

Planning

Chapter 11.1-11.3

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**.

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

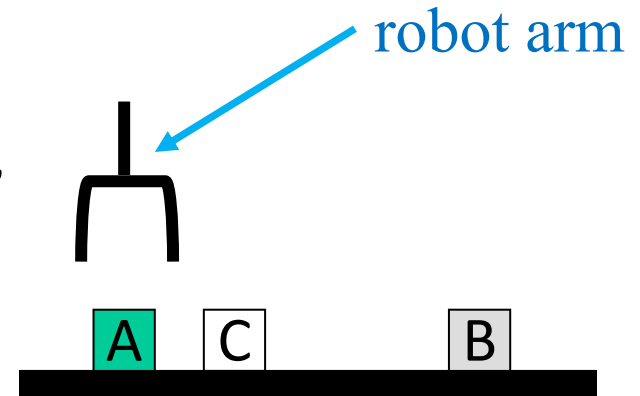
Typical Assumptions (2)

- 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

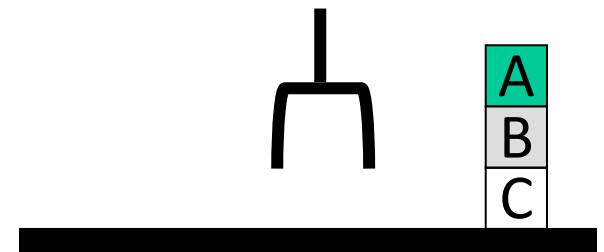
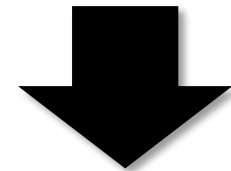
Classic Planning

Find **sequence of actions** to reach a **goal** in a discrete, deterministic, static, fully-observable environment

- State space search and logical reasoning could be used
- But classic planning developed custom representations & algorithms to do it more effectively
- The approach uses a **knowledge base** and reasoning about the state of the world and possible actions
- We'll look first at doing this in the simple **blocks world**

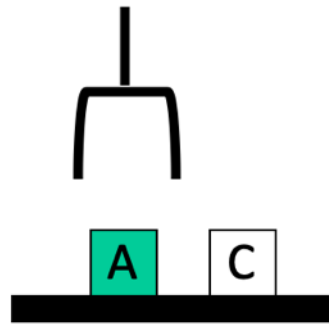


Initial State



Goal State

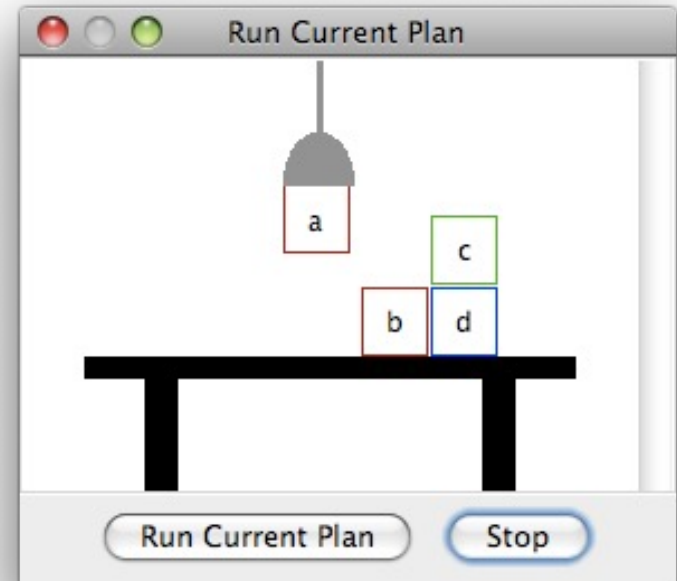
Blocks world



The blocks world is a “micro-world” with a **table**, a set of **blocks**, and a **robot hand**

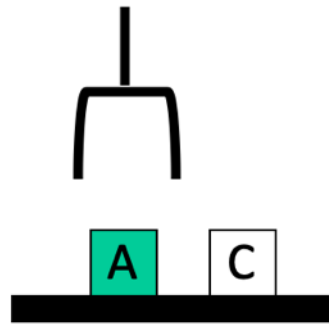
Some constraints for a simple model:

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



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

Blocks world



Typical representation uses a logic notation to represent the state of the world:

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

And possible **actions/ operators** with their preconditions and effects:

Pickup	Putdown
Stack	Unstack

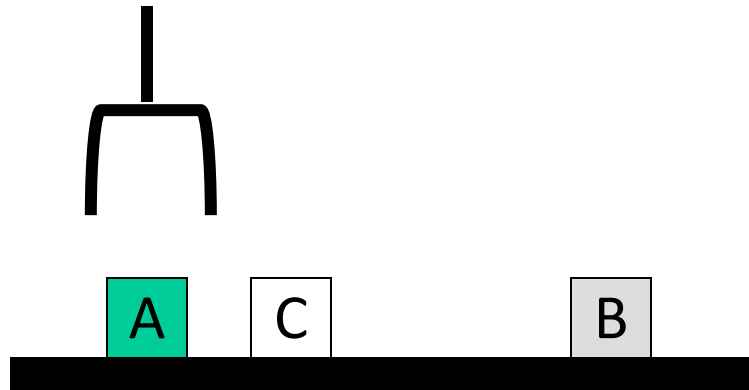
Typical BW planning problem

Initial state:

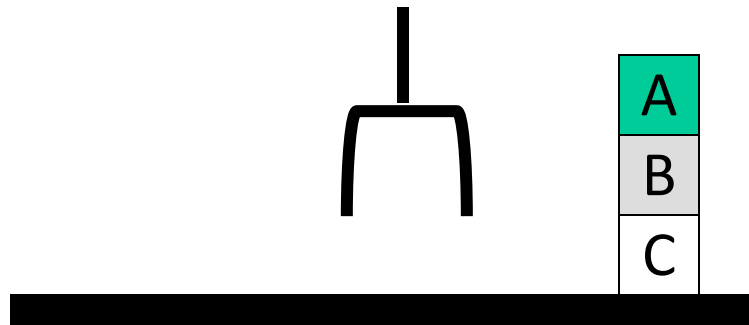
clear(a)
clear(b)
clear(c)
ontable(a)
ontable(b)
ontable(c)
handempty

Goal:

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



Initial state asserts everything that's true initially



Goal state asserts things we want to be true eventually

Typical BW planning problem

Initial state:

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

Goal state:

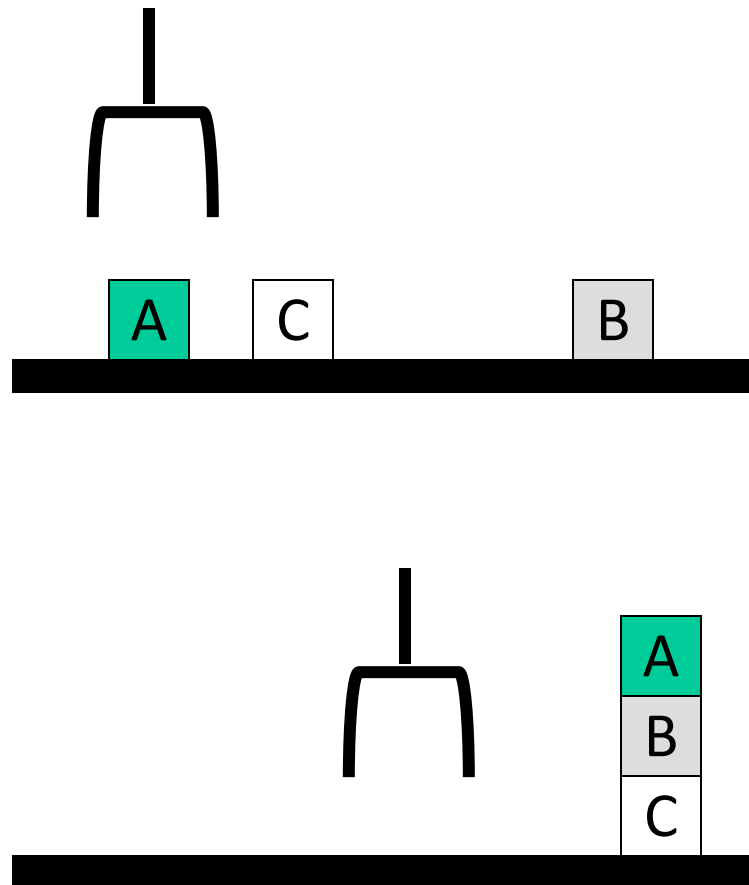
on(b,c)
on(a,b)
ontable(c)

Logical assertions/ **KB**
describing initial &
final states

Sequence
of robot
actions

Plan:

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



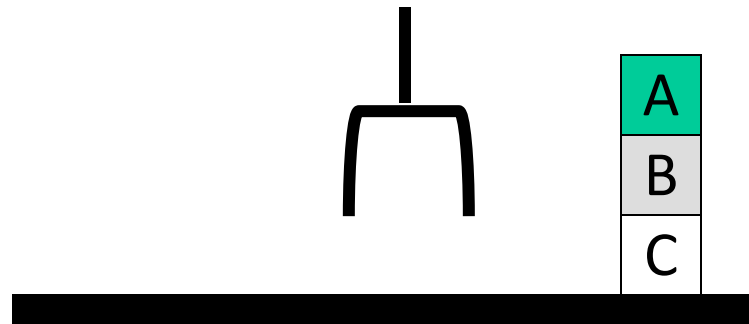
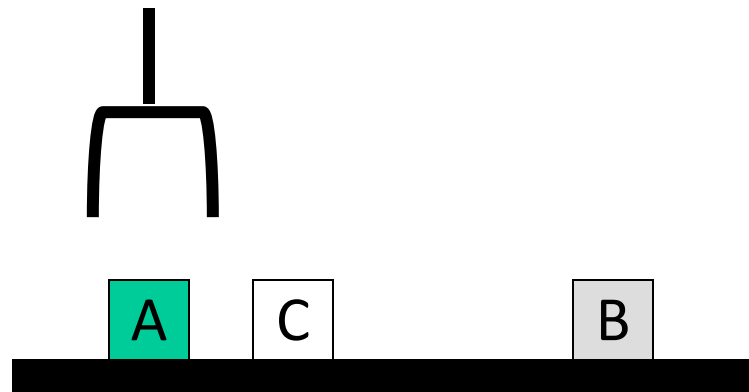
Another BW planning problem

Initial state:

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

Goal:

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



Simple approach:

- find a way to achieve each goal in order

Note: Goals in a different order!

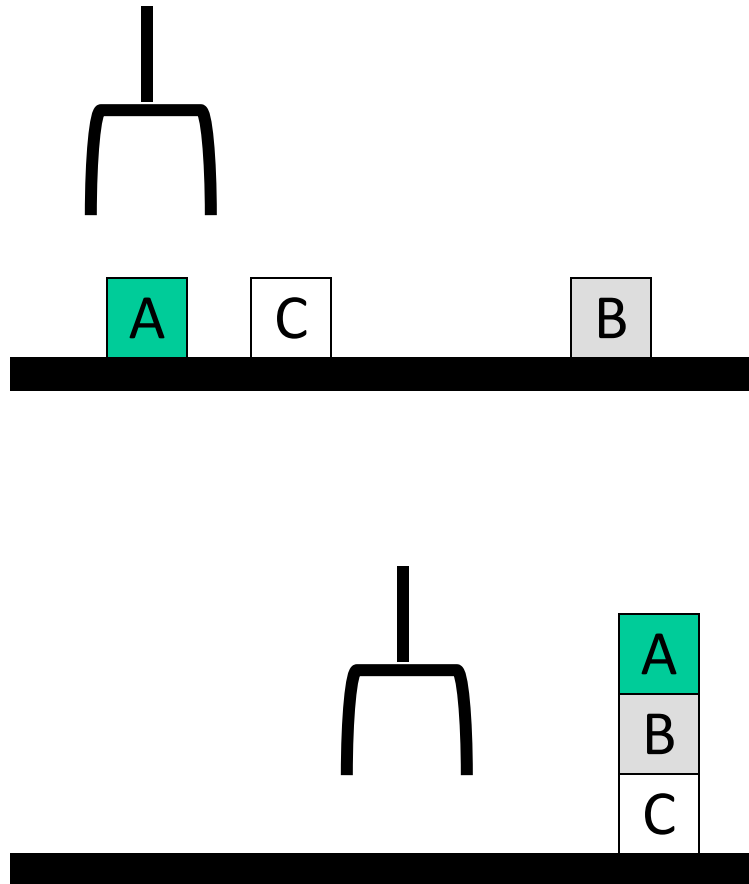
Another BW planning problem

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)

Note: Goals in a different order!

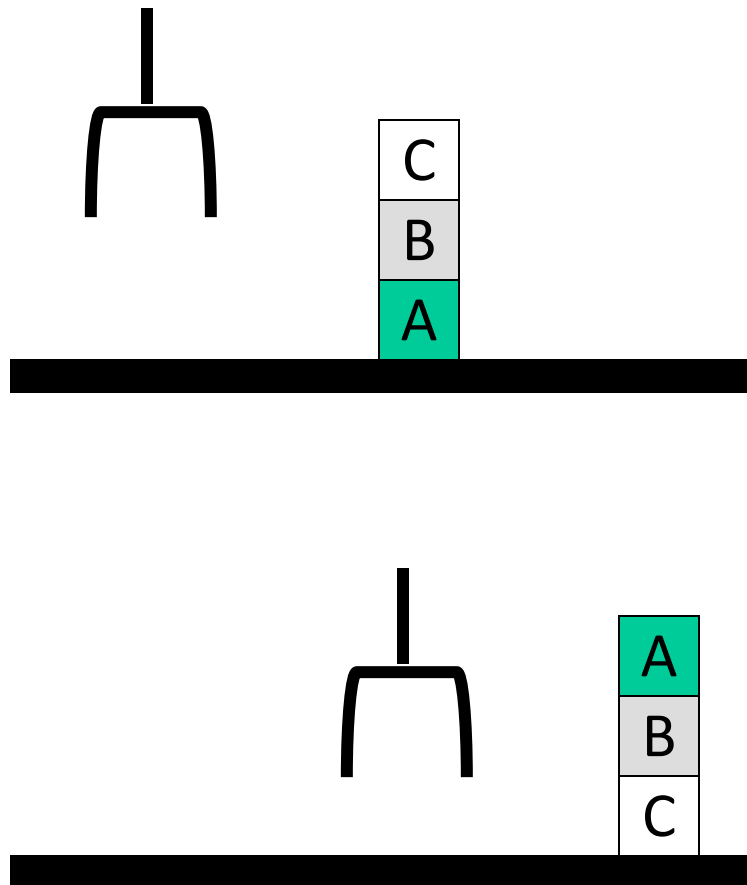
Yet Another BW planning problem

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)

Note: not very
efficient!

Planning vs. problem solving

- Problem solving methods 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

Major Approaches

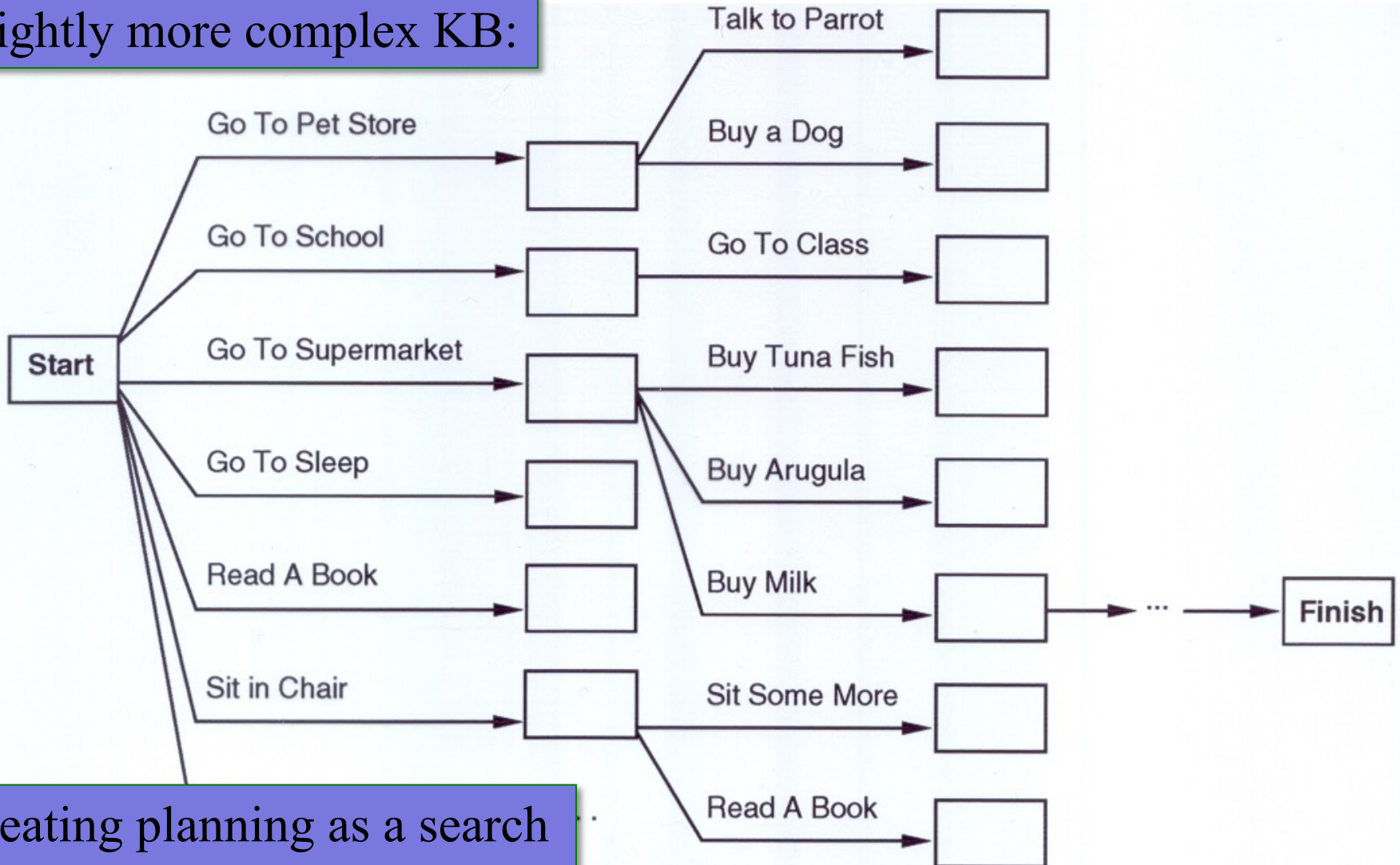
- **Planning as search**
- **GPS / STRIPS**
- **Situation calculus**
- **Partial order planning**
- **Hierarchical decomposition (HTN planning)**
- **Planning with constraints (SATplan, Graphplan)**
- *Reactive planning*

Planning as Search (?)

- 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

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

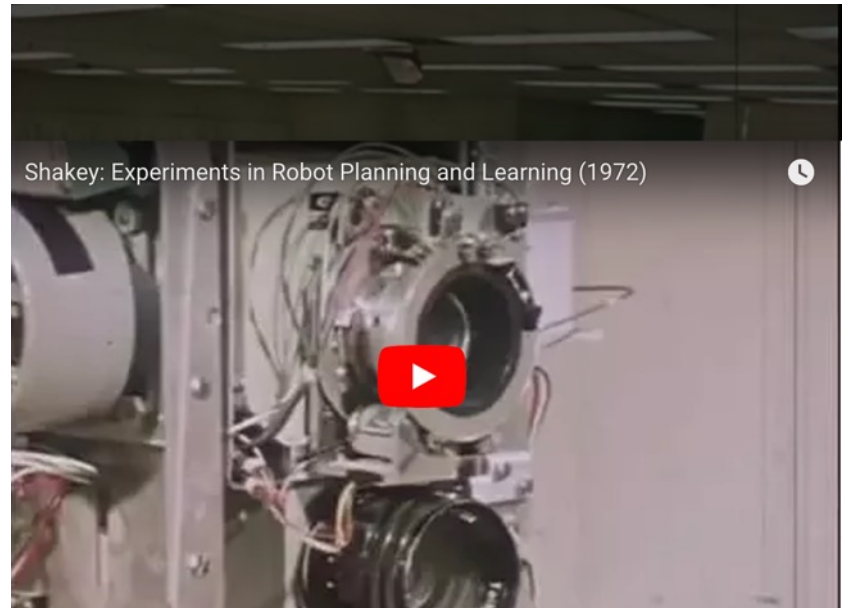
* an aspect of the world that changes - R&N 266

History: Shakey the robot

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



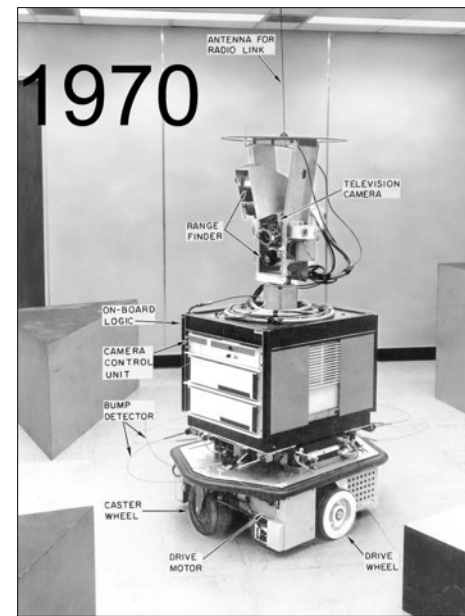
Shakey the Robot: 1st Robot to Embody Artificial Intelligence (2017, 6 min.)



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

Strips planning representation

- Classic approach first used in the [STRIPS](#) (Stanford Research Institute Problem Solver) planner
- A State is a conjunction of ground literals
 $\text{at}(\text{Home}) \wedge \neg \text{have}(\text{Milk}) \wedge \neg \text{have}(\text{bananas}) \dots$
- Goals are conjunctions of literals, but may have variables, assumed to be existentially quantified
 $\text{at}(\text{?x}) \wedge \text{have}(\text{Milk}) \wedge \text{have}(\text{bananas}) \dots$
- 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



[Shakey the robot](#)

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**

Blocks World Stack Action

stack(X,Y):

- **preconditions**(stack(X,Y), [holding(X), clear(Y)])
- **adds**(stack(X,Y), [handempty, on(X,Y), clear(X)])
- **deletes**(stack(X,Y), [holding(X), clear(Y)]).
- **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

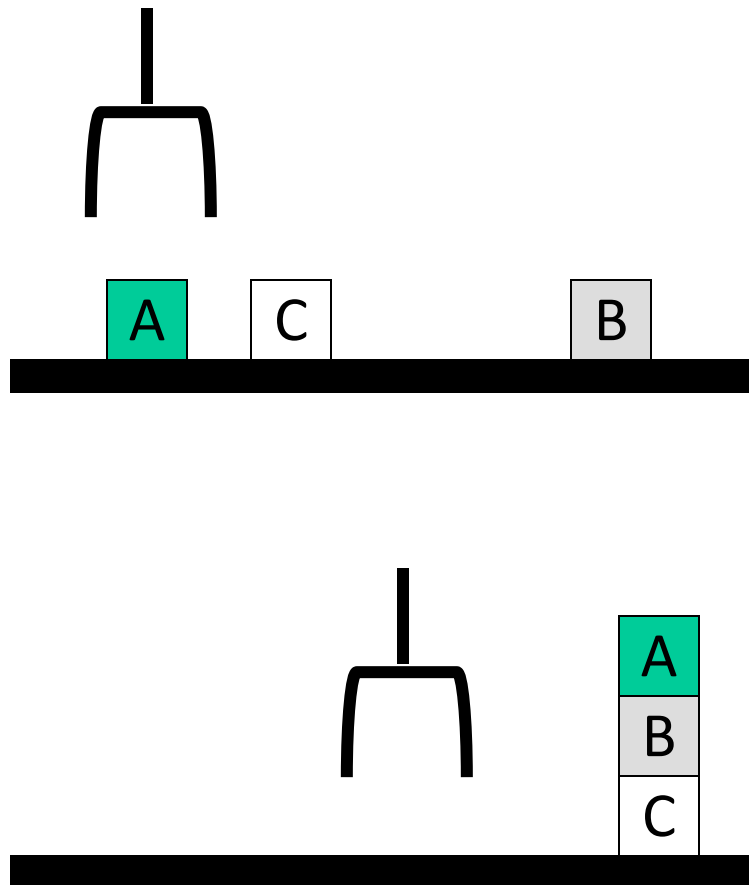
Typical BW planning problem

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)



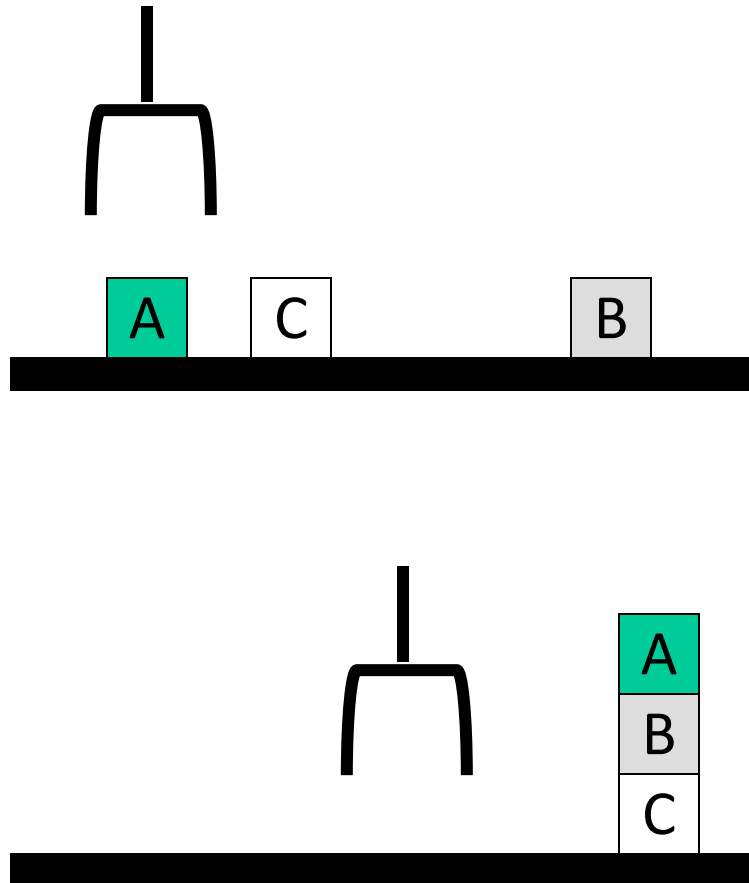

Another BW planning problem

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

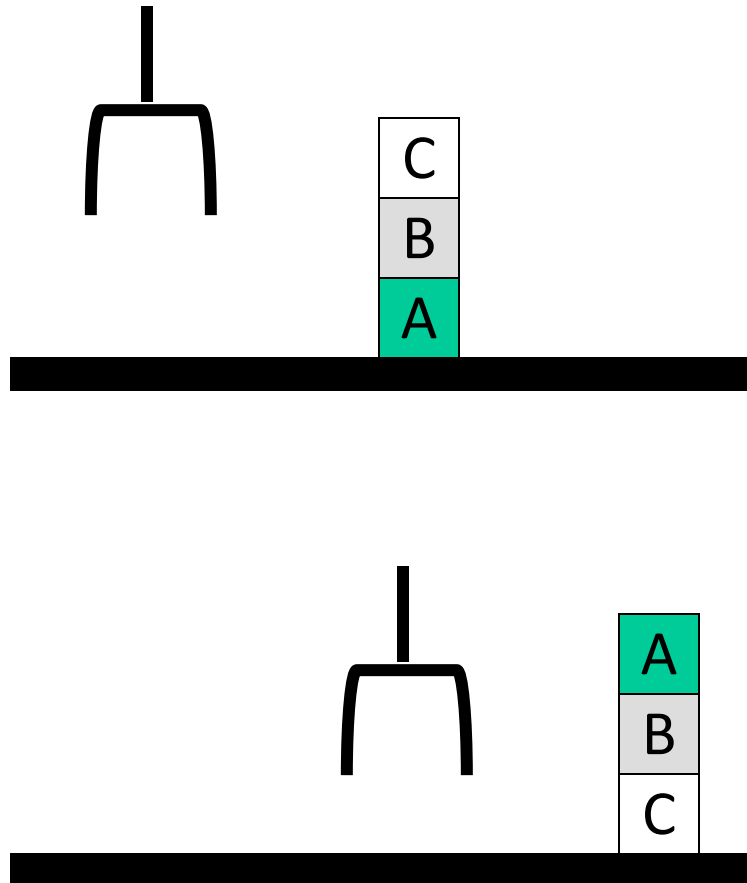


Initial state:

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)

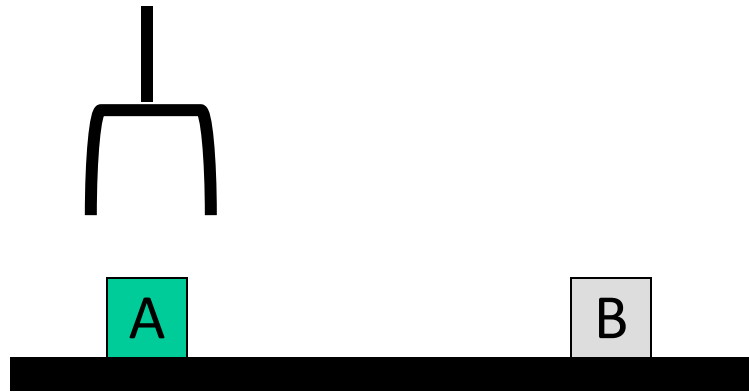
Yet Another BW planning problem

Initial state:

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

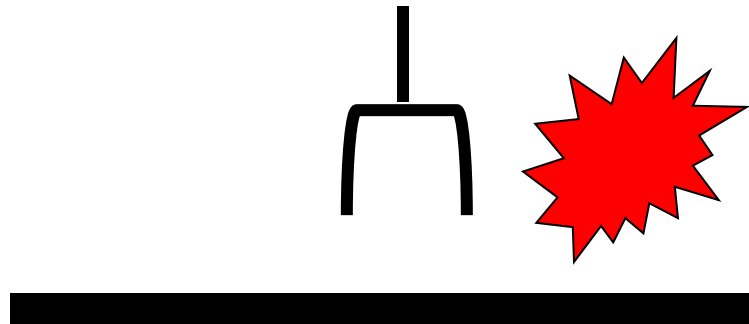
Goal:

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



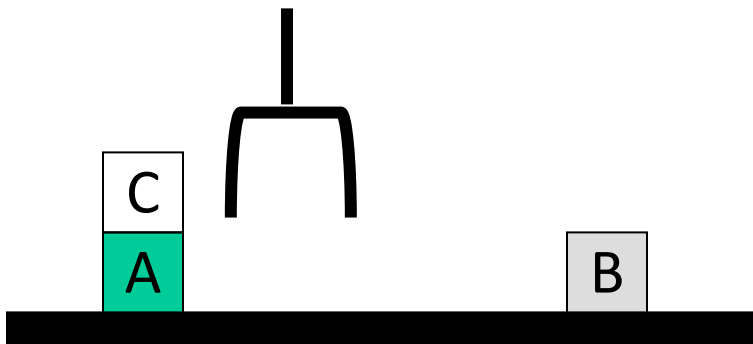
Plan:

??

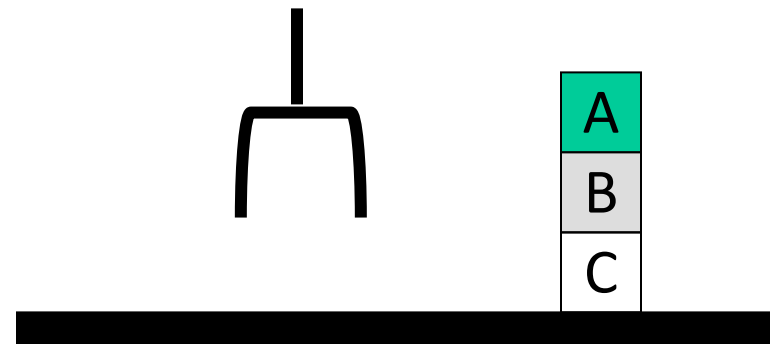


Goal interaction

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



Initial state



Goal state

State-Space Planning

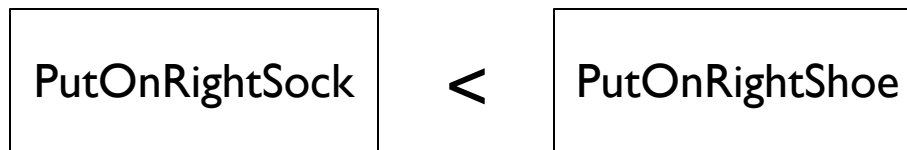
- STRIPS searches thru a space of situations (where you are, what you have, etc.)
- Find plan by searching **situations** to reach goal
- **Progression planner**: searches forward
 - From initial state to goal state
 - Prone to exploring irrelevant actions
- **Regression planner**: searches backward 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
 - but it's harder to define good heuristics – so most current systems favor forward search

Plan-Space Planning

- Alternative: **search through space of *plans***, not situations
- The system represents plans and the actions within those plans. The emphasis is on the order and structure of actions.
- 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

- 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)



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

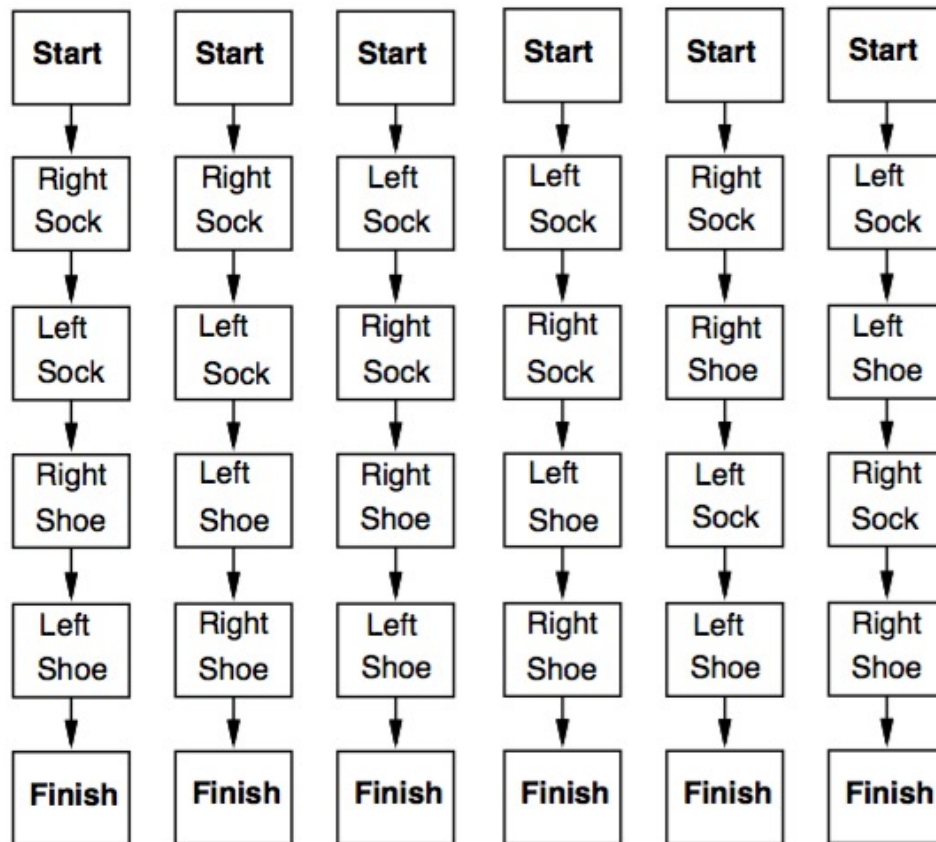
Partial-Order Planning

- 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

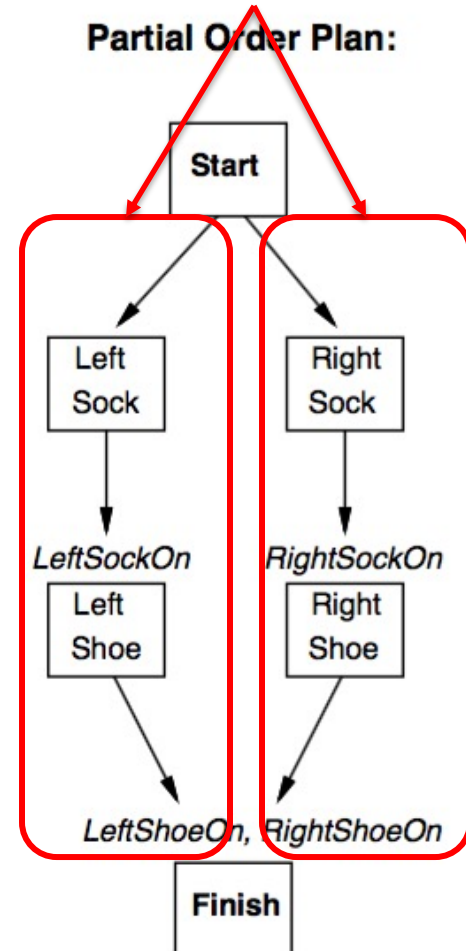
Linear vs. POP: Shoes

Do these
sequences in
any order

Total Order Plans:

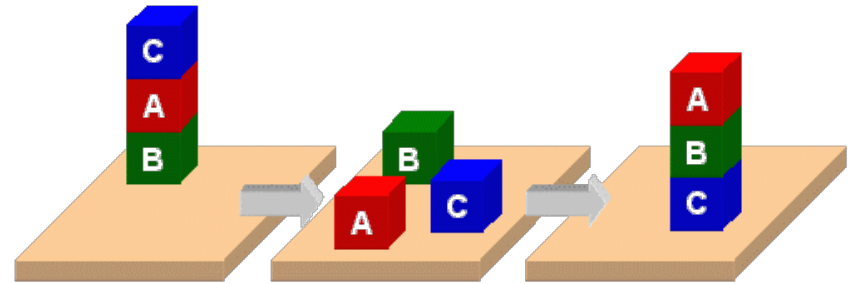


Partial Order Plan:



PDDL

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

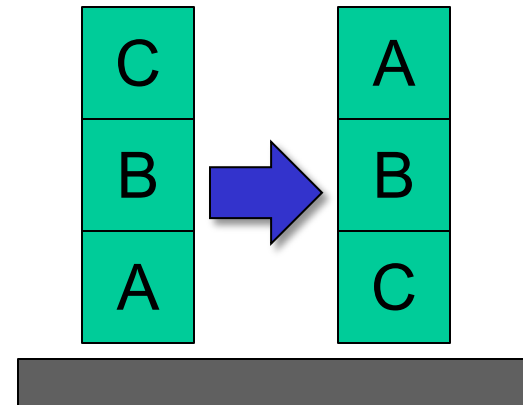
PDDL Representation

- 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 Problem File



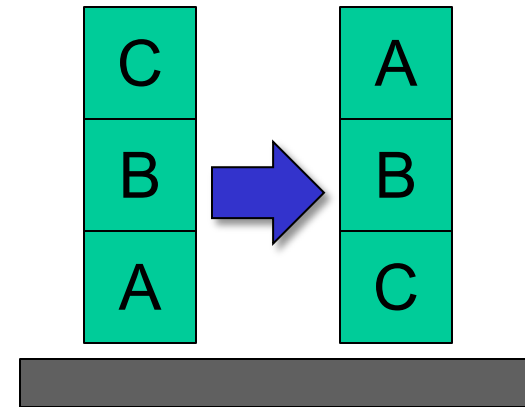
```
(define (problem 00)
  (:domain BW)
  (:objects A B C)
  (:init (arm-empty)
    (on B A)
    (on C B)
    (clear C))
  (:goal (and (on A B)
    (on B C))))
```



Blocks Word Problem File

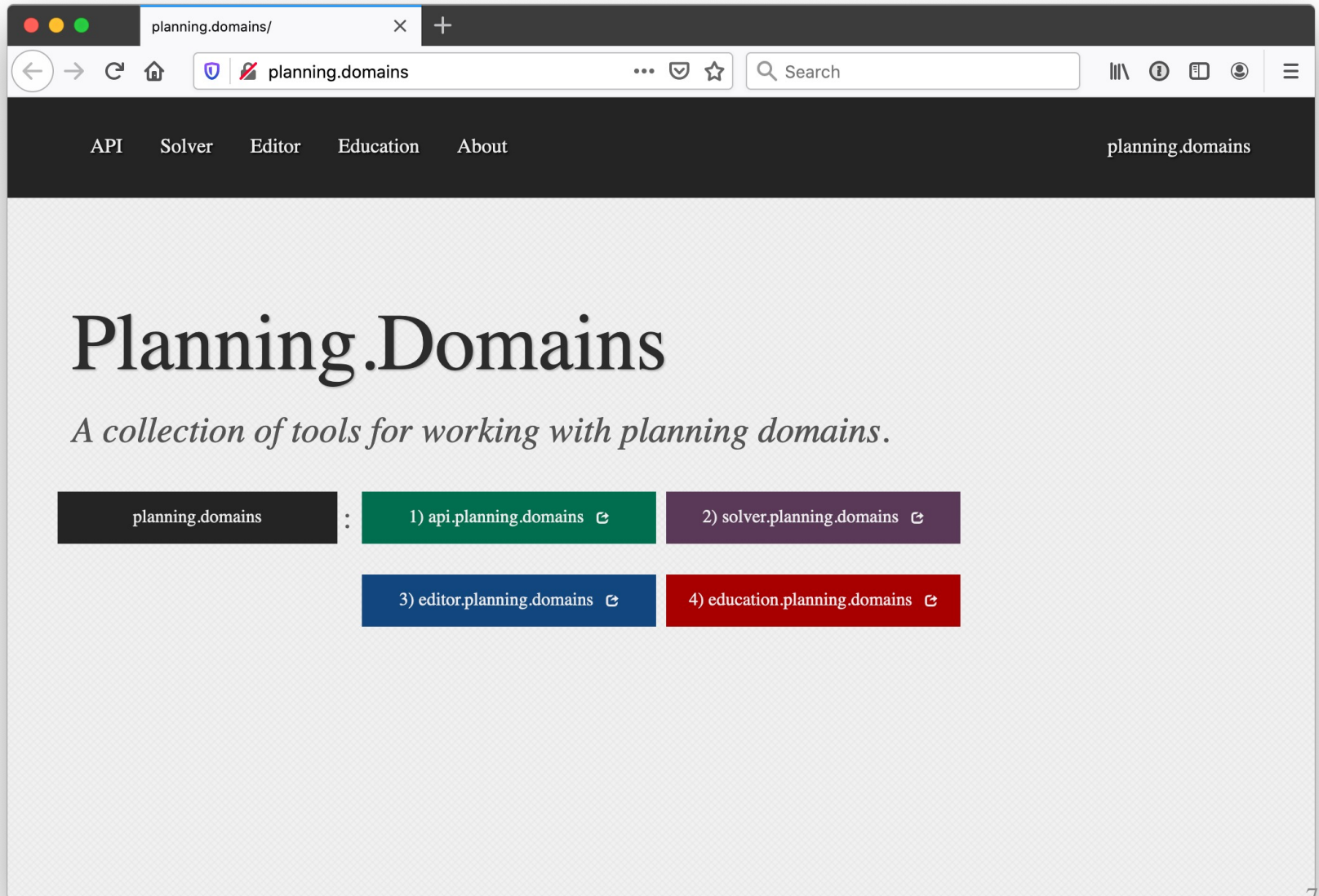


```
(define (problem 00)
  (:domain BW)
  (:objects A B C)
  (:init (arm-empty)
    (on B A)
    (on C B)
    (clear C))
  (:goal (and (on A B)
    (on B C))))
```



```
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

- 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

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
 - Conditional outcomes of actions
 - Uncertain outcomes of actions
 - Exogenous events
 - Incremental plan development
 - Dynamic real-time replanning
- } Scheduling
- } Planning under uncertainty
- } 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