CS 309: Autonomous Robots
FRI I

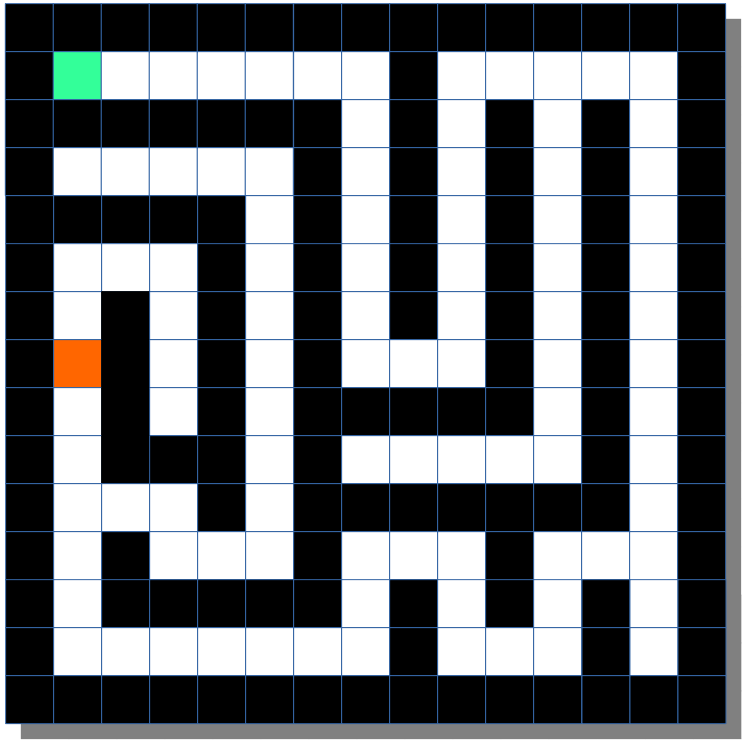
Symbolic Reasoning and Search

Instructor: Justin Hart

http://justinhart.net/teaching/2020_spring_cs309/

AI as search

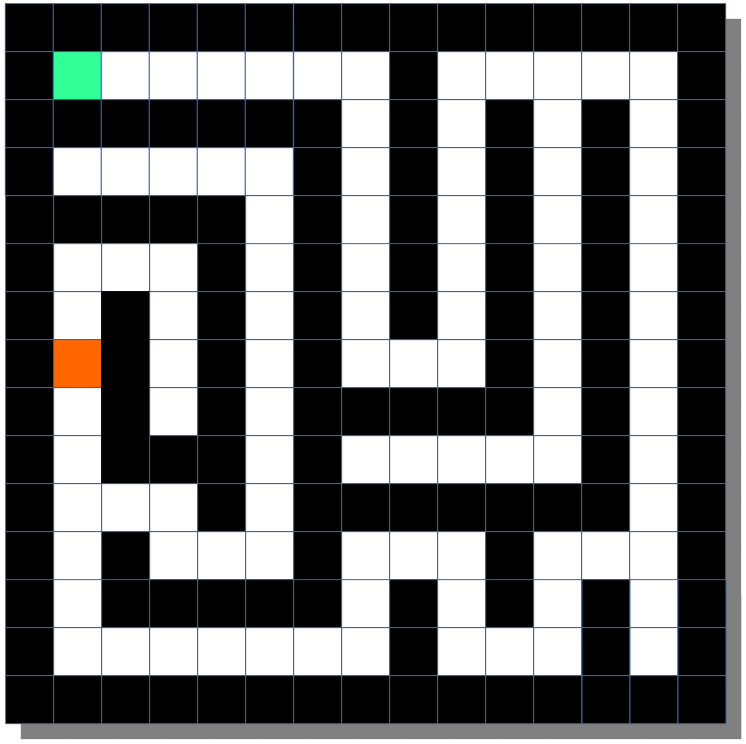
- Imagine a computer solving a maze
- There are many options for how to do this
- A search algorithm will test each action an agent can take until it finds a solution



AI as search

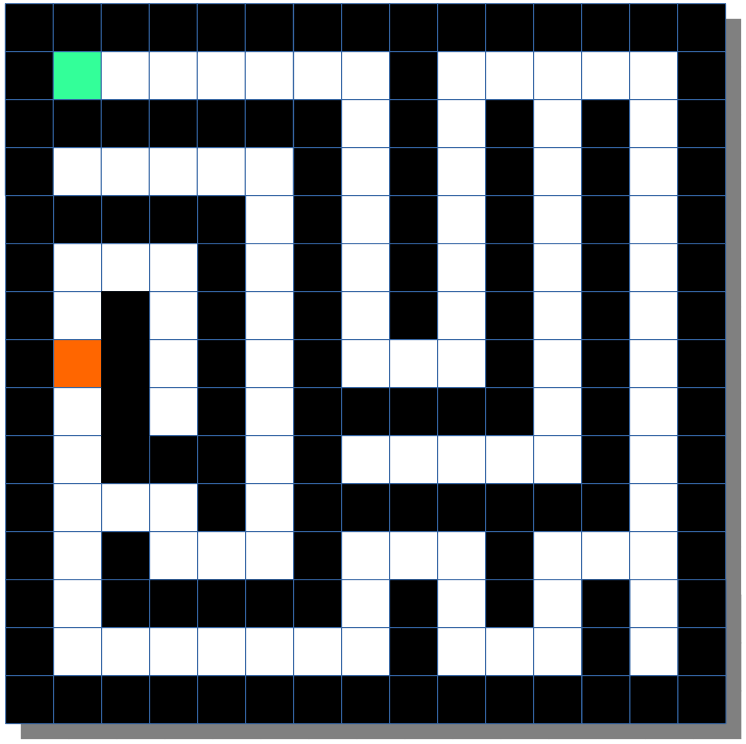
- The agent is the orange dot
  - It is supposed to get to the green dot

- Possible moves are up, down, left right
  - But here, left and right do not work
  - The search algorithm may try them, but they will fail
  - Up and down work



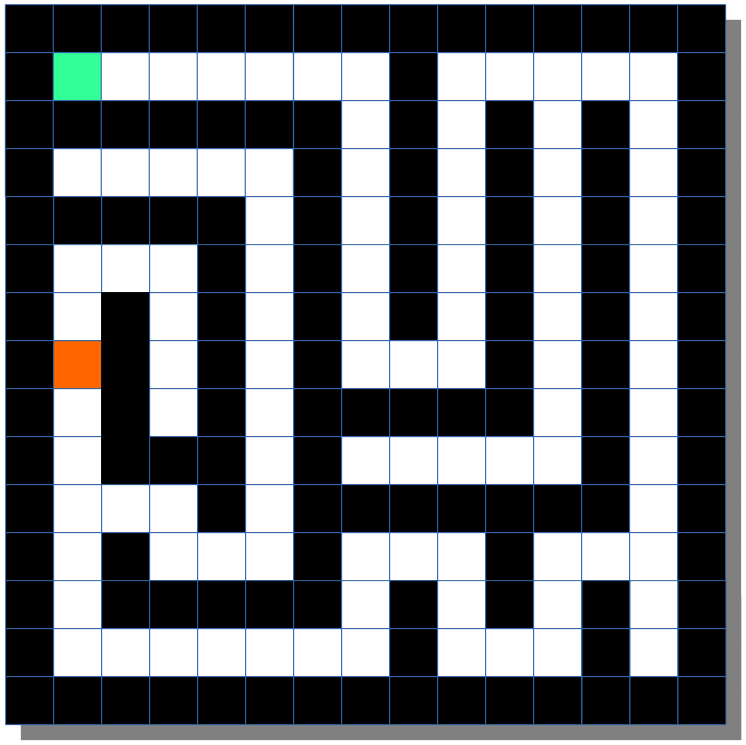
Satisfying vs Optimizing

- Satisficing solutions
  - Work, but are not known to be optimal
- Optimal solutions
  - Are intended to be optimal



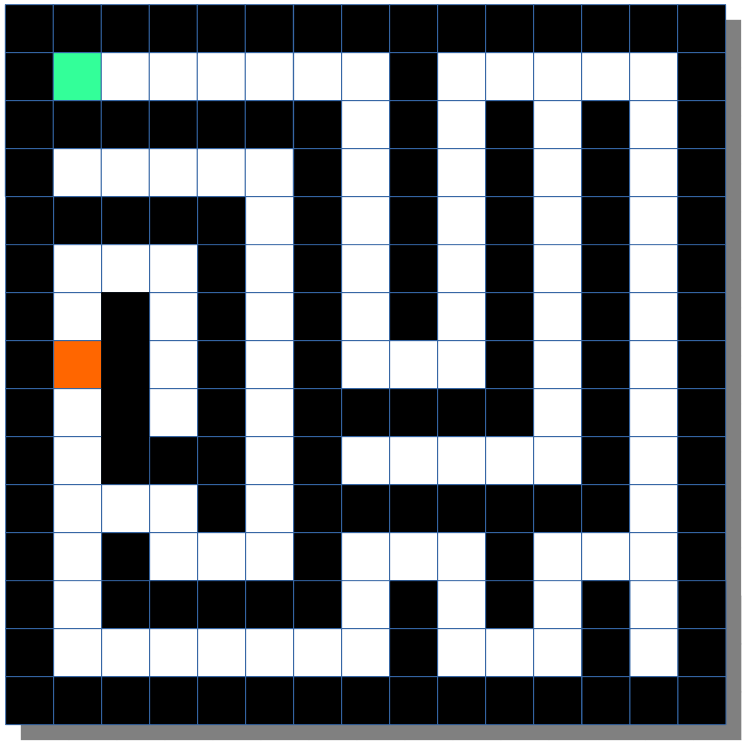
Three introductory search patterns

- Breadth-first
- Depth-first
- A*



The “ready queue”

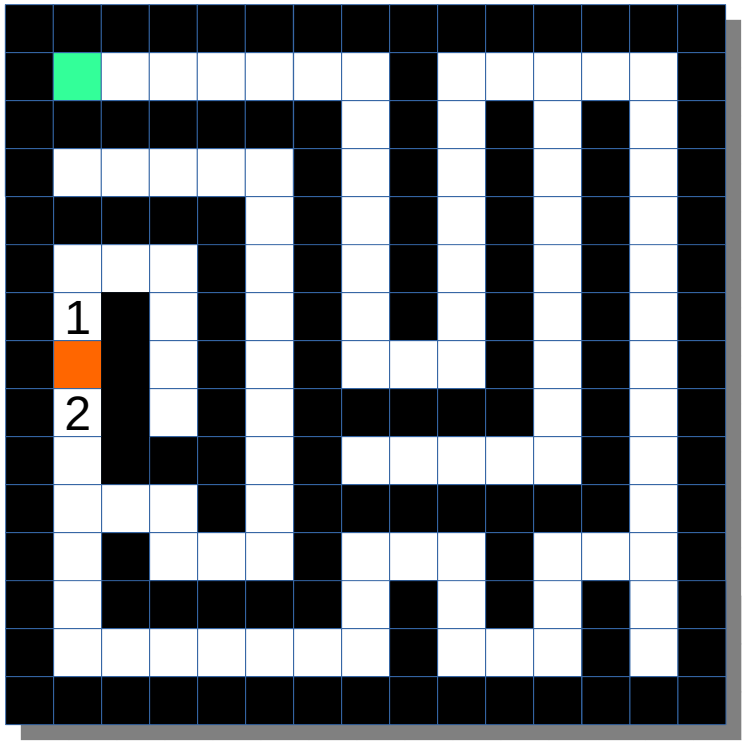
- A “state” can be thought of as a configuration of the “world” or “problem”
- The orange dot here represents the state of the agent occupying that cell
  - Let’s call this state 0
- When starting to solve the problem, the “start state” is placed into the ready queue
- A “search algorithm” will take the “start state” out of the ready queue and “expand” it, placing the resulting states into the ready queue



Breadth-first search (BFS)

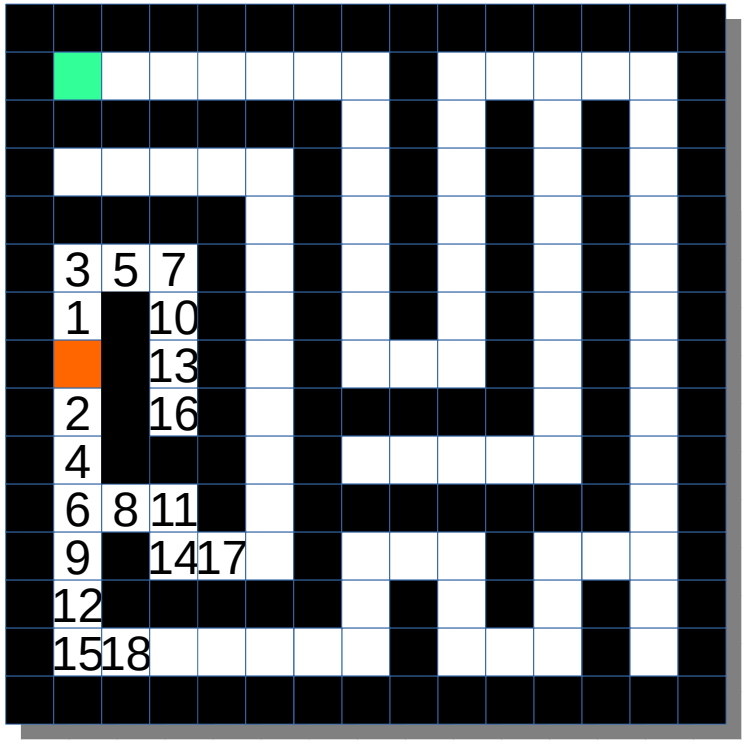
- Take a state out of the queue (FIFO.. We'll come back to this)
  - Up - works
  - Down – works
  - Left – fails
  - Right – fails

- Enter these into the ready queue



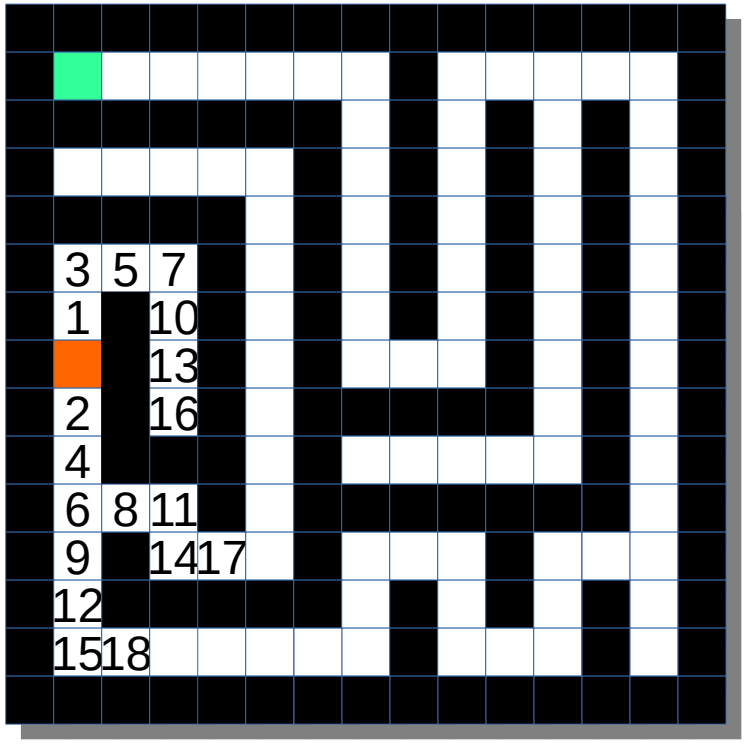
Breadth-first search

- Now take states out of the ready queue and expand them
  - You can ignore states that you've entered into before (if your algorithm can detect this)
- Continue to do this until a solution is found
- Breadth-first is FIFO
  - First-In-First-Out
  - It expands its search horizon at the breadth meaning that each potential solution is expanded at a "depth" (taking the same number of moves) until that layer is full.



Breadth-first search

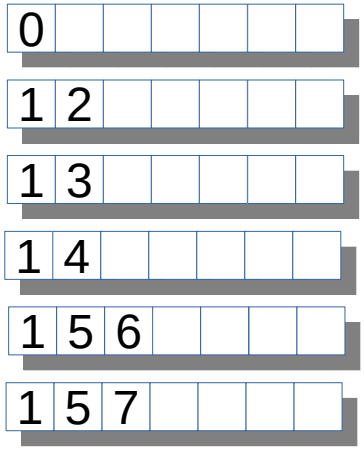
- BFS is “complete”
  - It will eventually explore the entire space
  - It is optimal in that the first solution found is guaranteed to take the fewest steps



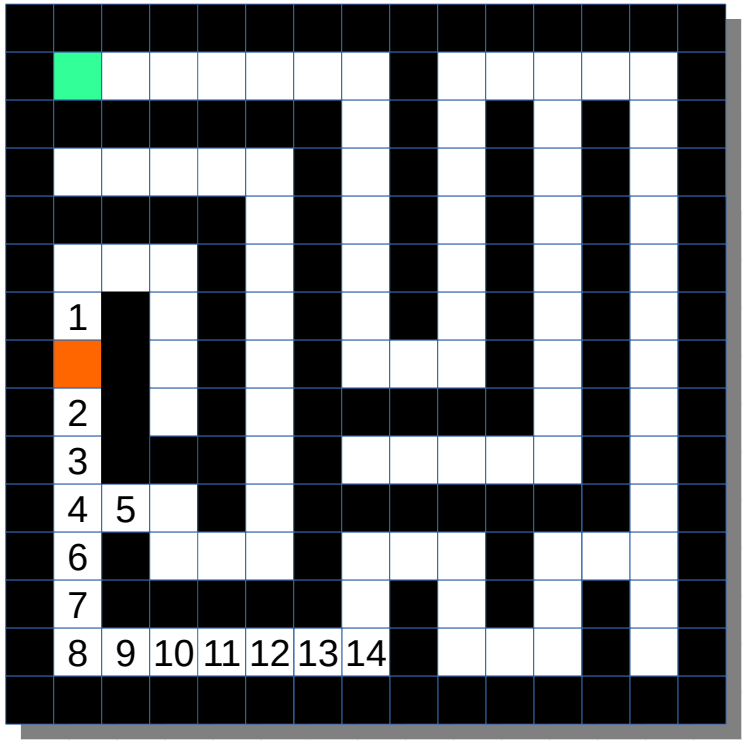
0	1	2	3	4		

Depth-first search

- FILO – First In - Last Out

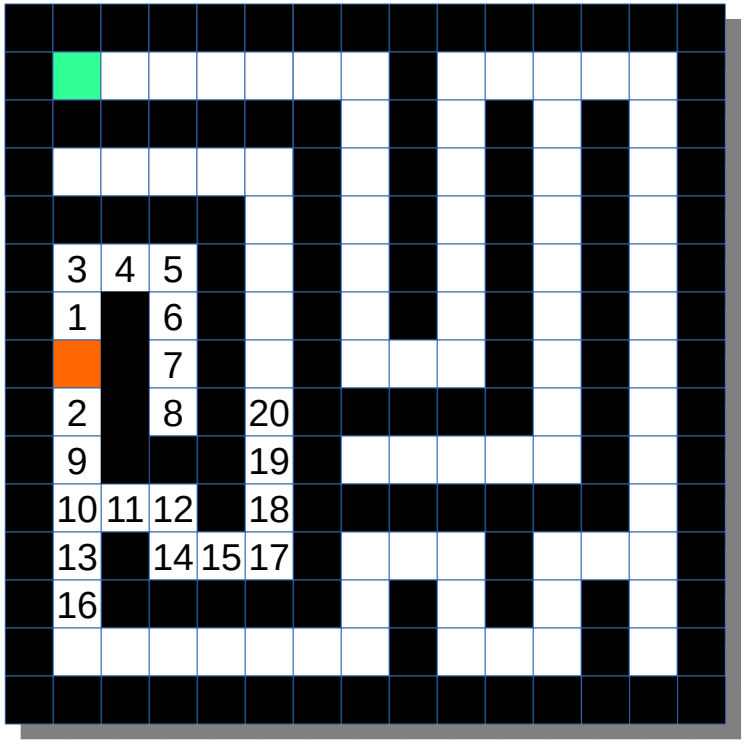


- DFS may not find the optimal solution
- Generally requires less memory than BFS



A* search

- BFS – queue (FIFO)
- DFS – stack (FILO)
- A* – priority queue (best-first)

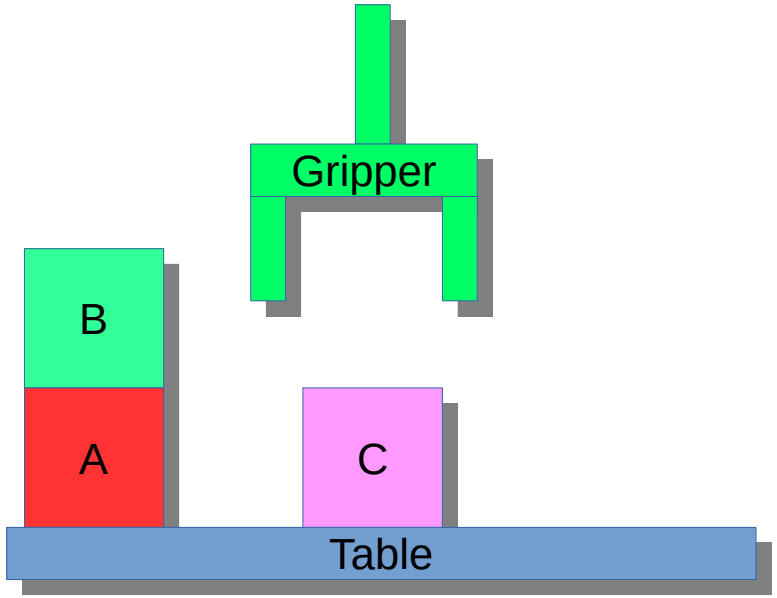


- What counts as “best”?
  - Use heuristics to “score” states
  - This maze uses Euclidean distance
- Depending on how you choose your heuristic A* may be complete
  - ..but that’s for your AI class
- Planning algorithms have come a long way, but build on these basic ideas

Blocks World

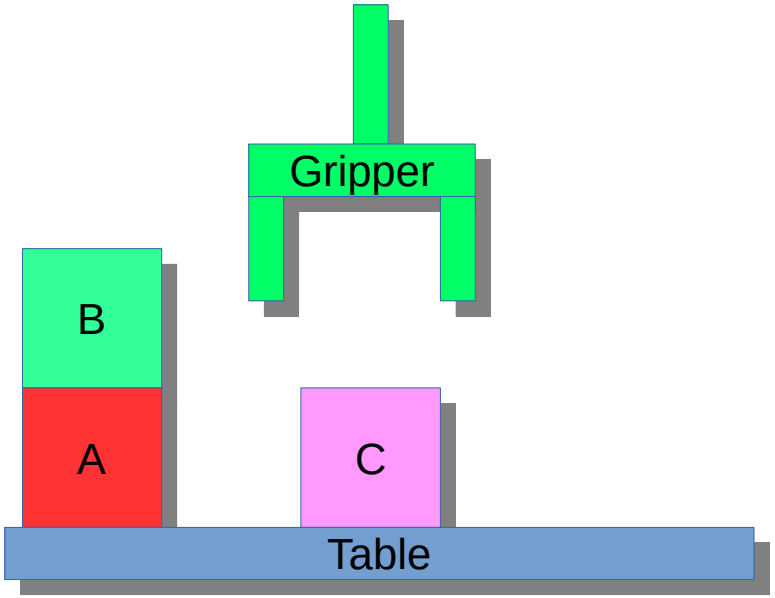
- Planning equivalent of “Hello World” -> “Blocks World”

- Blocks arranged on a table with a robot gripper



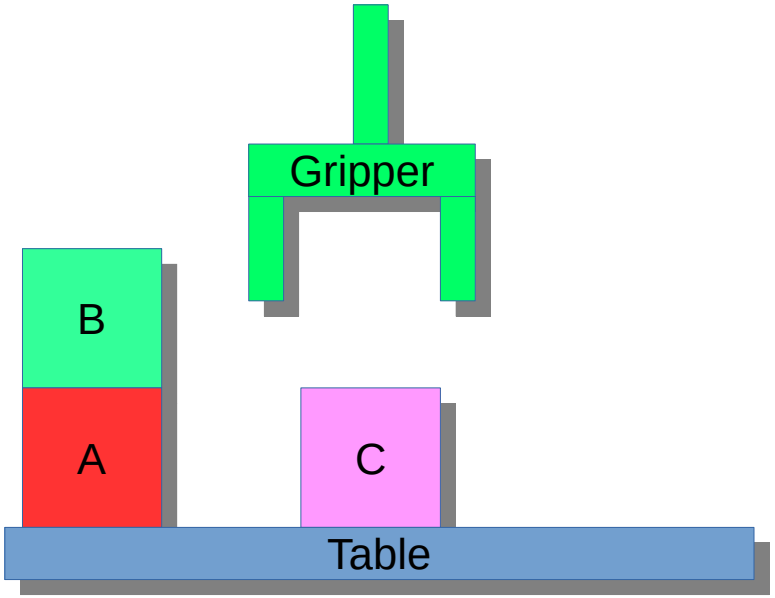
Atoms

- Represent things we can reason about in the world
  - block_a, block_b, block_c
  - table_a
  - gripper_a



Predicates

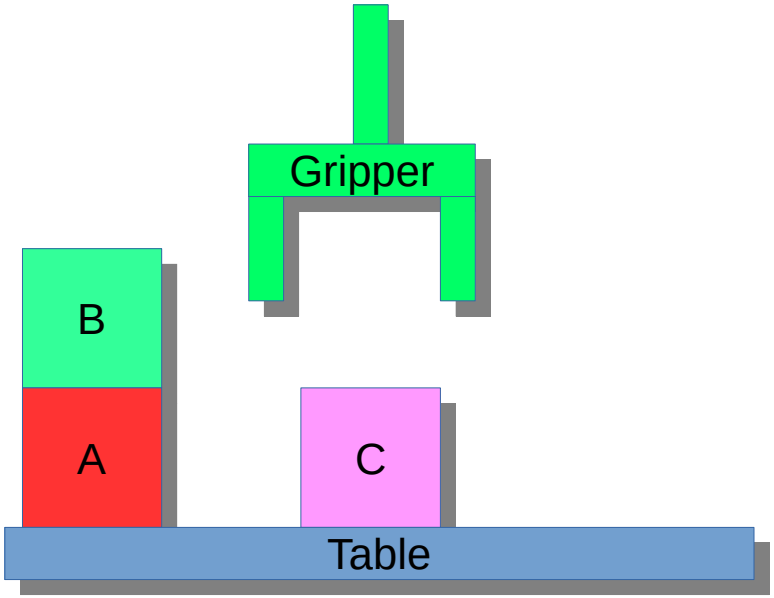
- Modify and describe atoms
  - on_table(block_a), on_table(block_c)
  - stacked(block_b, block_a)
  - clear(block_b), clear(block_c)
  - gripper_empty(gripper_a)



Predicates

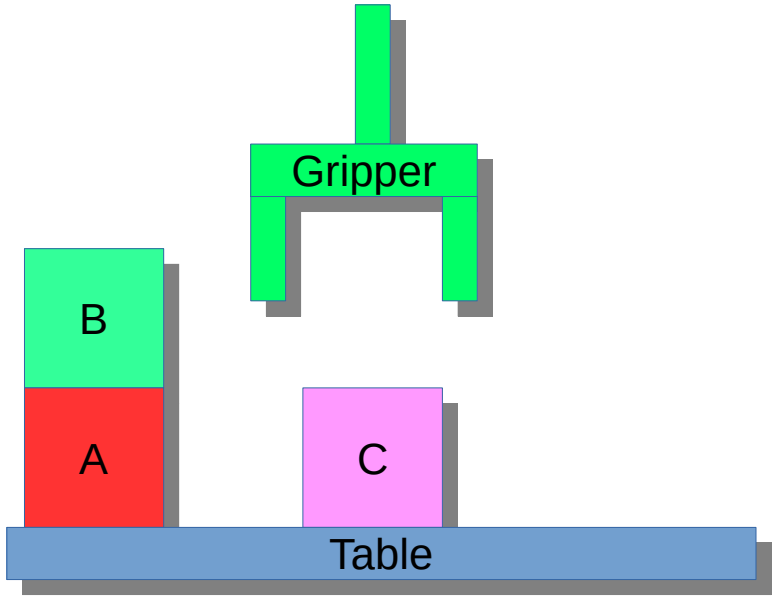
- Traditionally, predicates are used like types
  - block(block_a), block(block_b)..

- PDDL has types and type-checking
  - (:types
    block_a, block_b, block_c – block
    gripper_a – gripper
    table_a - table)



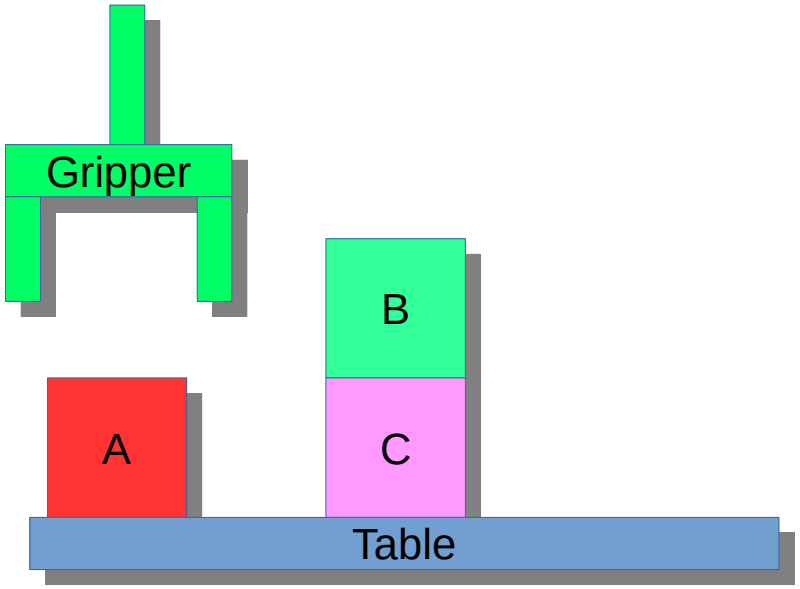
World States

- Predicates describe the state of the world
  - on_table(block_a), on_table(block_c)
  - stacked(block_b, block_a)
  - clear(block_b), clear(block_c)
  - gripper_empty(gripper_a)



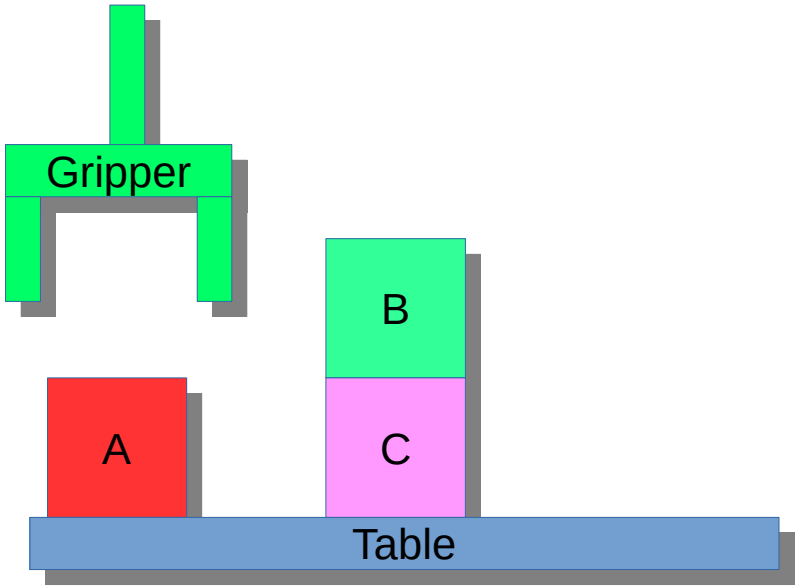
World States

- A different world state uses different predicates
  - on_table(block_a), on_table(block_c)
  - stacked(block_b, block_c)
  - clear(block_b), clear(block_a)
  - gripper_empty(gripper)

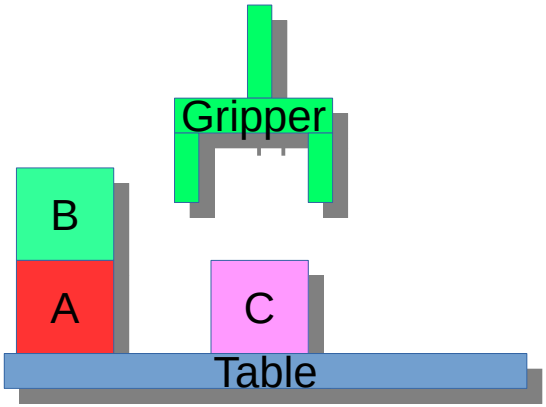


Start States and Goal States

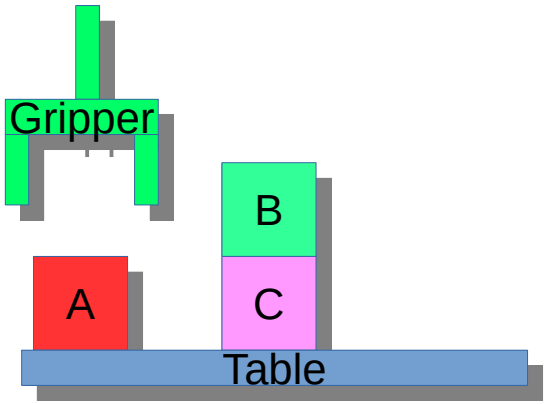
- Start state
  - Current state of the world, or the starting state of your plan
- Goal state
  - The state that you wish to reach



Start States and Goal States



- on_table(block_a), on_table(block_c)
- stacked(block_b, block_c)
- clear(block_b), clear(block_a)
- gripper_empty(gripper)



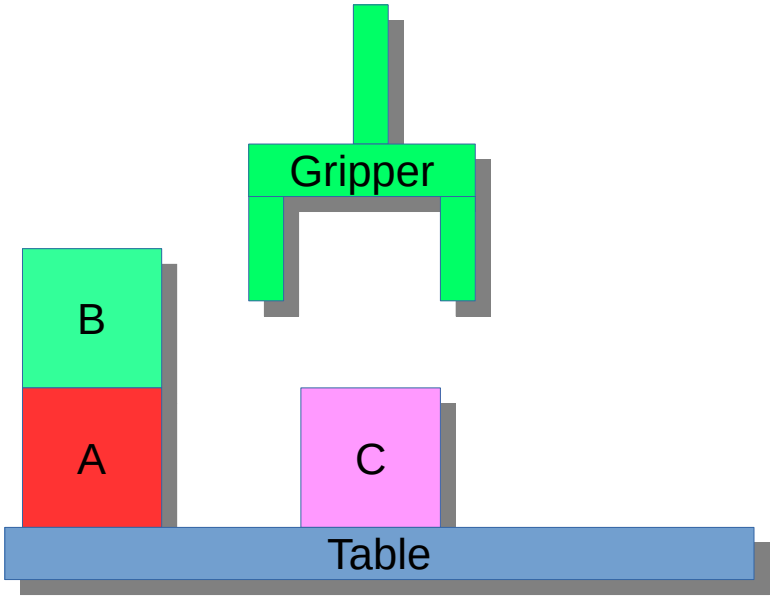
stacked(block_b, block_c)

- While your start state must be complete, generally your goal state can state only those predicates that you require to be true

Actions

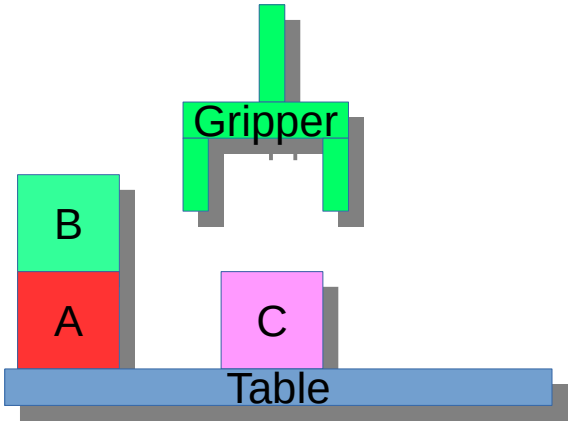
- Actions permute world state

- Actions have
  - A name
  - Parameters
  - Preconditions
  - Effects



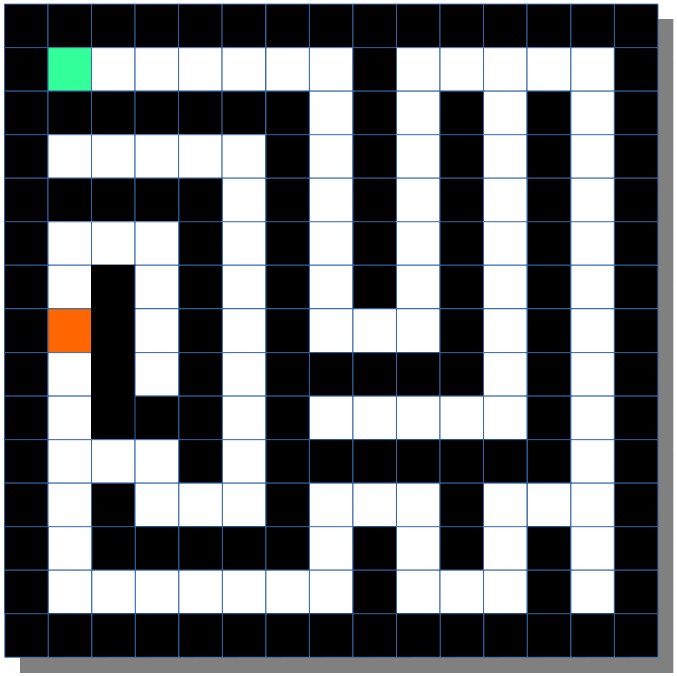
Actions

- (:action grasp-block
  :parameters (?g – gripper ?b – block)
  :precondition (and (empty ?g) (clear ?b))
  :effect (and
  (not (empty ?g))
  (not (clear ?b))
  (in_gripper ?b ?g)
)
)



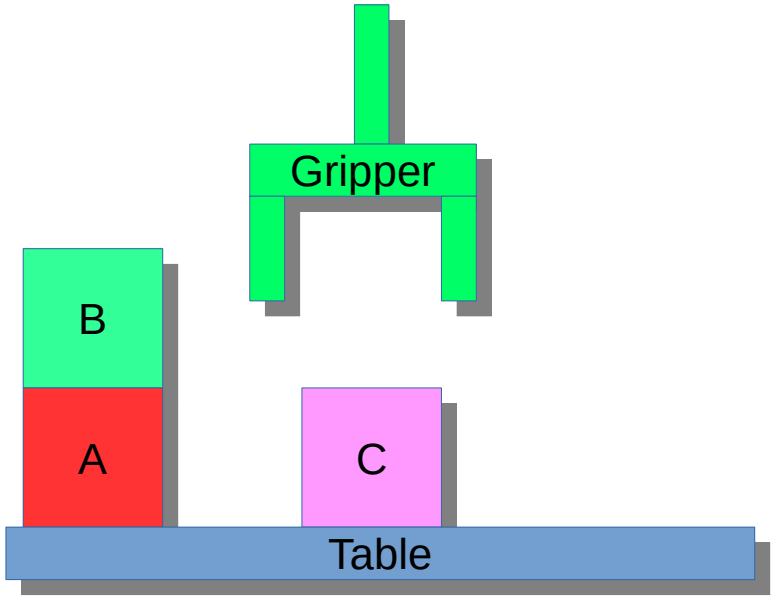
Actions

- Preconditions tell us what must be true for us to be able to take an action
- Effects tell us how the action changes the world
- The ready queue is filled with possible permutations based on the effects of actions whose preconditions are satisfied
  - Can't go left
  - Can't go right
  - Can go up → resulting in the agent being 1 square up
  - Can go down → resulting in the agent being 1 square down

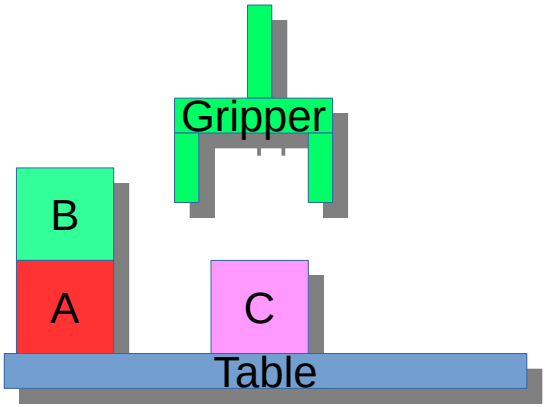


Actions

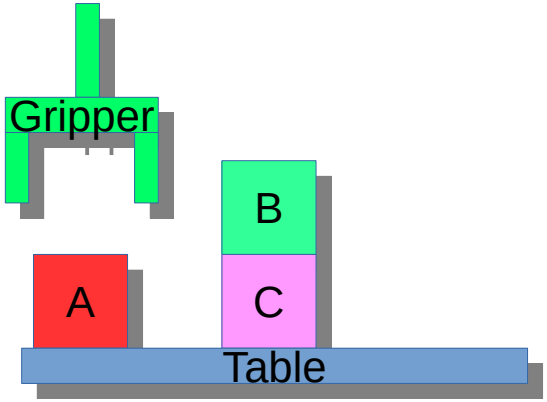
- Can't grasp-block(gripper_a, block_a)
  - So this action isn't taken
- Can grasp-block(gripper_a, block_b)
  - Goes into ready queue
- Can grasp-block(gripper_a, block_c)
  - Goes into ready queue



Plans



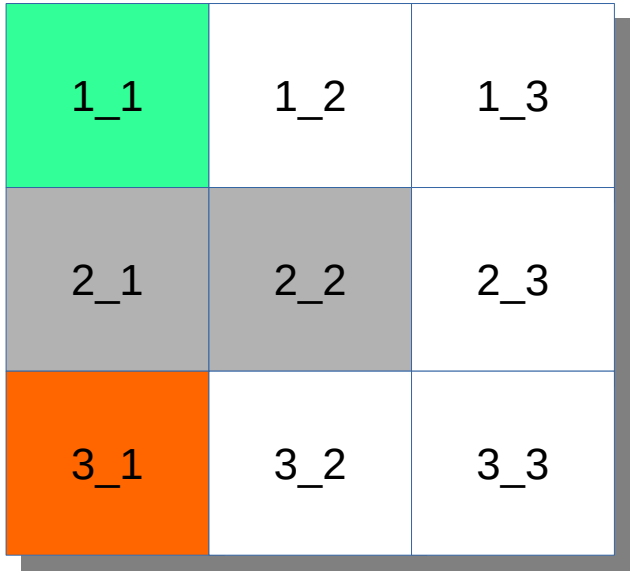
A plan takes the world from a start state to a goal state



grasp-block(gripper_a, block_b)

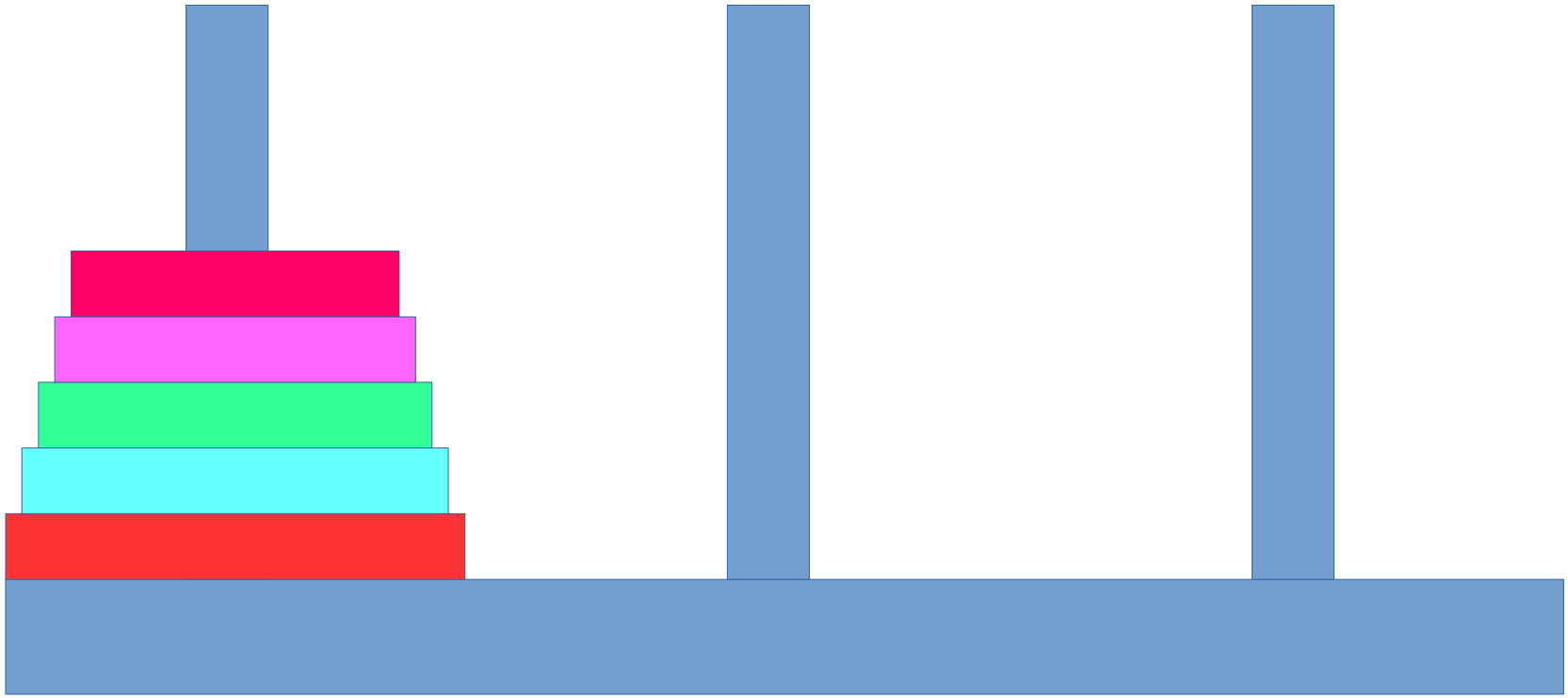
unstack-block(gripper_a, block_b, block_a)

stack-block(gripper_a, block_b, block_c)

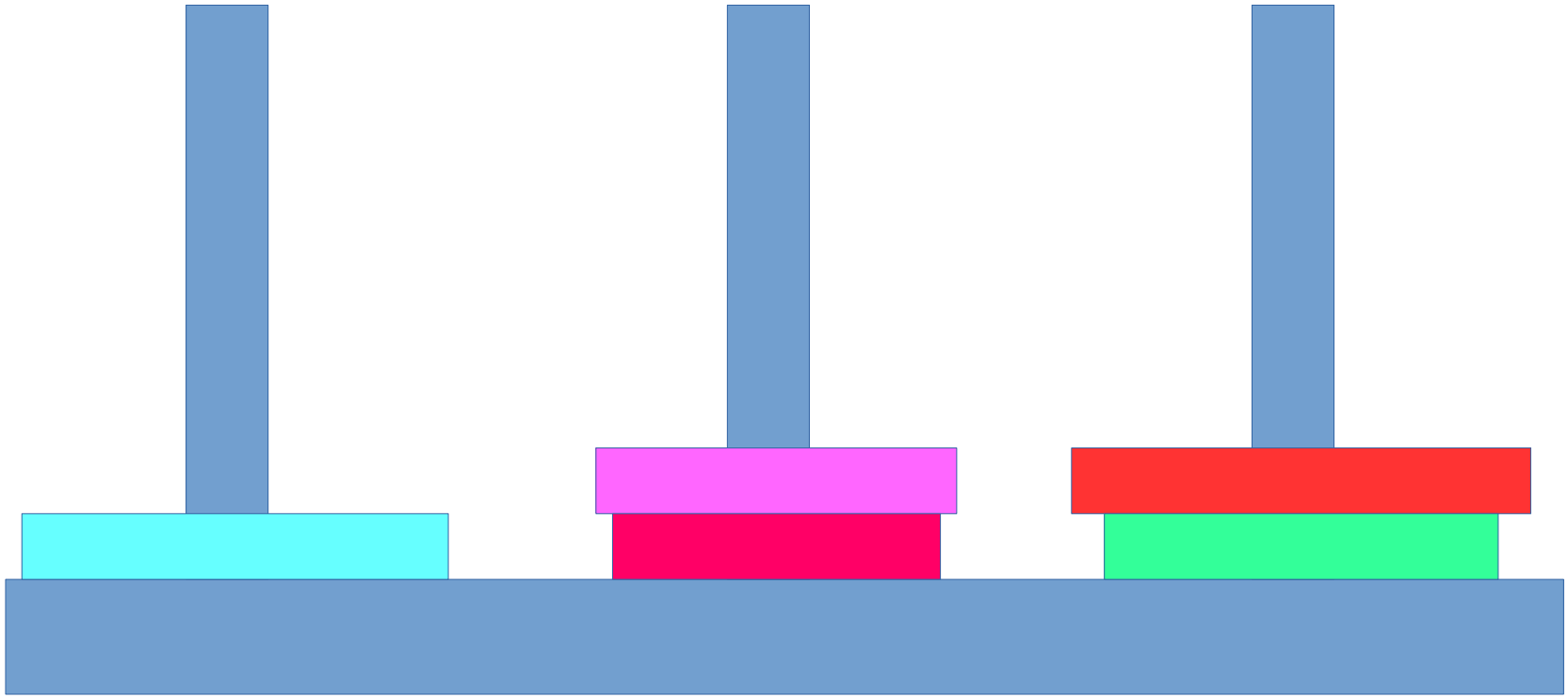

1_1
1_2
1_3
2_1
2_2
2_3
3_1
3_2
3_3

1_1	1_2	1_3
2_1		2_3
3_1	3_2	3_3

Tower of Hanoi



Tower of Hanoi



Tower of Hanoi

- Rules
  - Generally three posts
  - However many disks
  - Goal: Get all of the disks on the same post, with the biggest disk on bottom and progressively smaller disks towards the top.
  - Main constraint: You can only stack a disk onto a smaller disk
- Okay! Let's write this!