# CS 309: Autonomous Intelligent Robotics
# FRI I

## Lecture 7: AI as Search and PDDL

Instructor: Justin Hart

http://justinhart.net/teaching/2019_spring_cs309/

# A couple of quick notes

- You should be able to use the lab machines in the 3rd floor computing lab in GDC to do your homework.
- Mentors are available for your help in GDC 3.414

# Makefiles

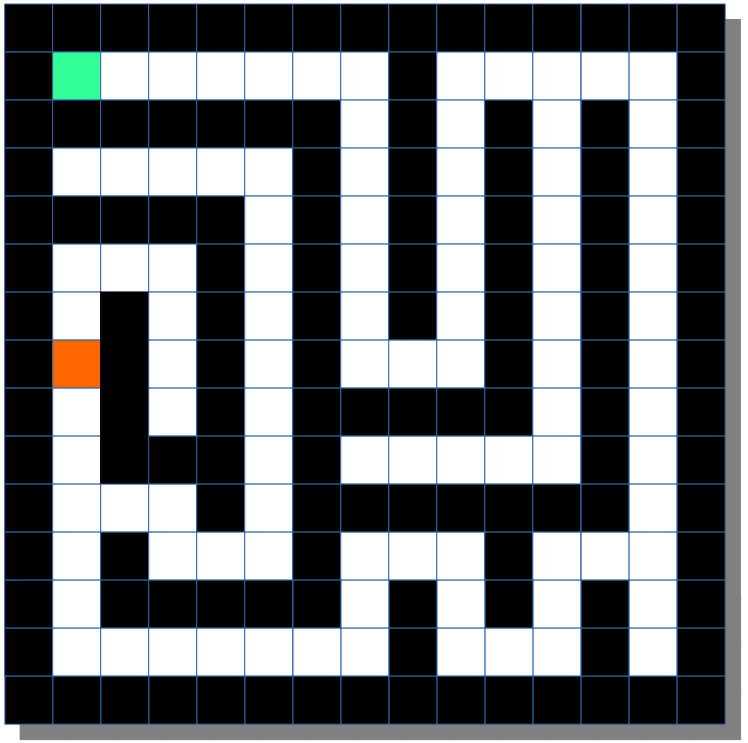
- Run with the command “make”
- Named “Makefile”

- Variables
  - CXX, $(CXX)
- Targets
  - ex14
  - Clean
- Dependencies
  - ex14.cpp
- Commands
  - $(CXX) ex14.cpp -o ex14
  - rm ex14 *~

```
1 CXX=g++
2
3 ex14: ex14.cpp
4   $(CXX) ex14.cpp -o ex14
5
6 clean:
7   rm ex14 *~
```

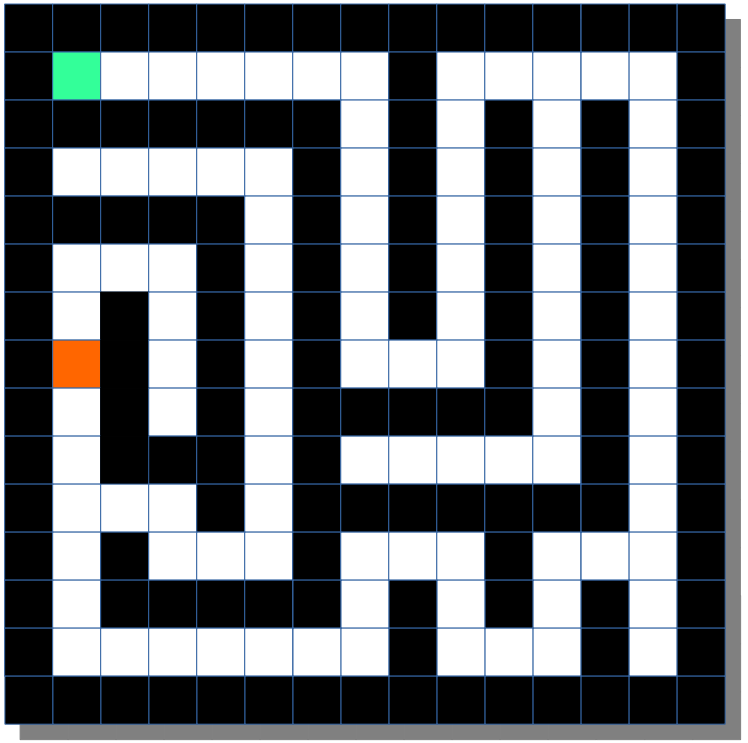
# AI as search

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



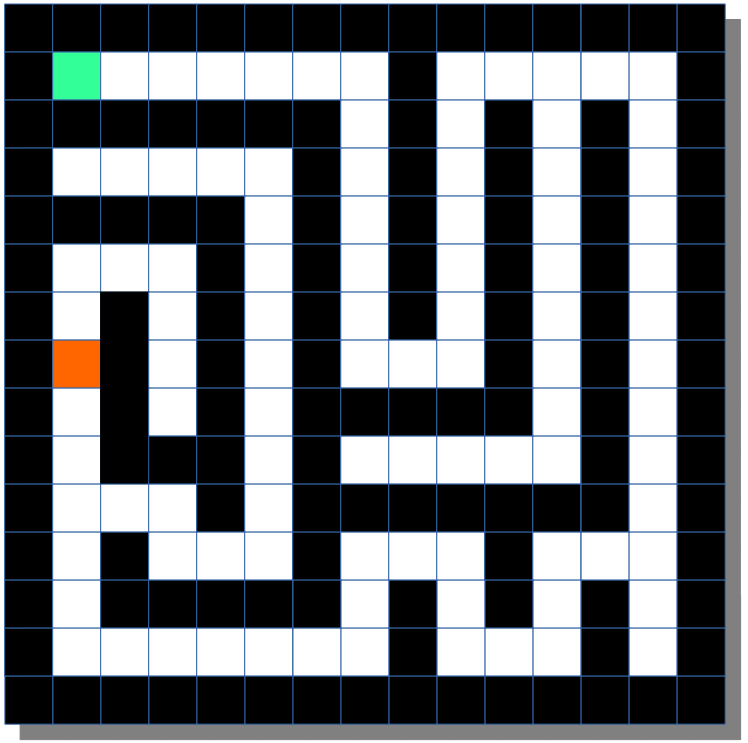
# AI as search

- There are two basic types of solutions
  - Satisficing solutions
    - Work, but are not known to be optimal
  - Optimal solutions
    - Are intended to be optimal



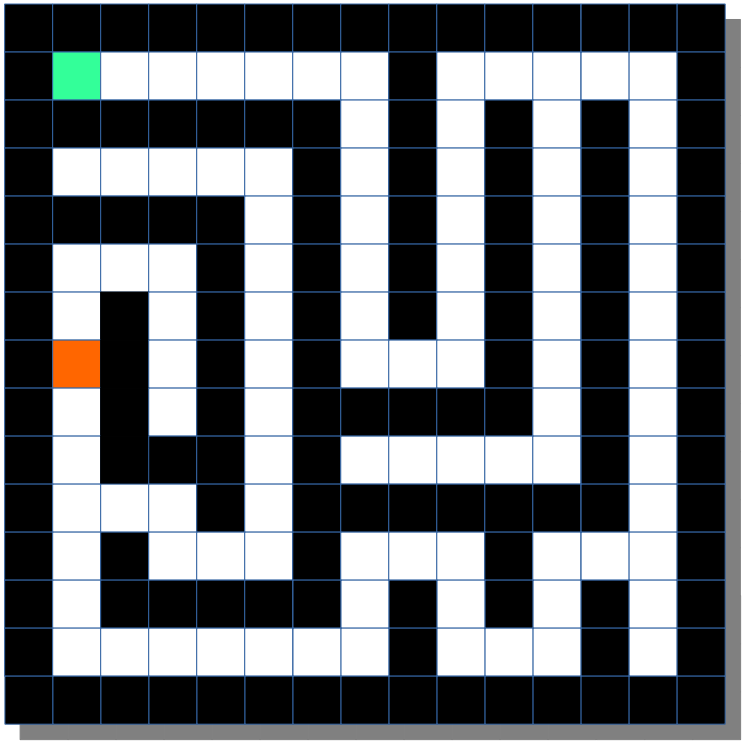
# AI as search

- The agent is the orange dot, trying to get to the green dot.
- Possible moves are up, down, left, right.
- Here, left and right are not possible, so when the search algorithm attempts them, they fail.
- Up and down work.



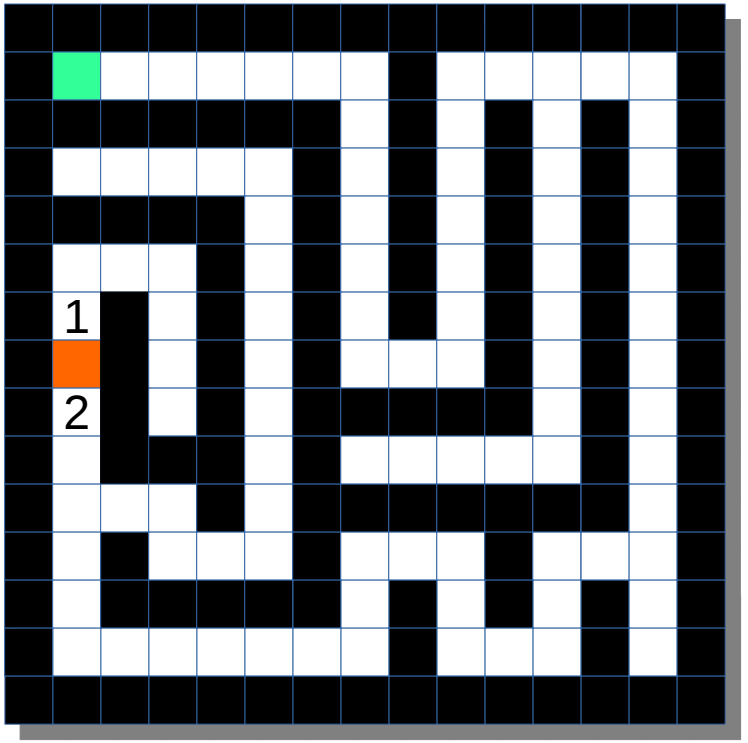
# Three basic search patterns

- Breadth-first search
- Depth-first search
- A*



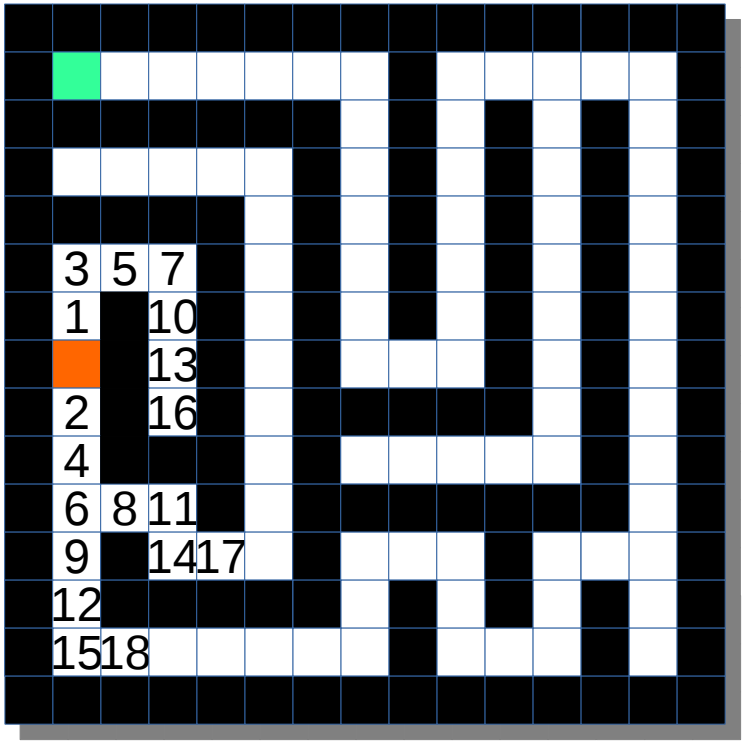
# Breadth-first search

- Expand search nodes
  - Up - Works
  - Down - Works
  - Left – Fails
  - Right – Fails
- Enter these into the “ready queue”



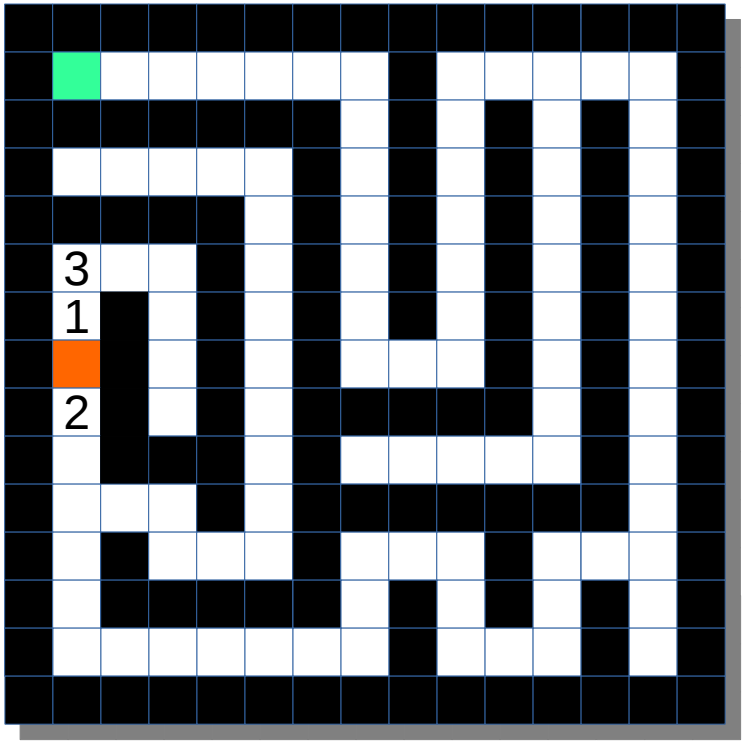

# Breadth-first search

- Continue until you have a solution



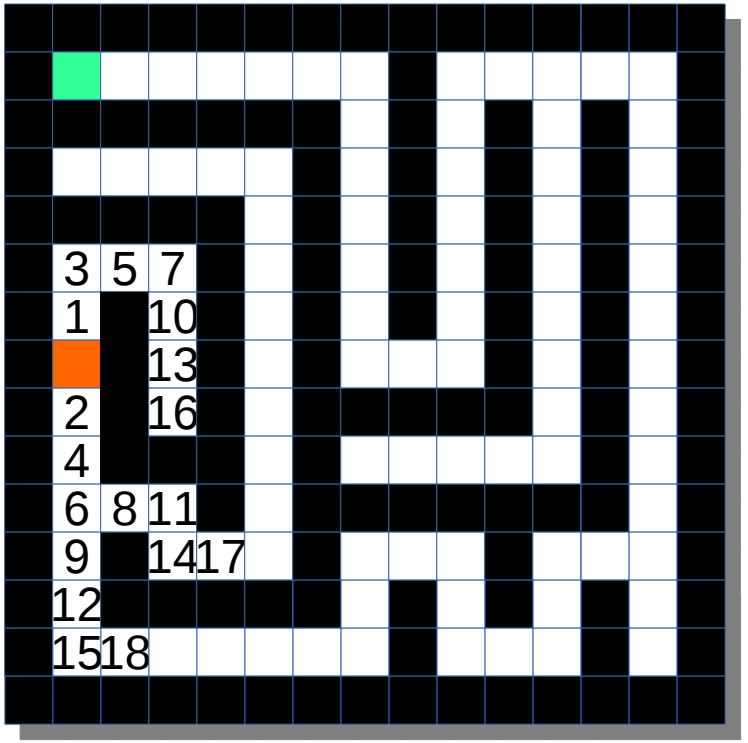

# Breadth-first search

- Now try the ones in the ready queue in First In First Out (FIFO) order



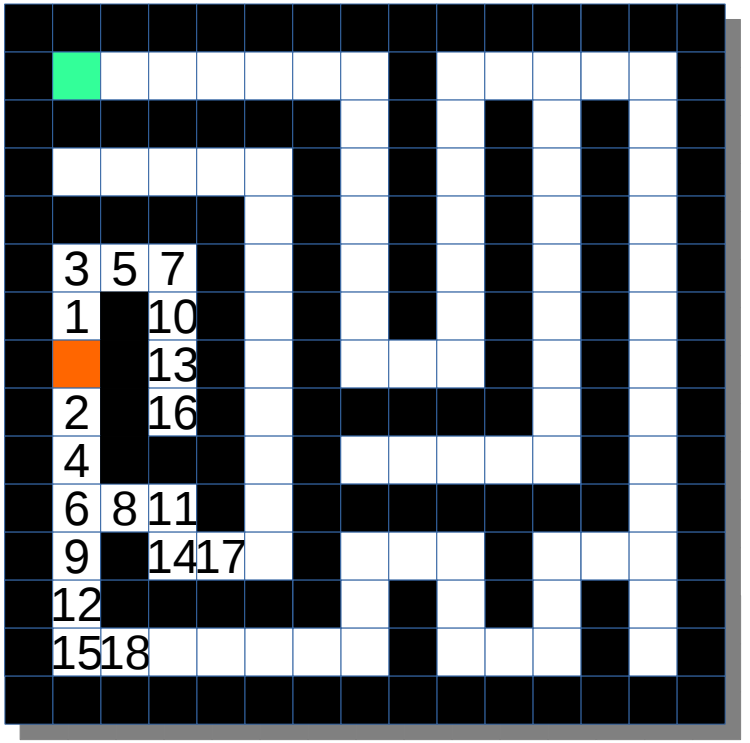

# Breadth-first search

- Continue until you have a solution



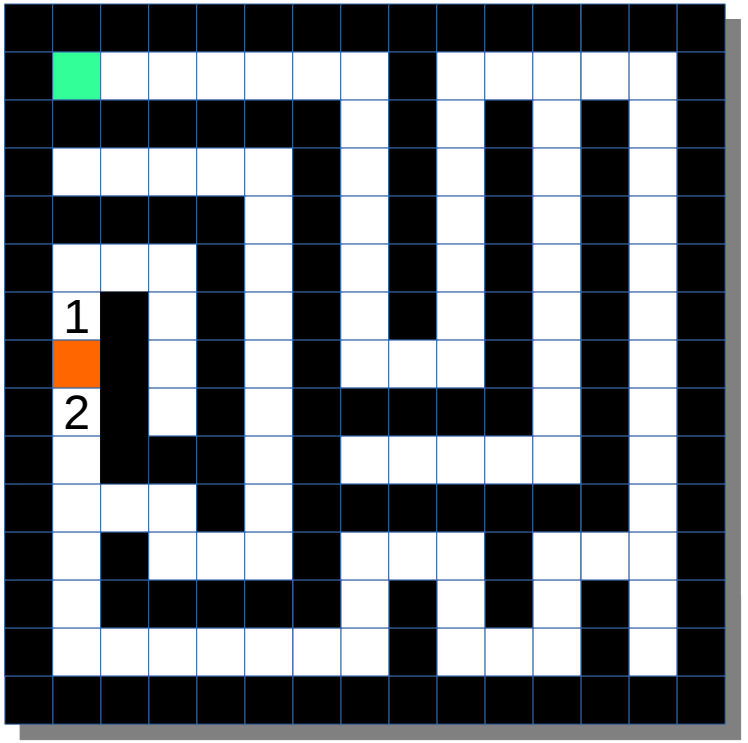

# Breadth-first search

- Breadth-first search is “complete” in that in will eventually explore the entire space
- It is “optimal” in that the first solution found takes the fewest steps

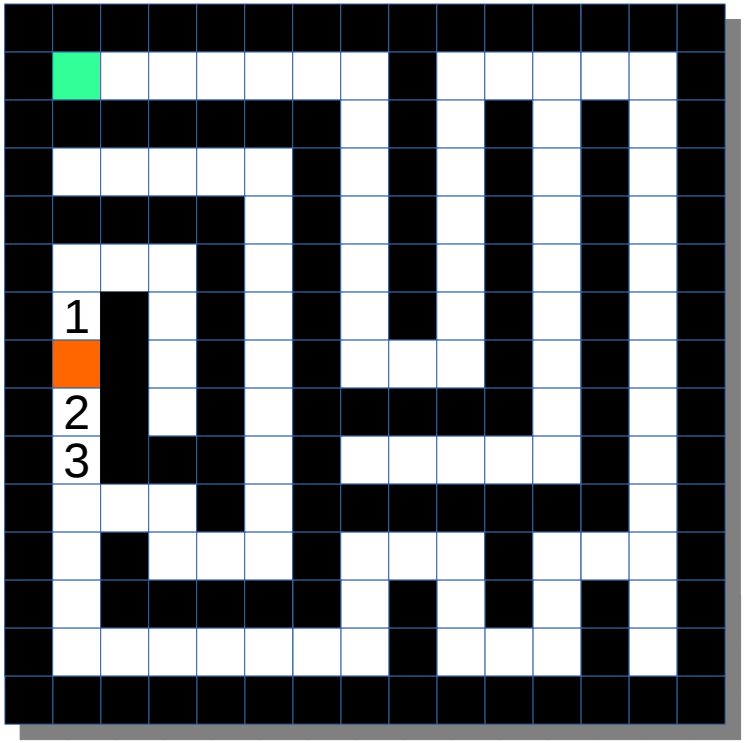
# Depth-first search

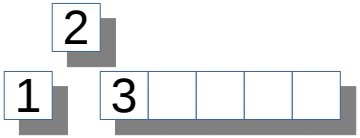
- Depth-first search tries to explore one path completely before moving on
- May faster than breadth-first, but may miss solutions if it takes the first found.




# Depth-first search

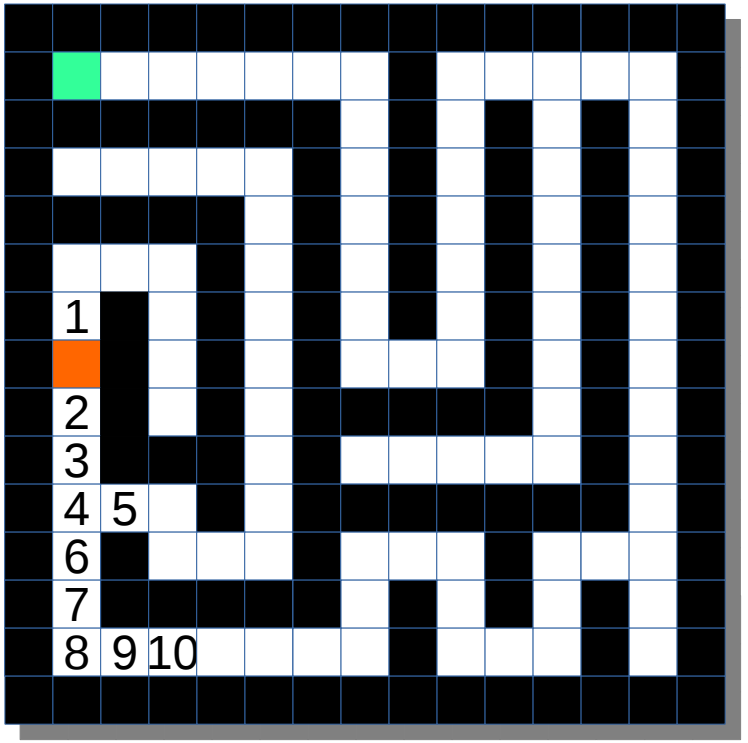
- Uses a First In Last Out (FILO) pattern

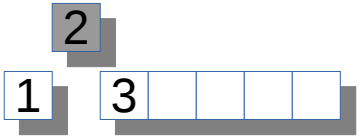


# Depth-first search

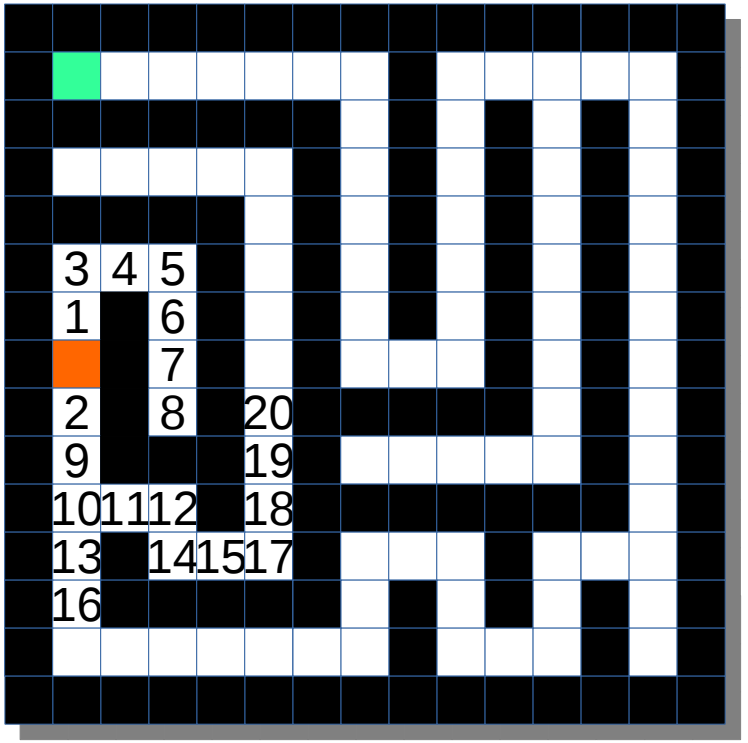
- Uses a First In Last Out (FILO) pattern



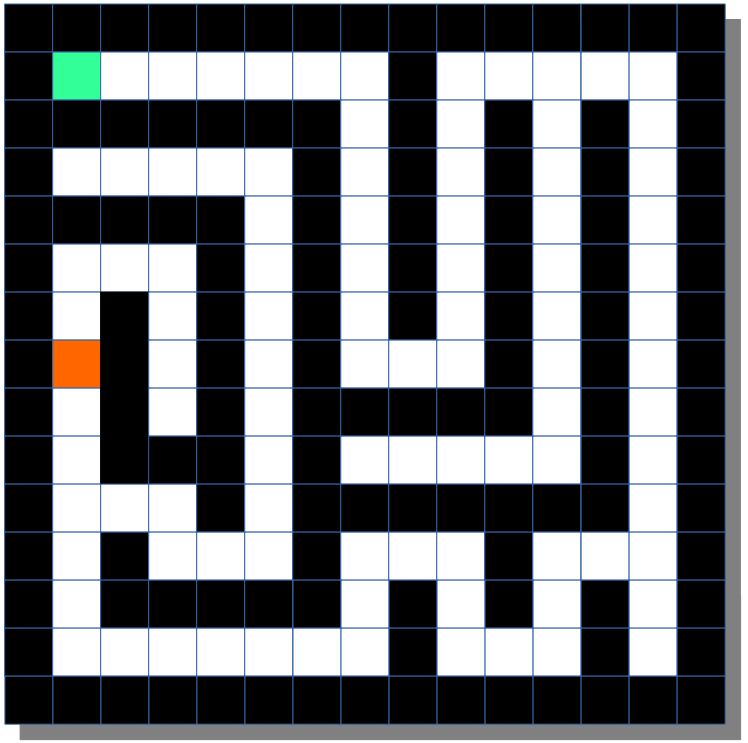
# A* search

- The queue becomes a priority queue, with those nodes assumed to have the lowest cost going to the front



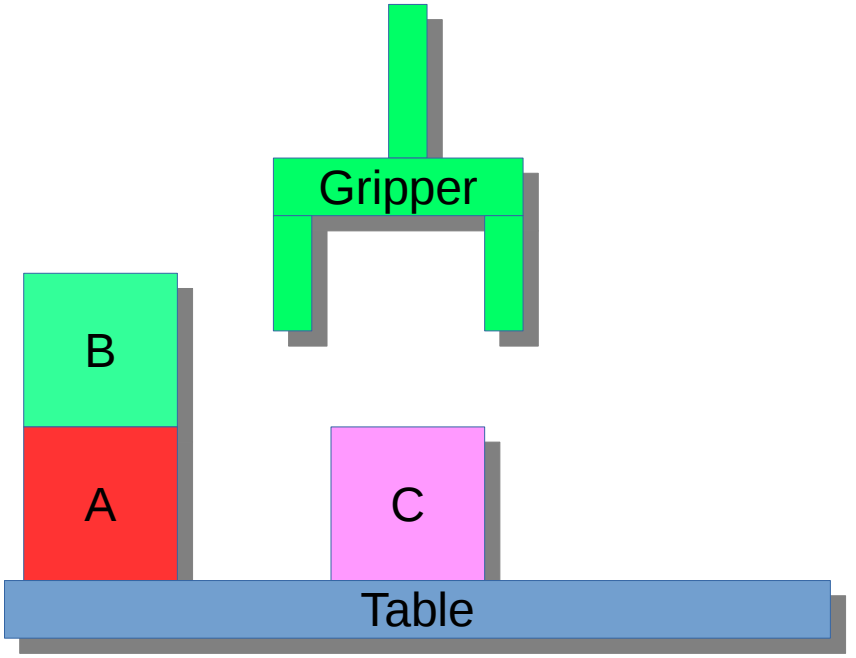

# Modern planning

- Planning algorithms have come a long way but still integrate these basic ideas
- The development of these algorithms is often its own class



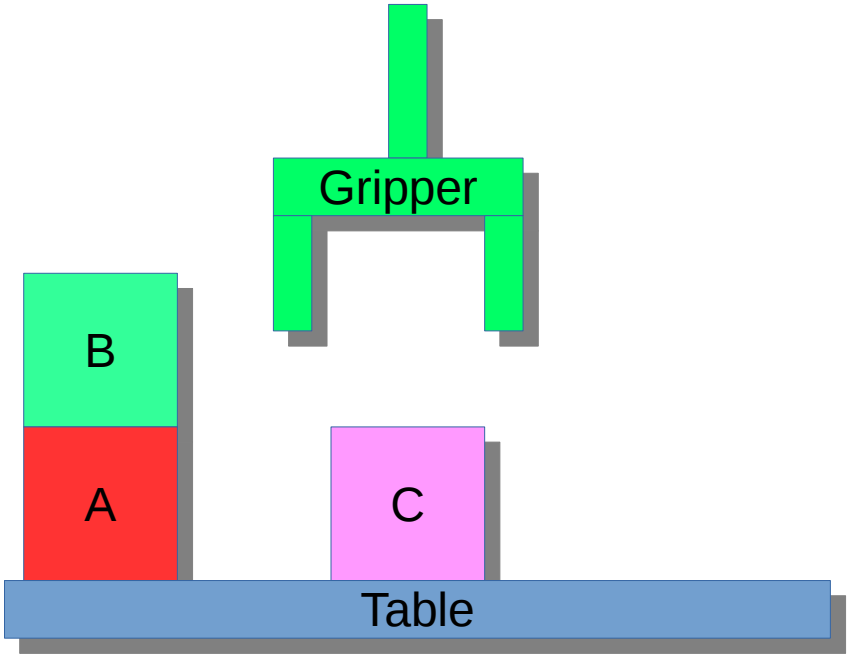
# Blocks world

- The planning equivalent of “Hello World” is “Blocks World”
- Blocks arranged on a table with a robot gripper



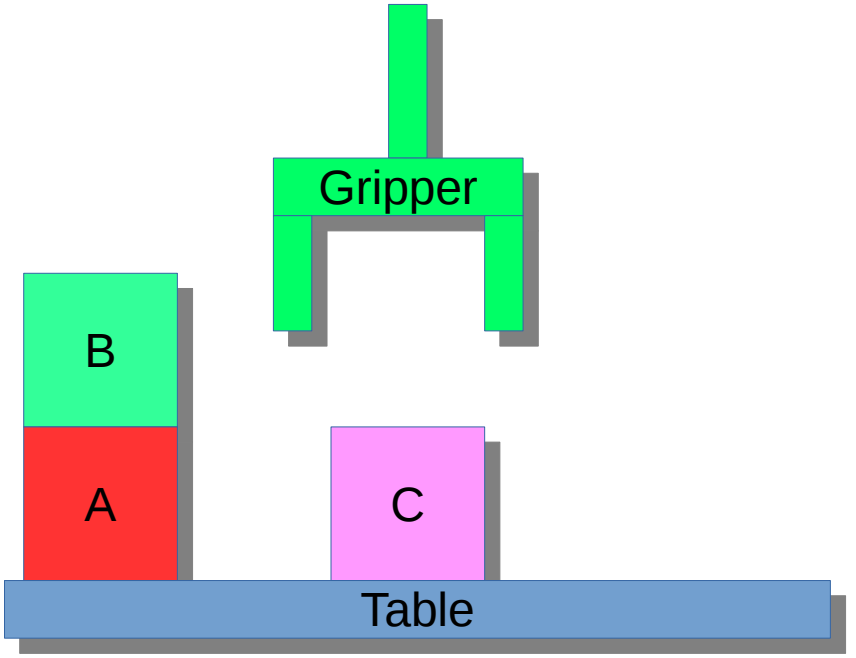
# Atoms

- Atoms represent the things we can talk about in the world
  - block_a, block_b, block_c
  - table_a
  - gripper_a



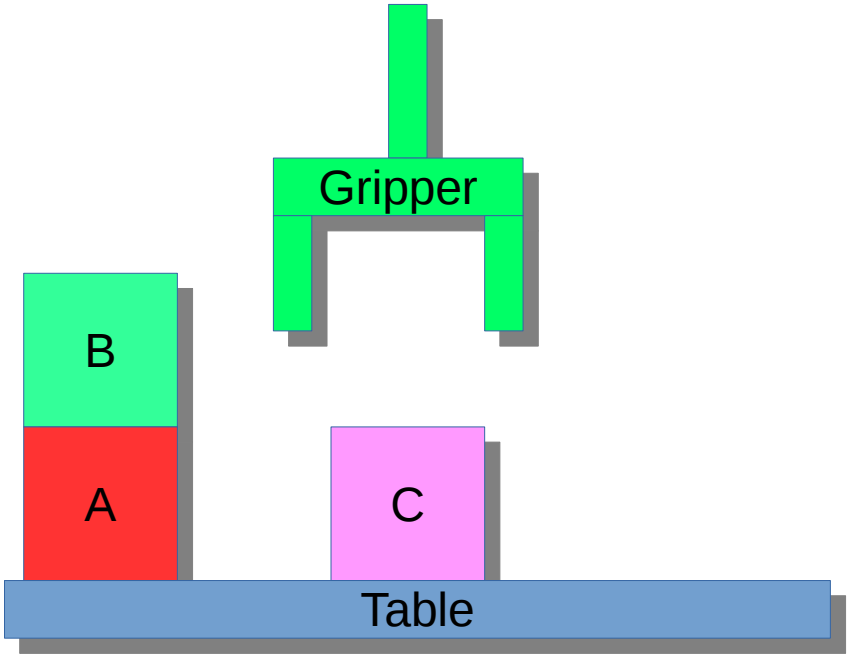
# Predicates

- 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)



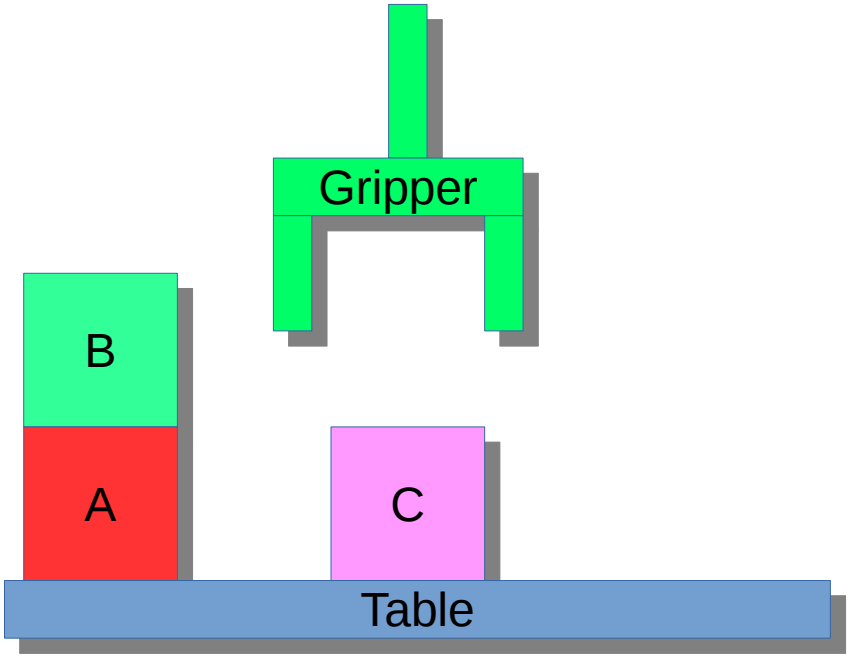
# Predicates

- 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)



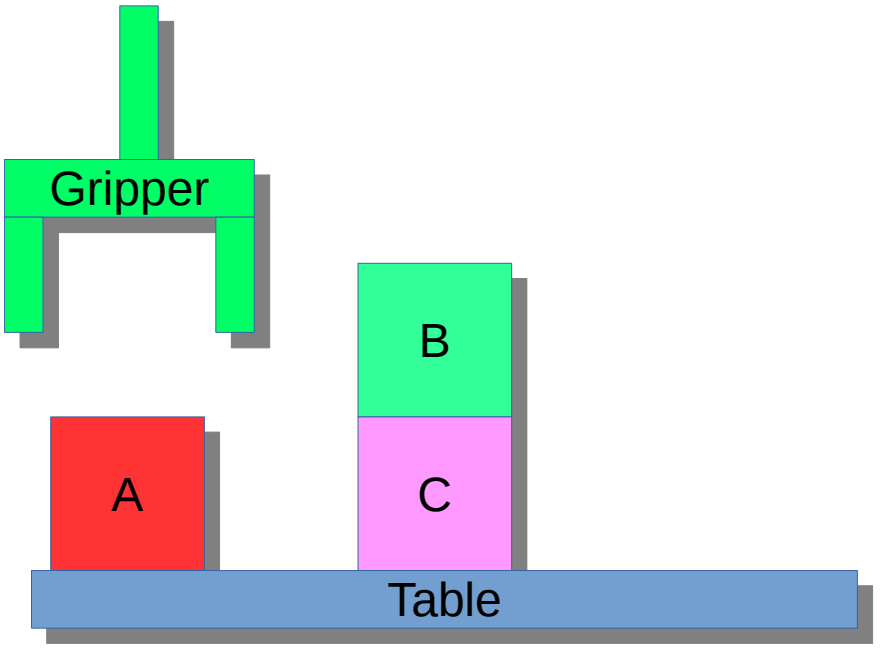
# World states

- The predicates used in the previous slide 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)



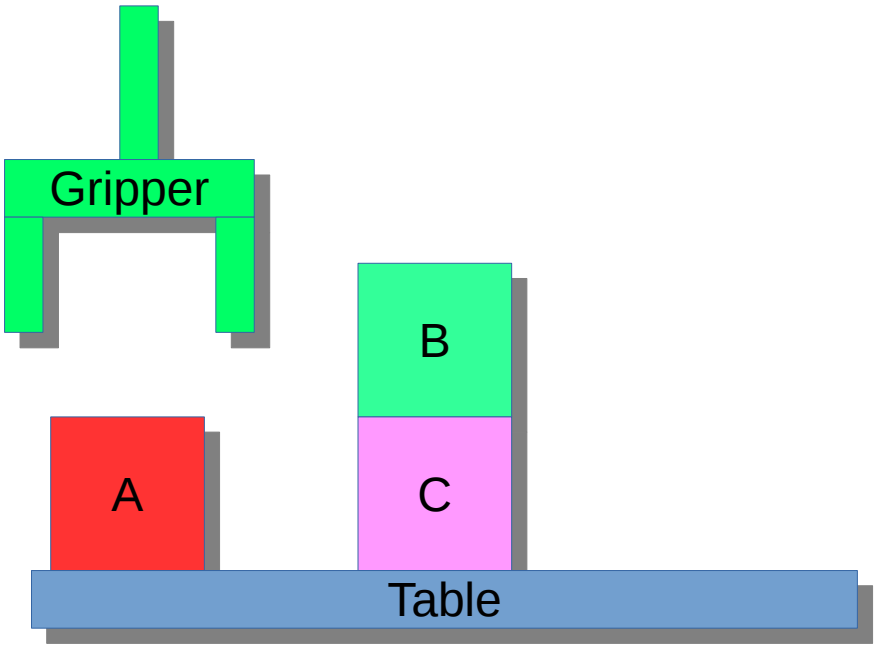
# World states

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

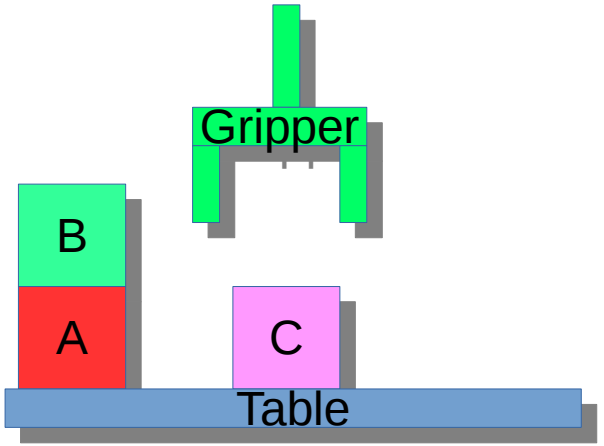


# Start states and end states

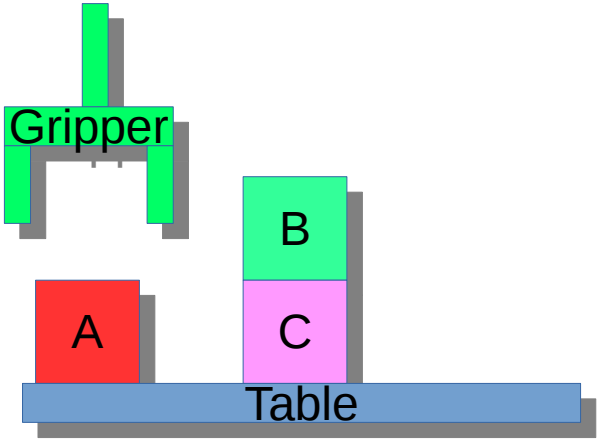
- Start state
  - The current state of the world, or the starting state of your plan
- Goal state
  - The state that you wish to reach



# 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)

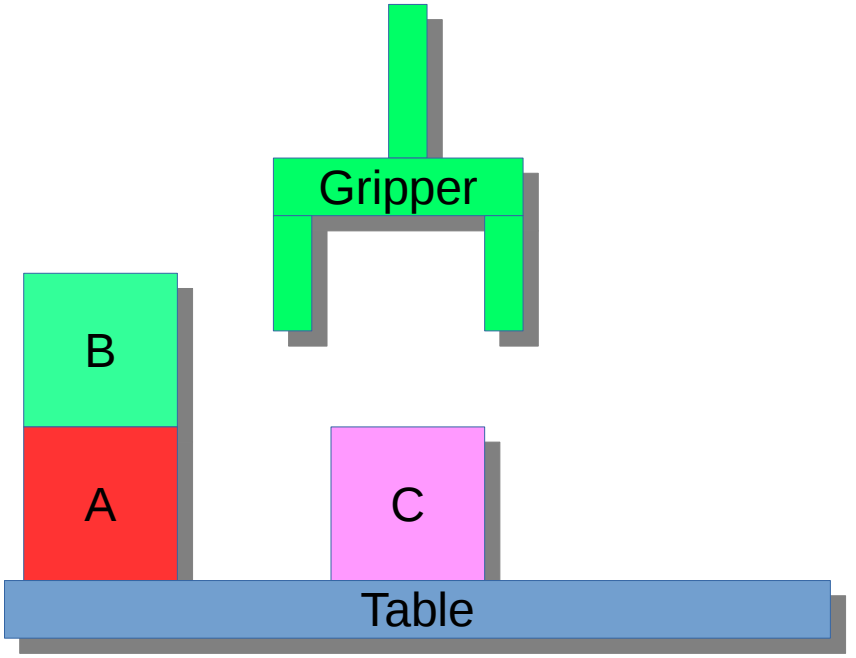



stacked(block_b, block_c)

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

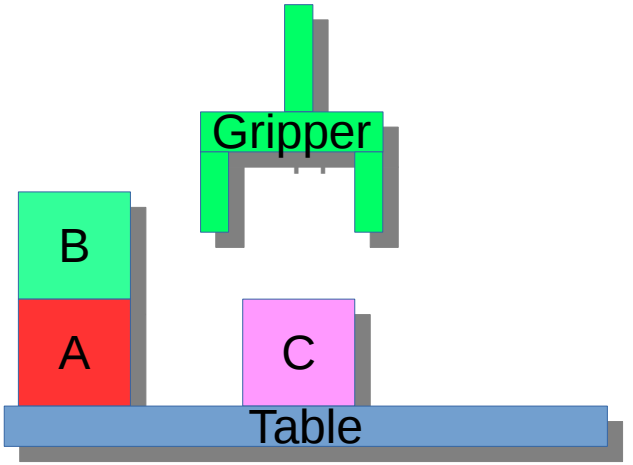
# Actions

- Actions permute world state
- Actions have
  - A name
  - Parameters
  - Preconditions
  - Effects



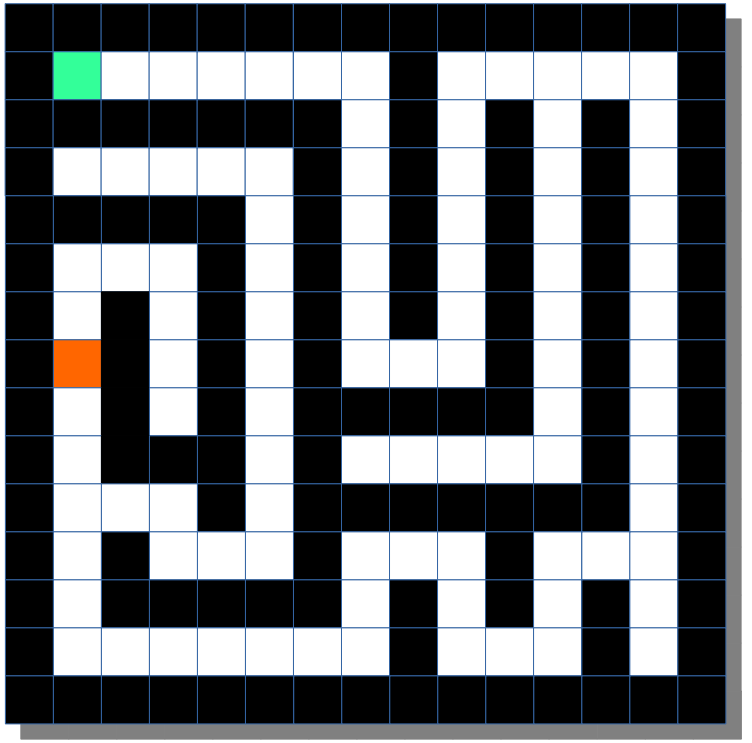
# Actions

```
(: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)
            )
)
```



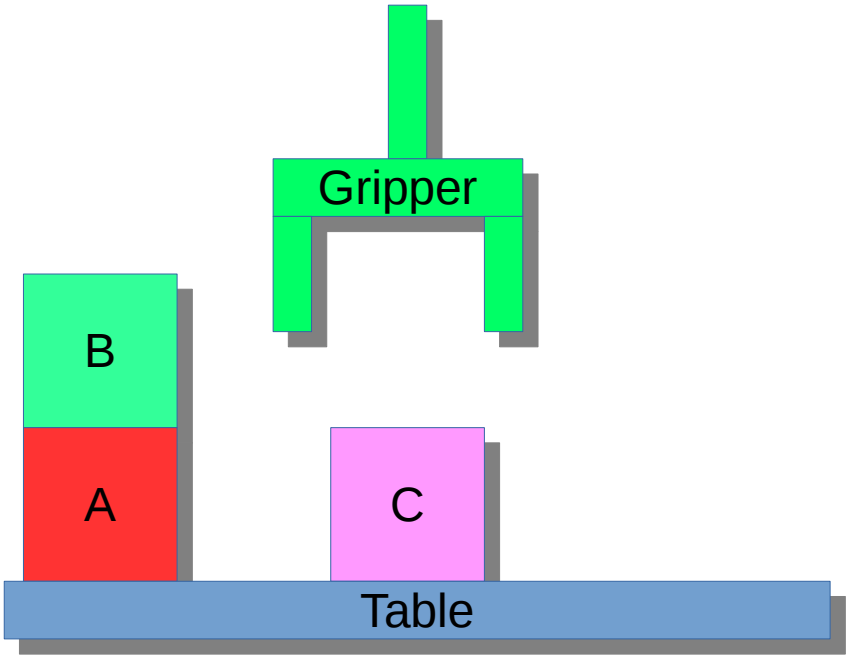
# Actions

- Think back to the maze
- 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
- Our 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

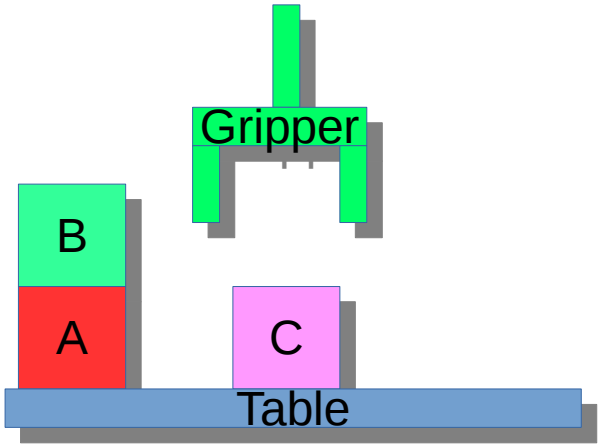


# Actions

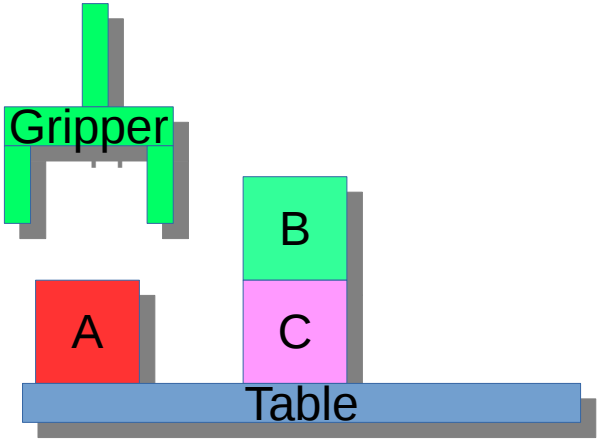
- Can't grasp-block(gripper_a, block_a)
  - So this action isn't taken
- Can grasp-block(gripper_a, block_b)
  - Goes into ready queue
- Can grasp-block(gripper_a, block_c)
  - Goes into ready queue



# Plans



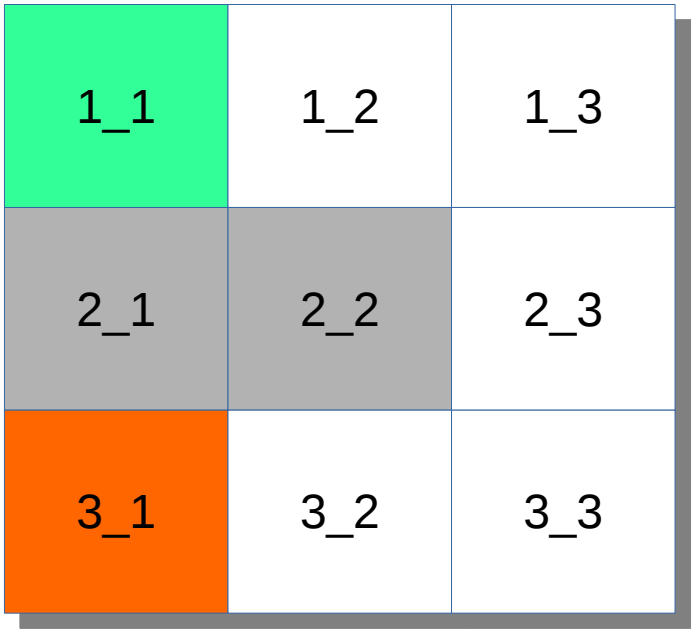

A plan takes the world from a start state to a goal state




grasp-block(gripper_a, block_b)

unstack-block(gripper_a, block_b, block_a)

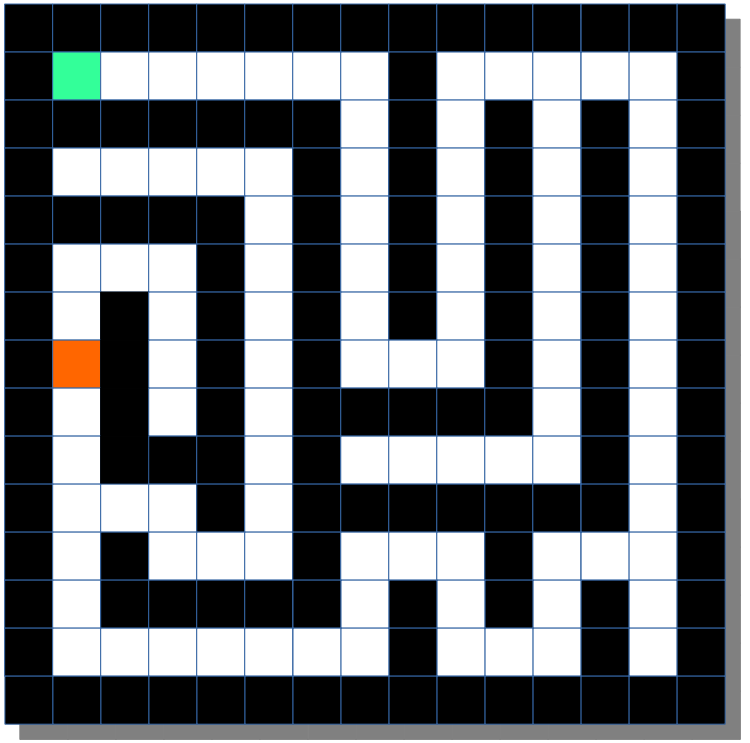
stack-block(gripper_a, block_b, block_c)


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 |

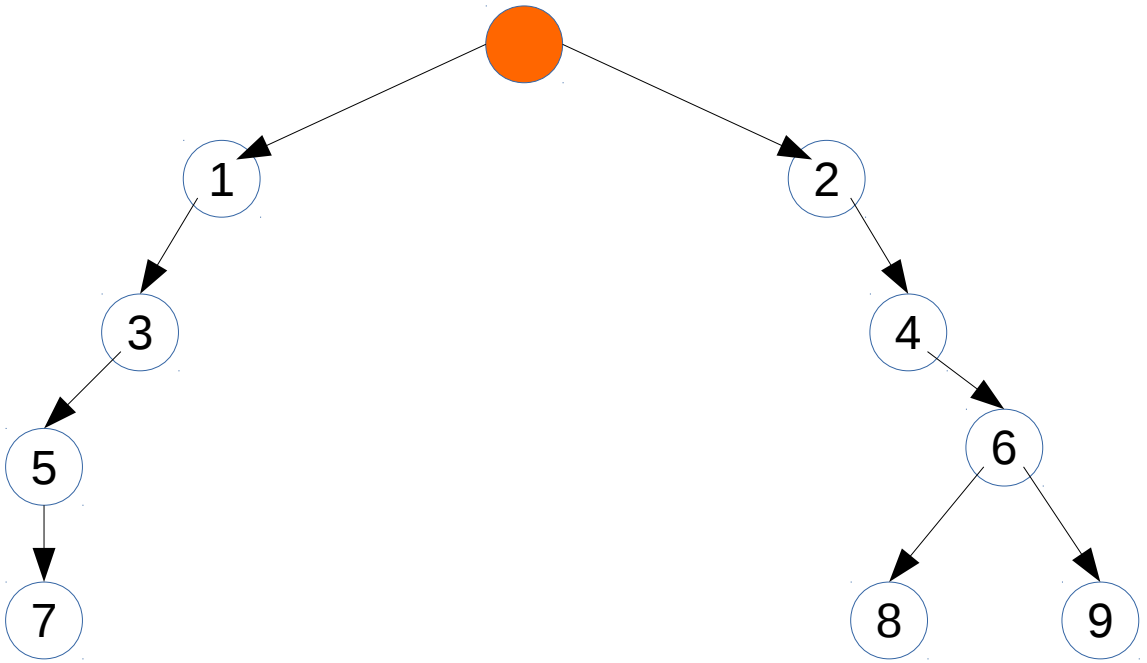
# Last time: AI as search

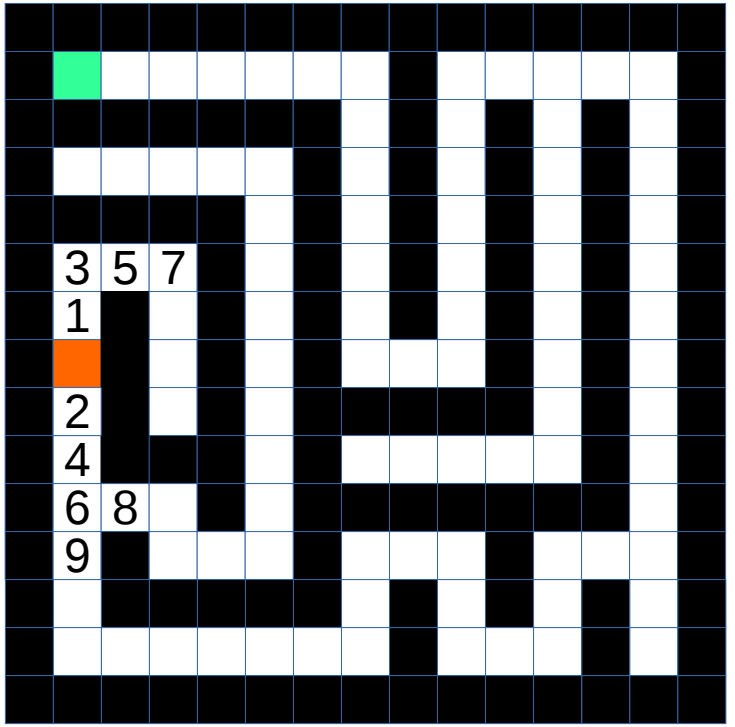
- Search algorithms attempt every action whose preconditions are met.
- These action are put into a queue then tested in the same way.
- This happens until a sequence of actions leads to the goal.



# Breadth-First Search

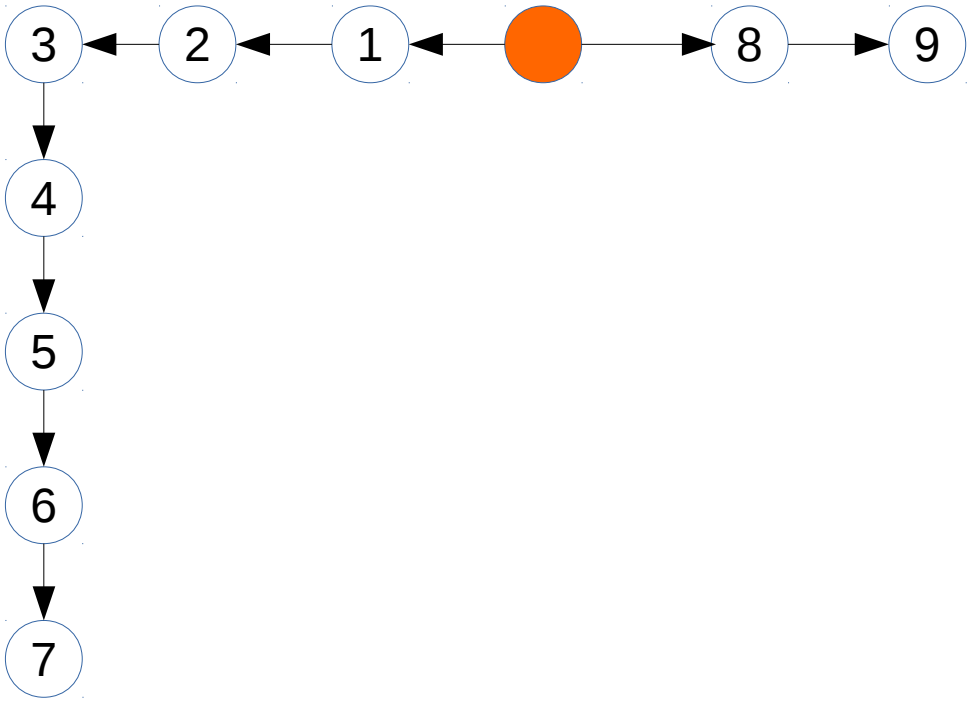
- Adds layers to the graph

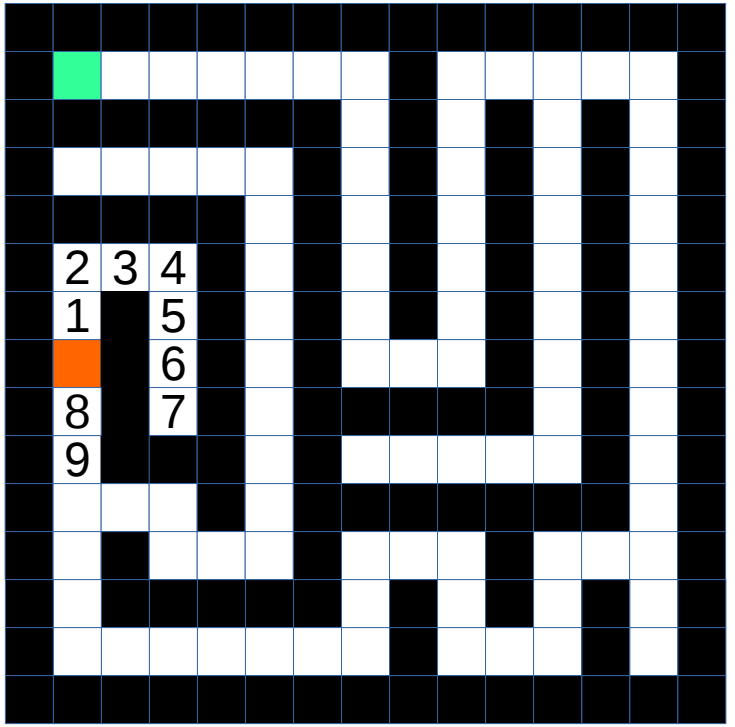


# Depth-First Search

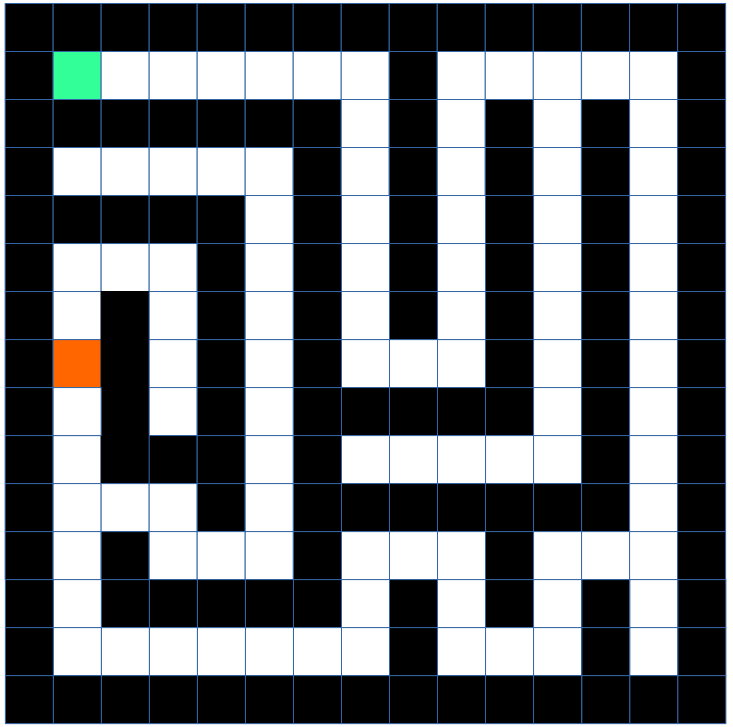
- Goes as deep in the graph as possible first



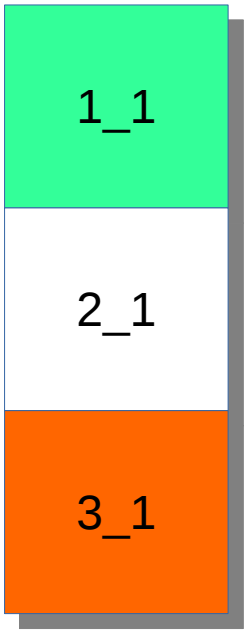



# A*

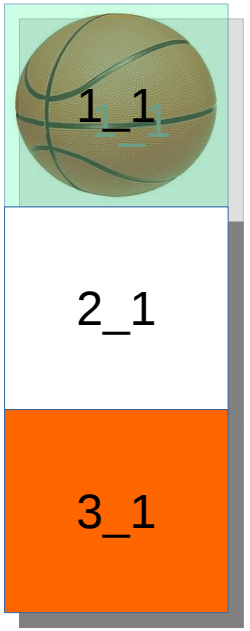
- Prioritizes the “best” points according to a heurstic
- (On this maze, A* would look the same as Depth-First Search)



# Hallway



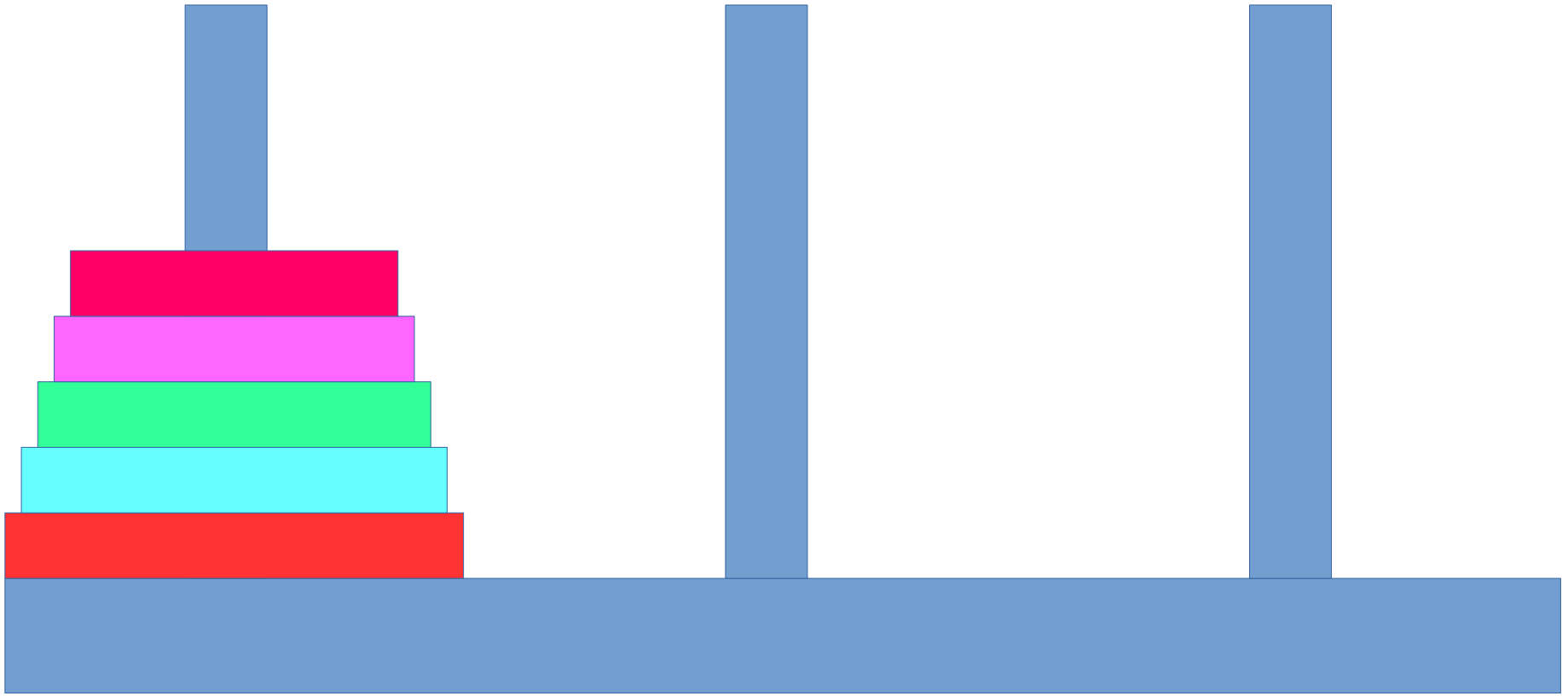
# Hallway



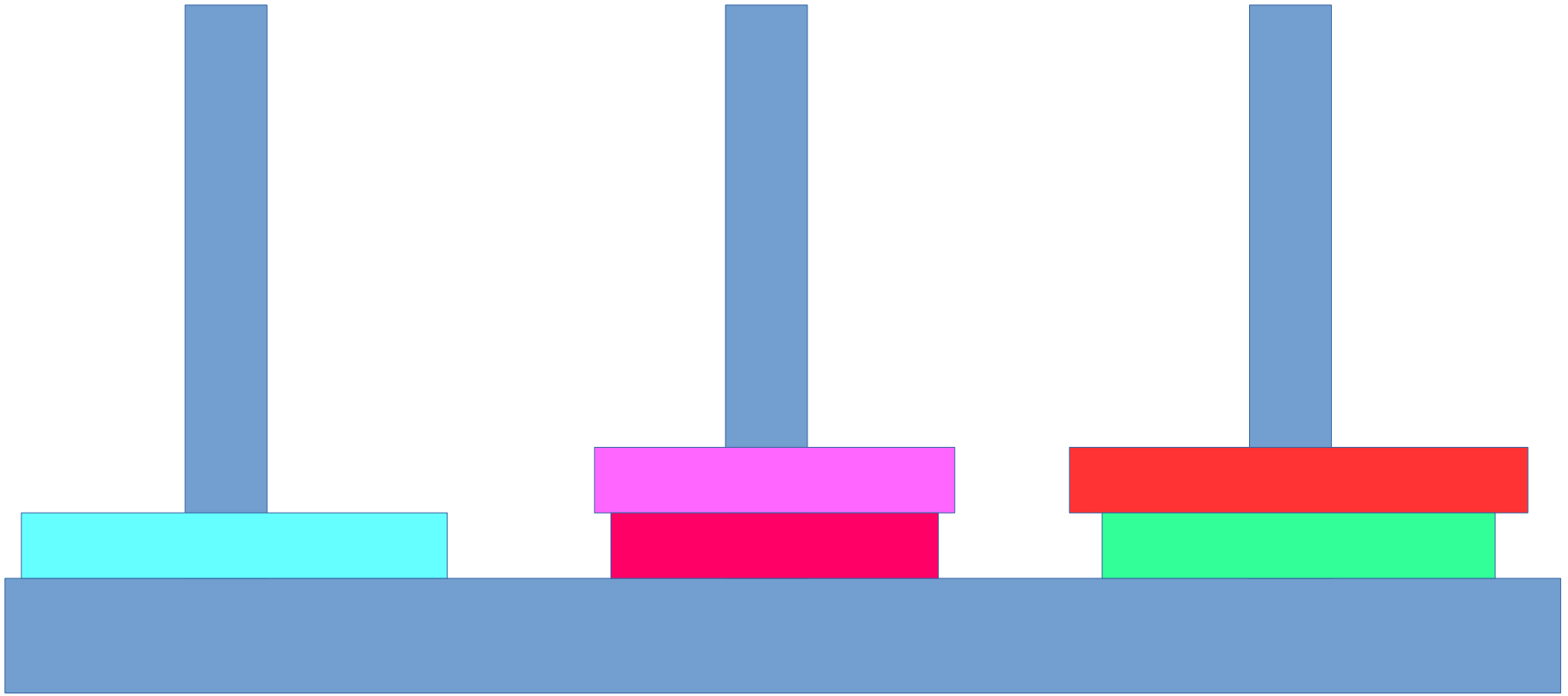
| | | |
|---|---|---|
|  1_1 | 1_2 | 1_3 |
| 2_1 | 2_2 | 2_3 |
| 3_1 | 3_2 | 3_3 |

| | | |
|---|---|---|
| 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
  - 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!