

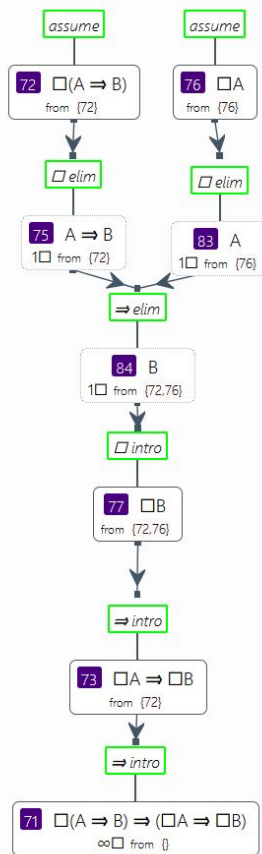
Automated Planning

James Oswald

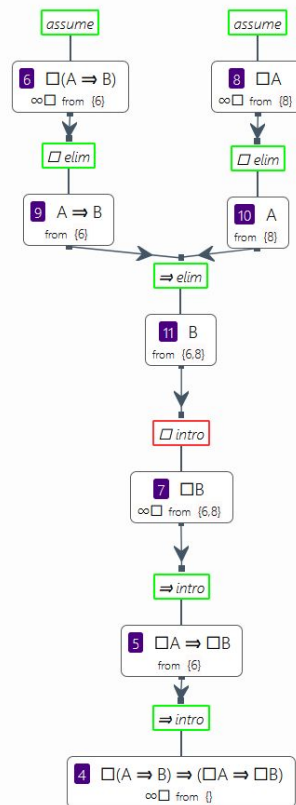
From Last Week

Proof of the K axiom

In K



In S5

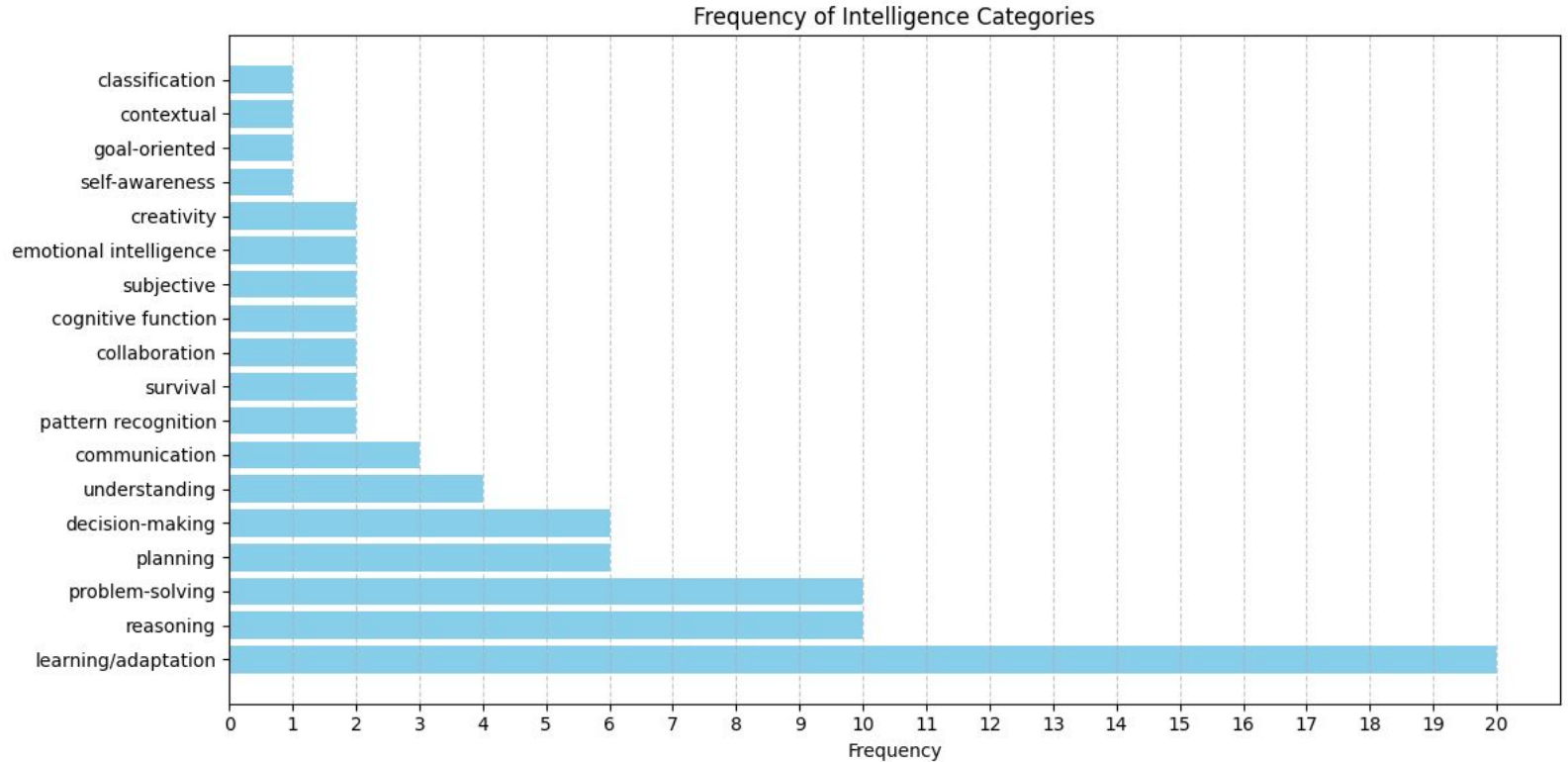


Agenda

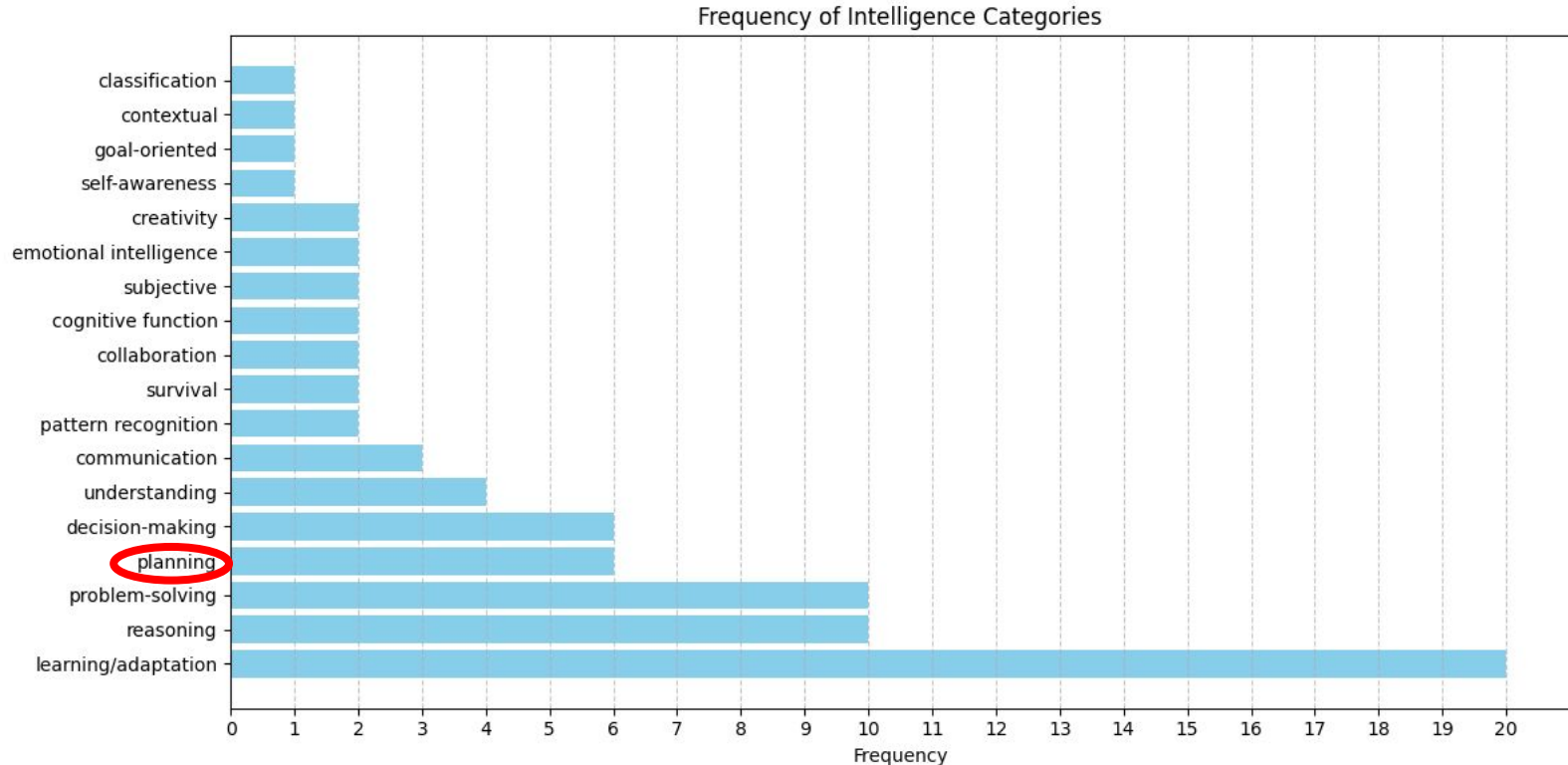
This class:

- Review Survey of definitions of intelligence
- Planning Survey and Discussion
- Planning in Natural Language
- STRIPS planning
- Programming in PDDL
- Planning and You Activity

ILBAI Fall 2024 Intelligence Poll



ILBAI Fall 2024 Intelligence Poll



What Is Planning? Discussion

- 1) Write your own definition of “**Planning**”, make it as general as possible.
- 2) What is a **Plan**?
- 3) With your definitions, write a high level pseudocode planning algorithm that generates a plan from whatever inputs you need. If you don't code, write out your steps required for planning in plain english.

Are your notions general enough to....

- Help a robot get from point A to point B on a 2d plane?
- Navigate from this room back to your dorm?
- Navigate from this room to my house (a place you have never seen)?
- Help sherlock holmes catch a criminal from (maybe false) witness testimony?

Email me your answers: oswalj@rpi.edu

Discussion: What underlies our definitions?

- Action?
- Time?
 - Discrete time? Continuous time?
- State?
 - Epistemic / Cognitive States?
 - Full / Partial Observability?
 - Sensing?

Warm Up Exercise : Cabbage, Goat, Wolf

A farmer wants to cross a river and take with him a wolf, a goat, and a cabbage.

There is a boat that can fit himself plus either the wolf, the goat, or the cabbage.

If the wolf and the goat are alone on one shore, the wolf will eat the goat. If the goat and the cabbage are alone on the shore, the goat will eat the cabbage.

How can the farmer bring the wolf, the goat, and the cabbage across the river?

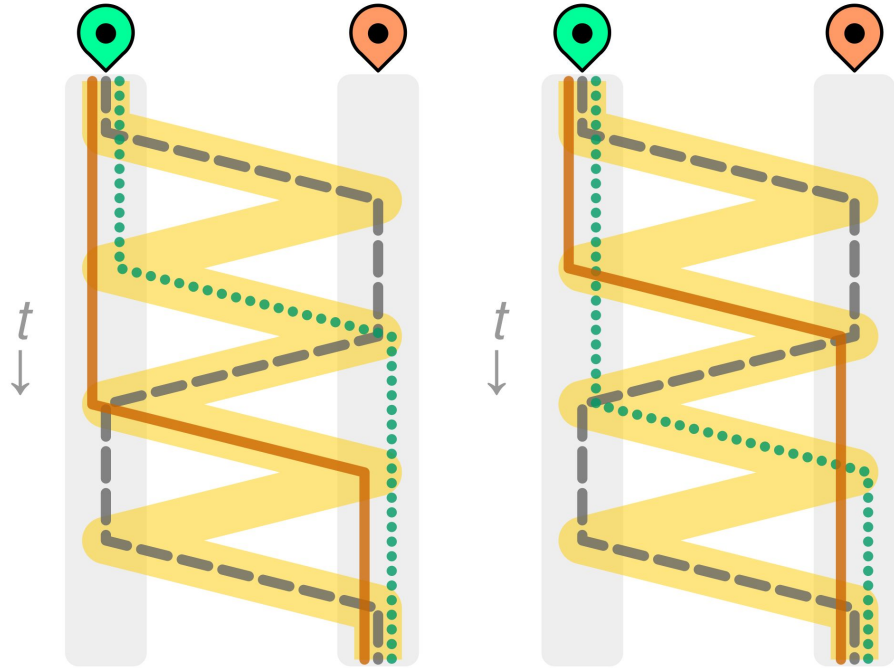
Hint: The shortest plan is 7 steps, but your plan may be longer



Short Solutions:

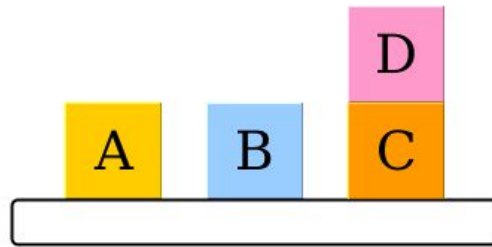
Multiple plans, one is:

1. Cross with goat
2. Return with nothing
3. Cross with cabbage
4. Return with the goat
5. Cross with the wolf
6. Return with nothing
7. Take the goat over

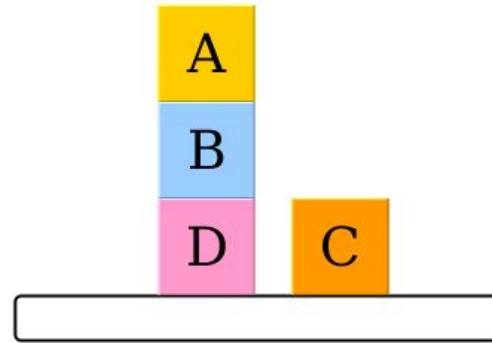


What About Robots?

You are given a single robot hand that can hold one cube. The robot may pick a cube up as long as it has nothing on top of it. It may place the cube on the table (create a new stack) or on top of another cube. Create a plan for the robot that takes us from the initial state to the goal state.



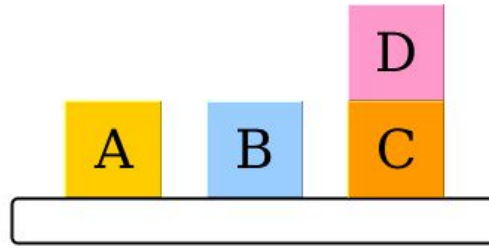
Initial state



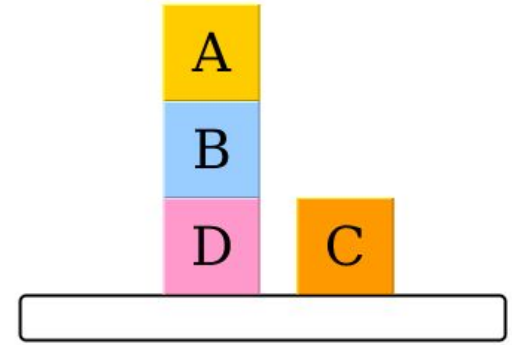
Goal

Solution:

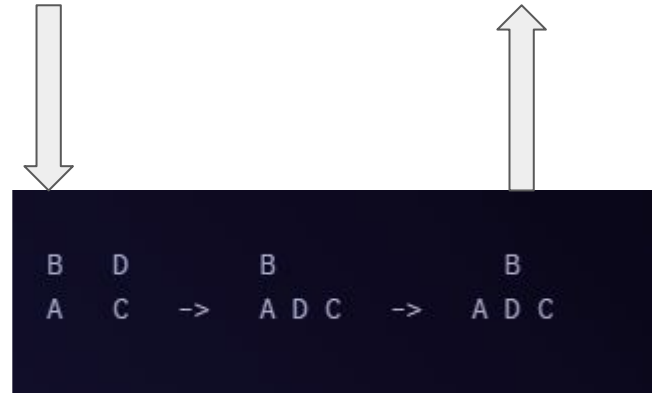
- 1) Pick up B
- 2) Stack B on A
- 3) Unstack D from C
- 4) Place D on the table
- 5) Unstack B from A
- 6) Stack B on D
- 7) Pick up A
- 8) Stack A on B



Initial state



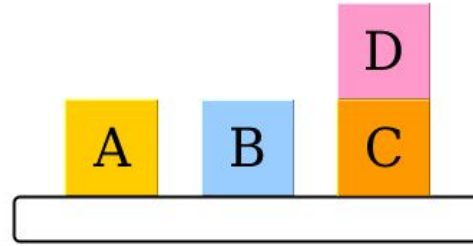
Goal



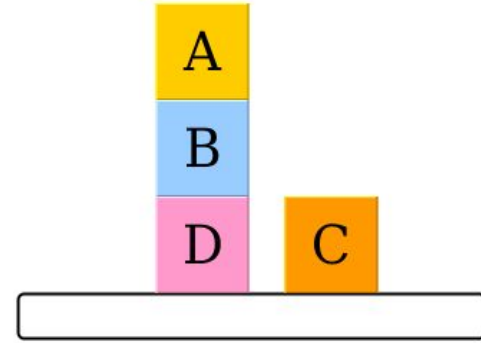
Formalizing State:



Turnus intous
Predicatesus



Initial state



Goal

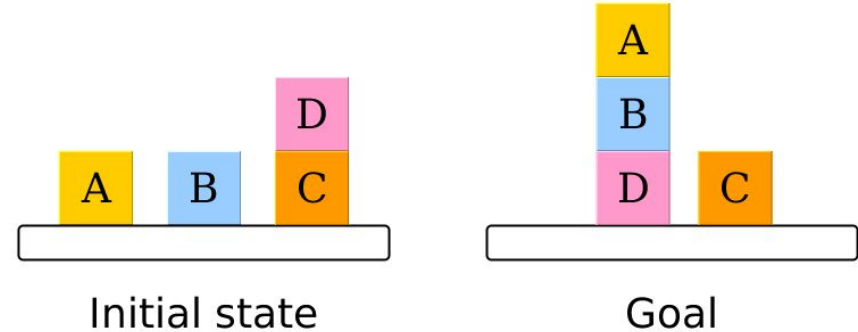
“The world is everything that is the case. The world is the totality of facts, not of things. The world is determined by the facts, and by these being all the facts.”

-Opening to Wittgenstein's Tractatus Logico-Philosophicus, 1922

Exercise: Formalize State in Predicate Logic

- 1) Come up with a list of “objects” in the problem (for example cubes)
- 2) Come up with a list of predicates that relate these objects.
For example: (on-top-of A B).
- 3) Write a list of predicates that formalize the initial state and the goal state.

If you finish early try to formalize state for Cabbage, Goat, Wolf.



Possible Solutions:

Initial State:

(on-table A) (on-table B)

(on-table C) (on-top-of D C)

Goal State:

(on-table D) (on-table C)

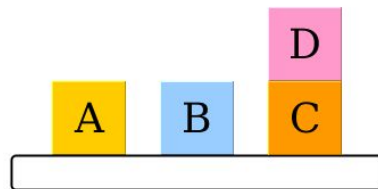
(on-top-of B D) (on-top-of A B)

Or maybe make the table an object:

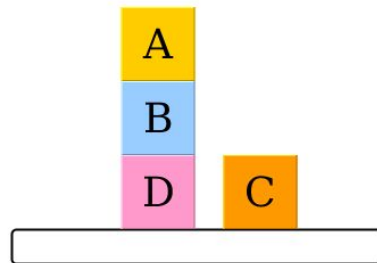
(on-top-of A T) (on-top-of B T)

(on-top-of C T) (on-top-of D C)

Are all other possible predicates
negated?



Initial state



Goal

We are missing something...

We forgot the Robot!

We formalized the start and end states without taking into account how we represent intermediate states where we are moving blocks.

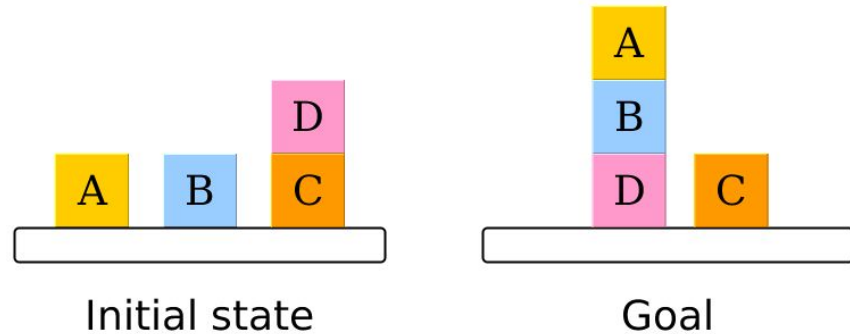
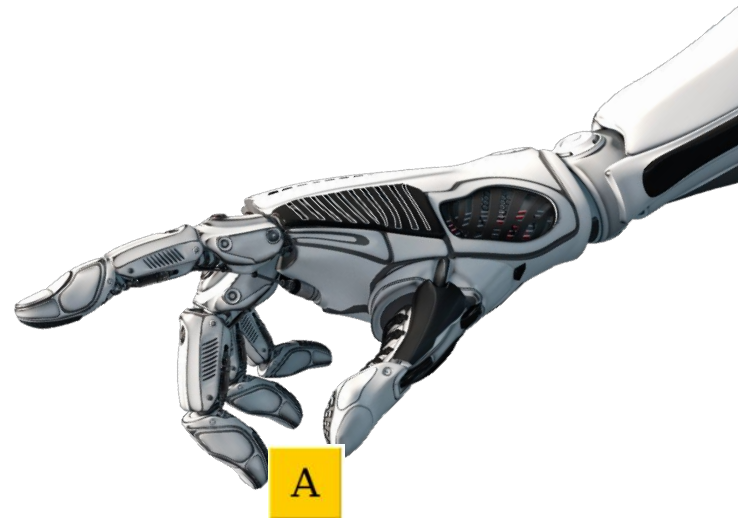
(holding ?block) = The robot is holding the given block.

(handempty) = The robot hand is empty

Note:

(holding A) and (holding B) is a contradiction
as is

(holding A) and (handempty)



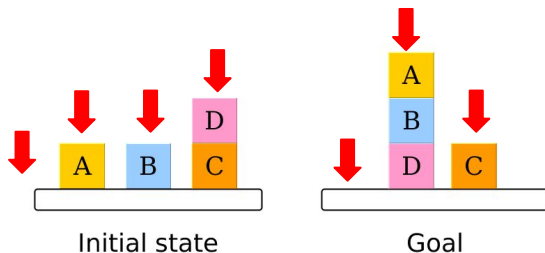
Predicates

While we make our plan, we often think about WHERE we can put a block.

- 1) We can place a block we are holding onto the table
- 2) Or on top of another block that has nothing on top of it.

Case 1 is easy to formalize as $(\text{holding } B) \Rightarrow (\text{on-table } B)$

Case 2 is more complex: $(\text{holding } B1) \Rightarrow (\text{on-top-of } B1 \ B2)$ but only if “there does not exist a block on-top of $B2$ ” formally: $(\text{not } (\text{exists } x (\text{on-top-of } x \ B2)))$.

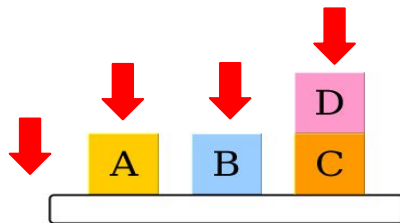


Precondition Predicates

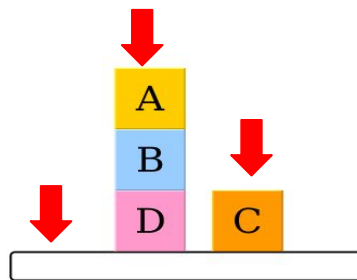
We can “cheat” and define a new predicate (clear X) to mean “there is nothing on top of X”.

As long as we keep it consistent with (not (exists x (on-top-of x B2))) there is no contradiction in the state.

(clear A)
(clear B)
(clear D)



Initial state



Goal

(clear A)
(clear C)

Updated State Exercise

We now have the following predicates:

(handempty) (holding ?B) (clear ?B)
(on-table ?B) (on-top-of ?B1 ?B2)

On the left we show the state after picking up B from the initial state, write the state after stacking B on A and unstacking D from C.

- 1) Pick up B
- 2) **Stack B on A**
- 3) **Unstack D from C**

Initial State

(handempty) 

(on-table A)

(on-table B) 

(on-table C)

(clear A)

(clear B) 

(clear D)

(on-top-of D C)

Pickup B

(holding B)

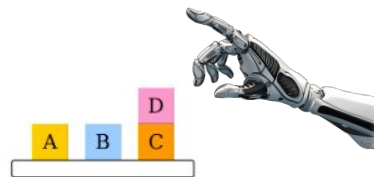
(on-table A)

(on-table C)

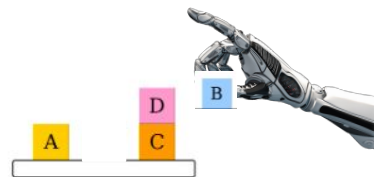
(clear A)

(clear D)

(on-top-of D C)

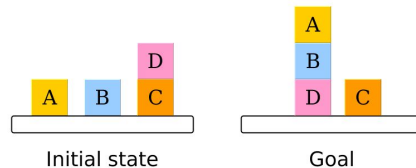


Initial state



Initial state

Solution



Initial State

(handempty)

(on-table A) —

(on-table B)

(on-table C) —

(clear A) —

(clear B)

(clear D) —

(on-top-of D C) —

Pickup B

(holding B)

(on-table A) —

(on-table C) —

(clear A)

(clear D) —

(on-top-of D C) —

Stack B on A

(handempty)

(on-table A) —

(on-table C) —

(clear B) —

(clear D)

(on-top-of D C)

(on-top-of B A) —

Unstack D from C

(holding D)

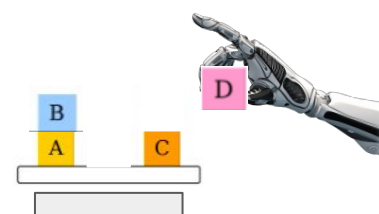
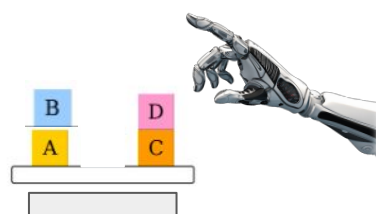
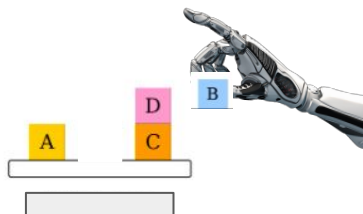
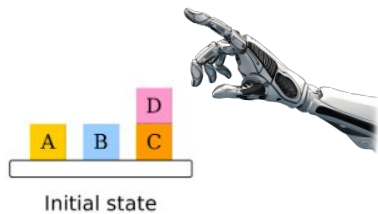
(on-table A)

(on-table C)

(clear B)

(clear C)

(on-top-of B A)



Formalizing Action: What moves can we make?

Let's recall our plan

- 1) Pick up B
- 2) Stack B on A
- 3) Unstack D from C
- 4) Place D on the table
- 5) Unstack B from A
- 6) Stack B on D
- 7) Pick up A
- 8) Stack A on B

It seems we repeat four distinct actions

- 1) Picking up a block off the table
- 2) Placing a block on the table
- 3) Stacking a block on a clear block
- 4) Unstacking a block from another block

All that changes are the objects the actions are applied to.

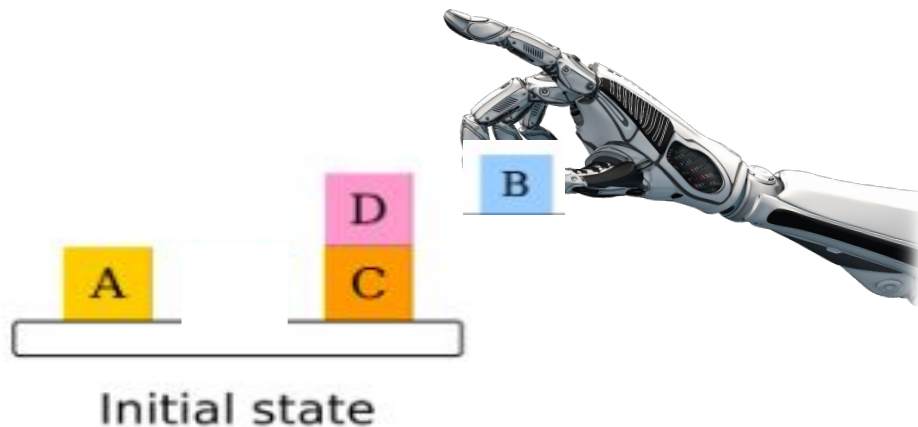
Formalizing Action

Action clearly modifies the state.

But we are missing something...

Can I perform any action in any state?

In this state, can I unstack D?

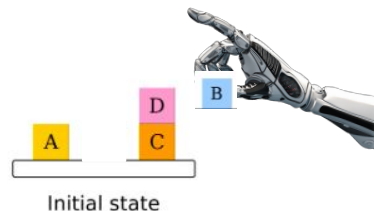
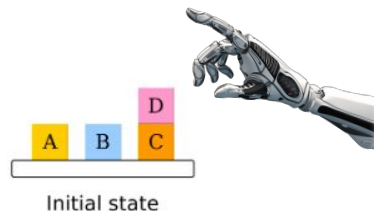


Preconditions: Restrictions on Action

What needs to hold for us to pick up a block from the table?

When would it not make sense to pick up a block?

<u>Initial State</u>	<u>Pickup B</u>
(handempty) 	(holding B)
(on-table A)	(on-table A)
(on-table B) 	(on-table C)
(on-table C)	(clear A)
(clear A)	(clear D)
(clear B) 	(on-top-of D C)
(clear D)	
(on-top-of D C)	



Preconditions for Pickup

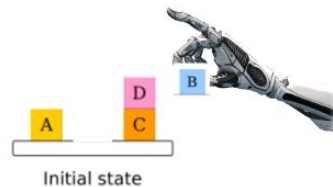
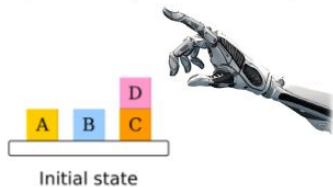
Preconditions for picking up block ?B from the table:

(handempty) - The robot's hand is empty

(clear ?B) - The block has nothing on top of it

(on-table ?B) - The block is on the table

<u>Initial State</u>	<u>Pickup B</u>
(handempty) 	(holding B)
(on-table A)	(on-table A)
(on-table B) 	(on-table C)
(on-table C)	(clear A)
(clear A)	(clear D)
(clear B) 	(on-top-of D C)
(clear D)	
(on-top-of D C)	



Effects of picking a block up off the table?

Two types of effects:

- 1) Deletions from the previous state
- 2) Additions to the new state

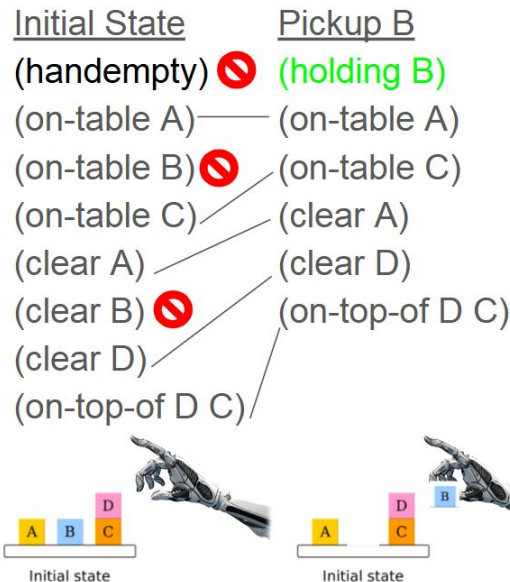
Effects of picking up ?B off the table?

Delete (handempty) from state

Delete (on-table ?B) from state

Delete (clear ?B) from state

Add (holding ?B) to state



Formalizing Action

We can succinctly write out the entire schema for picking up a block ?B in an S-Expression format.

Note: instead of deleting, we will write this as adding the negation of the predicate in the effects.

```
(:action pickup
  :parameters (?B)
  :precondition (and
    (clear ?B)
    (on-table ?B)
    (hand-empty)
  )
  :effect (and
    (holding ?B)
    (not (clear ?B))
    (not (on-table ?B))
    (not (hand-empty))
  )
)
```

Exercise: Formalize Put Down, Stack, Unstack

Formalize the other 3 actions in this format.

What are the preconditions and effects of put down (on table), stack, and unstack?

If you need help look back at slide 18 to see what changes.

```
(:action pickup
  :parameters (?B)
  :precondition (and
    (clear ?B)
    (on-table ?B)
    (hand-empty)
  )
  :effect (and
    (holding ?B)
    (not (clear ?B))
    (not (on-table ?B))
    (not (hand-empty))
  )
)
```

Solutions

```
(:action putdown
  :parameters (?B)
  :precondition (and
    (holding ?B)
  )
  :effect (and
    (not (holding ?B))
    (hand-empty)
    (clear ?B)
    (on-table ?B)
  )
)
```

```
(:action stack
  :parameters (?A ?B)
  :precondition (and
    (clear ?B)
    (holding ?A)
  )
  :effect (and
    (not (holding ?A))
    (hand-empty)
    (clear ?A)
    (on-top-of ?A ?B)
  )
)
```

```
(:action unstack
  :parameters (?A ?B)
  :precondition (and
    (on-top-of ?A ?B)
    (clear ?A)
    (hand-empty)
  )
  :effect (and
    (holding ?A)
    (not (hand-empty))
    (not (on-top-of ?A ?B))
    (not (clear ?A))
  )
)
```

Formalizing Planning: STRIPS

A (lifted) planning domain is defined as:

- 1) A set of (lifted) predicates that describe the world, W
- 2) A set of (lifted) actions, each of which is composed of
 - a) A set of predicates in W required to be true to perform the action
 - b) A set of predicates in W that are added to the state after the action
 - c) A set of predicates in W removed from the state after the action

A (lifted) planning problem consists of:

- 1) A set of objects that instantiate (ground) predicates
- 2) An initial state, composed of grounded predicates from W
- 3) A goal state (or set of goal states, described as a partial goal state)

Lifted vs Grounded / Predicates & Actions

To plan we need to *ground* all of our predicates

- (on-top-of ?B1 ?B2) - lifted predicate, we have free variables.

Grounding the predicate consists of filling in all possible blanks giving us a set of *grounded predicates*.

- (on-top-of A B) (on-top-of B C) (on-top-of C D) (on-top-of A C) (on-top-of A D),

Grounding Actions consists of doing the same thing for the action's parameters:

- (unstack ?B1 ?B2) is a lifted action with lifted preconditions and effects
- (unstack A B) is a grounded action, we have filled it in with concrete objects.
 - Its preconditions and effects are filled in with the matching grounded predicates

Before we plan, we ground out the domain! Our set of actions is actually the set of grounded actions, and state is the set of grounded predicates. This is typically an exponential process.

Formalizing Application and Plans

Definition of applicable: An action A is said to be *applicable* in a state S iff the preconditions of A ($\text{pre}(A)$) hold in S .

Definition of application: “Application” is a function taking an action A and state S a new state S' such that $S' = (S / \text{del}(A)) \cup \text{add}(A)$

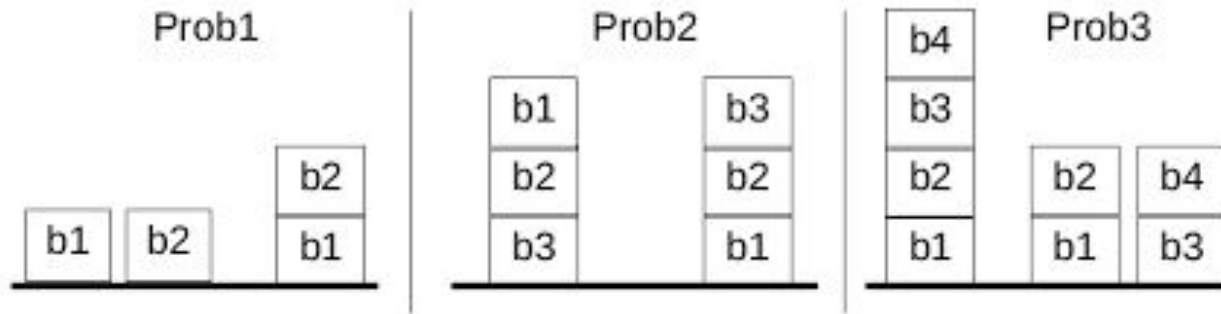
Definition of a Plan: A plan from an initial state S_1 to a goal state G is a sequence of actions $O_1, O_2, \dots$ such that each action is applicable to the state generated by applying the previous action. Formally

$(O_n \text{ is applicable to } S_n) \text{ and } (\text{forall } n: S_{(n+1)} = \text{Apply}(O_n, S_n)).$

Why Separate Out Domain and Problem?

You can have many different problems in the same domain.
Consider the blocksworld states:

- Each has different objects
- Same predicates and actions



Planning in Practice: Programming PDDL

PDDL is a description language to formalize planning domains and problems.

PDDL stands for “Planning Domain Definition Language”

You are already a PDDL Programmer!

You wrote PDDL Actions. Lets go the final mile...



PDDL Domains

Consist of

- 1) Predicates
- 2) Actions

```
(define (domain ILBAI-BLOCKS)
  (:predicates
    (hand-empty)
    (clear ?B)
    (on-table ?B)
    (holding ?B)
    (on-top-of ?A ?B)
  )

  //our actions here
)
```

PDDL Problems

Consist of:

- 1) Objects
- 2) Initial State
- 3) Goal State

```
(define (problem ex1)
  (:domain ILBAI-BLOCKS)
  (:objects A B C D)
  (:init
    (hand-empty)
    (on-table A)
    (on-table B)
    (on-table C)
    (clear A)
    (clear B)
    (clear D)
    (on-top-of D C)
  )

  (:goal (and
    (on-table D)
    (on-table C)
    (on-top-of B D)
    (on-top-of A B)
  ))
)
```

Why PDDL?

Planners accept PDDL as valid input.

If you have PDDL for your problem you can have the world's most powerful automated planners come up with a plan for you.

Let's check it out:

[PDDL Editor \(planning.domains\)](http://planning.domains)

Found Plan (output)

(unstack d c)

(putdown d)

(pickup b)

(stack b d)

(pickup a)

(stack a b)

```
(:action unstack
:parameters (d c)
:precondition
  (and
    (on-top-of d c)
    (clear d)
    (hand-empty)
  )
:effect
  (and
    (holding d)
    (not
      (hand-empty)
    )
    (not
      (on-top-of d c)
    )
    (not
      (clear d)
    )
  )
)
```

Ok but how do I program a Planner myself?

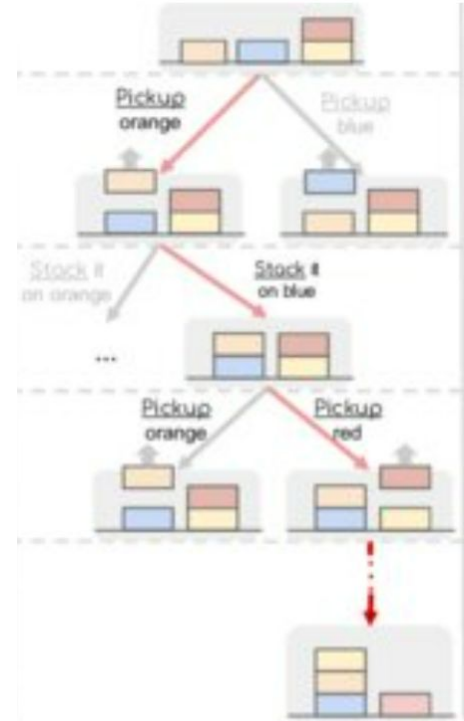
Naive Approach: Breadth first search on the applicable action tree.

Better Approach: A* search with custom heuristics.

Current Best Approaches: Based on Landmark Heuristics.

Landmarks are intermediate states that **MUST** be reached before the goal.

IE. To leave the room I must pass through the door
=> there is a landmark state S' where i am passing through the door.



Rest of Class: Pick One Activity, Planning and You

- 1) Formalize a simple game of your choice in PDDL (Use Planning)
 - a) <https://planning.wiki/ref/pddl> - PDDL Reference, Don't use types or anything
 - b) Pick something SIMPLE, cabbage goat wolf? Maze search? Transporting boxes?
 - c) Run it on editor.planning.domains
- 2) In a programming language of your choice (Become Planning):
 - a) implement the mathematical formalism as a structure/record/class
 - b) Instead of having objects, pick a small example & ground everything manually.
 - c) Implement applicability and application, if you're brave implement grounding.
 - d) Write a plan verifier that takes a state, goal, and sequence of actions and returns if the plan is valid
 - e) Write a naive breadth or depth first search planning algo, feel free to use online references.

Regardless of what activity you pick, right before class ends,
email me your pddl domain or code at: oswalj@rpi.edu