CMSC 471: Planning

Spring 2021 (Sections 01 & 03)

Slides courtesy Tim Finin, Cynthia Matuszek, Marie DesJardines. Some material adopted from notes by Andreas Geyer-Schulz and Chuck Dyer.

Overview

- What is planning?
- Approaches to planning
  - GPS / STRIPS
  - Situation calculus formalism
  - Partial-order planning

Planning Problem

- Find a **sequence of actions** that achieves a **goal** when executed from an **initial state**.
- That is, given
  - A set of operators (possible actions)
  - An initial state description
  - A goal (description or conjunction of predicates)
- Compute a sequence of operations: a **plan**.

Planning Proble

- put on right shoe
- put on left shoe
- put on pants
- put on right sock
- put on left sock
- put on shirt

- Find a **sequence of actions** that achieves a **goal** when executed from an **initial state**.
- That is, given
  - A set of operators (possible actions)
  - An initial state description
  - A goal (description or conjunction of predicates)
- Compute a sequence of operations: a **plan**.

Planning Proble

- put on right shoe
- put on left shoe
- put on pants
- put on right sock
- put on left sock
- put on shirt

- Find a **sequence of actions** that achieves a **goal** when executed from an **initial state**
- That is, given
  - – A set of operators (possible actio
  - – An initial state description
  - – A goal (description or conjunction of predicates)
- Compute a sequence of operations: a **plan**.

- pants off
- right shoe off
- right sock off
- right shoe off

(etc)

Planning Proble

- put on right shoe
- put on left shoe
- put on pants
- put on right sock
- put on left sock
- put on shirt

- Find a **sequence of actions** that achieves a **goal** when executed from an **ini**
- That is, given
  - A set of operators (possible actio
  - An initial state description
  - A goal (description or conjunction of predicates)
- Compute a sequence of operati

- pants off
- right shoe off
- right sock off
- right shoe off

(etc)

- pants on

(etc)

Some example domains

- We'll use some simple problems 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)

Typical Assumptions (1)

- **Atomic time**: Each action is indivisible
  - Can’t be interrupted halfway through putting on pants
- **No concurrent actions** allowed
  - Can’t put on socks at the same time
- **Deterministic actions**
  - The result of actions are completely known – no uncertainty

Typical Assumptions

- Agent is the **sole cause of change** in the world
  - Nobody else is putting on your socks
- Agent is **omniscient:**
  - Has complete knowledge of the state of the world
- **Closed world assumption**:
  - Everything known-true about the world is in the *state description*
  - Anything not known-true is known-false

Blocks World

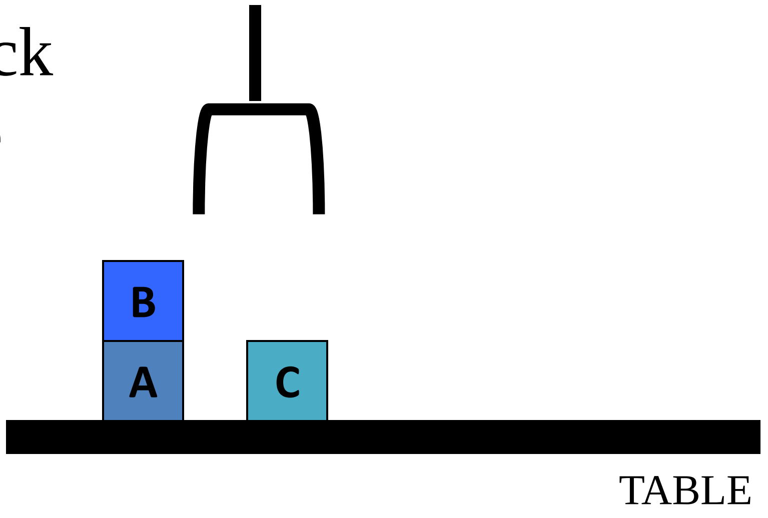
The **blocks world** consists of a table, set of blocks, and a robot gripper

Some domain constraints:

- Only one block on another block
- Any number of blocks on table
- Hand can only hold one block

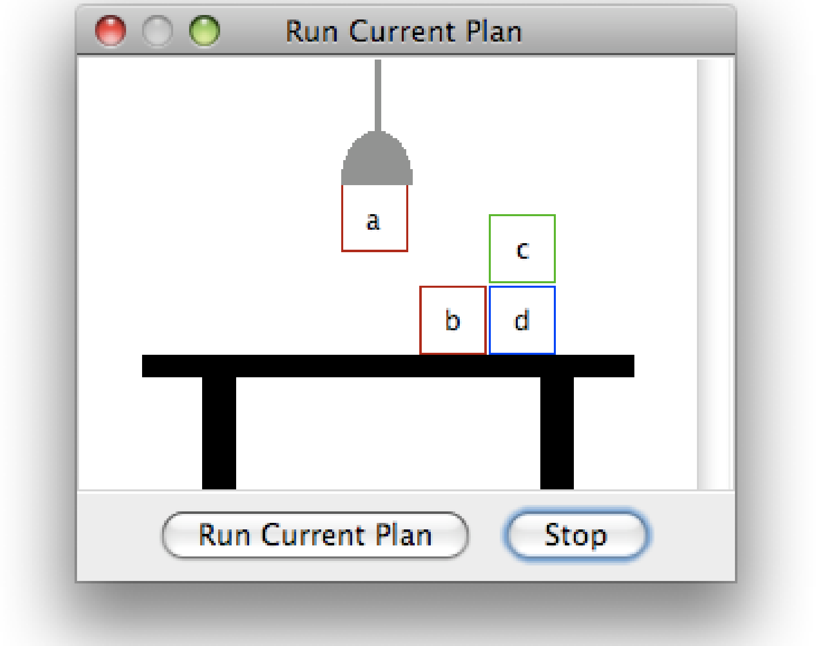
Typical representation:

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



Blocks world

- A micro-world
- 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

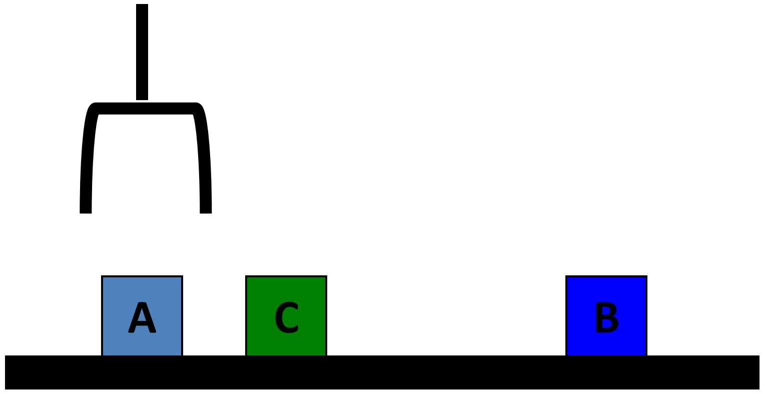


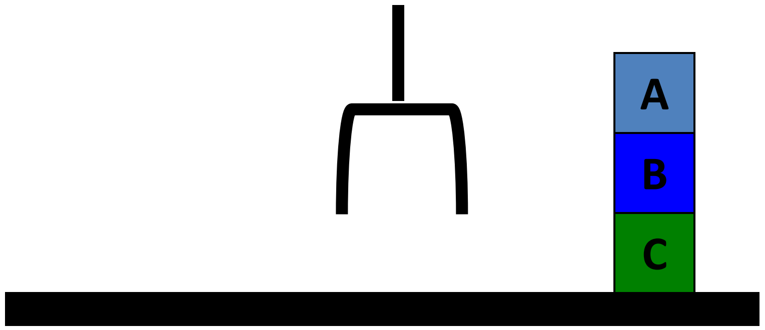
Meant to be a simple model! (Applet demo at: http://aispace.org/planning/index.shtml)

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)





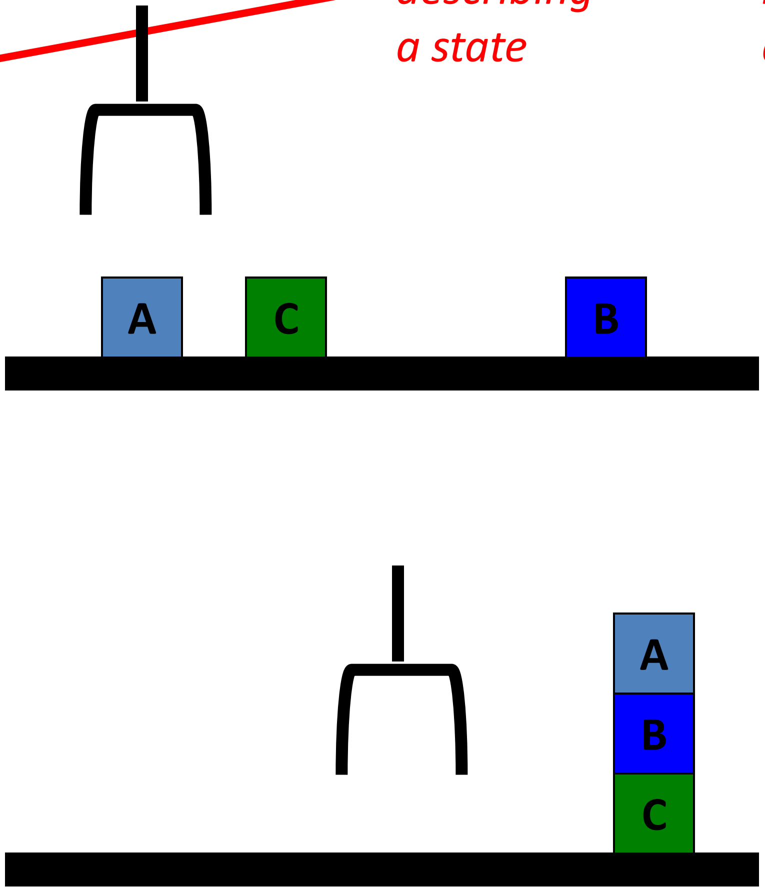
Typical BW planning problem

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)

Major Approaches

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

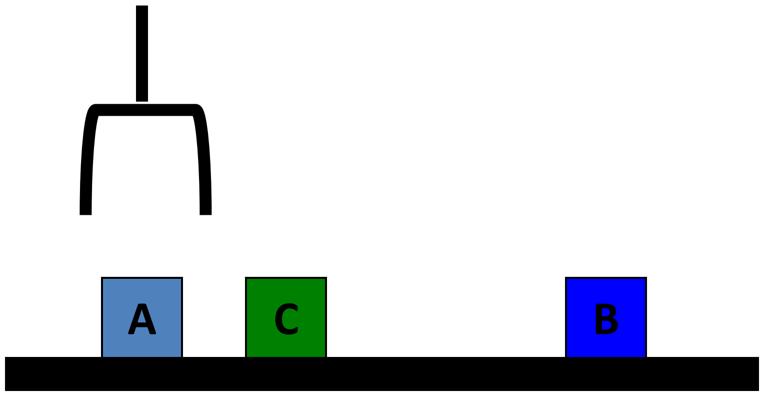
Planning vs. problem solving

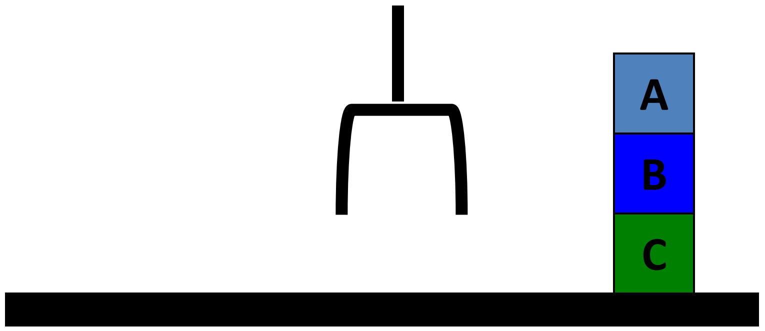
- Planning *vs.* problem solving: can often solve similar problems
- Planning is more powerful and efficient because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Search often proceeds through *plan space* rather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

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)

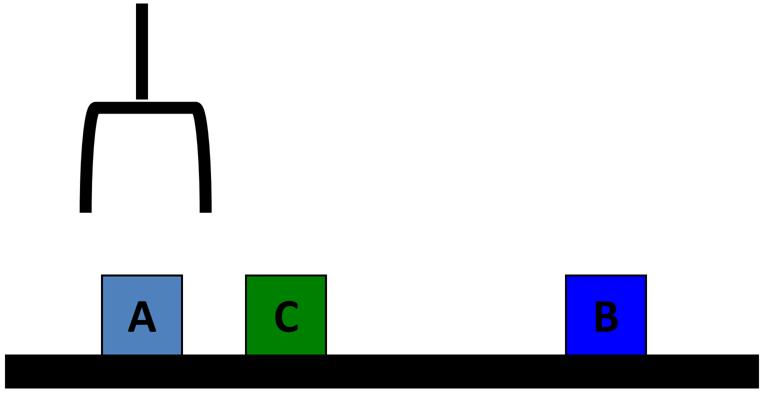
Yet Another BW planning problem

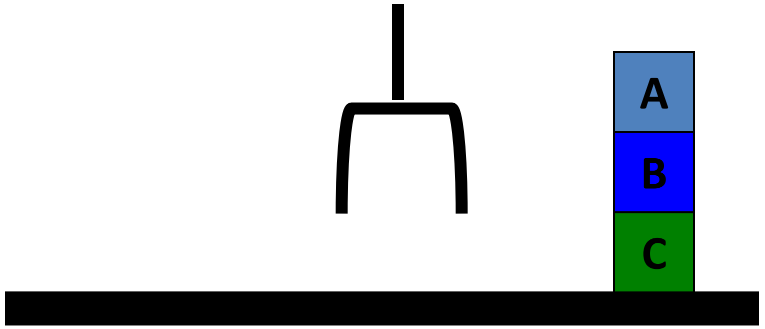
Initial state:

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

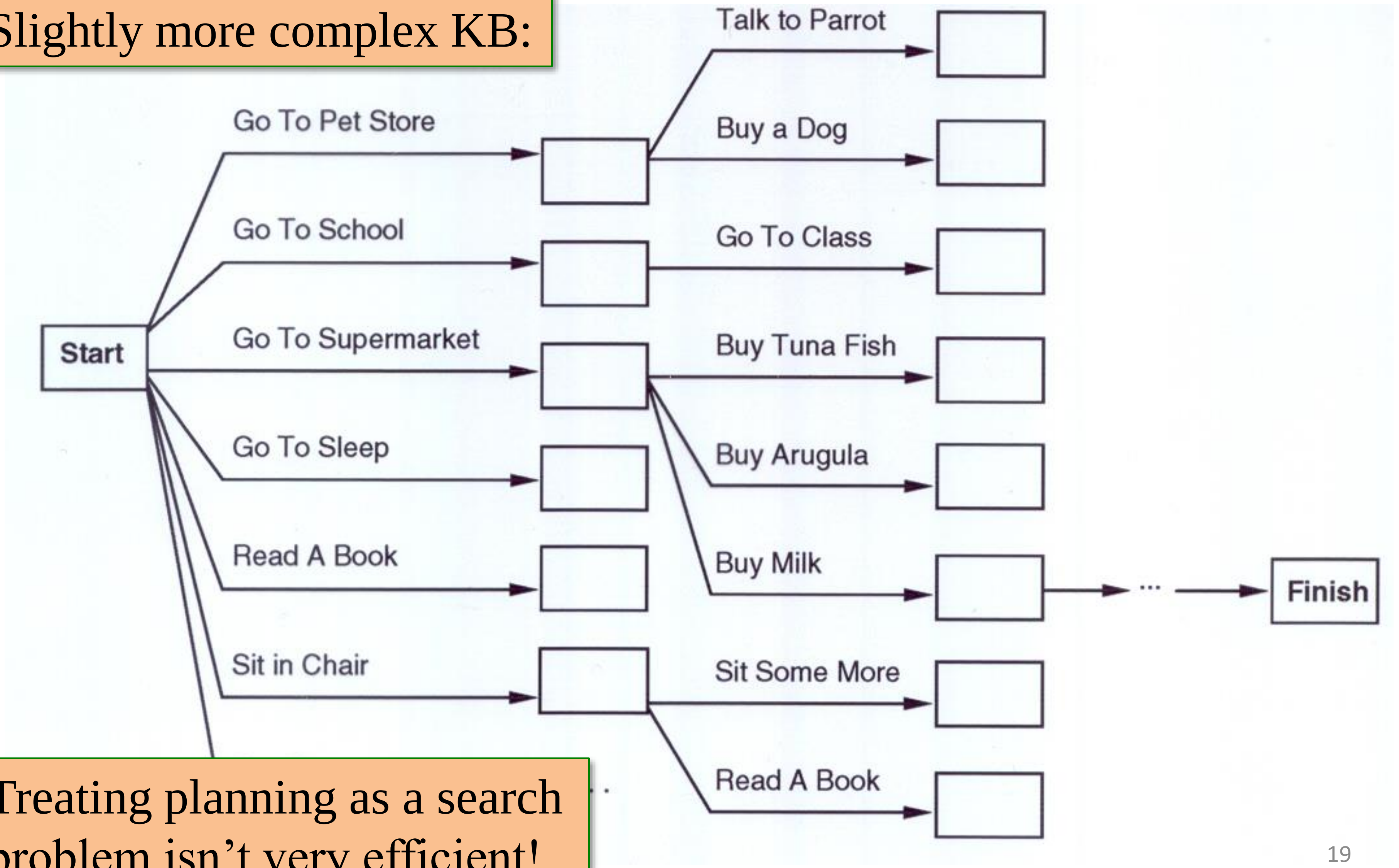
$$\text{backtracking}\left\{\begin{array}{l}\text{pickup(a)}\\\text{stack(a,b)}\\\text{unstack(a,b)}\\\text{putdown(a)}\end{array}\right.$$

Planning as Search (?)

- Can think of planning as a search problem
  - **Actions:** generate successor states
  - **States:** completely described & only used for successor generation, heuristic fn. evaluation & goal testing
  - **Goals:** represented as a 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."

Slightly more complex KB:



Treating planning as a search problem isn't very efficient!

General Problem Solver

- The **General Problem Solver (GPS)** system
  - An early planner (Newell, Shaw, and Simon)
- Generate actions that *reduce difference* between current state and goal state
- Uses *Means-Ends Analysis*
  - Compare what is **given** or **known** with what is desired
  - Select a reasonable thing to do next
  - Use a **table of differences** to identify procedures to reduce differences
- GPS is a **state space planner**
  - Operates on state space problems specified by an initial state, some goal states, and a set of operations

Situation Calculus Planning

- Intuition: Represent the **planning problem** using first-order logic
  - Situation calculus lets us reason about **changes** in the world
  - Use theorem proving to show ("prove") that a sequence of actions will lead to a desired result, when applied to a world state / situation

Situation Calculus Planning, cont.

- **Initial state**: a logical sentence about (situation) S_0
- **Goal state:** usually a conjunction of logical sentences
- **Operators**: descriptions of how the world changes as a result of the agent's actions:
  - Result(*a,s*) names the situation resulting from executing action *a* in situation *s*.
- Action sequences are also useful:
  - Result'(l,s): result of executing list of actions *l* starting in *s*

Situation Calculus Planning, cont.

- **Initial state**:

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

- **Goal state**:

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

- **Operators:**

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

- **Result(a,s)**: situation after executing action a in situation s

  $(\forall s)\ Result'([\ ],s) = s$

  $(\forall a,p,s)\ Result'([a|p]s) = Result'(p,Result(a,s))$

p=plan

Situation Calculus, cont.

- Solution: a **plan** that when applied to the **initial state** gives a situation satisfying the **goal query**:

  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))

- 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)]

Situation Calculus: Blocks World

- Example situation calculus rule for blocks world:
  - clear(X, Result(A,S)) ↔

    [clear(X, S) ∧

    (¬(A=Stack(Y,X) ∨ A=Pickup(X))

    ∨ (A=Stack(Y,X) ∧ ¬(holding(Y,S))

    ∨ (A=Pickup(X) ∧ ¬(handempty(S) ∧ ontable(X,S) ∧ clear(X,S))))]

    ∨ [A=Stack(X,Y) ∧ holding(X,S) ∧ clear(Y,S)]

    ∨ [A=Unstack(Y,X) ∧ on(Y,X,S) ∧ clear(Y,S) ∧ handempty(S)]

    ∨ [A=Putdown(X) ∧ holding(X,S)]
- English translation: a block is **clear** if

???

Situation Calculus: Blocks World

- Example situation calculus rule for blocks world:
  - clear(X, Result(A,S)) ↔
    [clear(X, S) ∧
    (¬(A=Stack(Y,X) ∨ A=Pickup(X))
    ∨ (A=Stack(Y,X) ∧ ¬(holding(Y,S))
    ∨ (A=Pickup(X) ∧ ¬(handempty(S) ∧ ontable(X,S) ∧ clear(X,S))))]
    ∨ [A=Stack(X,Y) ∧ holding(X,S) ∧ clear(Y,S)]
    ∨ [A=Unstack(Y,X) ∧ on(Y,X,S) ∧ clear(Y,S) ∧ handempty(S)]
    ∨ [A=Putdown(X) ∧ holding(X,S)]
- English translation: a block is **clear** if

(a) in the previous state it was clear AND we didn't pick it up or stack something on it successfully, or

(b) we stacked it on something else successfully, or

(c) something was on it that we unstacked successfully, or

(d) we were holding it and we put it down.

Situation Calculus Planning: Analysis

- Fine in theory, but:
  - Problem solving (search) is exponential in the worst case
  - Resolution theorem proving only finds *a* proof (plan), not necessarily a *good* plan
- So what can we do?
  - Restrict the language
    - Blocks world is already pretty small…
  - **Use a special-purpose planner rather than general theorem prover**

Basic Representations for Planning

- Classic approach first used in the STRIPS planner circa 1970
- **States** represented as conjunction of ground literals
  - at(Home) $\wedge$ $\neg$have(Milk) $\wedge$ $\neg$have(bananas) ...
- Goals are conjunctions of literals, but may have variables*
  - at(?x) $\wedge$ have(Milk) $\wedge$ have(bananas) ...
- Don't need to fully specify state
  - Un-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 finding whether the goal is **true**, but whether there is a sequence of actions to attain it

*generally assume $\exists$

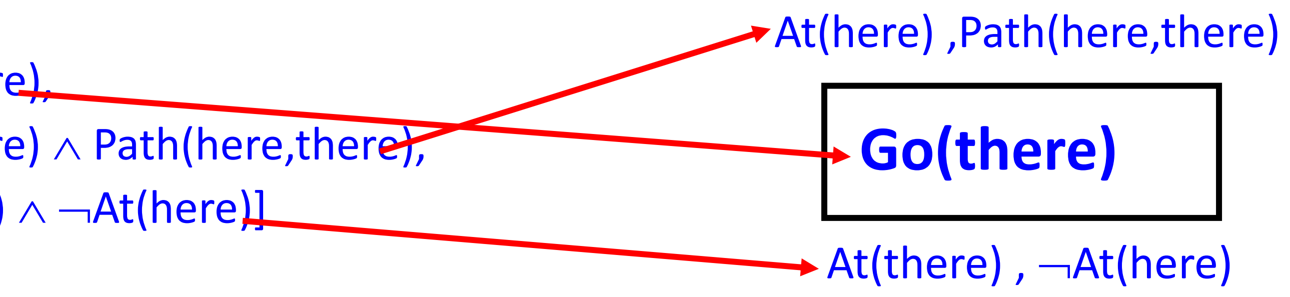
Operator/Action Representation

- **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) $\wedge$ Path(here,there),

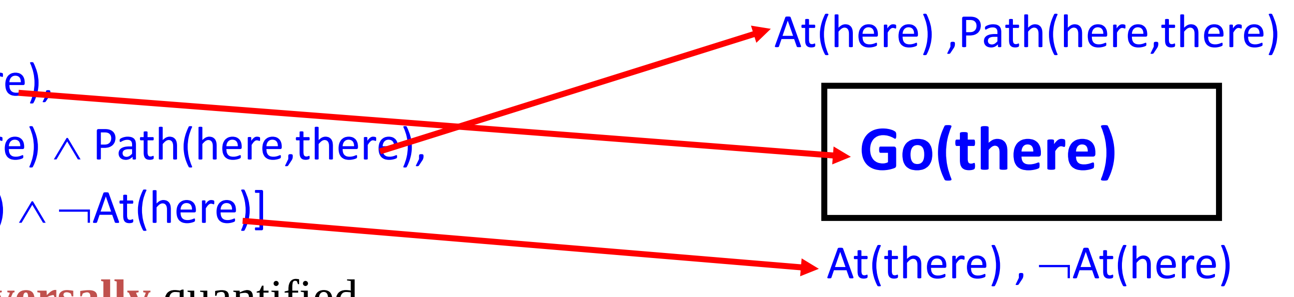
  Effect: At(there) $\wedge$ $\neg$At(here)]



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 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 will be represented by
  - Preconditions
  - New facts to be added (add-effects)
  - Facts to be removed (delete-effects)
  - A set of (simple) variable constraints (optional!)

Blocks World Operators

- So given these operations:
  - stack(X,Y), unstack(X,Y), pickup(X), putdown(X)
- Need:
  - Preconditions, facts to be added (add-effects), facts to be removed (delete-effects), optional variable constraints

Example: stack

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 II

operator(stack(X,Y),
  Precond [holding(X), clear(Y)],
  Add [handempty, on(X,Y), clear(X)],
  Delete [holding(X), clear(Y)],
  Constr [X≠Y, Y≠table, X≠table]).

operator(pickup(X),
  [ontable(X), clear(X), handempty],
  [holding(X)],
  [ontable(X), clear(X), handempty],
  [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(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

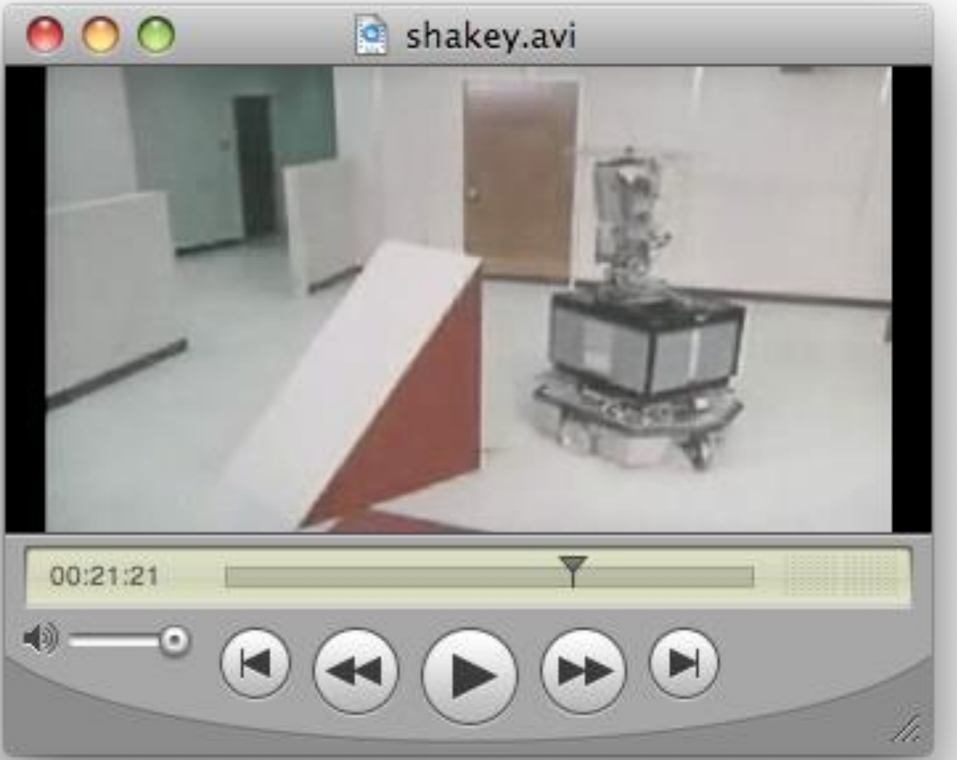
STRIPS planning

- 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 action that adds it and push action and its preconditions (subgoals) on stack

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 action that adds it and push action and its preconditions (subgoals) on stack
- When a current goal is satisfied, POP from stack
- When an action is on top stack, record its application on plan sequence and use its add and delete lists to update current state

Shakey video circa 1969

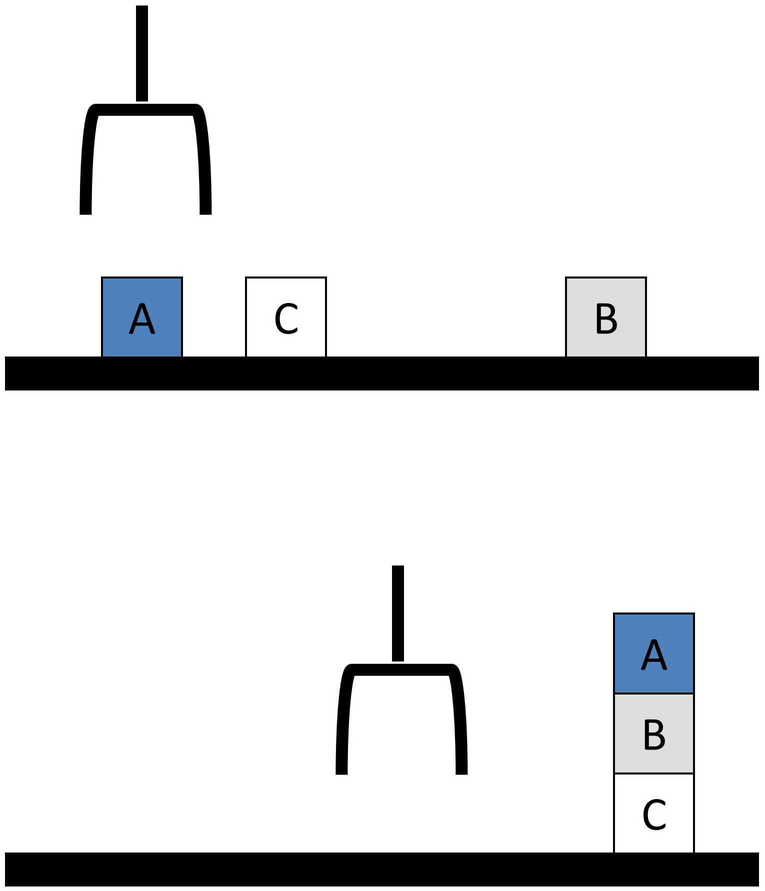


https://youtu.be/qXdn6ynwpiI or
https://youtu.be/7bsEN8mwUB8

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)

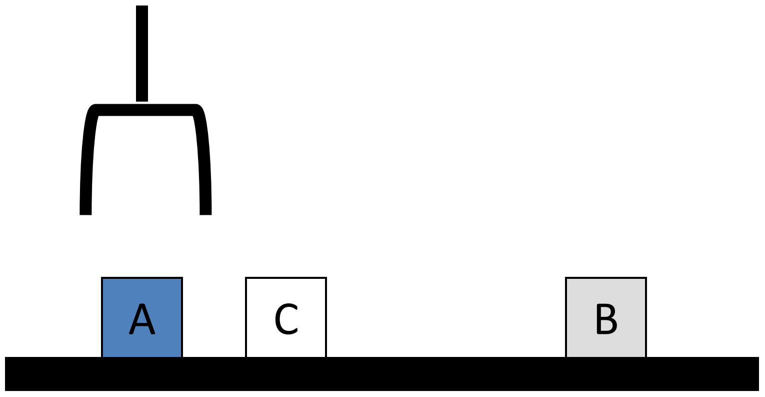


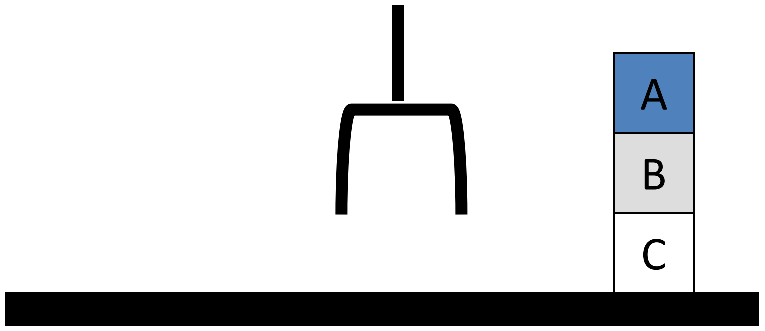
A plan:
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

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)

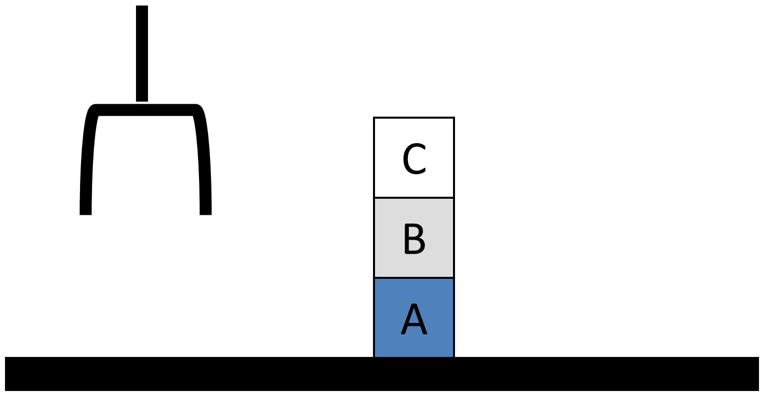
Yet Another BW planning problem

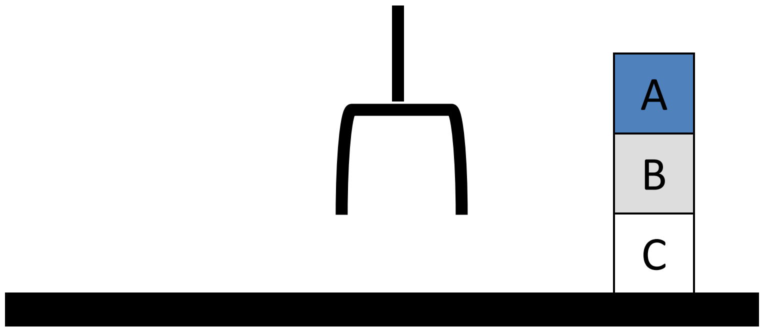
Initial state:

- 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(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)

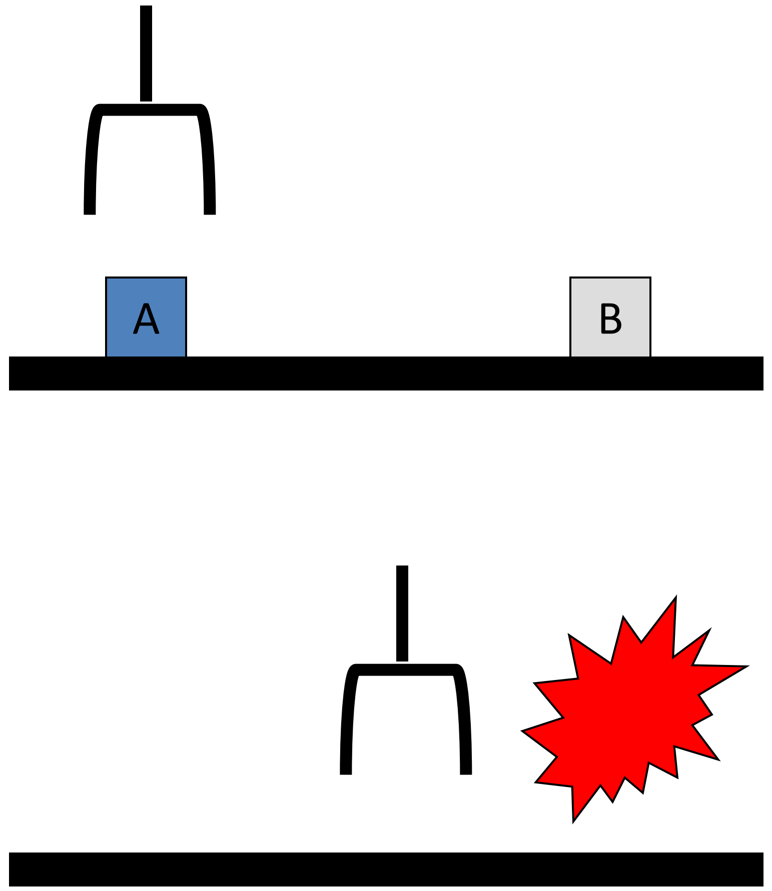
Yet Another BW planning problem

Initial state:

- ontable(a)
- ontable(b)
- clear(a)
- clear(b)
- handempty

Goal:

- on(a,b)
- on(b,a)

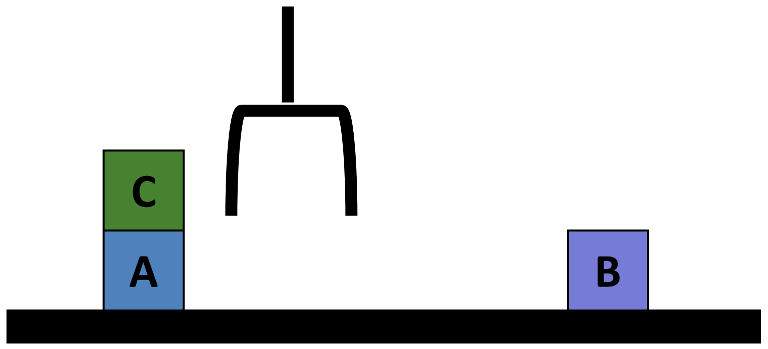


Plan:

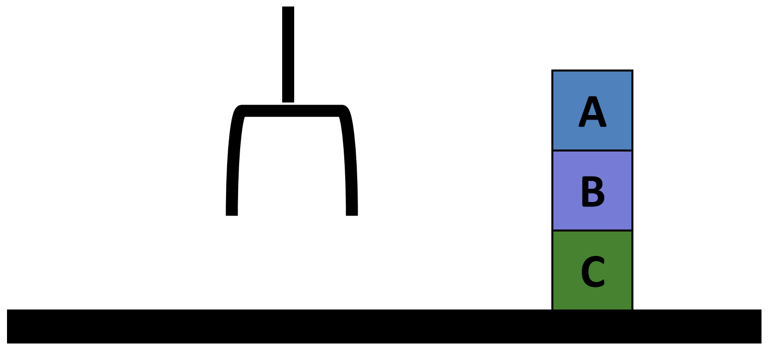
??

Goal Interactions

- Simple planning assumes that goals are **independent**
  - Each can be solved separately and then the solutions concatenated
- Let’s look at when that fails



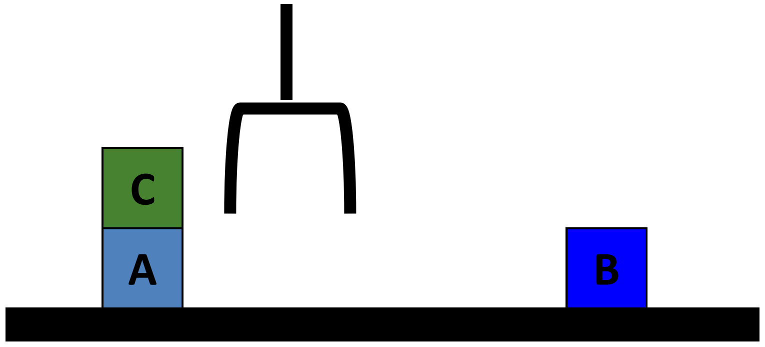
Initial state



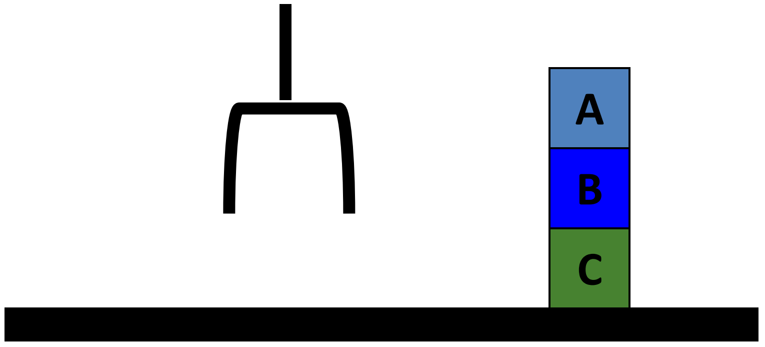
Goal state

Goal Interactions

- The “Sussman Anomaly”: classic **goal interaction problem**
  - Solving on(A,B) first (by doing unstack(C,A), stack(A,B))
  - Solve on(B,C) second (by doing unstack(A,B), stack(B,C))
- Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS can’t handle this (minor modifications can do simple cases)



Initial state

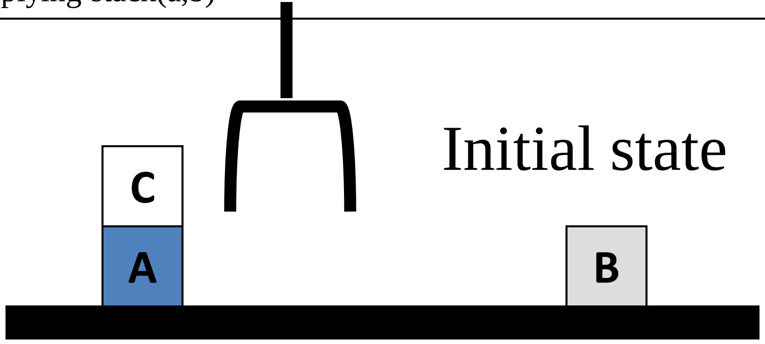


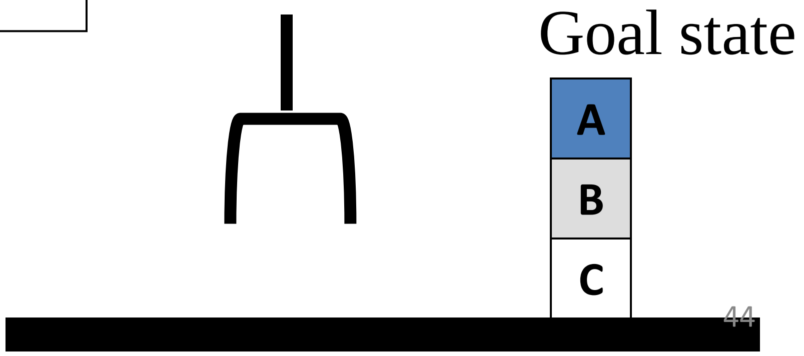
Goal state

Sussman Anomaly

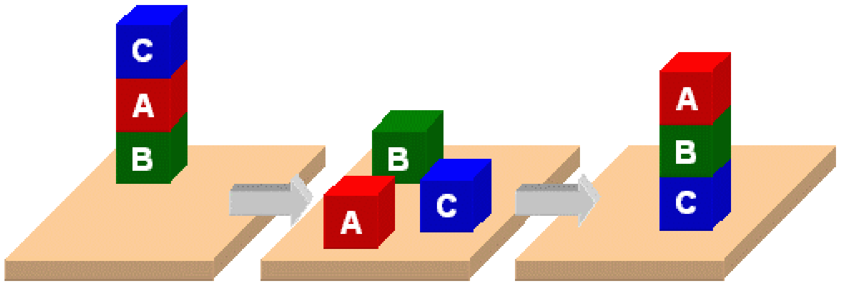
```
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
||Achieve clear(a) via unstack(_1584,a) with preconds:
[on(_1584,a),clear(_1584),handempty]
||Applying unstack(c,a)
||Achieve handempty via putdown(_2691)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





PDDL



- **Planning Domain Description Language**
- Based on STRIPS with various extensions
- First defined by Drew McDermott (Yale) et al.
  - Classic spec: PDDL 1.2; good reference guide
- Used in biennial International Planning Competition (IPC) series (1998-2020)
- Many planners use it as a standard input

PDDL Representation

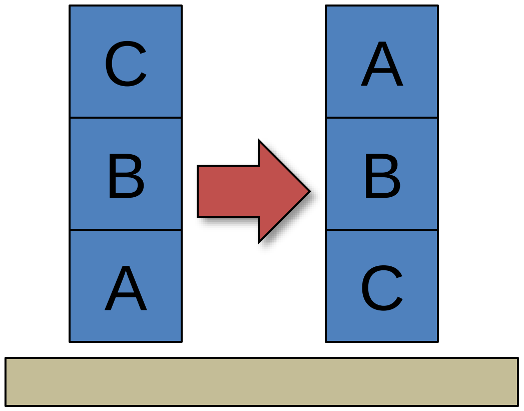
- Task specified via two files: **domain file** and **problem file**
  - Both use a logic-oriented notation with Lisp syntax
- **Domain file** defines a domain via *requirements*, *predicates*, *constants*, and *actions*
  - Used for many different problem files
- **Problem file:** defines problem by describing its *domain*, *objects*, *initial state* and *goal state*
- **Planner:** takes a domain and a problem and produces a plan

Blocks Word Domain File

```
(define (domain BW)
  (:requirements :strips)
  (:constants red green blue yellow small large)
  (:predicates (on ?x ?y) (on-table ?x) (color  ?x ?y) … (clear ?x))
  (:action pick-up
     :parameters (?obj1)
     :precondition (and (clear ?obj1) (on-table ?obj1)
                        (arm-empty))
     :effect (and (not (on-table ?obj1))
                  (not (clear ?obj1))
                  (not (arm-empty))
                  (holding ?obj1)))
  … more actions ...)
```

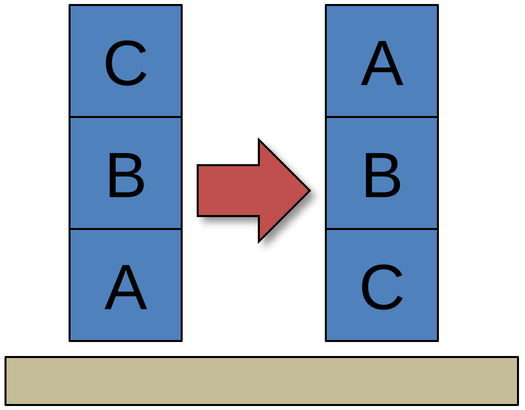
Blocks Word Problem File

```
(define (problem 00)
    (:domain BW)
    (:objects A B C)
    (:init (arm-empty)
           (on B A)
           (on C B)
           (clear C))
    (:goal (and (on A B)
                (on B C))))
```



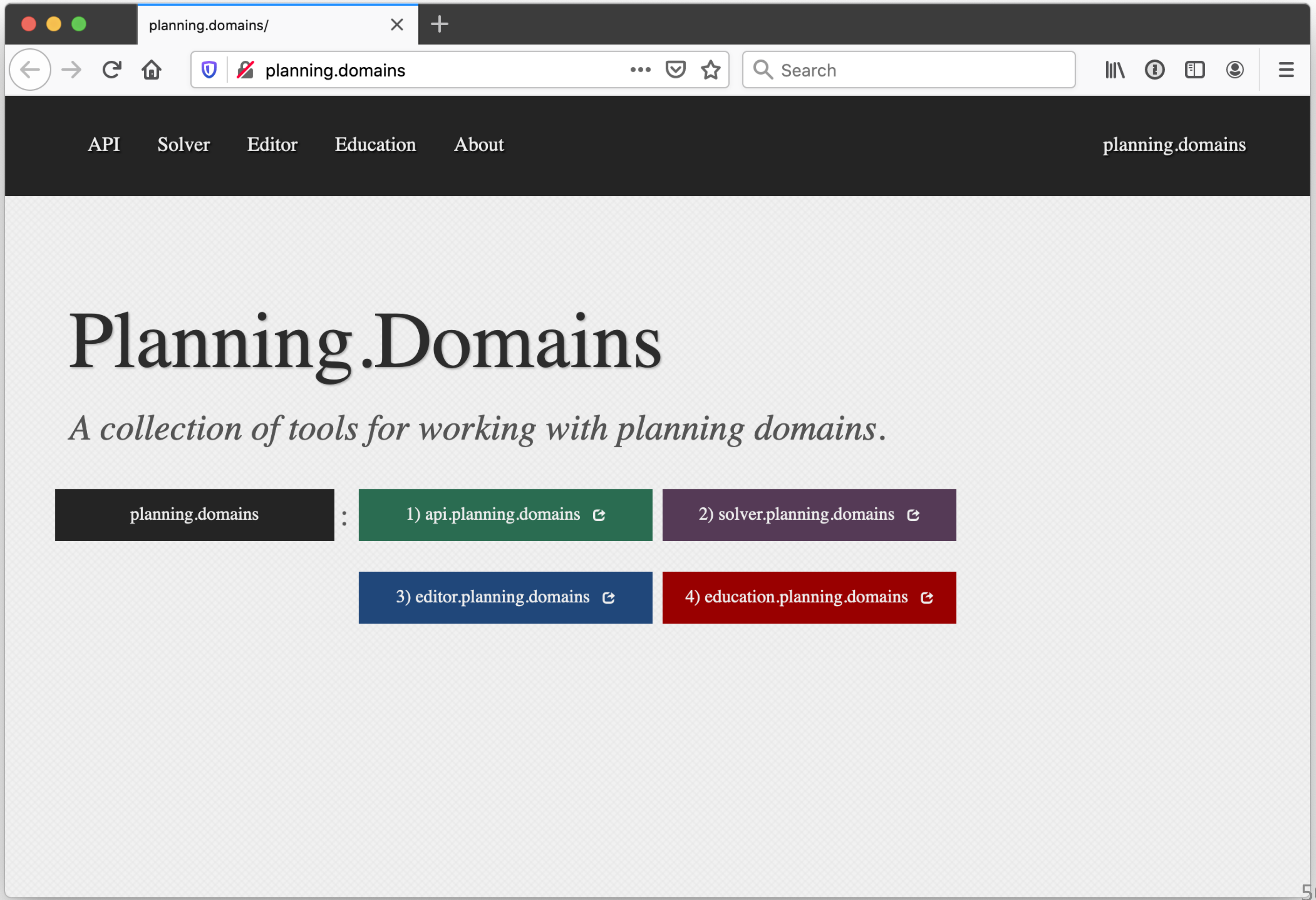
Blocks Word Problem File

```
(define (problem 00)
    (:domain BW)
    (:objects A B C)
    (:init (arm-empty)
            (on B A)
            (on C B)
            (clear C))
    (:goal (and (on A B)
                (on B C))))
```



```
Begin plan
1 (unstack c b)
2 (put-down c)
3 (unstack b a)
4 (stack b c)
5 (pick-up a)
6 (stack a b)
End plan
```

http://planning.domains/



Planning.Domains

A collection of tools for working with planning domains.

planning.domains :
1) api.planning.domains
2) solver.planning.domains
3) editor.planning.domains
4) education.planning.domains

Planning.domains

- Open source environment for providing planning services using PDDL (GitHub)
- Default planner is ff
  - very successful forward-chaining heuristic search planner producing sequential plans
  - Can be configured to work with other planners
- Use interactively or call via web-based API

State-Space Planning

- STRIPS searches thru a space of situations (where you are, what you have, etc.)
- Find plan by searching **situations** to reach goal
- **Progression planner**: searches forward
  - From initial state to goal state
- **Regression planner**: searches backward from goal
  - Works **iff** operators have enough information to go both ways
  - Ideally leads to reduced branching: planner is only considering things that are relevant to the goal

Planning Heuristics

- Need an **admissible** heuristic to apply to planning states
  - Estimate of the distance (number of actions) to the goal
- Planning typically uses **relaxation** to create heuristics
  - Ignore all or some selected preconditions
  - Ignore delete lists: Movement towards goal is never undone)
  - Use state abstraction (group together "similar" states and treat them as though they are identical) – e.g., ignore fluents*
  - Assume subgoal independence (use max cost; or, if subgoals actually are independent, sum the costs)
  - Use pattern databases to store exact solution costs of recurring subproblems

* an aspect of the world that changes - *R&N 266*

Plan-Space Planning

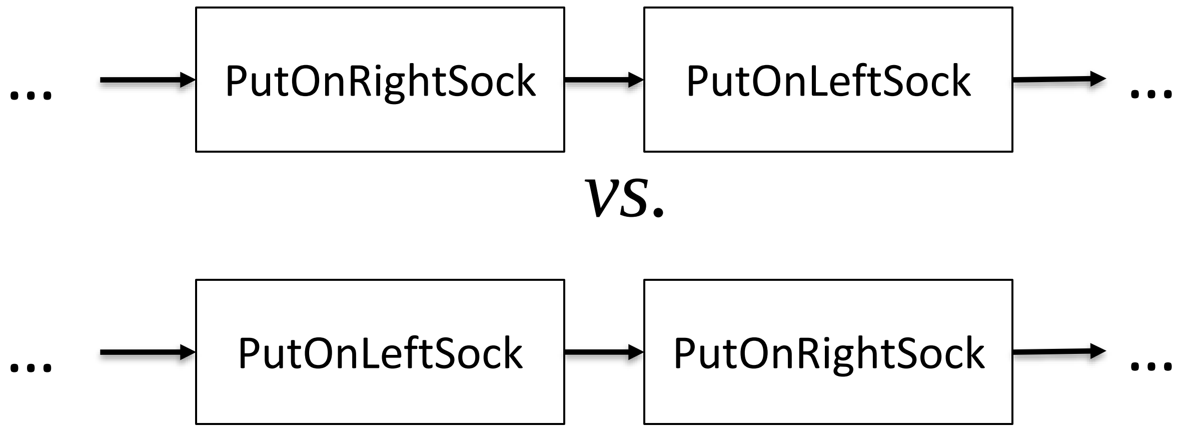
- Alternative: **search through space of *plans***, not situations
- Start from a **partial plan**; expand and refine until a complete plan that solves the problem is generated
- **Refinement operators** add constraints to the partial plan and modification operators for other changes
- We can still use STRIPS-style operators:

  Op(ACTION: PutOnRightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)
  Op(ACTION: PutOnRightSock, EFFECT: RightSockOn)
  Op(ACTION: PutOnLeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)
  Op(ACTION: PutOnLeftSock, EFFECT: LeftSockOn)

Partial-Order Planning

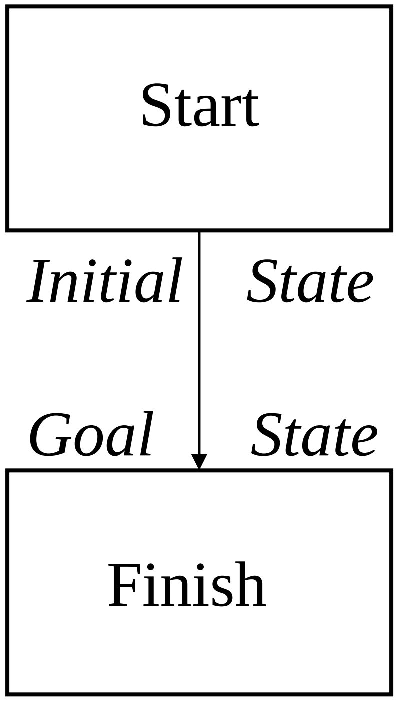
Partial-Order Planning

- The big idea: Don’t specify the order of steps if you don’t have to.

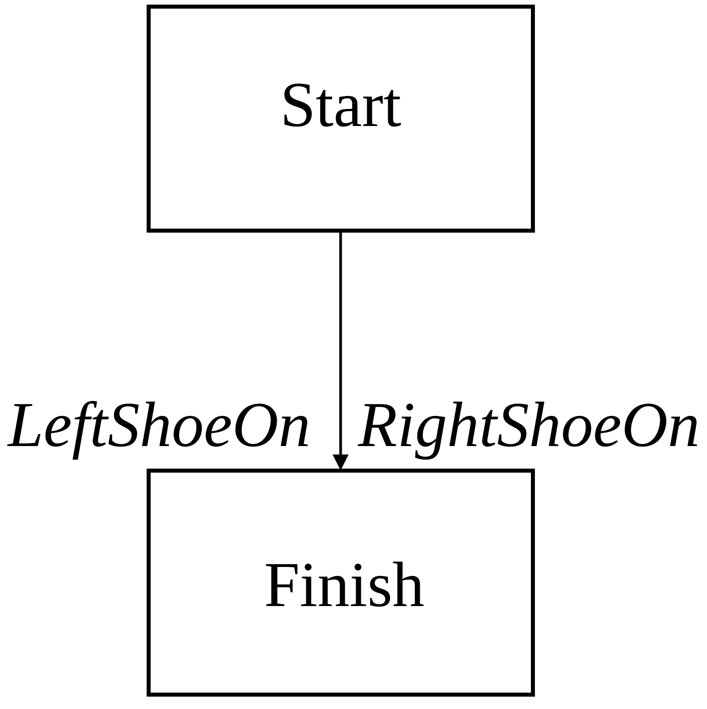


- Doesn’t matter, but a regular planner has to consider and specify all the options.

A simple graphical notation



(a)



(b)

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
  - E.g., S1<S2 (step S1 must come before S2)

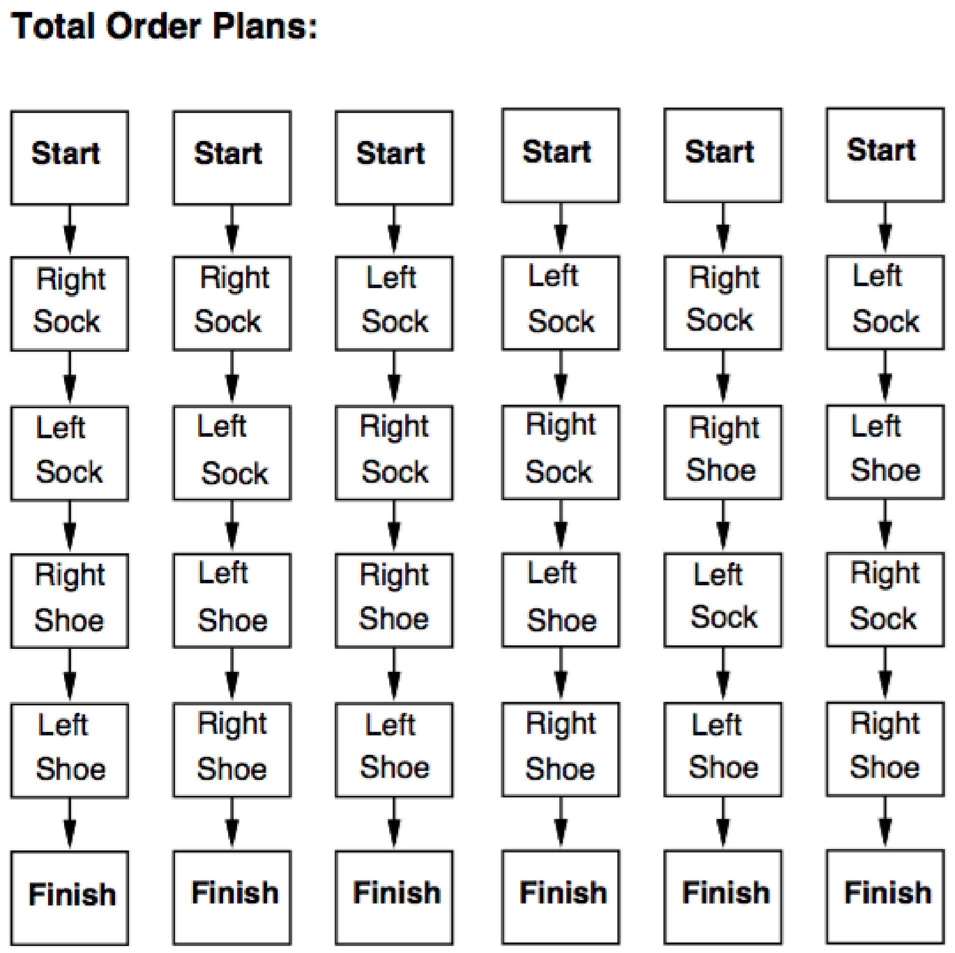
PutOnRightSock < PutOnRightShoe

The order here *does* matter, so the planner has to know that.

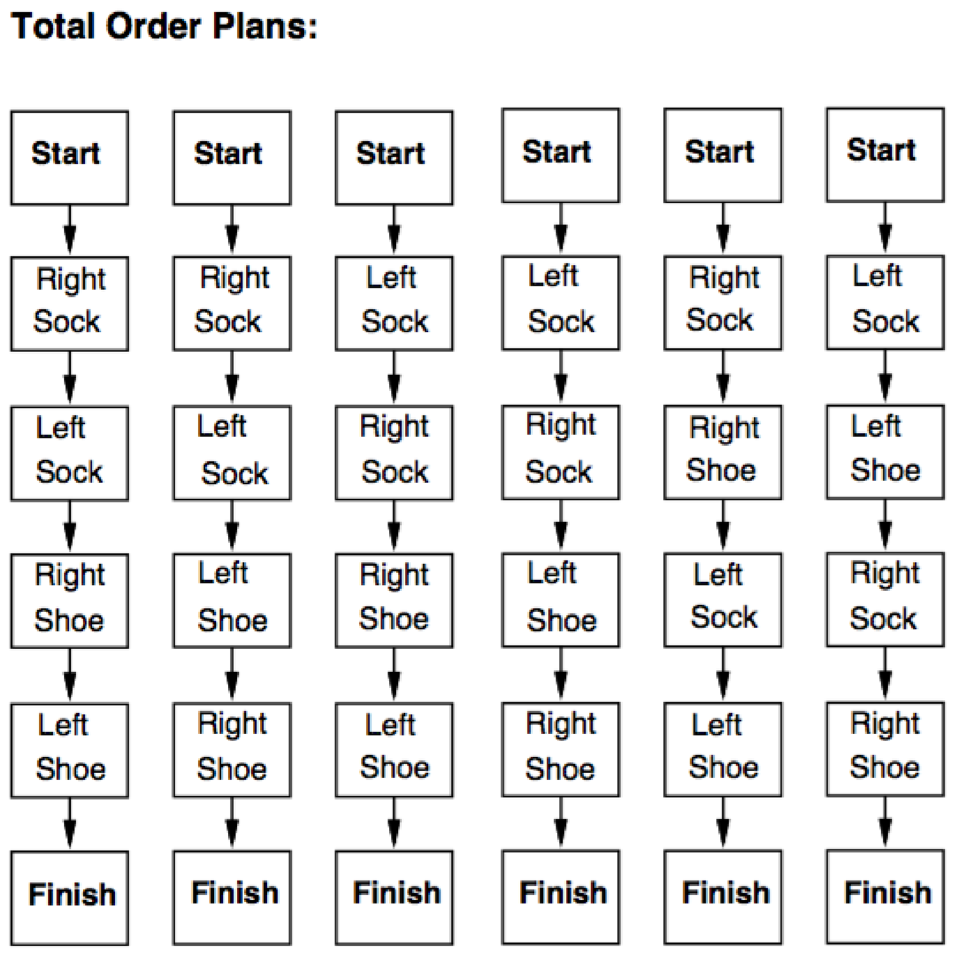
Partial-Order Planning

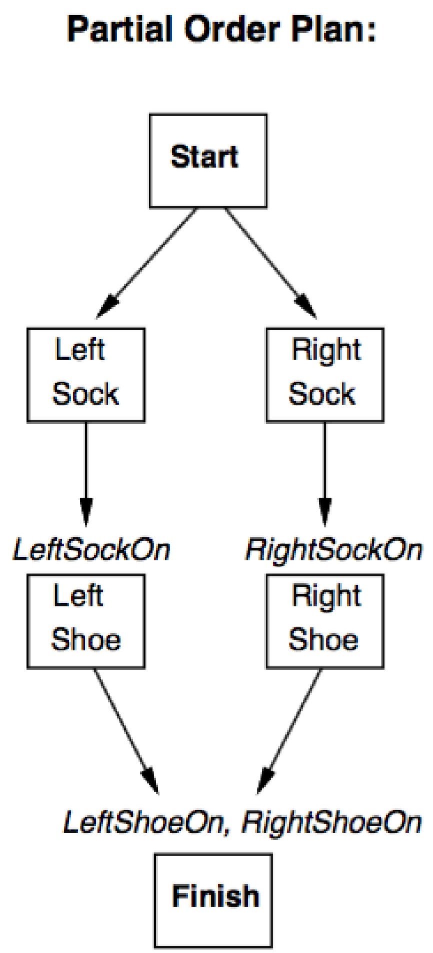
- A **linear planner** builds a plan as a **totally ordered sequence** of plan steps
- A **non-linear planner (aka partial-order planner)** builds up a plan as a set of steps with some temporal constraints
  - E.g., S1<S2 (step S1 must come before S2)
- Partially ordered plan (POP) **refined** by either:
  - adding a new **plan step**, or
  - adding a new **constraint** to the steps already in the plan.
- Linearize a POP by topological sort

Linear *vs.* POP: Shoes

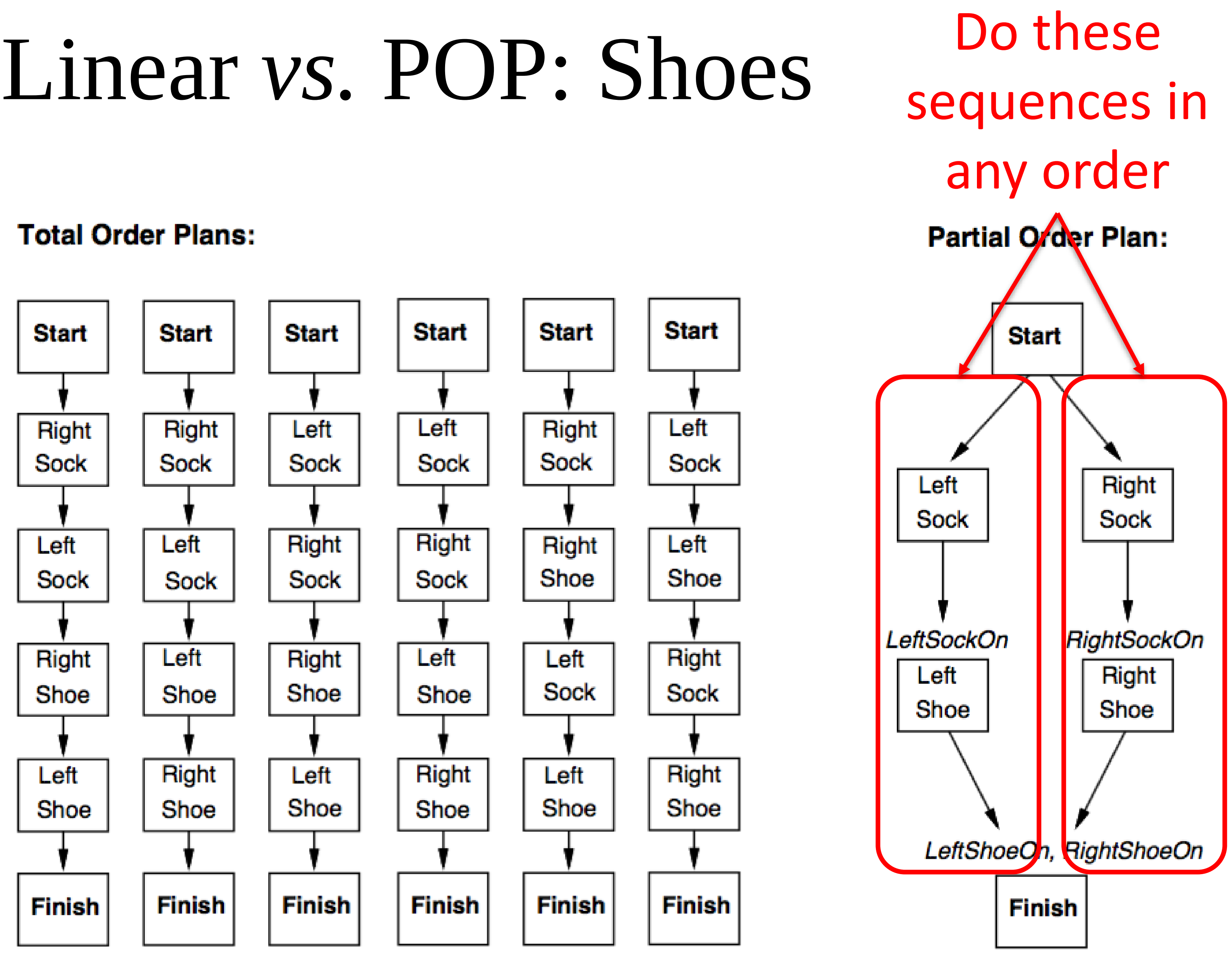


Linear *vs.* POP: Shoes



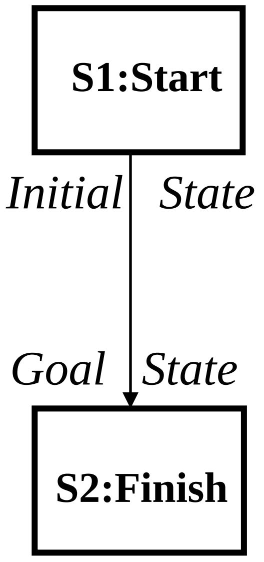


Linear *vs.* POP: Shoes



The Initial Plan

Every plan starts the same way



Least Commitment

- Non-linear planners embody the principle of **least commitment**
  - Only choose actions, orderings and variable bindings absolutely necessary, postponing other decisions
  - Avoid 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

Non-Linear Plan Components

1) A set of **steps** $\{S_1, S_2, S_3, S_4 \ldots\}$
   - Each step has an **operator description**, **preconditions** and **post-conditions**
   - ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn

2) A set of **causal links** $\{ \ldots (S_i,C,S_j) \ldots\}$
   - (One) goal of step S_i is to achieve precondition C of step S_j
   - ⟨PutOnLeftShoe, LeftShoeOn, Finish⟩
     - This says: No action that undoes LeftShoeOn is allowed to happen after PutOnLeftShoe and before Finish. Any action that undoes LeftShoeOn must either be before PutOnLeftShoe or after Finish.

3) A set of **ordering constraints** $\{ \ldots S_i<S_j \ldots \}$
   - If step S_i must come before step S_j
   - PutOnSock < Finish

Non-Linear Plan: Completeness

- A non-linear plan consists of
  (1) A set of **steps** $\{S_1, S_2, S_3, S_4 \ldots\}$
  (2) A set of **causal links** $\{ \ldots (S_i,C,S_j) \ldots\}$
  (3) A set of **ordering constraints** $\{ \ldots S_i<S_j \ldots \}$
- 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 $S_j<S_k$

Trivial Example

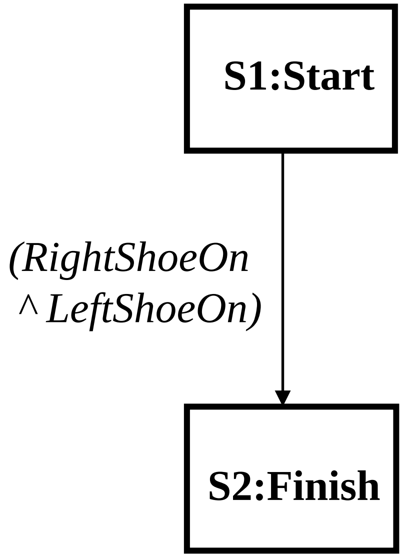
Operators:

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

Op(ACTION: RightSock, EFFECT: RightSockOn)

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

Op(ACTION: LeftSock, EFFECT: leftSockOn)



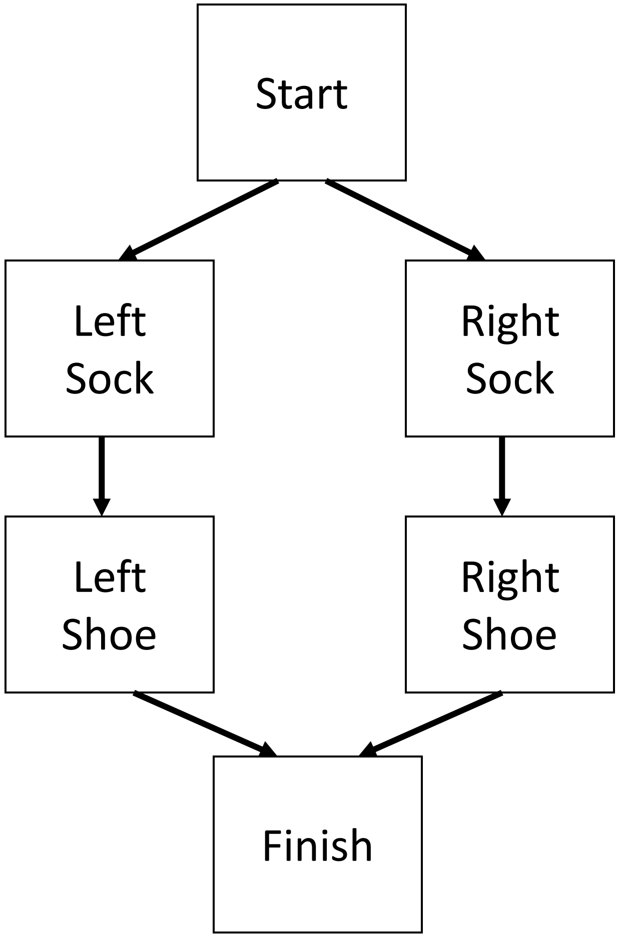
Steps: {S1:[Op(Action:Start)],

S2:[Op(Action:Finish,
Pre:
RightShoeOn^LeftShoeOn)]}

Links: {}

Orderings: {S1<S2}

Solution

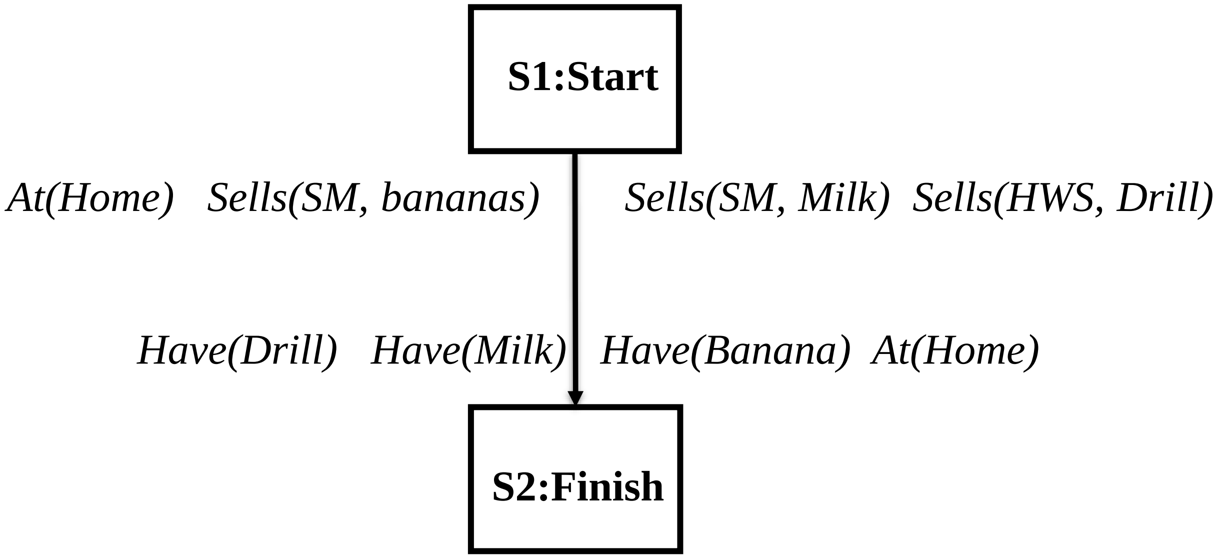


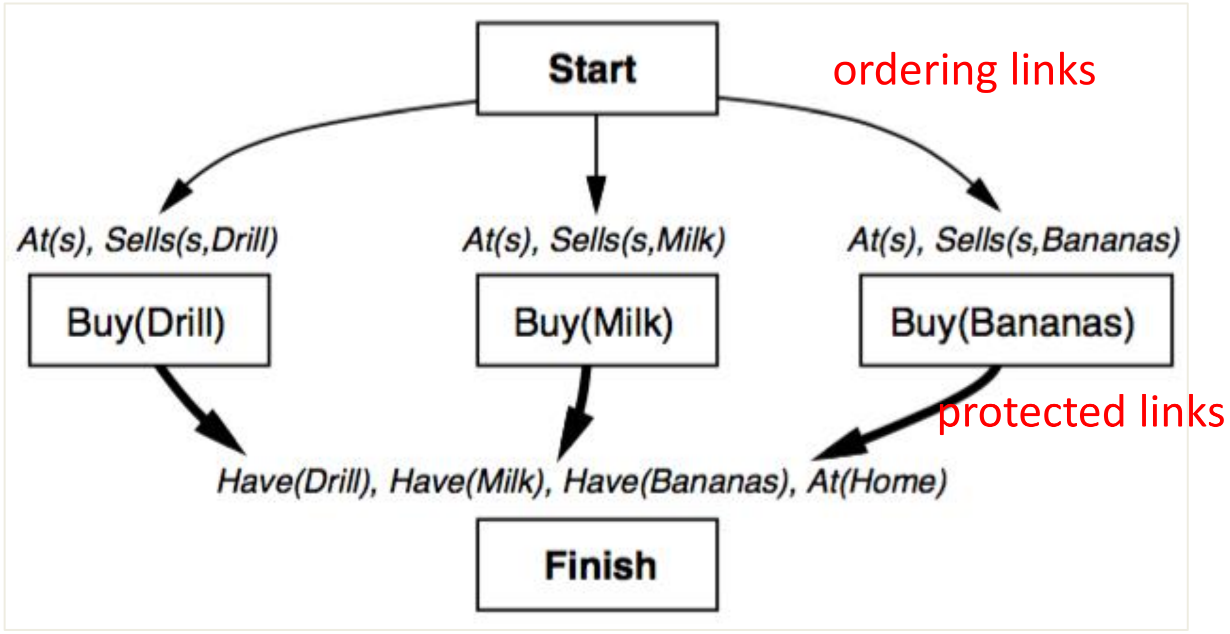
POP Constraints and Search Heuristics

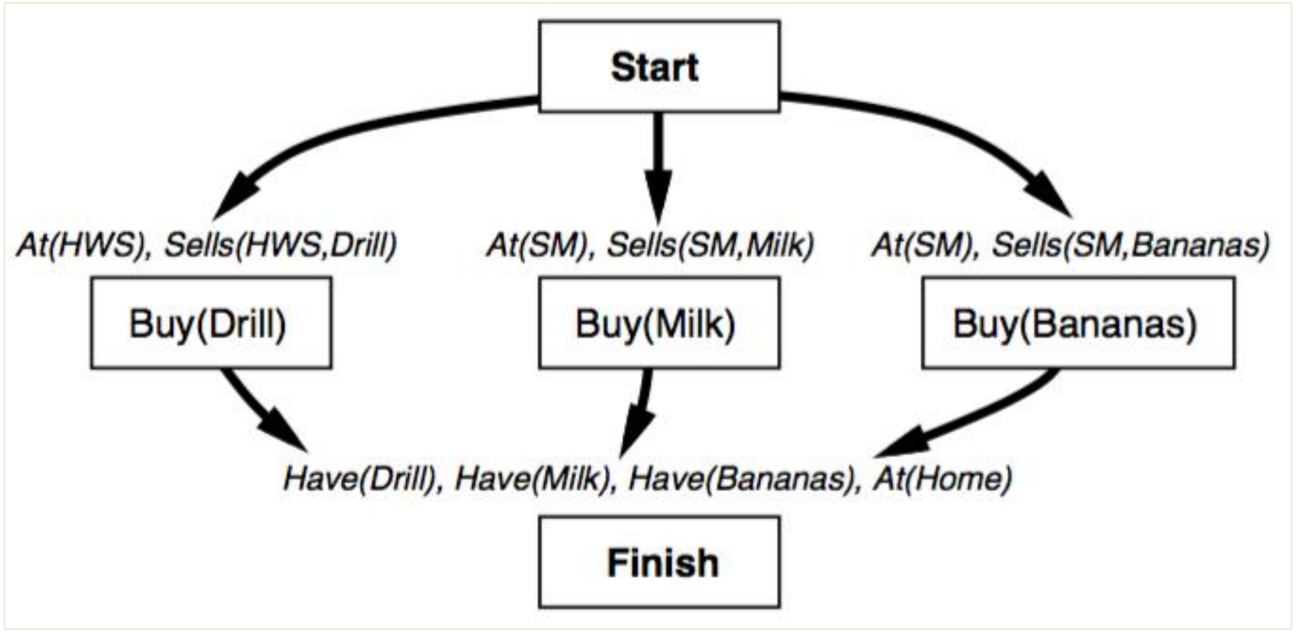
- Only add steps that reach a not-yet-achieved precondition
- Use a least-commitment approach:
  - Don’t order steps unless they need to be ordered
- Honor causal links $S_1 \xrightarrow[c]{} S_2$ that **protect** a condition *c*:
  - Never add an intervening step S_3 that violates *c*
  - If a parallel action **threatens** *c* (i.e., has the effect of negating or **clobbering** *c*), resolve that threat by adding ordering links:
    - Order S_3 before S_1 (**demotion**)
    - Order S_3 after S_2 (**promotion**)

Partial-Order Planning Example

- **Initially:** at home; SM sells bananas; SM sells milk; HWS sells drills
- **Goal:** Be home with milk, bananas, and a drill

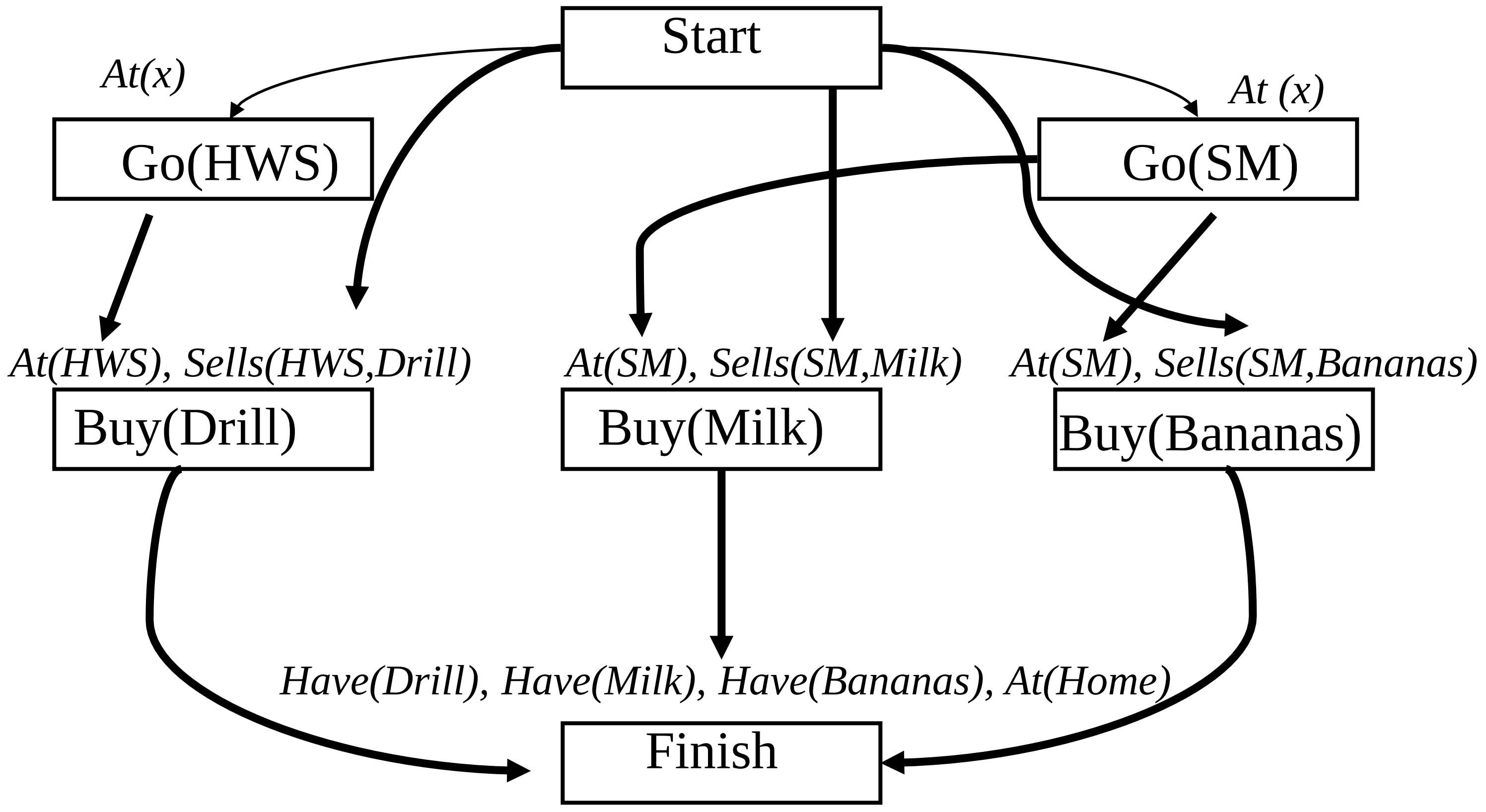

S1:Start
At(Home) Sells(SM, bananas) Sells(SM, Milk) Sells(HWS, Drill)
Have(Drill) Have(Milk) Have(Banana) At(Home)
S2:Finish

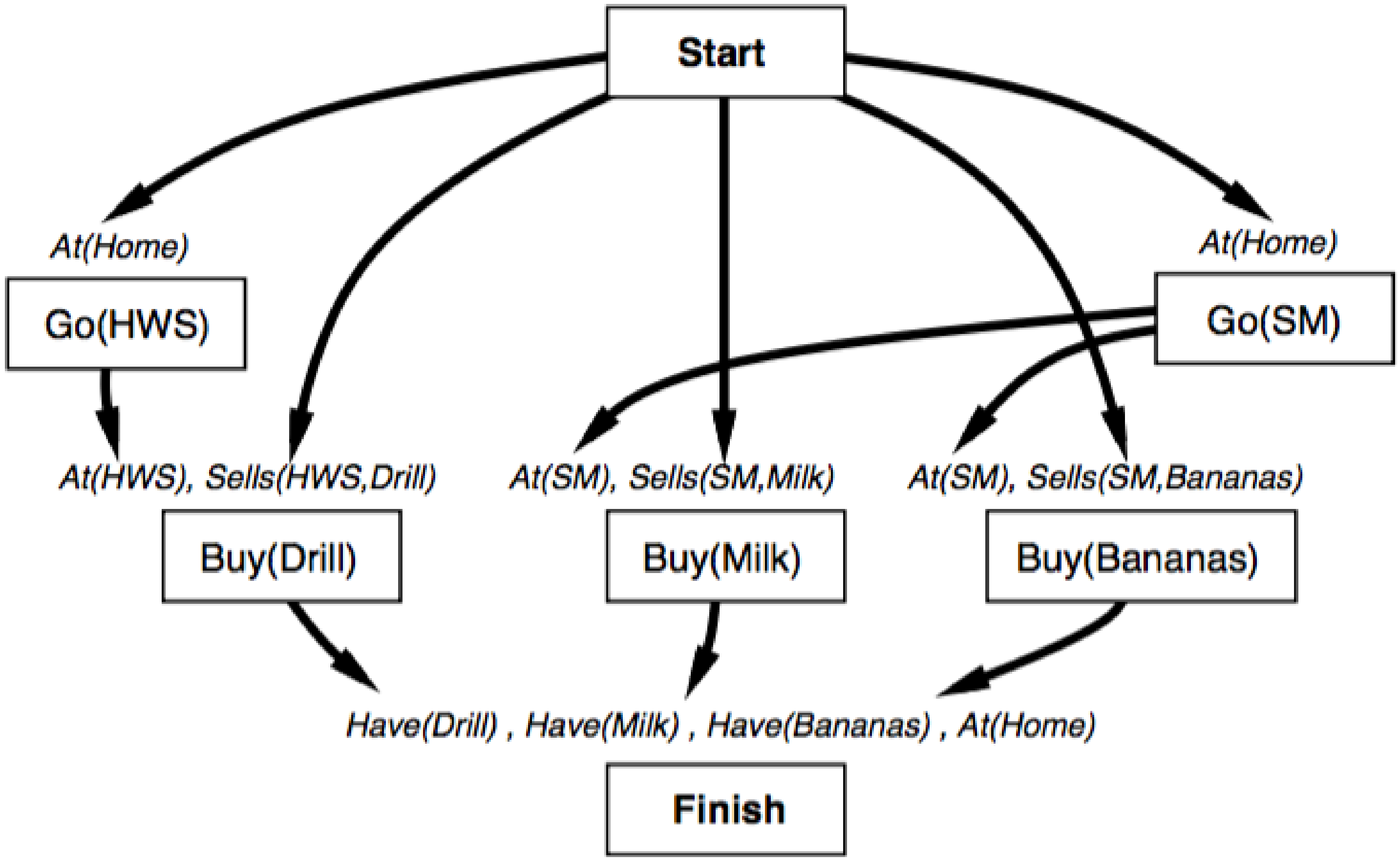



Start
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish

- Add three actions to achieve basic goals
- Use initial state to achieve the "Sells" preconditions
- Bold links are causal (protected), regular are just ordering constraints

Planning


Start
At(x)
At (x)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish


Start
At(Home)
At(Home)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill) , Have(Milk) , Have(Bananas) , At(Home)
Finish

Real-World Planning Domains

- Real-world domains are complex
- Don’t satisfy assumptions of STRIPS or partial-order planning methods
- Some of the characteristics we may need to deal with:
  - Modeling and reasoning about resources
  - Representing and reasoning about time
  - Planning at different levels of abstractions

  } Scheduling

  - Conditional outcomes of actions
  - Uncertain outcomes of actions
  - Exogenous events

  } Planning under uncertainty

  - Incremental plan development
  - Dynamic real-time replanning

  } HTN planning

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, HTNs, …)
  - *Constraint-based search (GraphPlan, SATplan, …)*
- Search strategies
  - Forward planning
  - Goal regression
  - Backward planning
  - Least-commitment
  - Nonlinear planning

Extended PDDL Examples

Classic Blocks World

- Starting with
  - BW: a domain file
  - Several problem files
- Use planning.domains to demonstrate solving the problems

bw.pddl 1

```
(define (domain bw)

 (:requirements :strips)

 (:predicates
    (on ?x ?y)        ; object ?x is on ?object ?y
    (on-table ?x)   ; ?x is directly on the table
    (clear ?x)         ; ?x has nothing on it
    (arm-empty)   ; robot isn't holding anything
    (holding ?x))   ; robot is holding ?x

 ;; the four classic actions for manipulating objects
 … actions in next four slides …
```

Allows basic add and delete effects in actions

List all the predicates with their arguments

bw.pddl 2

```
(:action pick-up
   :parameters (?ob1)

   :precondition
      (and (clear ?ob1)
           (on-table ?ob1)
           (arm-empty))

   :effect
      (and (not (on-table ?ob1))
           (not (clear ?ob1))
           (not (arm-empty))
           (holding ?ob1)))
```

Variable for the argument of a pick-up action

These three statements must be True before we can do a pick-up action

After doing a pick-up action, these become True

bw.pddl 3

```
(:action pick-up
    :parameters (?ob1)

    :precondition
        (and (clear ?ob1)
             (on-table ?ob1)
             (arm-empty))

    :effect
        (and (not (on-table ?ob1))
            (not (clear ?ob1))
            (not (arm-empty))
            (holding ?ob1)))
```

Variable for the argument of a pick-up action

These three statements must be True before we can do a pick-up action

After doing a pick-up action, these become True

bw.pddl 4

```
(:action put-down
   :parameters (?ob)
   :precondition (holding ?ob)
   :effect
      (and (not (holding ?ob))
           (clear ?ob)
           (arm-empty)
           (on-table ?ob)))
```

put-down means put the think you are holding on the table

```
(:action stack
   :parameters (?ob ?underob)
   :precondition (and (holding ?ob) (clear ?underob))
   :effect
      (and (not (holding ?ob))
           (not (clear ?underob))
           (clear ?ob)
           (arm-empty)
           (on ?sob ?underob)))
```

stack means put the thing you are holding on another object

bw.pddl 5

```
(:action unstack
  :parameters (?sob ?sunderob)
  :precondition
    (and (on ?sob ?sunderob)
         (clear ?sob)
         (arm-empty))
  :effect
    (and (holding ?sob)
         (clear ?sunderob)
         (not (clear ?sob))
         (not (arm-empty))
         (not (on ?sob ?sunderob)))

) ; this closes the domain definition
```

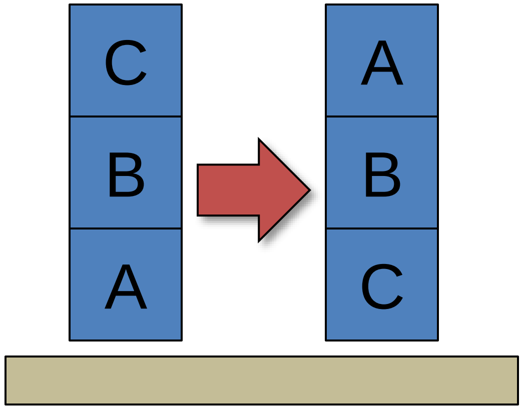
unstack means take the first arg off the second arg

First arg can't have anything on it and the robt cannot be holding anything

Here are the updates to our knowledge base describing the state of the world

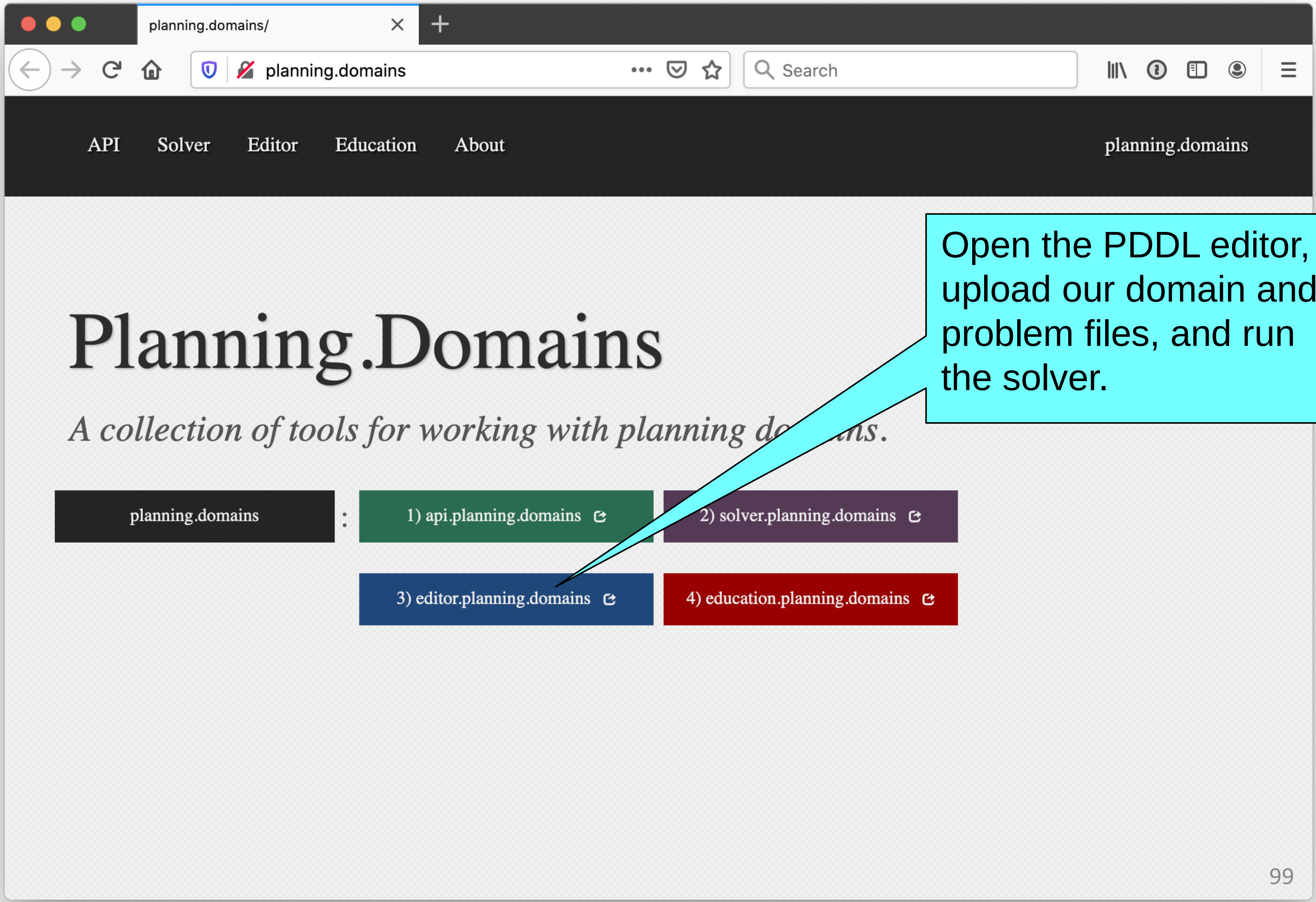
```
;; The arm is empty and there is a stack of three blocks: C is on B which is on A
;;  which is on the table.  The goal is to reverse the stack, i.e., have A on B and B
;;  on C.  No need to mention C is on the table, since domain constraints will enforce it.

(define (problem p03)
  (:domain bw)
  (:objects A B C)
  (:init (arm-empty)
          (on-table A)
          (on B A)
          (on C B)
          (clear C))
  (:goal (and (on A B)
              (on B C))))
```



p03.pddl

http://planning.domains/



Planning.Domains

A collection of tools for working with planning domains.

planning.domains
1) api.planning.domains
2) solver.planning.domains
3) editor.planning.domains
4) education.planning.domains

Open the PDDL editor, upload our domain and problem files, and run the solver.