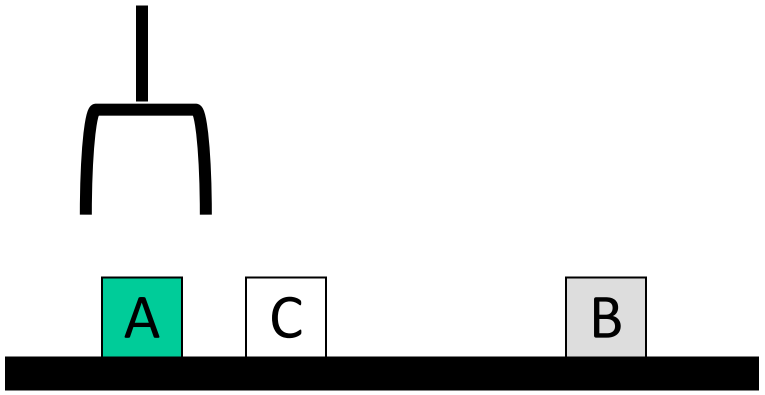
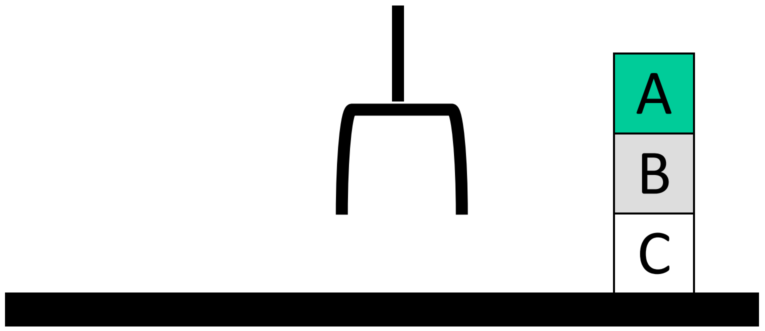
Planning

Chapter 11.1-11.3

Some material adopted from notes
by Andreas Geyer-Schulz
and Chuck Dyer

Blocks World Planning





Blocks world

The blocks world is a micro-world consisting of a table, a set of blocks and a robot hand

Some constraints for a simple model:

- 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 uses a logic notation:

ontable(b) ontable(d)

on(c,d)   holding(a)

clear(b)   clear(c)

Typical BW planning problem

Initial state:

clear(a)

clear(b)

clear(c)

ontable(a)

ontable(b)

ontable(c)

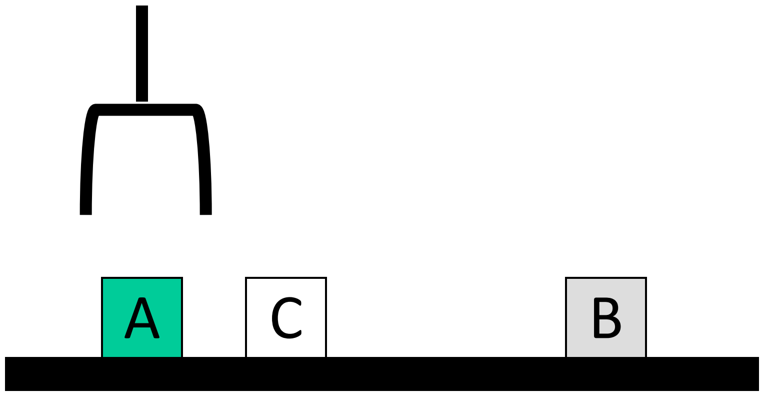
handempty

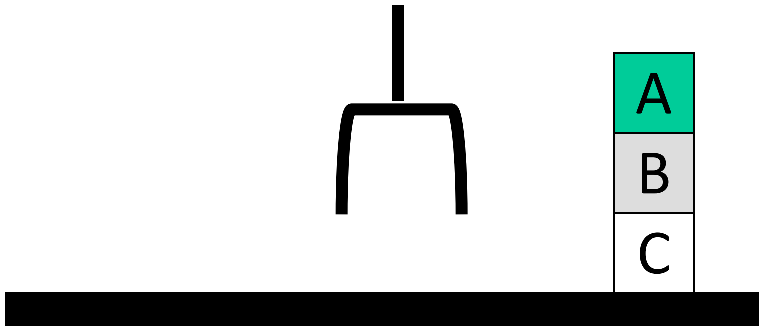
Goal:

on(b,c)

on(a,b)

ontable(c)





Typical BW planning problem

Initial state:

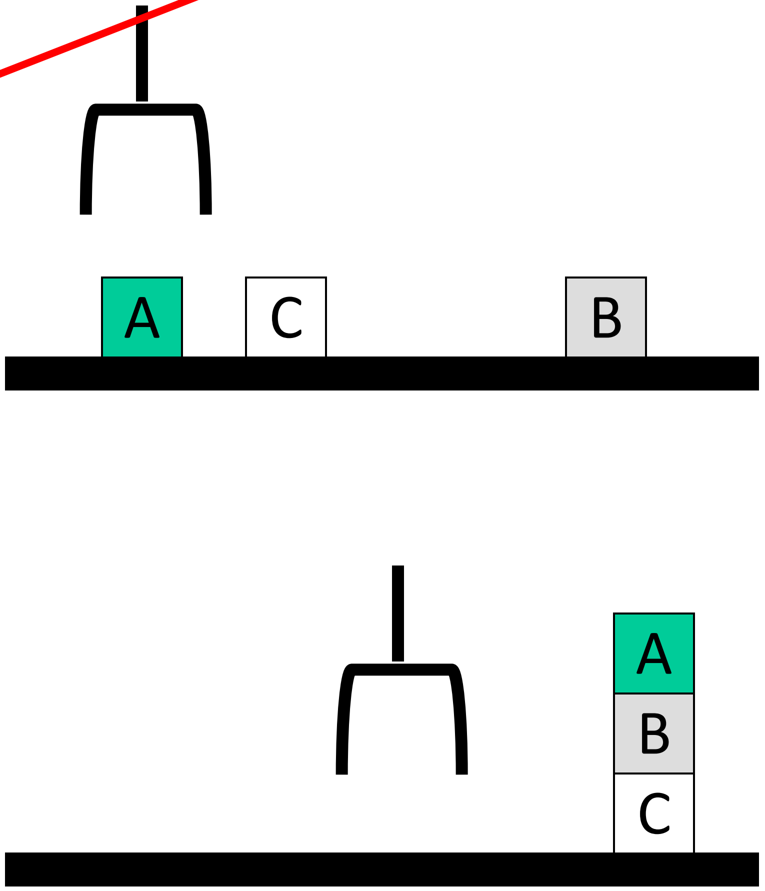
clear(a)
clear(b)
clear(c)
ontable(a)
ontable(b)
ontable(c)
handempty

Goal state:

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

assertions describing a state

atomic robot actions



Plan:

pickup(b)
stack(b,c)
pickup(a)
stack(a,b)

Planning problem

- Find sequence of actions to achieve goal state when executed from initial state given
  - set of possible primitive actions, including their *preconditions* and *effects*
  - initial state description
  - goal state description
- Compute plan as a sequence of actions that, when executed in initial state, achieves goal state
- States specified as conjunction of conditions, e.g., $ontable(a) \wedge on(b, a)$

Planning vs. problem solving

- Problem solving methods can often solve similar problems
- Planning is more powerful and efficient 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)
- Sub-goals can be planned independently, reducing the complexity of the planning problem

Typical simplifying assumptions

- Atomic time: Each action is indivisible
- No concurrent actions: but actions need not be ordered w.r.t. each other in the plan
- Deterministic actions: action results completely determined — no uncertainty in their effects
- Agent is the sole cause of change in the world
- Agent is omniscient with complete knowledge of the state of the world
- Closed world assumption: everything known to be true in world is included in state description and anything not listed is false

Blocks world

The blocks world consists of a table, a set of blocks and a robot hand

Some domain constraints:

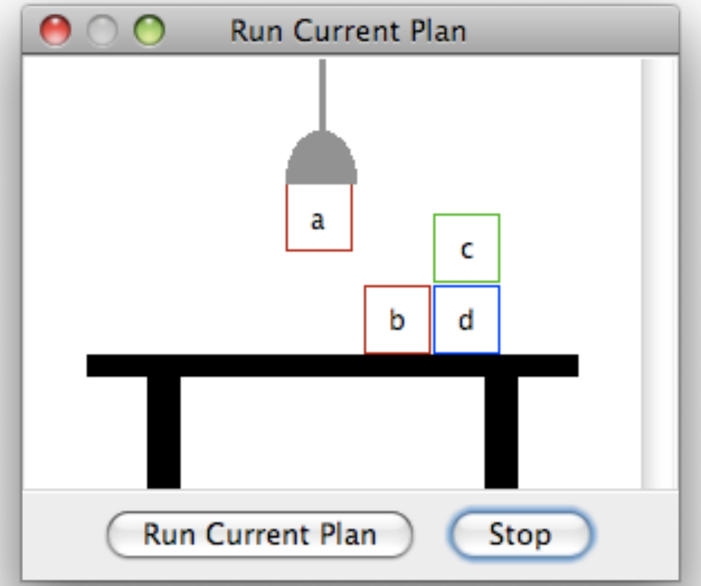
- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

ontable(b) ontable(d)

on(c,d) holding(a)

clear(b) clear(c)



Meant to be a simple model!

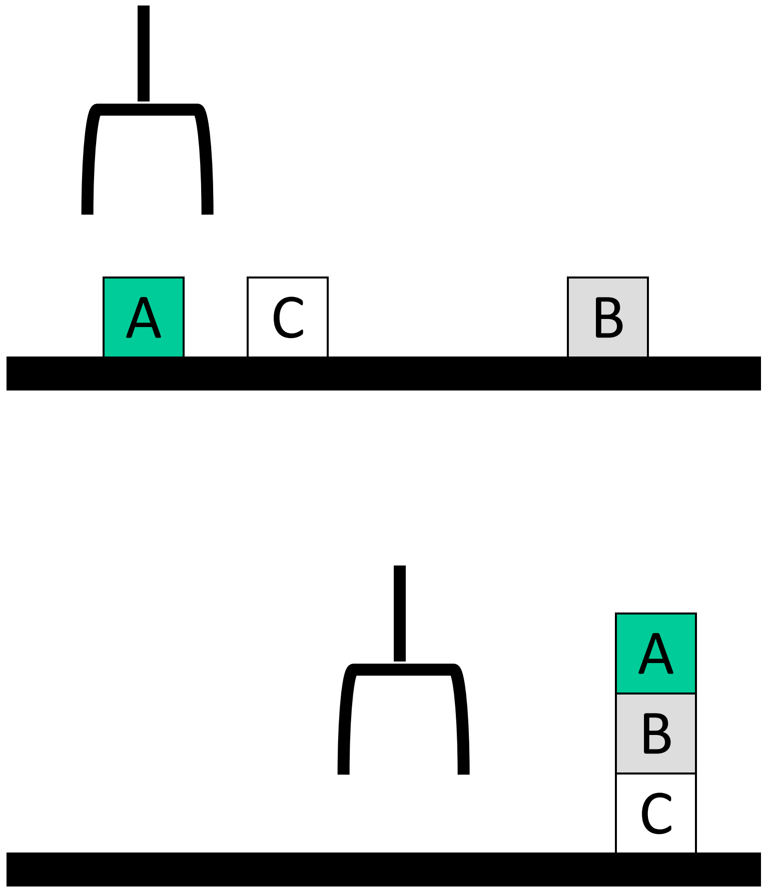
Typical BW planning problem

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)

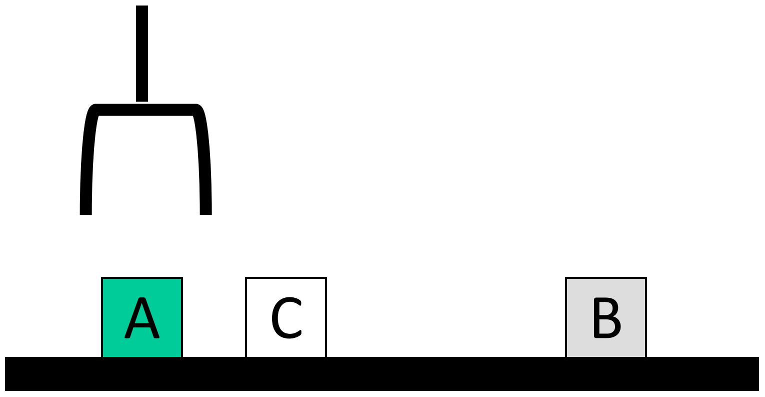
Another BW planning problem

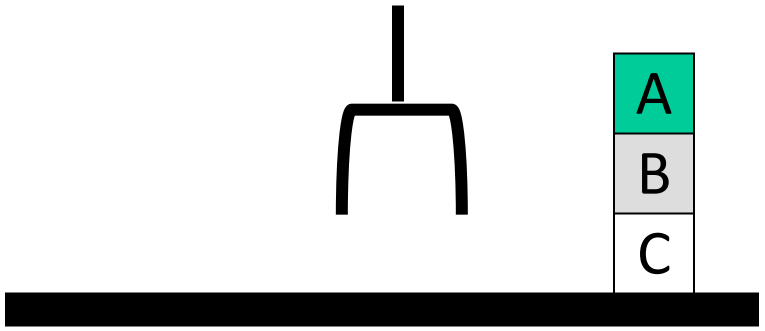
Initial state:

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)

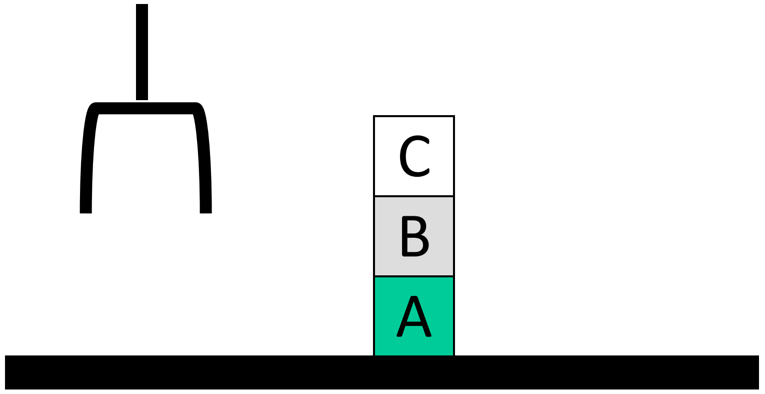
Yet Another BW planning problem

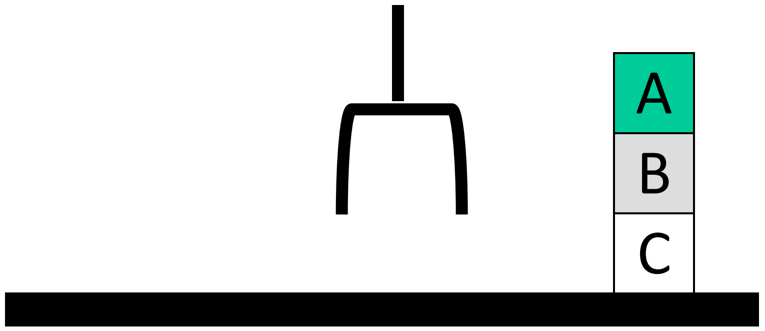
Initial state:

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

Goal:

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





Plan:

- unstack(c,b)
- putdown(c)
- unstack(b,a)
- putdown(b)
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

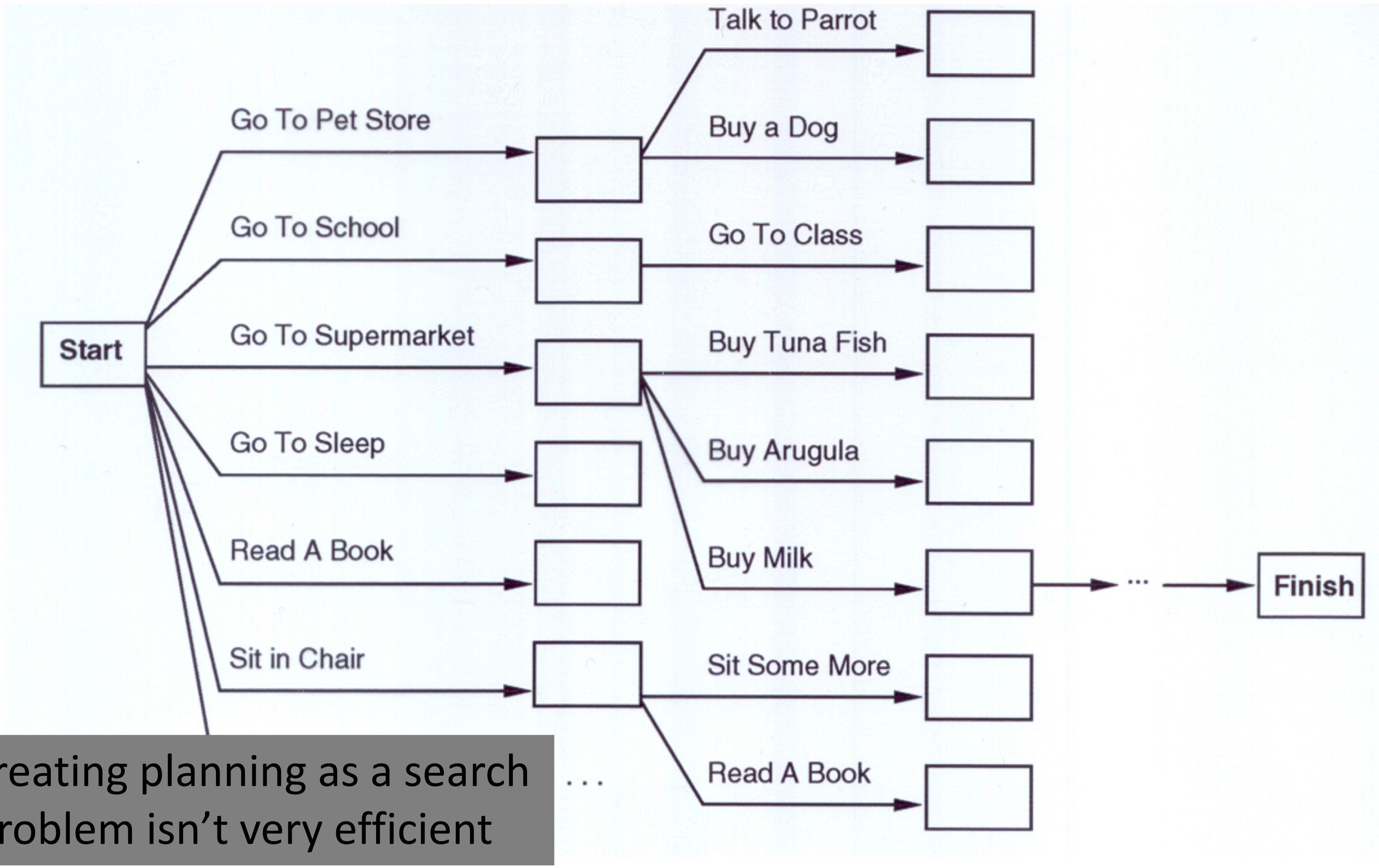
Major approaches

- Planning as search
- GPS / STRIPS
- Situation calculus
- Partial order planning
- Hierarchical decomposition (HTN planning)
- Planning with constraints (SATplan, Graphplan)
- Reactive planning

Planning as Search

- **Actions:** generate successor states
- **States:** completely described & only used for successor generation, heuristic fn. evaluation & goal testing
- **Goals:** represented as goal test and using a heuristic function
- **Plan representation:** unbroken sequences of actions forward from initial states or backward from goal state

"Get a quart of milk, a bunch of bananas and a variable-speed cordless drill."



Treating planning as a search problem isn't very efficient

General Problem Solver

- General Problem Solver (GPS) system was an early planner (Newell, Shaw, and Simon, 1957)
- GPS generated actions that reduced difference between some state and a goal state
- GPS used Means-Ends Analysis
  - Compare given to desired states; select best action to do next
  - Table of differences identifies actions to reduce types of differences
- GPS was a state space planner: operated in domain of state space problems specified by initial state, some goal states, and set of operations
- Introduced general way to use domain knowledge to select most promising action to take next

Situation calculus planning

- Intuition: Represent planning problem using first-order logic
  - Situation calculus lets us reason about changes in the world
  - Use theorem proving to find action sequence, when applied to initial situation leads to desired result
- How “neats” approach the problem

Situation calculus

- Initial state: logical sentence about (situation) S_0

  $At(Home, S_0) \wedge \neg Have(Milk, S_0) \wedge \neg\ Have(Bananas, S_0) \wedge \neg\ Have(Drill, S_0)$

- Goal state:

  $(\exists s)\ At(Home,s) \wedge Have(Milk,s) \wedge Have(Bananas,s) \wedge Have(Drill,s)$

- Actions describe how world changes:

  $\forall(a,s)\ Have(Milk,Result(a,s)) \Leftrightarrow$
  $((a=Buy(Milk) \wedge At(Grocery,s)) \vee (Have(Milk, s) \wedge a \neq Drop(Milk)))$

- Result(a,s) names situation resulting from doing action a in situation s

- Action sequences: Result'(l,s) is result of executing 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 II

- Solution is plan that when applied to initial state yields situation satisfying the goal:

  At(Home, Result'(p,S_0))

  $\wedge$ Have(Milk, Result'(p,S_0))

  $\wedge$ Have(Bananas, Result'(p,S_0))

  $\wedge$ Have(Drill, Result'(p,S_0))

- We expect a plan (i.e., variable assignment through unification) such as:

  p = [Go(Grocery), Buy(Milk), Buy(Bananas),
  Go(HardwareStore), Buy(Drill), Go(Home)]

Situation calculus: Blocks world

- A situation calculus rule for the blocks world

  Clear (X, Result(A,S)) $\leftrightarrow$
  [Clear (X, S) $\wedge$
  ($\neg$(A=Stack(Y,X) $\vee$ A=Pickup(X))
  $\vee$ (A=Stack(Y,X) $\wedge$ $\neg$(holding(Y,S))
  $\vee$ (A=Pickup(X) $\wedge$ $\neg$(handempty(S) $\wedge$ ontable(X,S) $\wedge$ clear(X,S))))]
  $\vee$ [A=Stack(X,Y) $\wedge$ holding(X,S) $\wedge$ clear(Y,S)]
  $\vee$ [A=Unstack(Y,X) $\wedge$ on(Y,X,S) $\wedge$ clear(Y,S) $\wedge$ handempty(S)]
  $\vee$ [A=Putdown(X) $\wedge$ holding(X,S)]

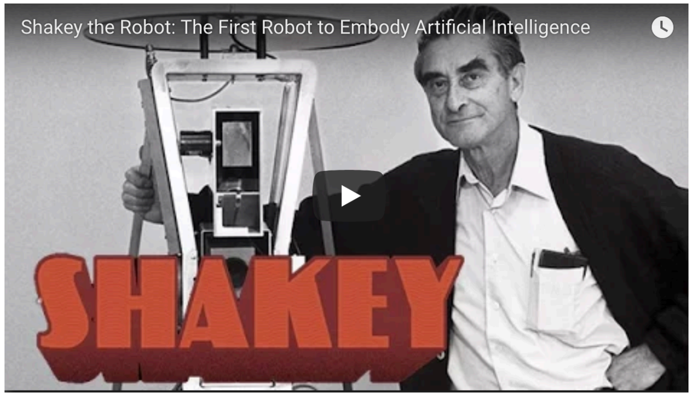
- Translation: A block is clear if (a) in previous state it was clear & we didn't pick it up or stack something on it, or (b) we stacked it on something else, or (c) something was on it that we unstacked, or (d) we were holding it and we put it down.
- Whew!!! There's gotta be a better way!

Situation calculus planning: Analysis

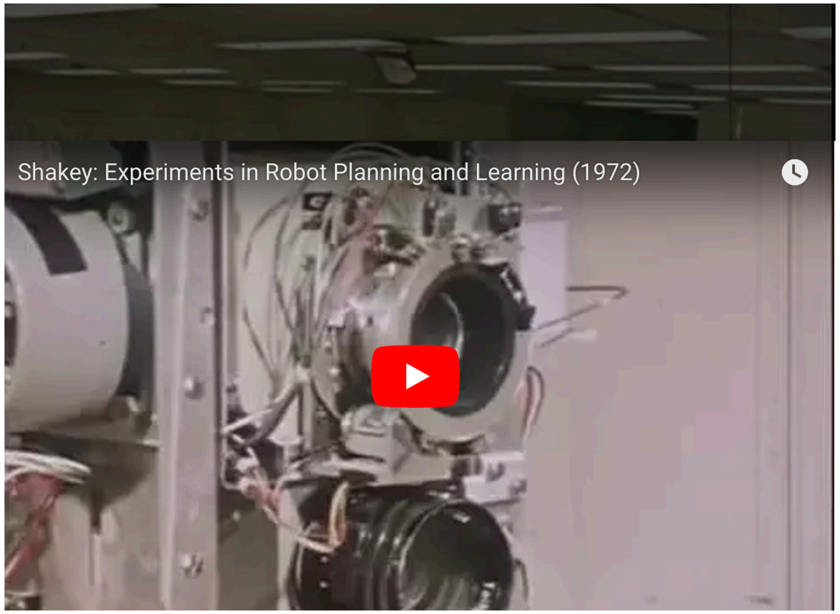
- Fine in theory, but problem solving (search) is exponential in worst case
- Resolution theorem proving only finds a proof (plan), not necessarily a **good** plan
- So, restrict language and use special-purpose algorithm (a planner) rather than general theorem prover
- Planning is a common task for intelligent agents, so it’s reasonable to have a special subsystem

Shakey the robot

First general-purpose mobile robot to be able to reason about its own actions

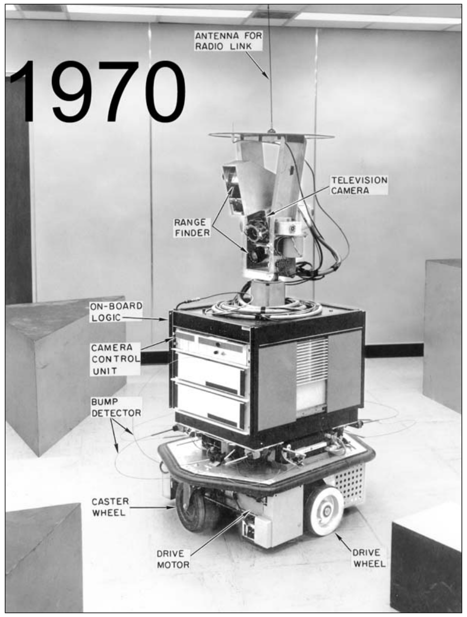

Shakey the Robot: The First Robot to Embody Artificial Intelligence
SHAKEY

Shakey the Robot: 1st Robot to Embody Artificial Intelligence (2017, 6 min.)



Shakey: Experiments in Robot Planning and Learning (1972, 24 min)

Strips planning representation



Shakey the robot

- Classic approach first used in the STRIPS (Stanford Research Institute Problem Solver) planner
- A State is a conjunction of ground literals

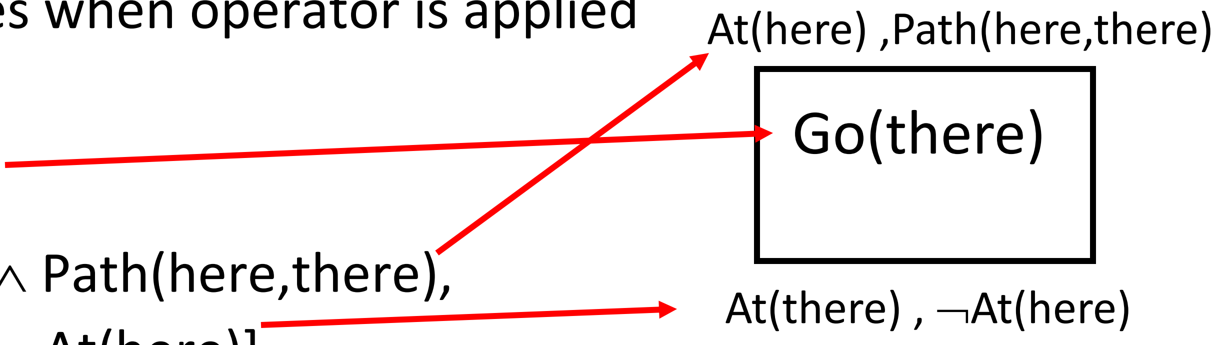
  at(Home) $\wedge$ $\neg$have(Milk) $\wedge$ $\neg$have(bananas) ...
- Goals are conjunctions of literals, but may have variables, assumed to be existentially quantified

  at(?x) $\wedge$ have(Milk) $\wedge$ have(bananas) ...
- Need not fully specify state
  – Non-specified conditions either don’t-care or assumed false
  – Represent many cases in small storage
  – May only represent changes in state rather than entire situation
- Unlike theorem prover, not seeking whether goal is true, but is there a sequence of actions to attain it

Operator/action representation

- Action operators have three components:
  - Action description
  - Precondition: conjunction of positive literals
  - Effect: conjunction of positive or negative literals describing 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 operators

- Classic basic operations for the blocks world
  - stack(X,Y): put block X on block Y
  - unstack(X,Y): remove block X from block Y
  - pickup(X): pickup block X
  - putdown(X): put block X on the table
- Each represented by
  - list of preconditions
  - list of new facts to be added (add-effects)
  - list of facts to be removed (delete-effects)
  - optionally, set of (simple) variable constraints
- For example stack(X,Y):

```
preconditions(stack(X,Y), [holding(X), clear(Y)])
deletes(stack(X,Y), [holding(X), clear(Y)]).
adds(stack(X,Y), [handempty, on(X,Y), clear(X)])
constraints(stack(X,Y), [X≠Y, Y≠table, X≠table])
```

Blocks world operators (Prolog)

operator(op, preconditions, adds, deletes, constraints)

```
operator(stack(X,Y),
   [holding(X), clear(Y)],
   [handempty, on(X,Y), clear(X)],
   [holding(X), clear(Y)],
   [X≠Y, Y≠table, X≠table]).
```

```
operator(unstack(X,Y),
      [on(X,Y), clear(X), handempty],
      [holding(X), clear(Y)],
      [handempty, clear(X), on(X,Y)],
      [X≠Y, Y≠table, X≠table]).
```

```
operator(pickup(X),
  [ontable(X), clear(X), handempty],
   [holding(X)],
   [ontable(X), clear(X), handempty],
   [X≠table]).
```

```
operator(putdown(X),
       [holding(X)],
       [ontable(X), handempty, clear(X)],
       [holding(X)],
       [X≠table]).
```

STRIPS planning

- STRIPS maintains two additional data structures:
  - State List - all currently true predicates.
  - Goal Stack - push down stack of goals to be solved, with current goal on top
- If current goal not satisfied by present state, find operator that adds it and push operator and its preconditions (subgoals) on stack
- When a current goal is satisfied, POP from stack
- When an operator is on top stack, record application of that operator on plan sequence and use operator’s add and delete lists to update current state

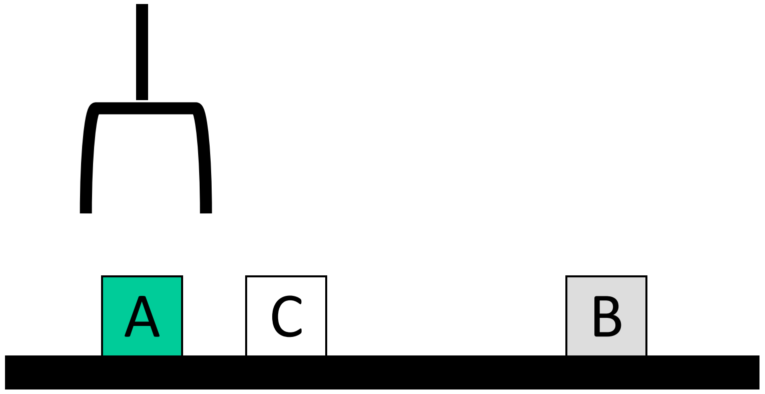
Typical BW planning problem

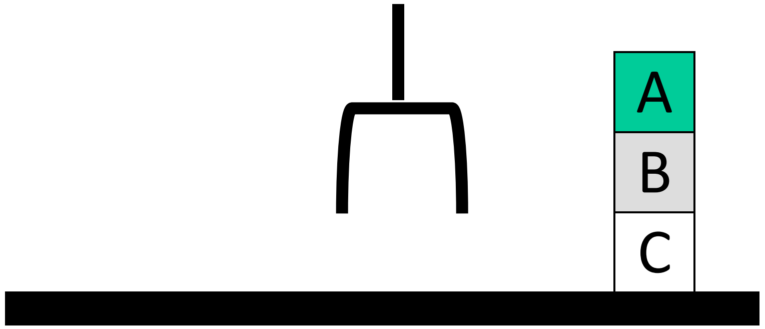
Initial state:

- 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)

Strips in Prolog

```
% strips(+Goals, +InitState, -Plan)
strips(Goal, InitState, Plan):-
  strips(Goal, InitState, [],  _, RevPlan),
  reverse(RevPlan, Plan).

% strips(+Goals,+State,+Plan,-NewState, NewPlan )
% Finished if each goal in Goals is true
% in current State.
strips(Goals, State, Plan, State, Plan) :-
  subset(Goals,State).
```

```
strips(Goals, State, Plan, NewState, NewPlan):-
  % Goal is an unsatisfied goal.
  member(Goal, Goals),
  (\+ member(Goal, State)),
  % Op is an Operator with Goal as a result.
  operator(Op, Preconditions, Adds, Deletes,_),
  member(Goal,Adds),
  % Achieve the preconditions
  strips(Preconditions, State, Plan, TmpState1,
    TmpPlan1),
  % Apply the Operator
  diff(TmpState1, Deletes, TmpState2),
  union(Adds, TmpState2, TmpState3).
  % Continue planning.
  strips(GoalList, TmpState3, [Op|TmpPlan1],
    NewState, NewPlan).
```

Trace (Prolog)

```
strips([on(b,c),on(a,b),ontable(c)],[clear(a),clear(b),clear(c),ontable(a),ontable(b),ontable(c),handempty],[])
Achieve on(b,c) via stack(b,c) with preconds: [holding(b),clear(c)]
    strips([holding(b),clear(c)],[clear(a),clear(b),clear(c),ontable(a),ontable(b),ontable(c),handempty],[])
    Achieve holding(b) via pickup(b) with preconds: [ontable(b),clear(b),handempty]
      strips([ontable(b),clear(b),handempty],[clear(a),clear(b),clear(c),ontable(a),ontable(b),ontable(c),handempty],[])
    Applying pickup(b)
    strips([holding(b),clear(c)],[clear(a),clear(c),holding(b),ontable(a),ontable(c)],[pickup(b)])
Applying stack(b,c)
strips([on(b,c),on(a,b),ontable(c)],[handempty,clear(a),clear(b),ontable(a),ontable(c),on(b,c)],[stack(b,c),pickup(b)])
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
    strips([holding(a),clear(b)],[handempty,clear(a),clear(b),ontable(a),ontable(c),on(b,c)],[stack(b,c),pickup(b)])
    Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
      strips([ontable(a),clear(a),handempty],[handempty,clear(a),clear(b),ontable(a),ontable(c),on(b,c)],[stack(b,c),picku
      p(b)])
    Applying pickup(a)
    strips([holding(a),clear(b)],[clear(b),holding(a),ontable(c),on(b,c)],[pickup(a),stack(b,c),pickup(b)])
Applying stack(a,b)
strips([on(b,c),on(a,b),ontable(c)],[handempty,clear(a),ontable(c),on(a,b),on(b,c)],[stack(a,b),pickup(a),stack(b,c),pickup(
  b)])
```

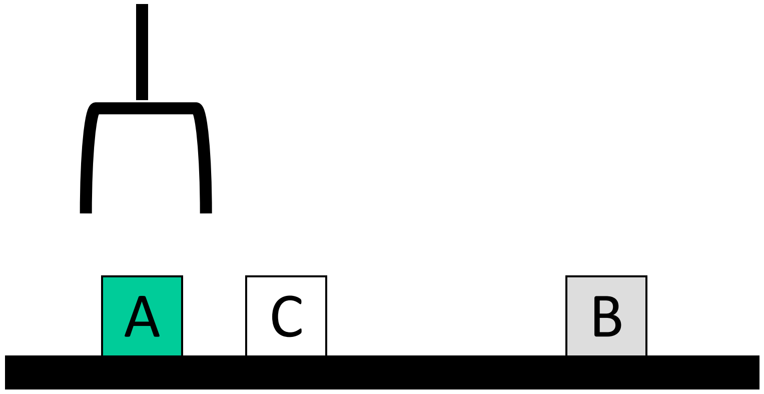
Another BW planning problem

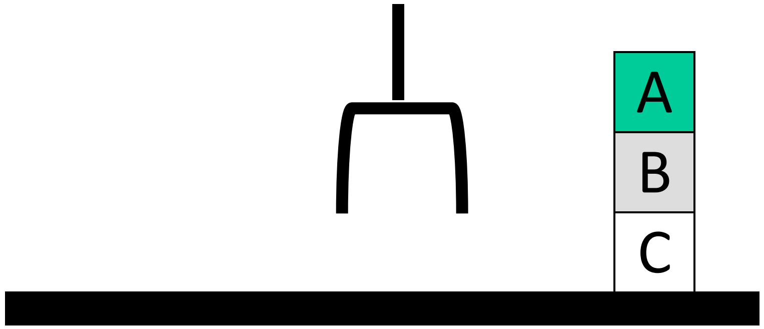
Initial state:

- 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)

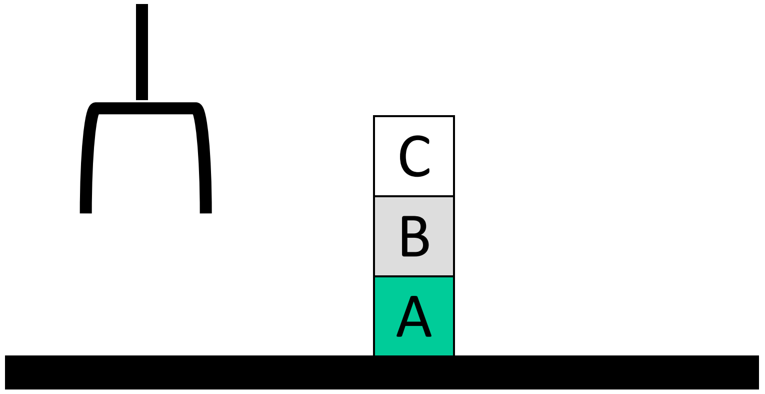
Yet Another BW planning problem

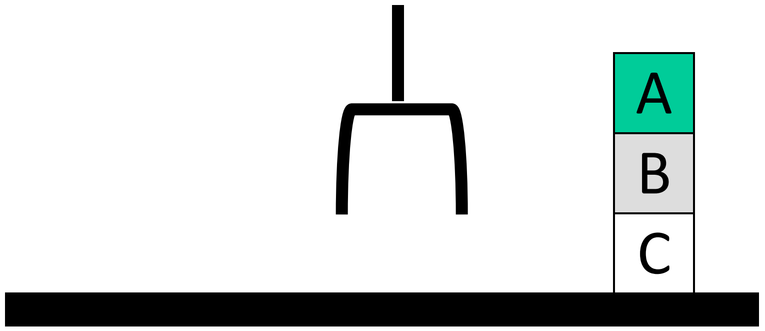
Initial state:

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

Goal:

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





Plan:

- unstack(c,b)
- putdown(c)
- unstack(b,a)
- putdown(b)
- pickup(b)
- stack(b,a)
- unstack(b,a)
- putdown(b)
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Yet Another BW planning problem

Initial state:

ontable(a)

ontable(b)

clear(a)

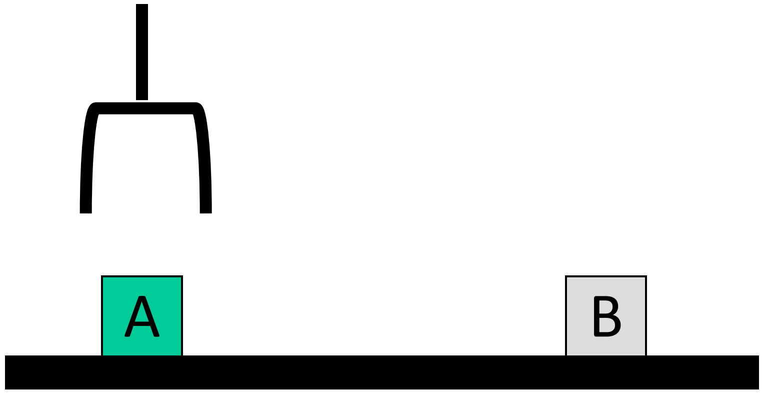
clear(b)

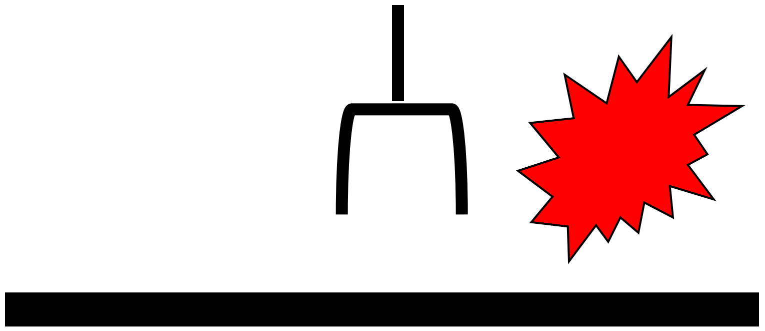
handempty

Goal:

on(a,b)

on(b,a)



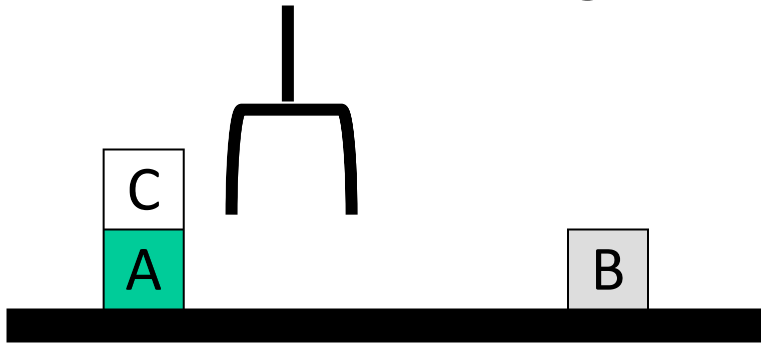


Plan:

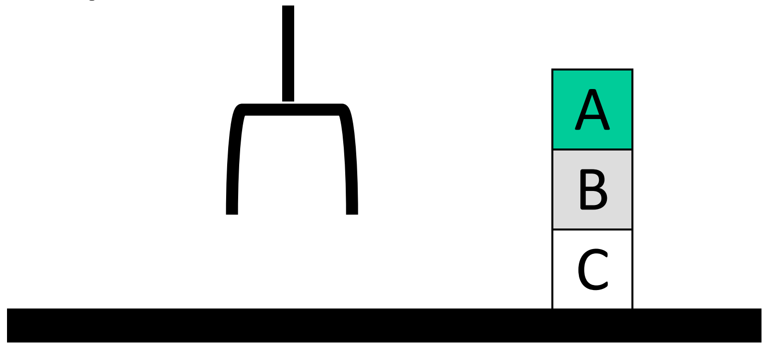
??

Goal interaction

- Simple planning algorithms assume independent sub-goals
  - Solve each separately and concatenate the solutions
- The "Sussman Anomaly" is the classic example of the goal interaction problem:
  - Solving on(A,B) first (via unstack(C,A), stack(A,B)) is undone when solving 2nd goal on(B,C) (via unstack(A,B), stack(B,C))
  - Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS couldn't handle this, although minor modifications can get it to do simple cases



Initial state

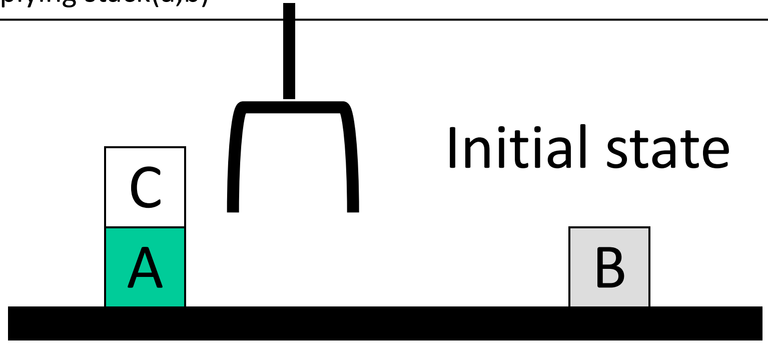


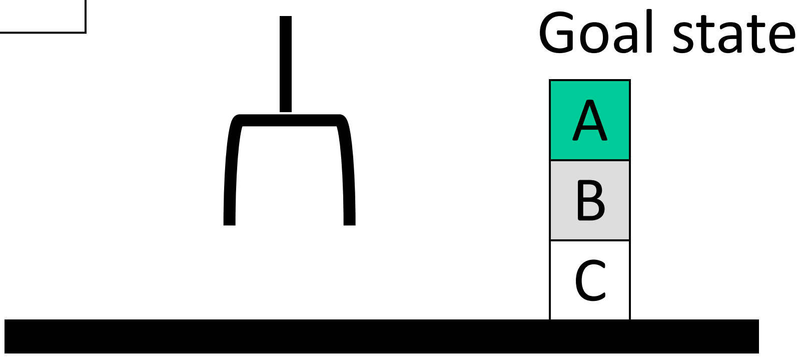
Goal state

Sussman Anomaly

```
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
||Achieve clear(a) via unstack(_1584,a) with preconds:
[on(_1584,a),clear(_1584),handempty]
||Applying unstack(c,a)
||Achieve handempty via putdown(_2691) with preconds: [holding(_2691)]
||Applying putdown(c)
|Applying pickup(a)
Applying stack(a,b)
Achieve on(b,c) via stack(b,c) with preconds: [holding(b),clear(c)]
|Achieve holding(b) via pickup(b) with preconds: [ontable(b),clear(b),handempty]
||Achieve clear(b) via unstack(_5625,b) with preconds:
[on(_5625,b),clear(_5625),handempty]
||Applying unstack(a,b)
||Achieve handempty via putdown(_6648) with preconds: [holding(_6648)]
||Applying putdown(a)
|Applying pickup(b)
Applying stack(b,c)
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
|Applying pickup(a)
Applying stack(a,b)
```

```
From
[clear(b),clear(c),ontable(a),ontable(b),on
(c,a),handempty]
 To [on(a,b),on(b,c),ontable(c)]
 Do:
     unstack(c,a)
     putdown(c)
     pickup(a)
     stack(a,b)
     unstack(a,b)
     putdown(a)
     pickup(b)
     stack(b,c)
     pickup(a)
     stack(a,b)
```





Sussman Anomaly

- Classic Strips assumed that once a goal had been satisfied it would stay satisfied
- Simple Prolog version selects any currently unsatisfied goal to tackle at each iteration
- This can handle this problem, at the expense of looping for other problems
  - e.g., achieving goal [on(a,b) , on (b,a)]
- What's needed? A notion of *protecting* a sub-goal so it's not undone by later steps

State-space planning

- STRIPS searches thru a space of situations (where you are, what you have, etc.)
  - Plan is a solution found by "searching" through situations to get to goal
- Progression planners search forward from initial state to goal state
  - Usually results in a high branching factor
- Regression planners search backward from goal
  - OK if operators have enough information to go both ways
  - Can reduce branching: you're only considering things relevant to goal
  - Handling a conjunction of goals is difficult (e.g., STRIPS)

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 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

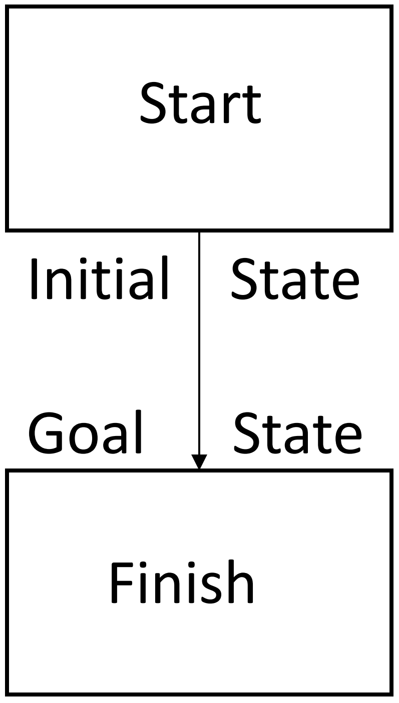
- Linear planners build plans as totally ordered sequences of steps
- Non-linear planners (aka partial-order planners) build plans as sets of steps with temporal constraints
  - constraints like S1<S2 if step S1 must come before S2
- One refines a partially ordered plan (POP) by either:
  - adding a 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

Some example domains

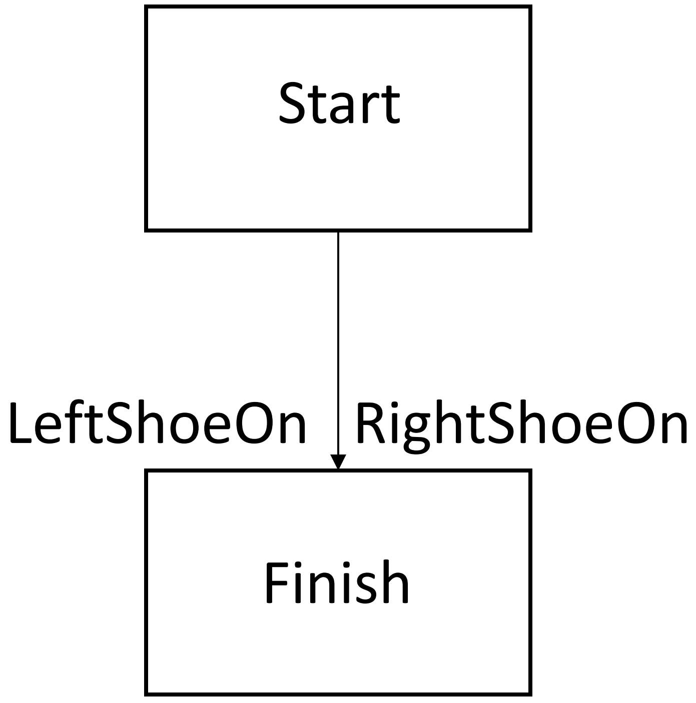
We'll use some simple problems with a real world flavor to illustrate planning problems and algorithms

- Putting on your socks and shoes in the morning
  - Actions like put-on-left-sock, put-on-right-shoe
- Planning a shopping trip involving buying several kinds of items
  - Actions like go(X), buy(Y)

A simple graphical notation

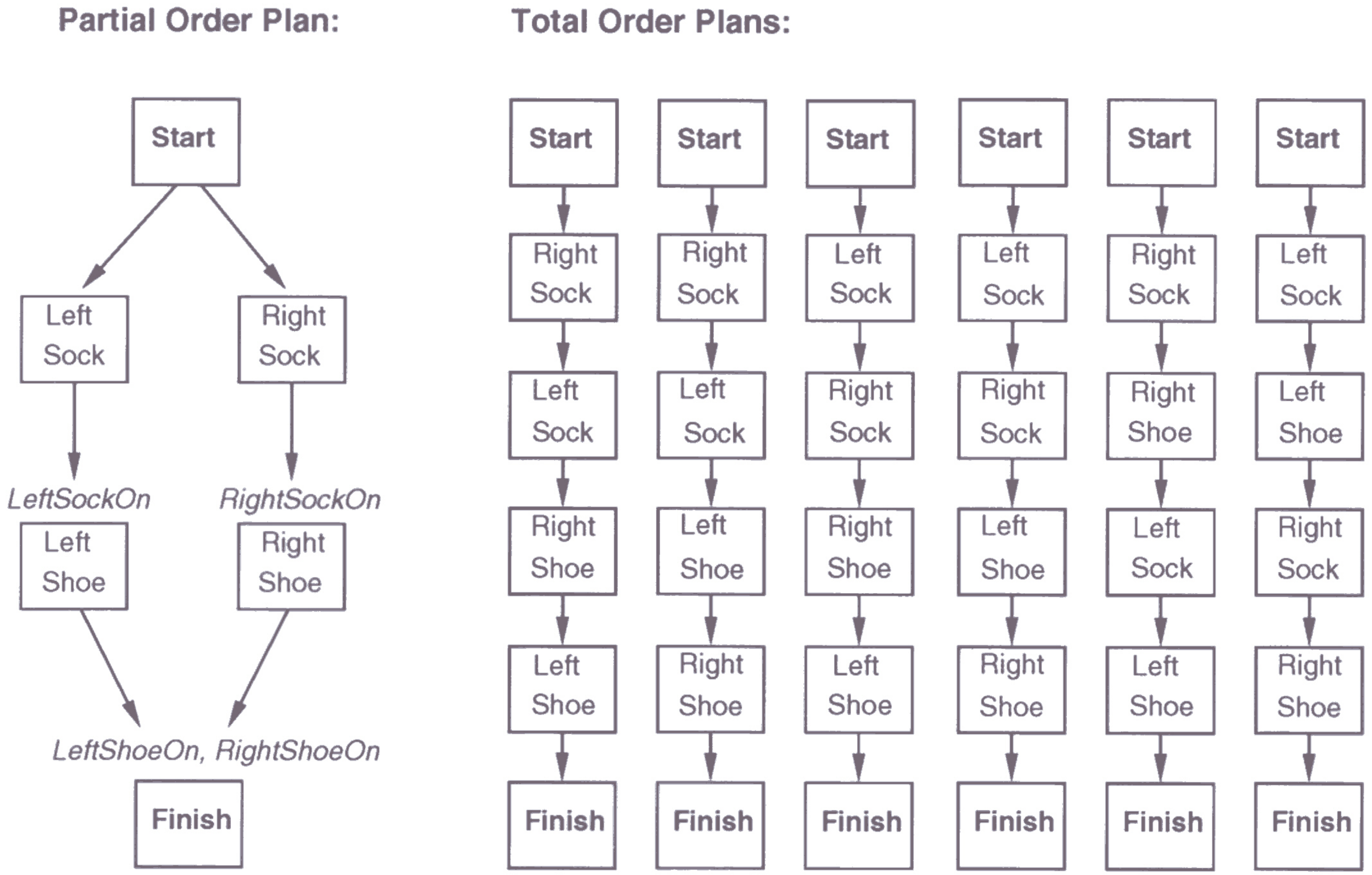


(a)



(b)

Partial Order Plan vs. Total Order Plan


Partial Order Plan:
Total Order Plans:
Start
Left Sock
Right Sock
LeftSockOn
RightSockOn
Left Shoe
Right Shoe
LeftShoeOn, RightShoeOn
Finish
Start → Right Sock → Left Sock → Right Shoe → Left Shoe → Finish
Start → Right Sock → Left Sock → Left Shoe → Right Shoe → Finish
Start → Left Sock → Right Sock → Right Shoe → Left Shoe → Finish
Start → Left Sock → Right Sock → Left Shoe → Right Shoe → Finish
Start → Right Sock → Right Shoe → Left Sock → Left Shoe → Finish
Start → Left Sock → Left Shoe → Right Sock → Right Shoe → Finish

The space of POPs is smaller than TOPs and hence involve less search

Least commitment

- Non-linear planners embody the principle of least commitment
  - only choose actions, orderings & variable bindings absolutely necessary, postponing other decisions
  - avoids early commitment to decisions that don’t really matter
- Linear planners always choose to add a plan step in a particular place in the sequence
- Non-linear planners choose to add a step and possibly some temporal constraints

Real-world planning domains

- Real-world domains are complex and don't satisfy assumptions of STRIPS or methods
- Some of the characteristics we may need to handle:
  - Modeling and reasoning about resources
  - Representing and reasoning about time

    } Scheduling
  - Planning at different levels of abstractions
  - Conditional outcomes of actions
  - Uncertain outcomes of actions

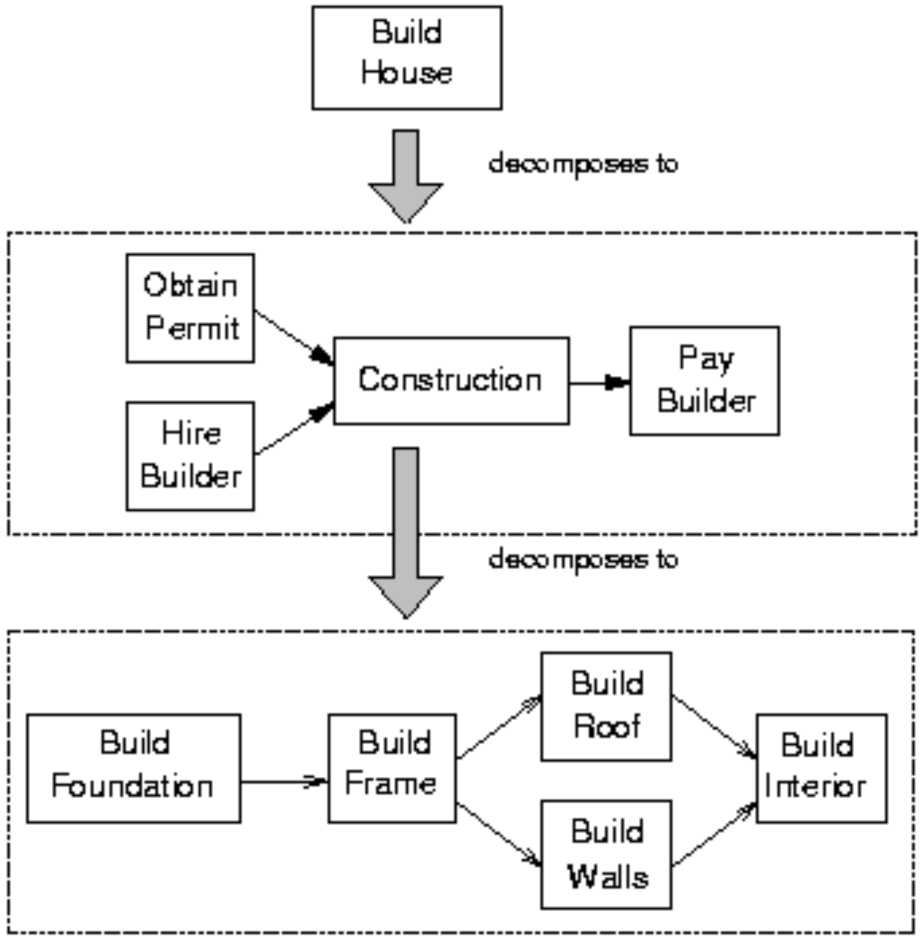
    } Planning under uncertainty
  - Exogenous events
  - Incremental plan development
  - Dynamic real-time re-planning

    } HTN planning

Hierarchical decomposition

- Hierarchical decomposition, or hierarchical task network (HTN) planning, uses abstract operators to incrementally decompose a planning problem from a high-level goal statement to a primitive plan network
- Primitive operators represent actions that are executable, and can appear in the final plan
- Non-primitive operators represent goals (equivalently, abstract actions) that require further decomposition (or operationalization) to be executed
- There is no “right” set of primitive actions: One agent’s goals are another agent’s actions!

HTN planning: example



HTN operator: Example

```
OPERATOR decompose
PURPOSE: Construction
CONSTRAINTS:
    Length (Frame) <= Length (Foundation),
    Strength (Foundation) > Wt(Frame) + Wt(Roof)
        + Wt(Walls) + Wt(Interior) + Wt(Contents)
PLOT: Build (Foundation)
      Build (Frame)
      PARALLEL
            Build (Roof)
            Build (Walls)
      END PARALLEL
      Build (Interior)
```

Planning summary

- Planning representations
  - Situation calculus
  - STRIPS representation: Preconditions and effects
- Planning approaches
  - State-space search (STRIPS, forward chaining, ….)
  - Plan-space search (partial-order planning, HTN, …)
  - Constraint-based search (GraphPlan, SATplan, …)
- Search strategies
  - Forward planning
  - Goal regression
  - Backward planning
  - Least-commitment
  - Nonlinear planning