PDDL)*

Differences Between STRIPS and ADL

STRIPS

- Only positive literals in the states
- Closed world assumption – unmentioned literals are false
- Effect $P \wedge \neg Q$ means add P and remove Q
- Only ground literals in goals
- Goals allow only conjunctions; disjunctions are not allowed
- Equality is not supported
- No support for types
- Effects are conjunctions

ADL

- Both positive and negative literals
- Open World Assumption – unmentioned literals are unknown
- Effect $P \wedge \neg Q$ means add P and $\neg$ Q; and remove $\neg$ P and Q
- Quantified variables in goals
- Goals allow conjunctions and disjunctions
- Equality built in
- Variables are typed
- Conditional effects are allowed; e.g., when P; E means E is an effect if and only if P is satisfied

Example: air cargo transport

Init (At(C1, SFO) ∧ At(C2,JFK) ∧ At(P1,SFO) ∧ At(P2,JFK) ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO))

Goal (At(C1,JFK) ∧ At(C2,SFO))

Action (Load(c,p,a)

PRECOND: *At(c,a) ∧At(p,a) ∧Cargo(c) ∧Plane(p) ∧Airport(a)*

EFFECT: *¬At(c,a) ∧In(c,p))*

Action (Unload(c,p,a)

PRECOND: *In(c,p) ∧At(p,a) ∧Cargo(c) ∧Plane(p) ∧Airport(a)*

EFFECT: *At(c,a) ∧ ¬In(c,p))*

Action (Fly(p,from,to)

PRECOND: *At(p,from) ∧Plane(p) ∧Airport(from) ∧Airport(to)*

EFFECT: *¬ At(p,from) ∧ At(p,to))*

[Load(C1,P1,SFO), Fly(P1,SFO,JFK), Load(C2,P2,JFK), Fly(P2,JFK,SFO)]

Example: Spare tire problem

Init(At(Flat, Axle) ∧ At(Spare,trunk))

Goal(At(Spare,Axle))

Action(Remove(Spare,Trunk)

PRECOND: *At(Spare,Trunk)*

EFFECT: *¬At(Spare,Trunk) ∧ At(Spare,Ground))*

Action(Remove(Flat,Axle)

PRECOND: *At(Flat,Axle)*

EFFECT: *¬At(Flat,Axle) ∧ At(Flat,Ground))*

Action(PutOn(Spare,Axle)

PRECOND: *At(Spare,Ground) ∧¬At(Flat,Axle)*

EFFECT: *At(Spare,Axle) ∧ ¬At(Spare,Ground))*

Action(LeaveOvernight

PRECOND:

EFFECT: *¬ At(Spare,Ground) ∧ ¬ At(Spare,Axle) ∧ ¬ At(Spare,trunk) ∧ ¬ At(Flat,Ground) ∧ ¬ At(Flat,Axle))*

This example goes beyond STRIPS: negative literal in pre-condition (ADL description)

Example: Blocks world

Init(On(A, Table) $\wedge$ *On(B,Table)* $\wedge$ *On(C,Table)* $\wedge$ *Block(A)* $\wedge$ *Block(B)* $\wedge$
Block(C) $\wedge$ *Clear(A)* $\wedge$ *Clear(B)* $\wedge$ *Clear(C))*

Goal(On(A,B) $\wedge$ *On(B,C))*

// Move Block b from top of Block x to Block y
Action(Move(b,x,y)
PRECOND: *On(b,x)* $\wedge$ *Clear(b)* $\wedge$ *Clear(y)* $\wedge$ *Block(b)* $\wedge$ *(b*$\neq$ *x)* $\wedge$ *(b*$\neq$ *y)* $\wedge$ *(x*$\neq$ *y)*
EFFECT: *On(b,y)* $\wedge$ *Clear(x)* $\wedge$ $\neg$ *On(b,x)* $\wedge$ $\neg$ *Clear(y))*

// Move Block b from top of Block x to the table
Action(MoveToTable(b,x)
PRECOND: *On(b,x)* $\wedge$ *Clear(b)* $\wedge$ *Block(b)* $\wedge$ *(b*$\neq$ *x)*
EFFECT: *On(b,Table)* $\wedge$ *Clear(x)* $\wedge$ $\neg$ *On(b,x))*

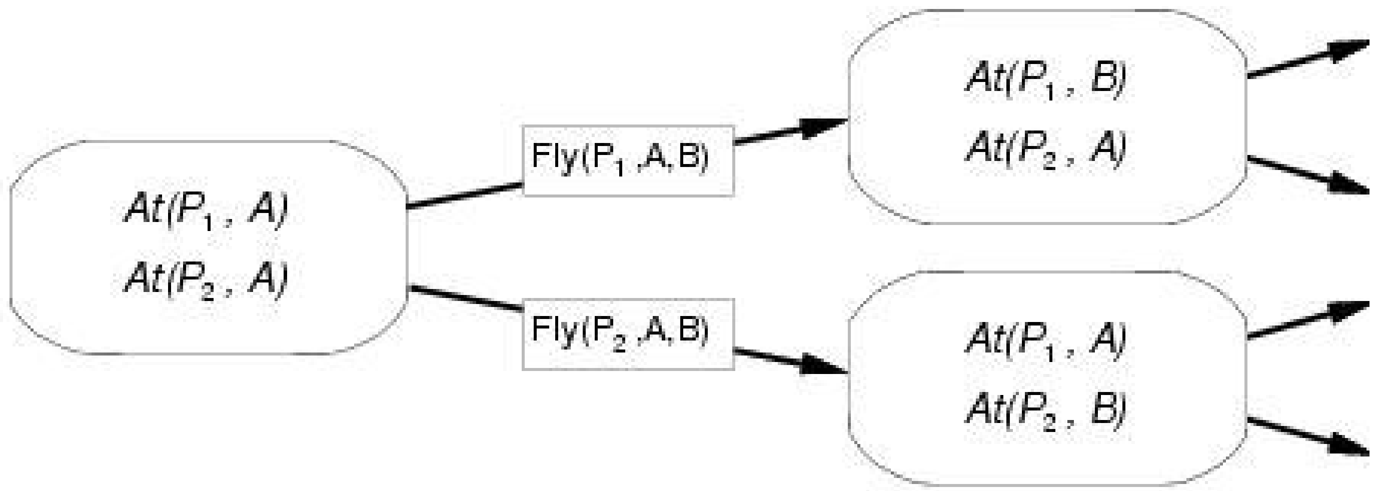
Spurious actions are possible: Move(B,C,C) – prevented by appropriate equality restrictions as above

Planning with state-space search

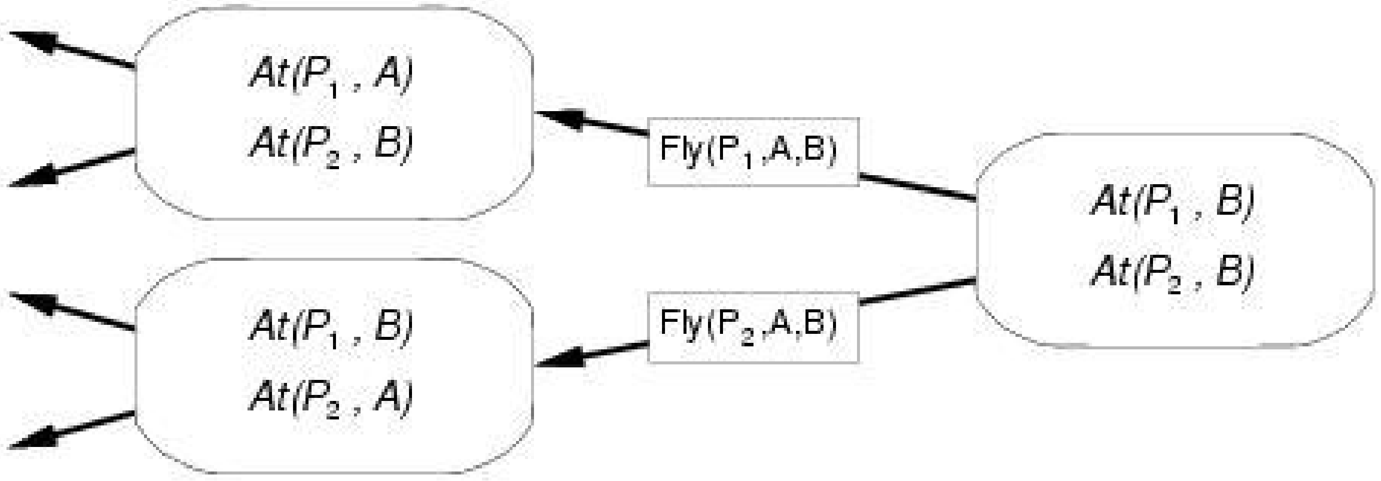
- Both forward and backward search possible
- Progression planners
  - forward state-space search
  - Consider the effect of all possible actions in a given state
- Regression planners
  - backward state-space search
  - To achieve a goal, what must have been true in the previous state.

Progression and regression – Cargo Problem

(a)



(b)



Progression algorithm

- Formulation as state-space search problem:
  - Initial state = initial state of the planning problem
    - Literals not appearing are false
  - Actions = those whose preconditions are satisfied
    - Add positive effects, delete negative
  - Goal test = does the state satisfy the goal
  - Step cost = each action costs 1
- No functions … any graph search that is complete is a complete planning algorithm.
- Inefficient: (1) irrelevant action problem (2) good heuristic required for efficient search

Regression algorithm

- How to determine predecessors?
  - What are the states from which applying a given action leads to the goal?

    Goal state = $At(C1, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$
    Relevant action for first conjunct: *Unload(C1,p,B)*
    Works only if pre-conditions are satisfied.
    Previous state= $In(C1, p) \wedge At(p, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$
    Subgoal At(C1,B) should not be present in this state.

- Actions must not undo desired literals (consistent)
- Main advantage: only relevant actions are considered.
  - Often much lower branching factor than forward search.

Regression algorithm

- General process for predecessor construction
  - Give a goal description G
  - Let A be an action that is relevant and consistent
  - The predecessors is as follows:
    - *Any positive effects of A that appear in G are deleted.*
    - *Each precondition literal of A is added , unless it already appears.*
- Any standard search algorithm can be added to perform the search.
- Termination when predecessor satisfied by initial state.
  - In FOL case, satisfaction might require a substitution.

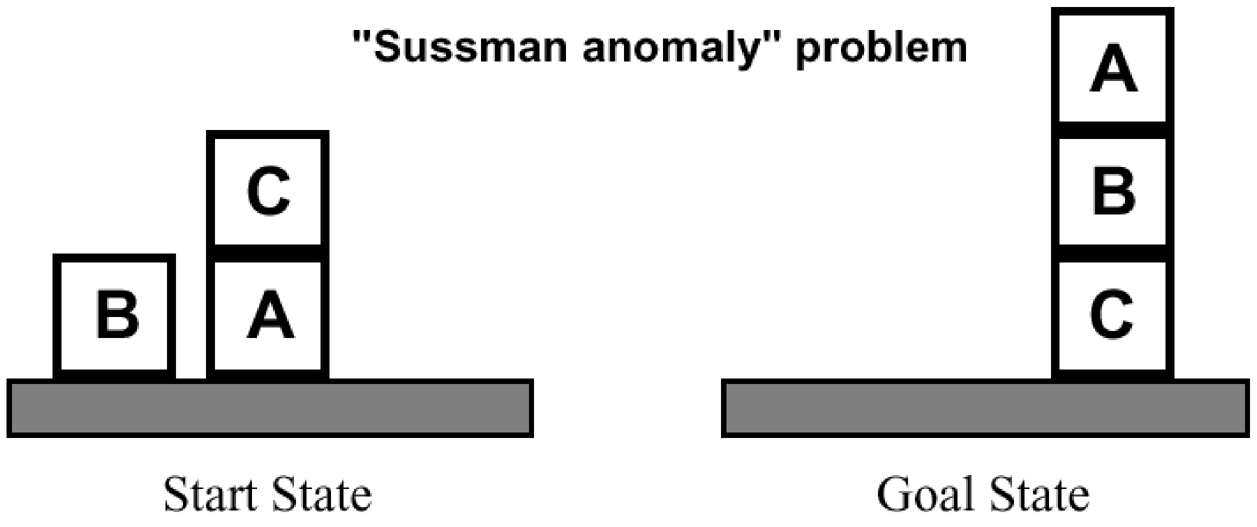
Heuristics for state-space search

- Neither progression or regression are very efficient without a good heuristic.
  - How many actions are needed to achieve the goal?
  - Exact solution is NP hard, find a good estimate
- Two approaches to find admissible heuristic:
  - The optimal solution to the relaxed problem.
    - *Remove all preconditions from actions*
  - The subgoal independence assumption:

    The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving the subproblems independently.

Example: block world

"Sussman anomaly" problem



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

$\boxed{\text{PutOn(x,y)}}$

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

Clear(x) On(x,z)

$\boxed{\text{PutOnTable(x)}}$

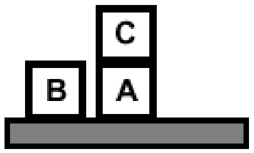
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Example (cont.)

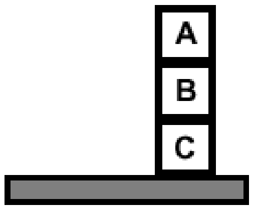
START

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

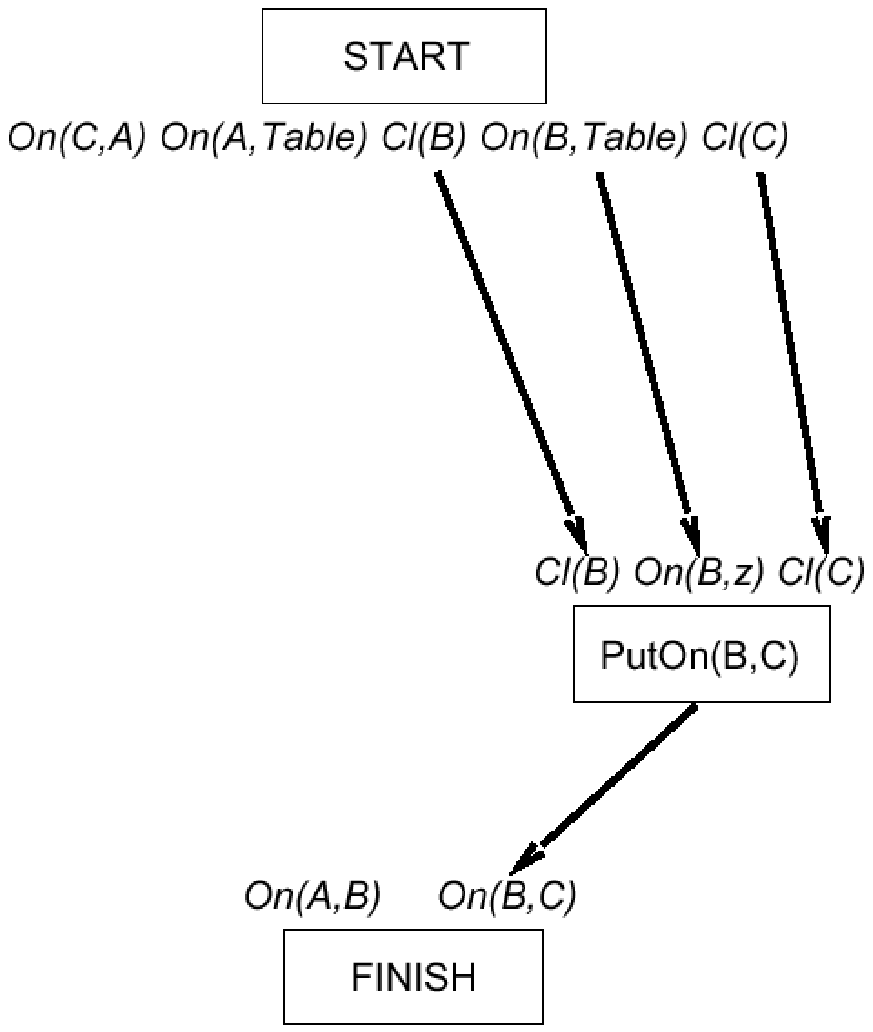


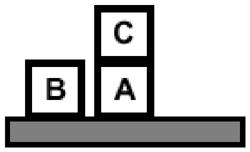
On(A,B) *On(B,C)*

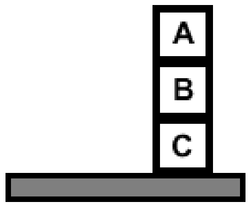
FINISH



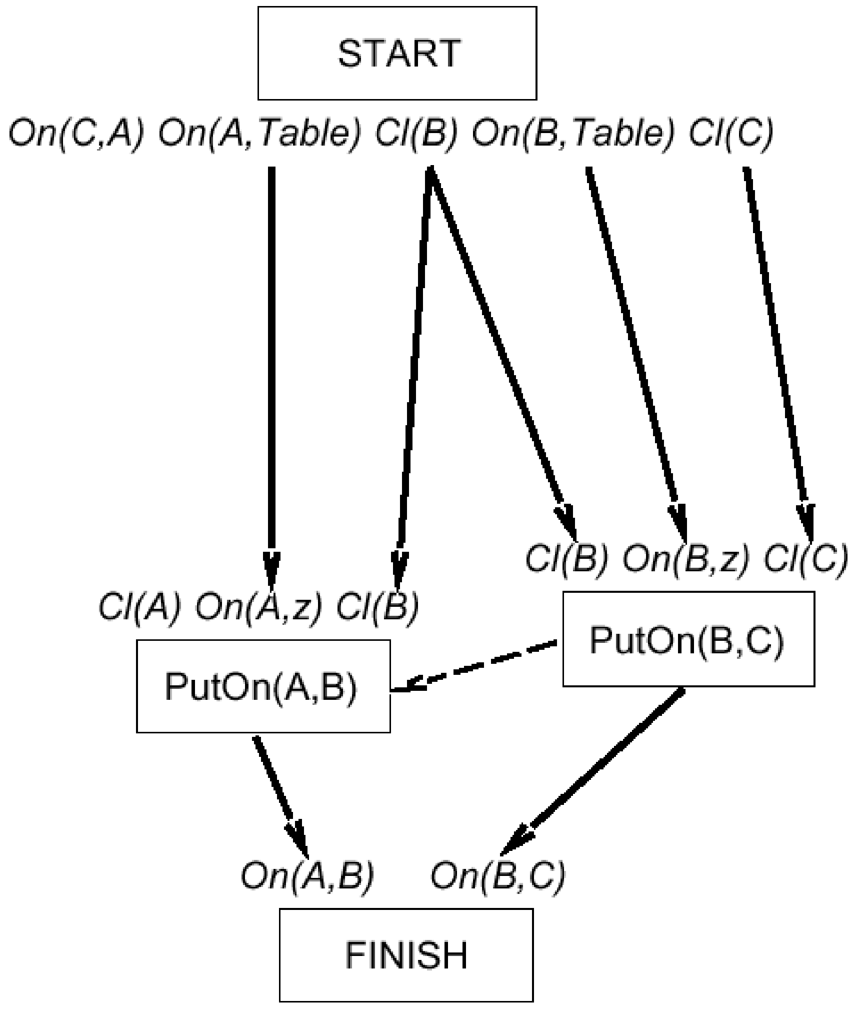
Example (cont.)

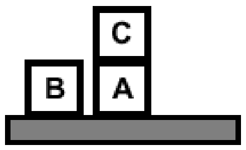




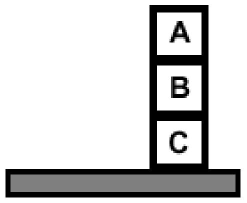


Example (cont.)

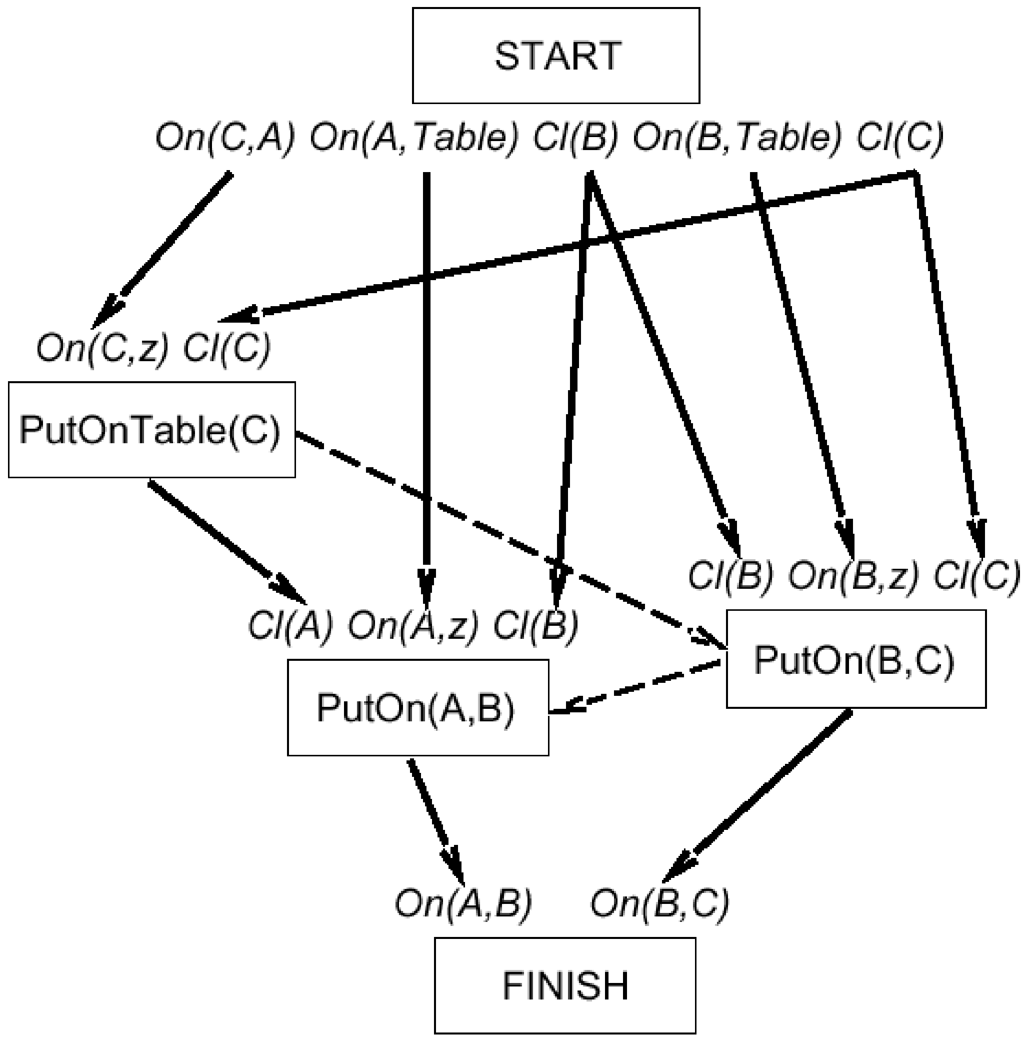


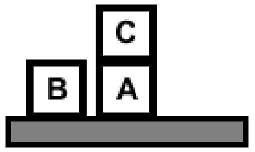


PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)



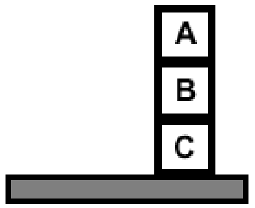
Example (cont.)





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

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



Conclusion from the Blocks Example

- Problem can be solved, BUT not by trying to apply ALL operators to achieve a single goal at a time sequentially – satisfying one goal seems to clobber earlier achieved goals.
- The issue: we are forcing an order on operators when they do not need to be mutually ordered.
- We need an approach that allows INTERLEAVING of steps for multiple goals
- This observation motivates the next planning approach: PARTIAL ORDER PLANNING - to be covered next class…