AI Robotics

Path Planning

Overview

Robot Model

Robot

plan

sense

learn

act

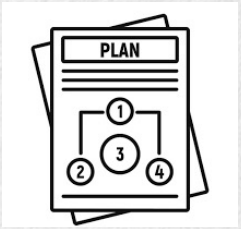
Environment

Robot Model

sensing ✓

localization ✓

mapping ✓

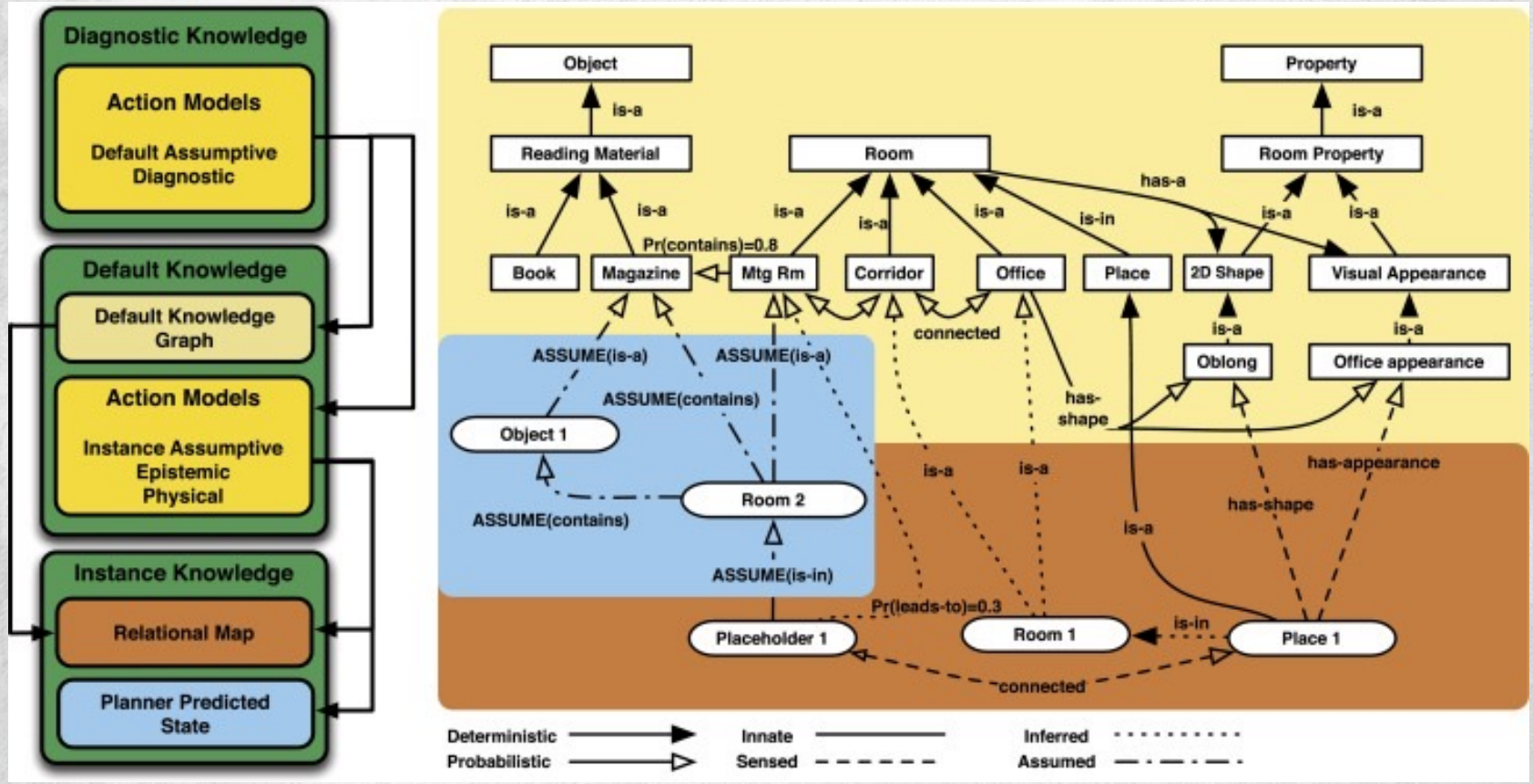


planning ?

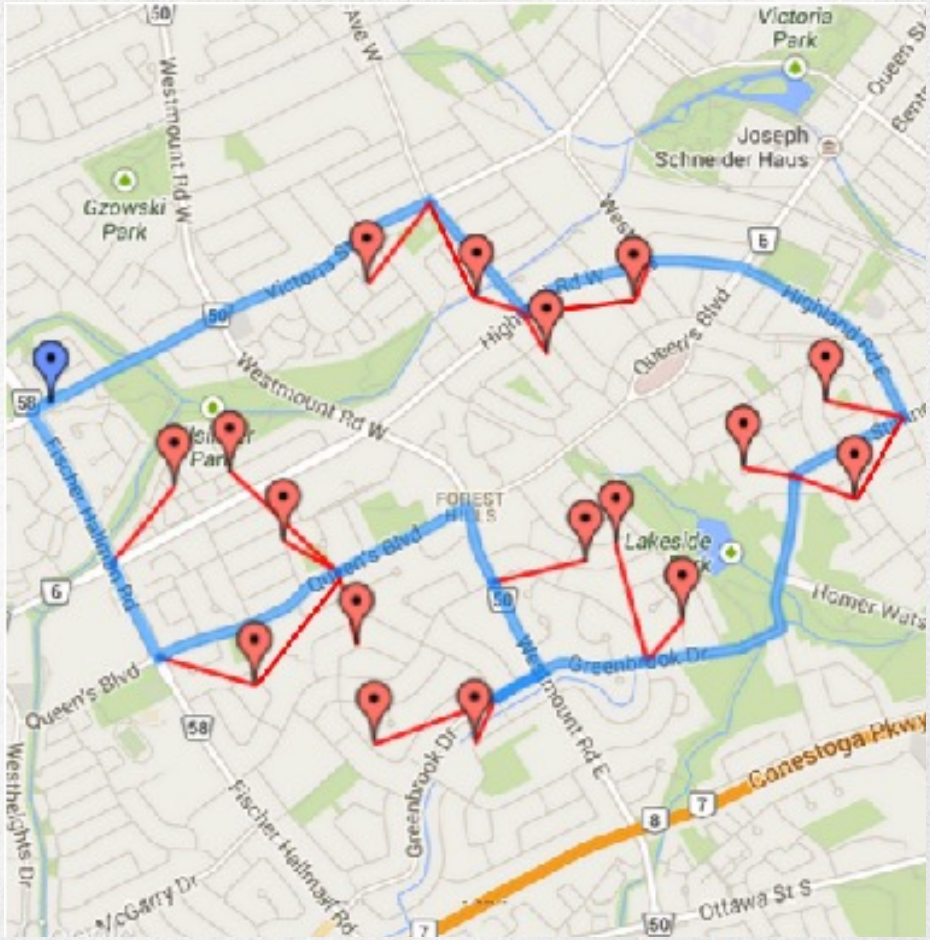
Planning

- Planning consists of any high level decision-making or problem-solving that the robot has to perform
- What might the planning portion of a robot need to know/do?
  - What tasks do I have to perform (Mission Planning)
  - What is the best way to get there? (Path planning)
  - How can I get in the correct configuration? (Motion planning)

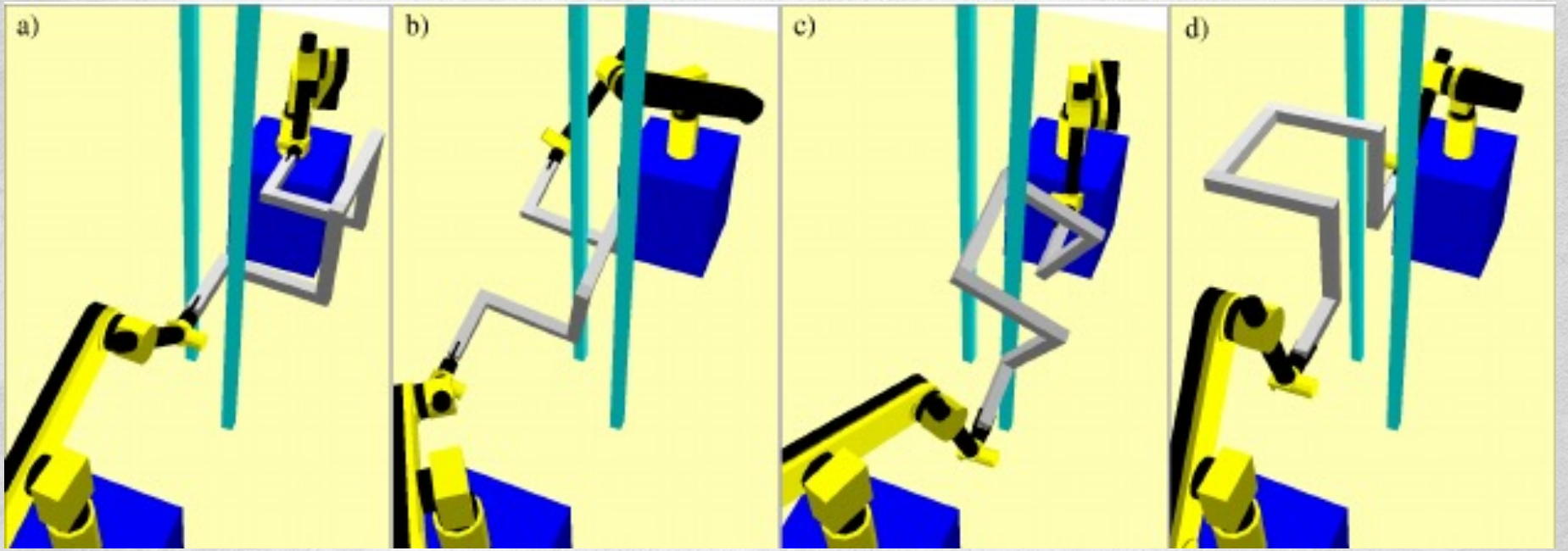
Mission Planning



Path Planning



Motion Planning

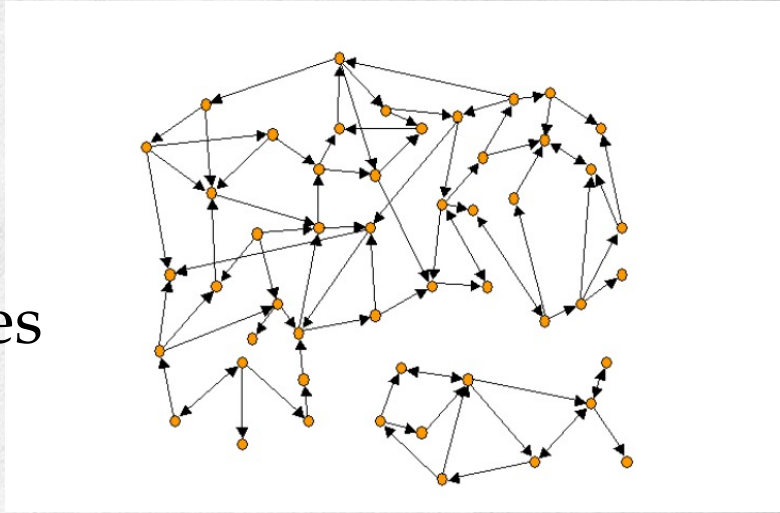


Planning

- Mission planning
  - Which tasks to perform and what order to perform them to achieve a specified goal
  - Tasks may have preconditions and effects
- Path-planning
  - Focus is on robot's (*x, y, theta*) pose
    - What is it now?
    - What do we want it to be?
    - What sequence of poses will get me from where it is now to where it should be?
- Motion-planning
  - Focus also on robot's pose (called a configuration), but in general this can be a high-dimensional pose, consisting of many joint angles (think a robotic arm)

Problem Representation

Search space



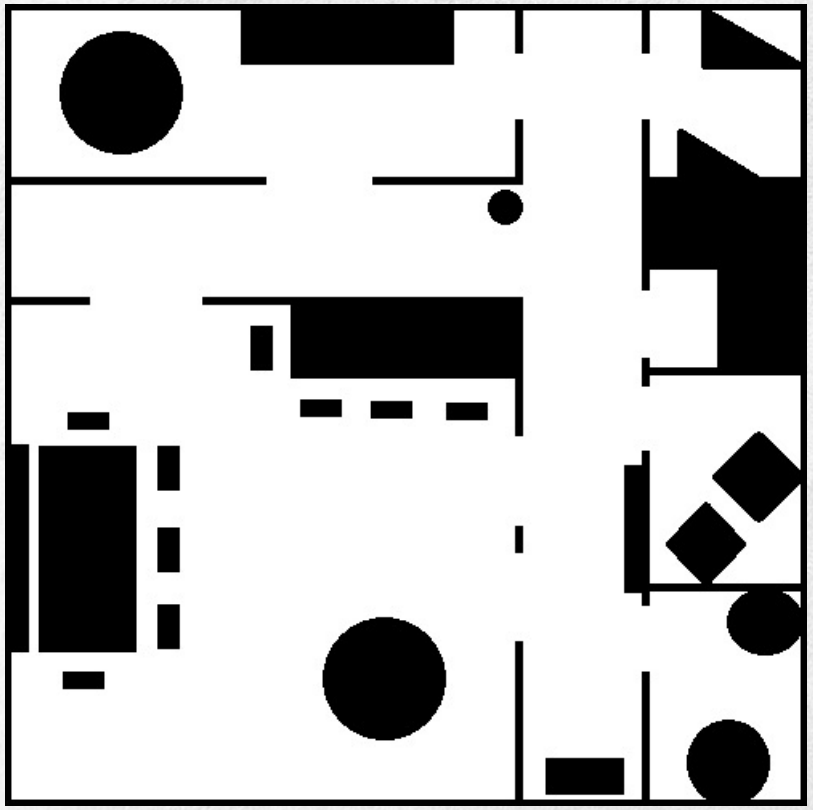
- Vertices – states
- Edges – connect neighboring states

- Planning algorithm needs to find a *solution* or a path from the initial state to the goal state
- The *cost* of a solution is the sum of the costs of each edge in the path
  - For our purposes, this will be the length of the path
- An *optimal solution* is a solution with the lowest cost of any solution

Search Space

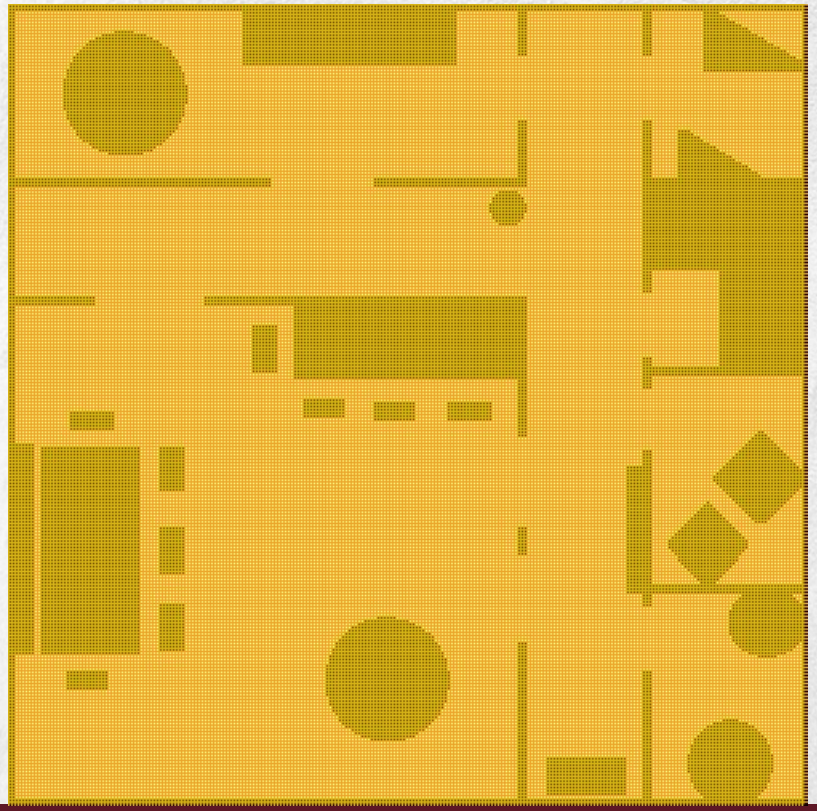
- The **Search Space** for a planning problem is the set of all poses/configurations/states that we will planning in
- In robotics problems the search spaces are typically *continuous*. This means that there is an uncountably infinite number of possible states
- The planning algorithms we will discuss today require a *discrete* state space to work with
- How can we come up with a discrete version of a continuous state space?

Original Map



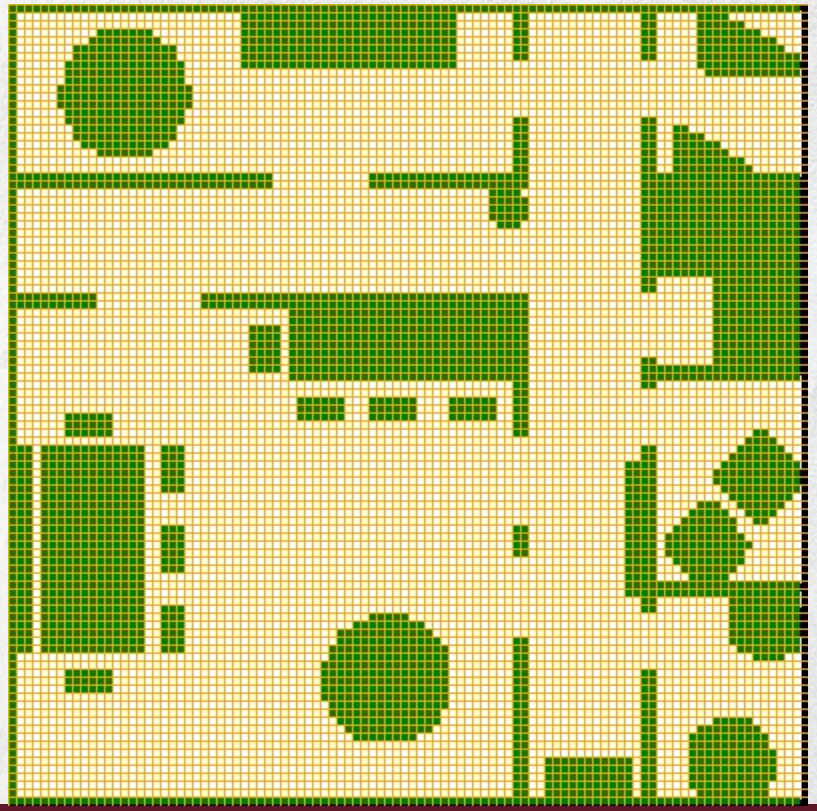
Grid-based discretization

2 pix per cell =
61752 xy-cells



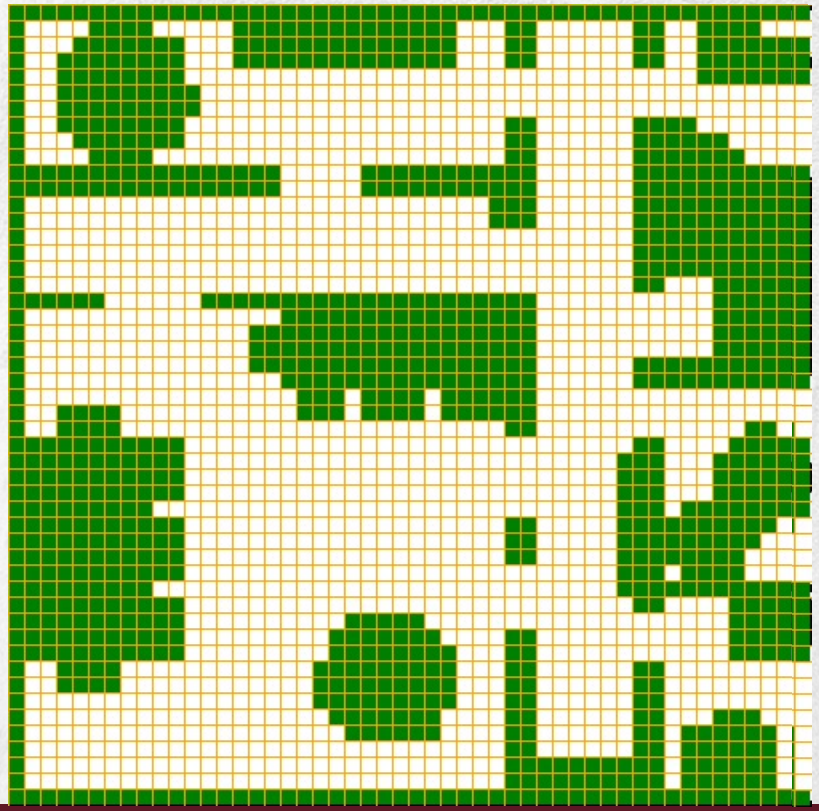
Grid-based Discretization

5 pix per cell =
9702 xy-cells



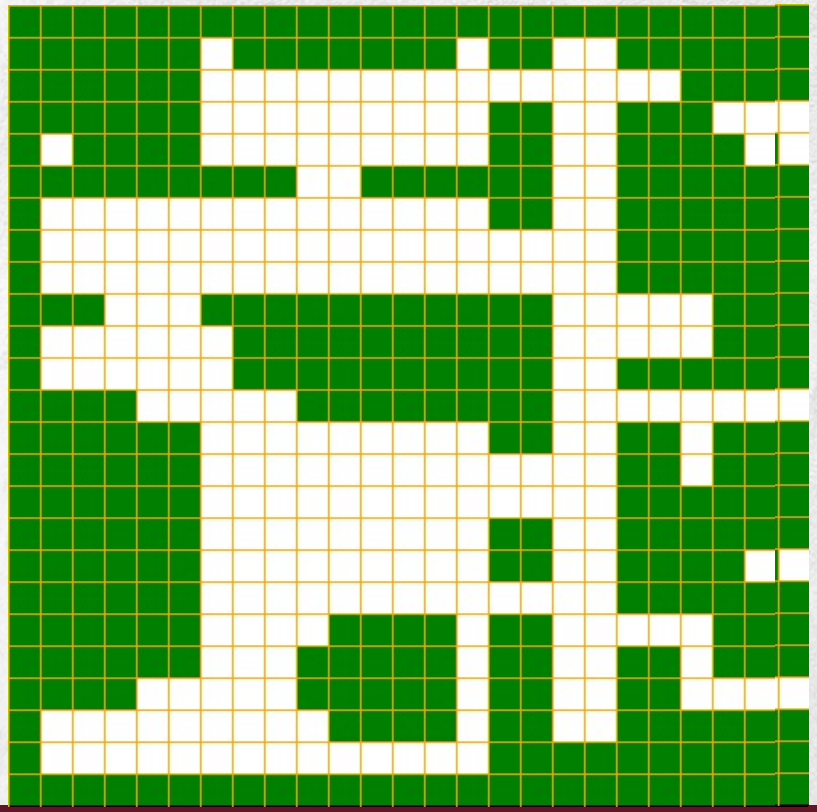
Grid-based Discretization

10 pix per cell =
2352 xy-cells



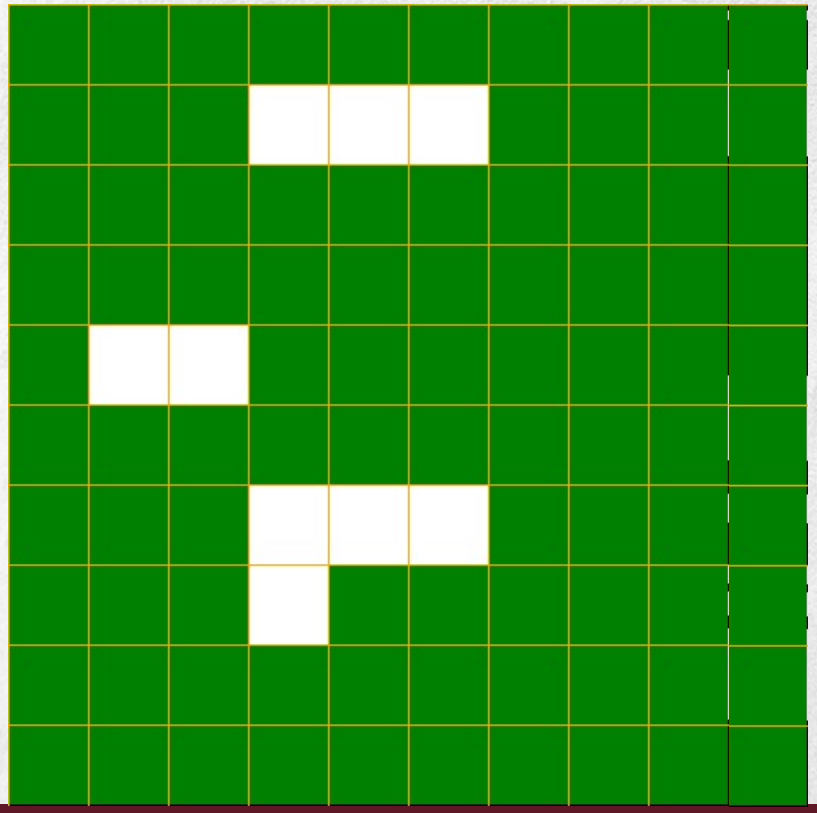
Grid-based Discretization

20 pix per cell =
552 xy-cells

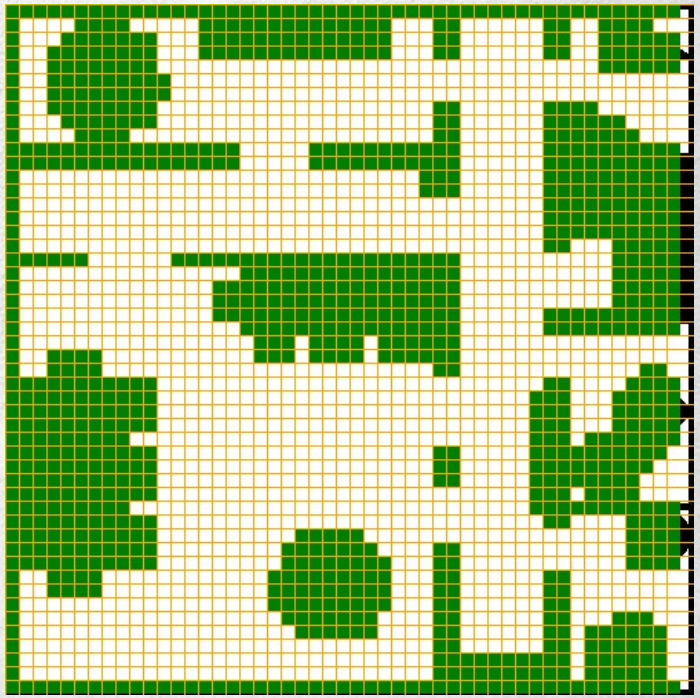


Grid-based Discretization

50 pix per cell =
72 xy-cells

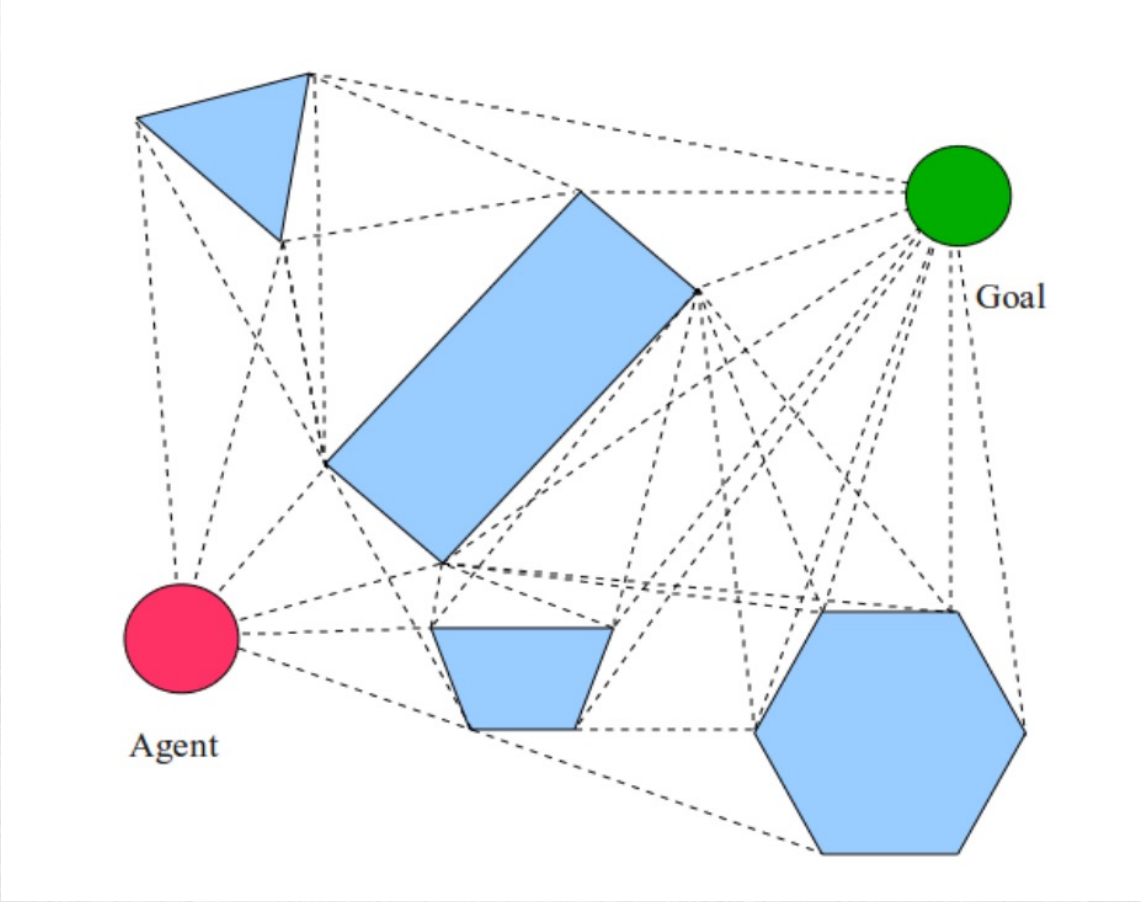


Grid-based discretization

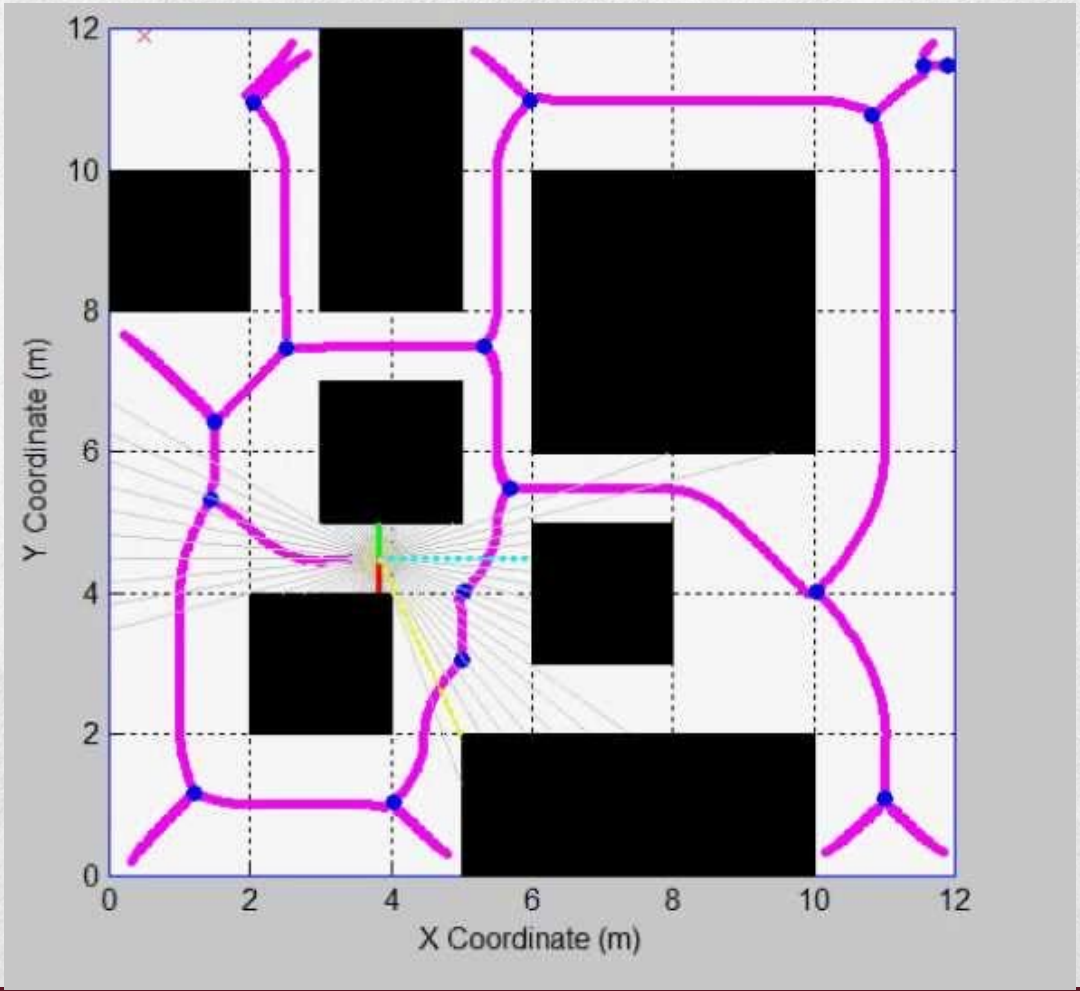


- Advantages
  - Easy to compute
  - Conceptually straightforward
- Disadvantages
  - Doesn't scale well
  - Ignores features of environment
  - Might make possible paths impossible if resolution is too large

Visibility Graph

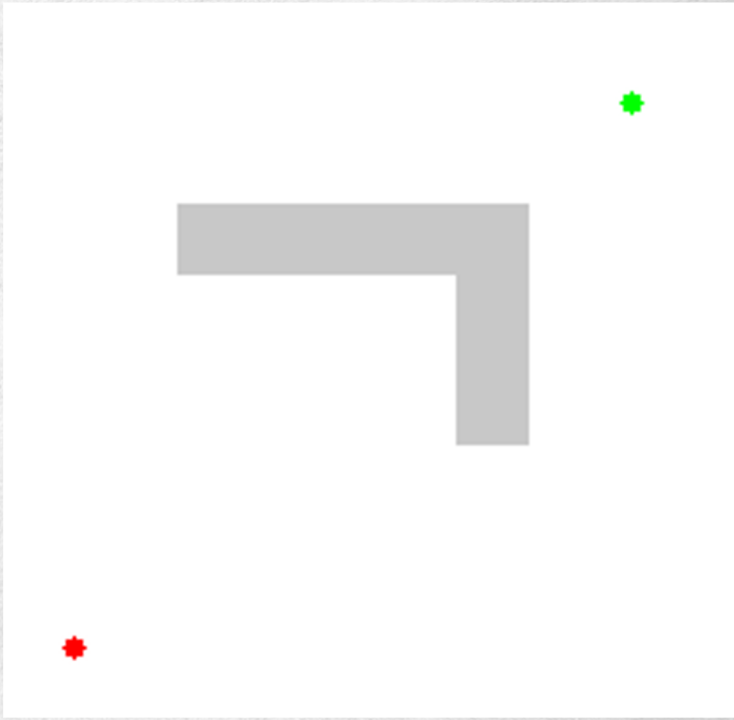


Voronoi Diagram



Search-Based Planning

1. Make a set of points
   - Each point = one possible robot state
2. Make connections between neighboring points
   - Forming a graph
3. Remove any connections that go through obstacles
4. Search for a path through the graph



Search Algorithms

Search Space and Planning Algorithms

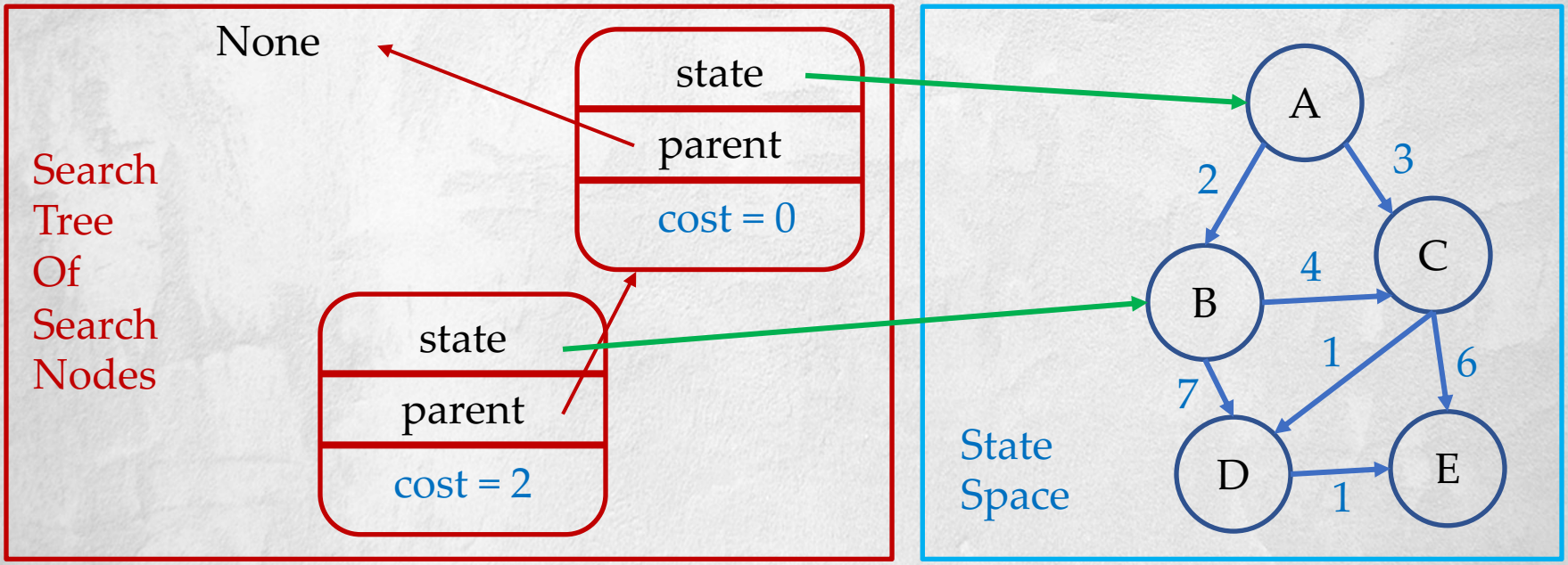
- So we have our discretized search space
- How do our algorithms interact with it?
- A search space provides:
  - An initial state
  - For any state, a set of neighboring states, that can be easily accessed from that state
    - i.e. neighbor grid cells
    - This will be impacted by how you decide to do it
      - All 8 neighbors on grid? Only up-down/left-right?
  - The cost for getting from any state to a neighbor
  - Test to see if a state is a goal (goal-test)

Search Algorithms

- The algorithms we discuss today will all work in a similar way
- They will each build up a tree of *search nodes*, where each search node corresponds to a path in the search space
- The algorithms differ in how they decide which search node, or partial path, to further explore at each time step
- Search will stop once a partial path has reached the goal state

Search Nodes

- These are data structures that have:
  - A search state they are pointing to
  - A pointer to their parent search node (tracing these from a given search node to the initial search node gives us the partial path represented by the search node)
  - The cost of the path from the initial node to this search node

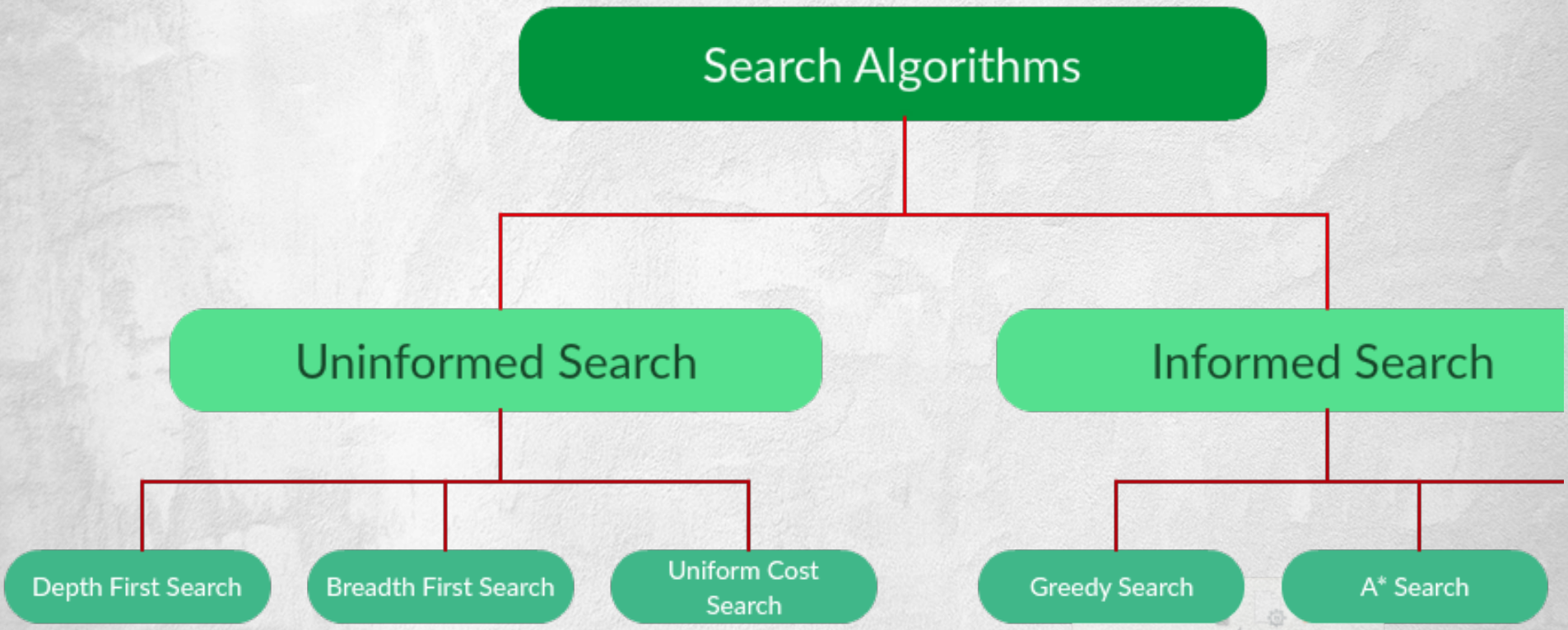


Basic Search Algorithm

- Create START node Node generation
  - State is initial state
  - Cost is 0
  - Parent pointer is None
- Insert START into FRINGE (queue which stores search nodes)
- While FRINGE is not empty:
  - Get *node* from FRINGE according to strategy
  - Is *node.state* a goal? If Yes, then we are done (return *node*)
  - If No, then for each neighboring state *s* of *node.state*:
    - Create search node *n* with Node generation
      - State = *s*
      - Cost = *node.cost* + cost of getting from *node.state* to *s*
      - Parent pointer = *node*

Node Expansion

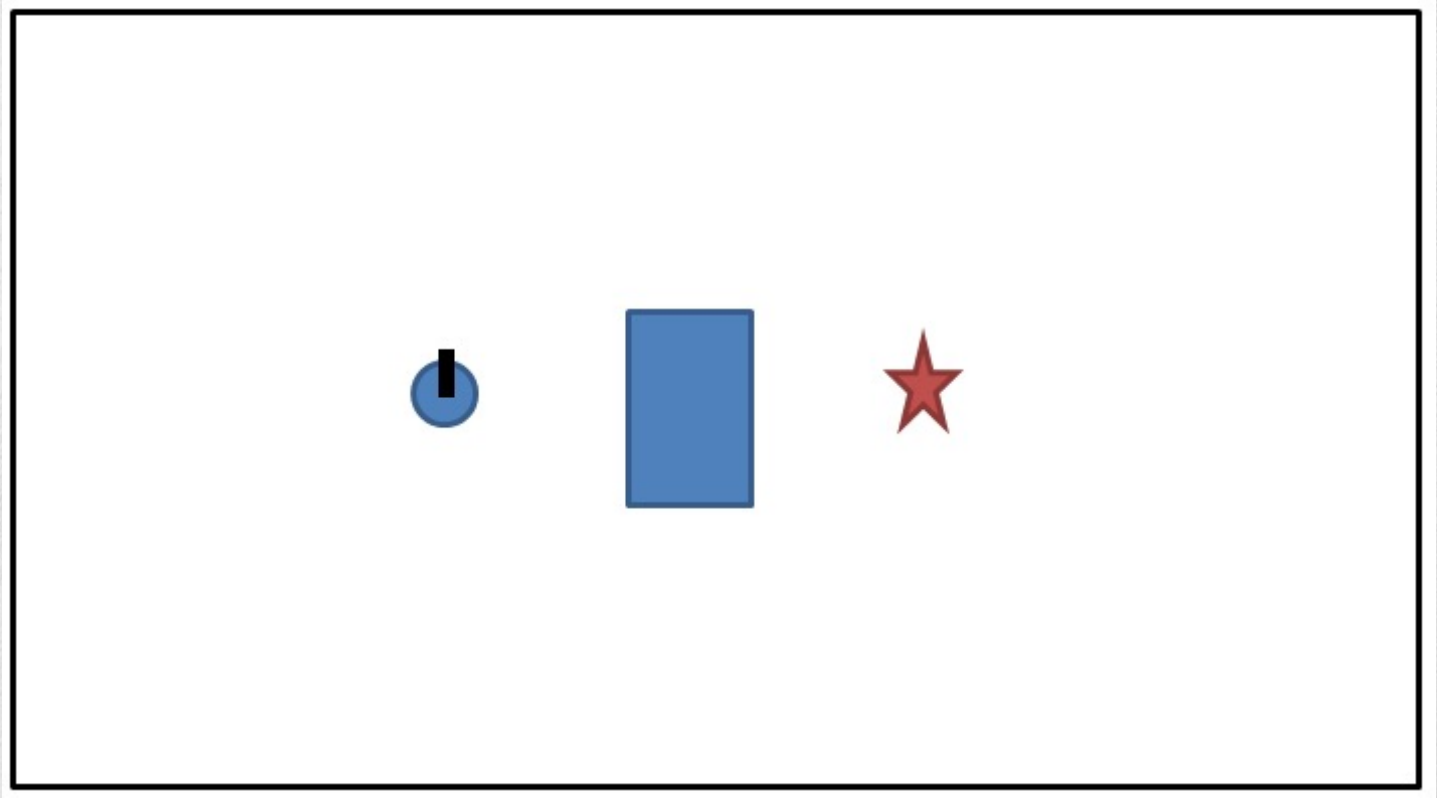
Search Algorithm Categories



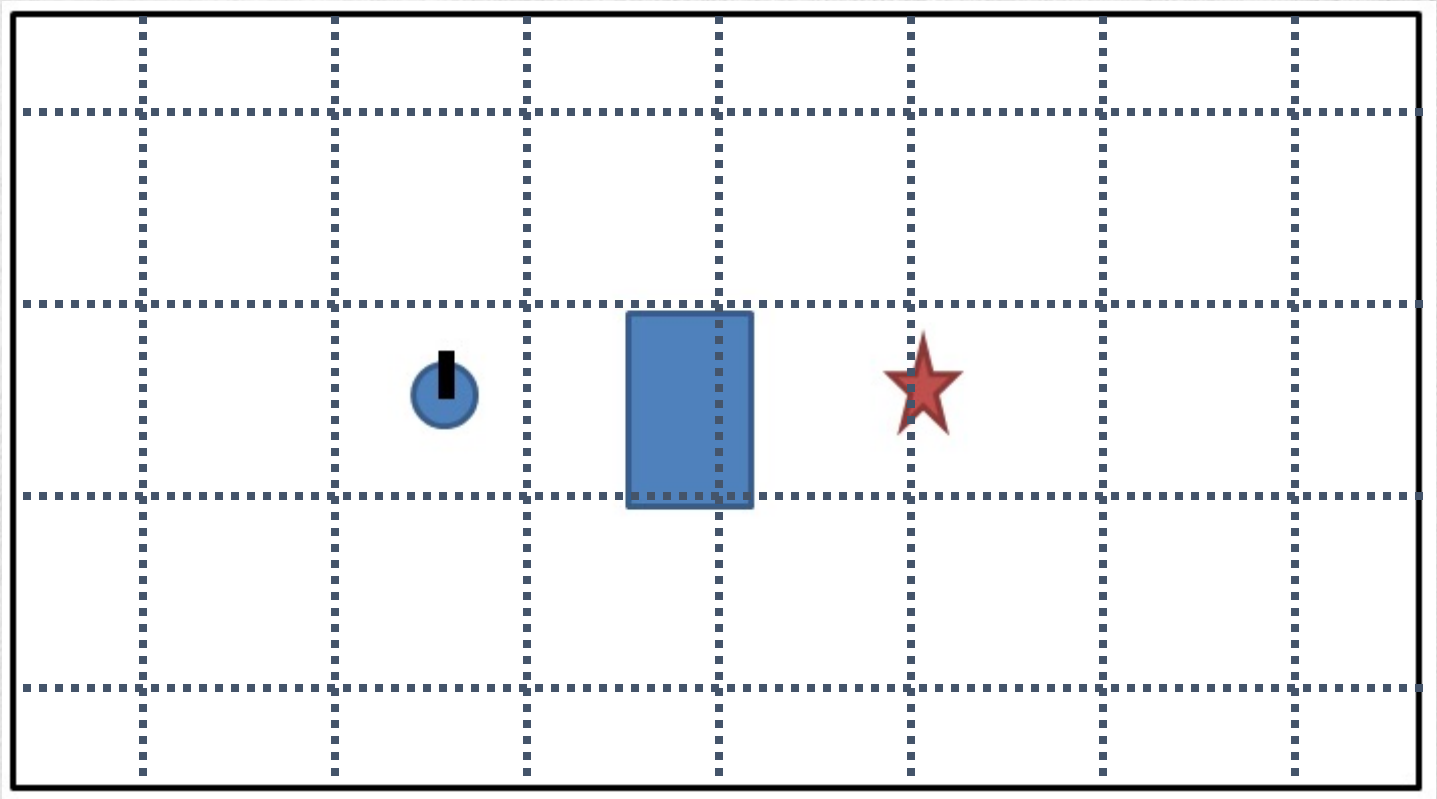
Uninformed Search Algorithms

- Uninformed = Blind = no information about each state (other than goal test)
- Depth-First Search
  - FRINGE is a LIFO queue / stack
  - Node that was added most recently is the next to explore
- Breadth-First Search
  - FRINGE is a FIFO queue
  - Node that has been in the longest is the next to explore
- Uniform-Cost Search
  - FRINGE is a priority queue, sorted by cost
  - Like Breadth-First, but takes cost into account
  - Always expand the node with the lowest cost
- Which of these will always find a solution? Always find an optimal solution?

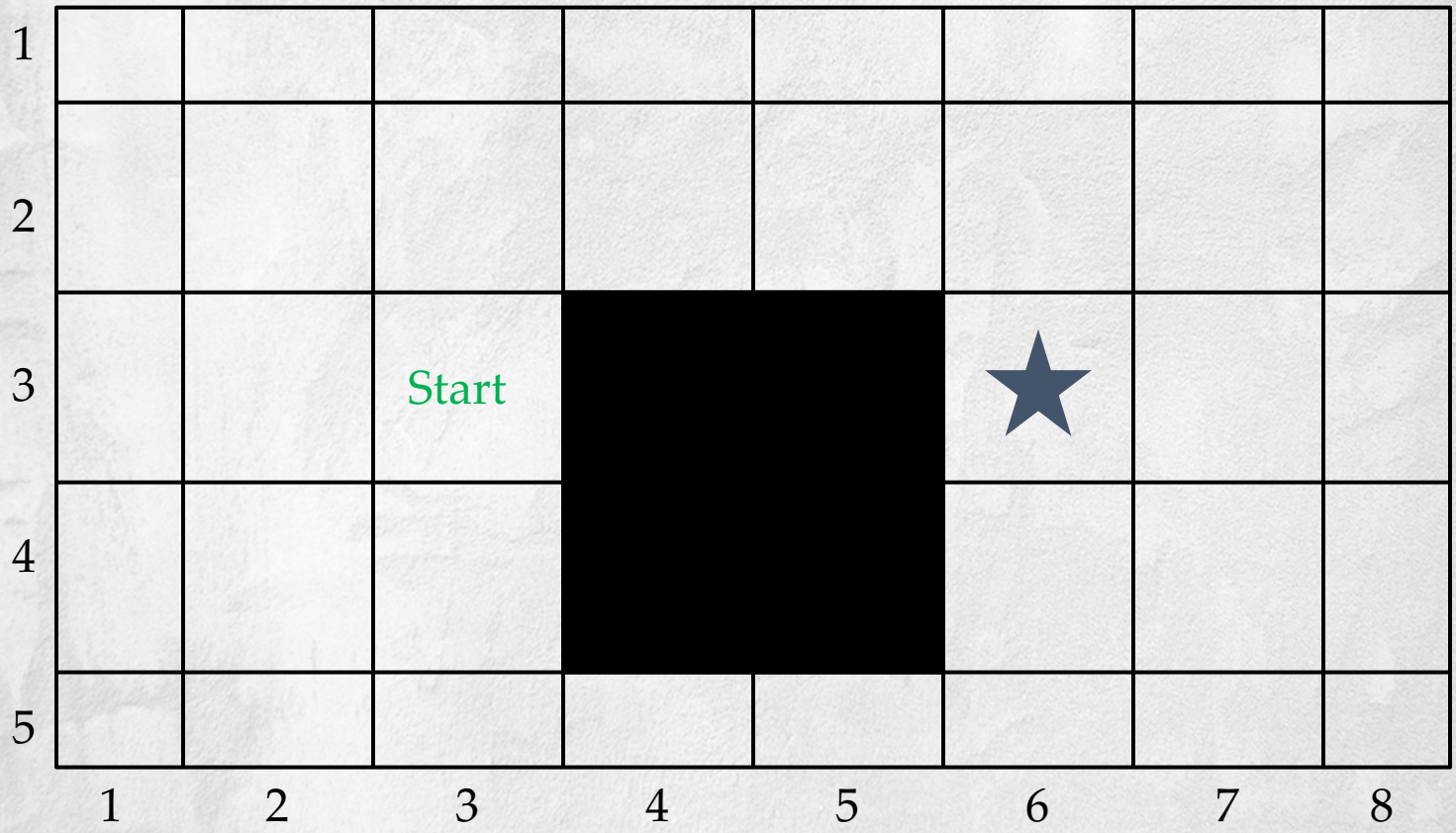
Let's work some examples



Discretize the search space



Discretized search space

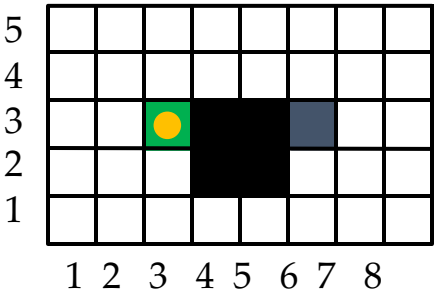


Start = (3,3) Goal = (6,3)
Neighbors = N,E,S,W (in that order)

Depth-First Search

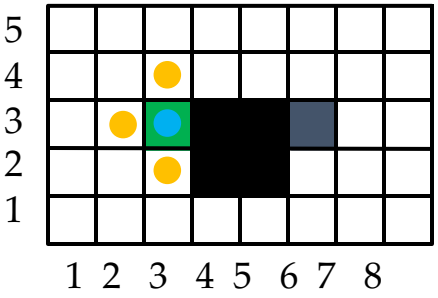
Depth First Search

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```



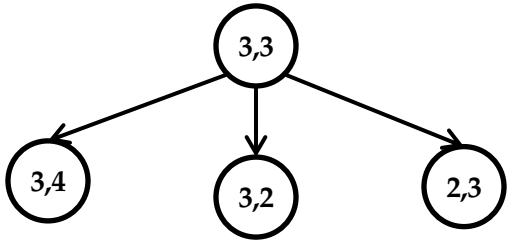
Neighbors = N,E,S,W (in that order)

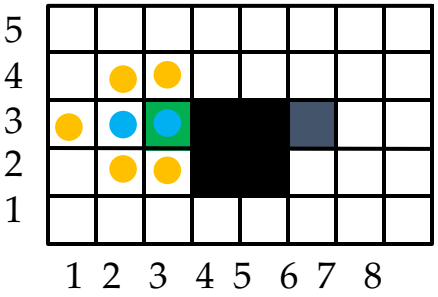
Search Tree: Depth-first Search (no duplicates in fringe)



Neighbors = N,E,S,W (in that order)

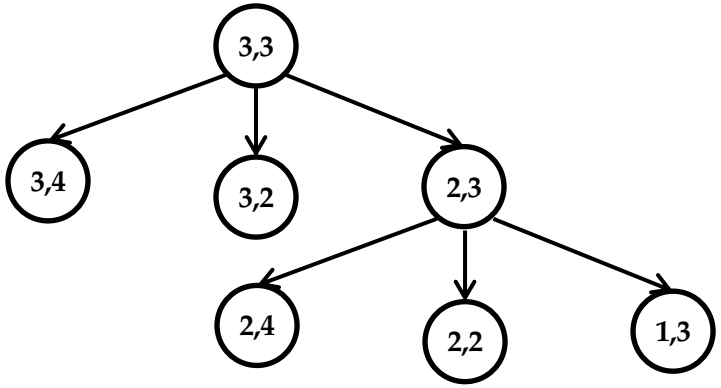
Search Tree: Depth-first Search (no duplicates in fringe)

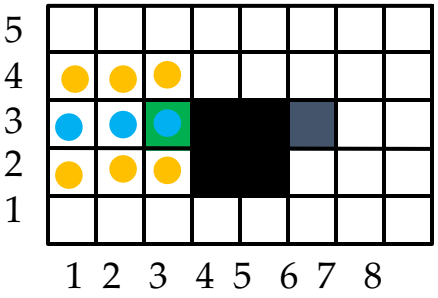




Neighbors = N,E,S,W (in that order)

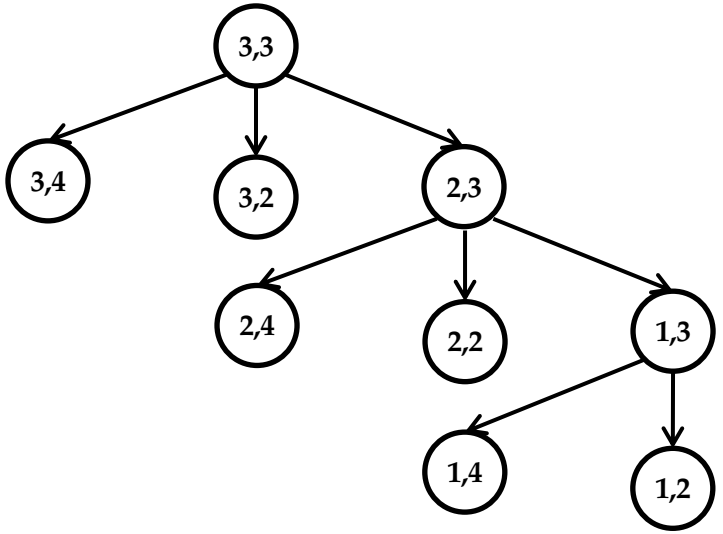
Search Tree: Depth-first Search (no duplicates in fringe)

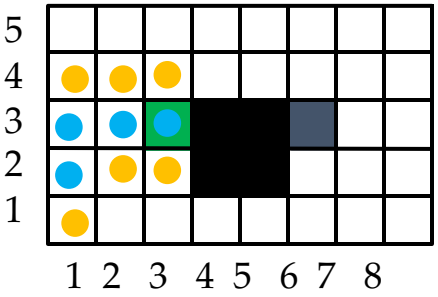




Neighbors = N,E,S,W (in that order)

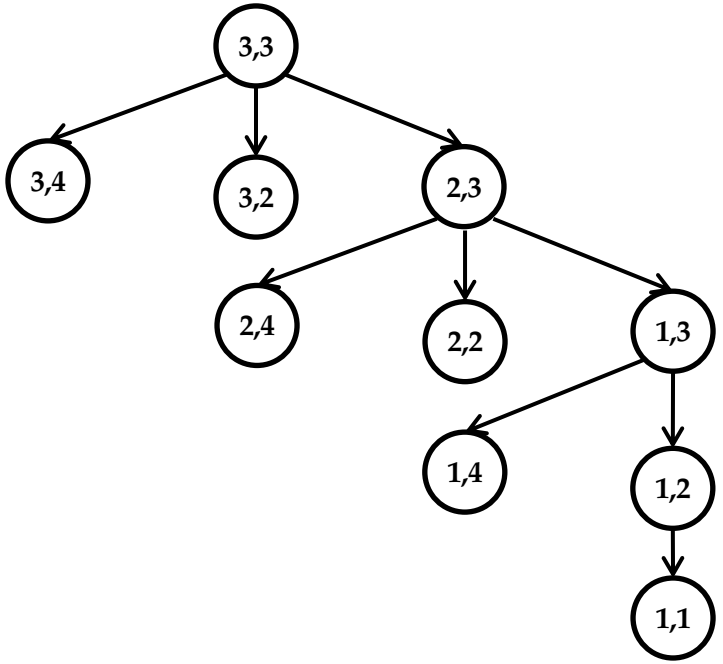
Search Tree: Depth-first Search (no duplicates in fringe)

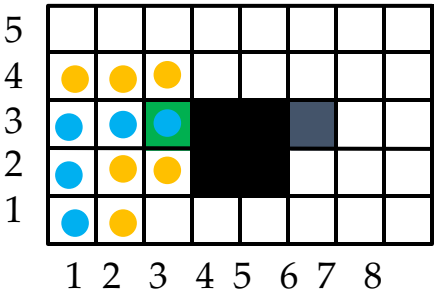




Neighbors = N,E,S,W (in that order)

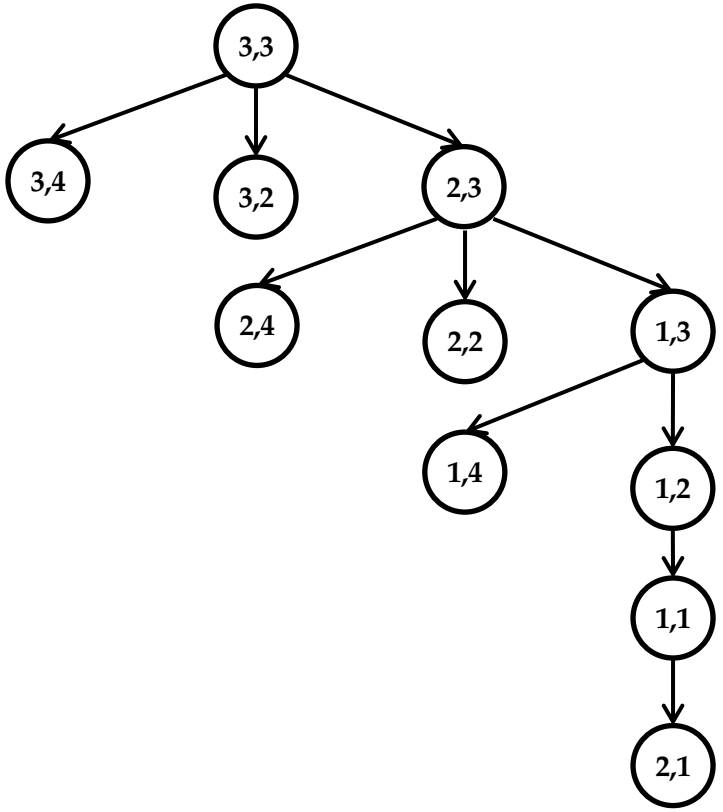
Search Tree: Depth-first Search (no duplicates in fringe)

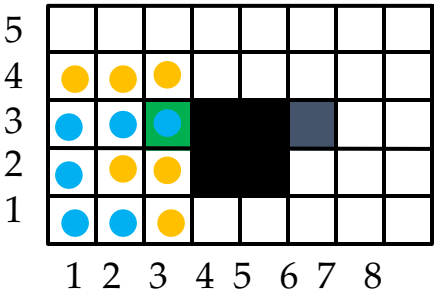




Neighbors = N,E,S,W (in that order)

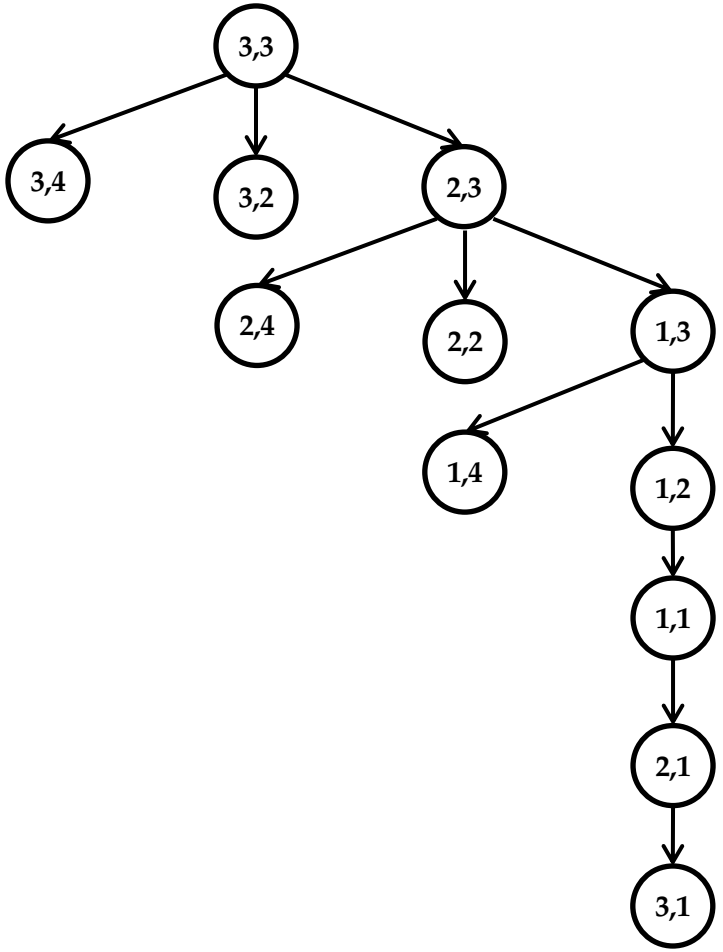
Search Tree: Depth-first Search (no duplicates in fringe)

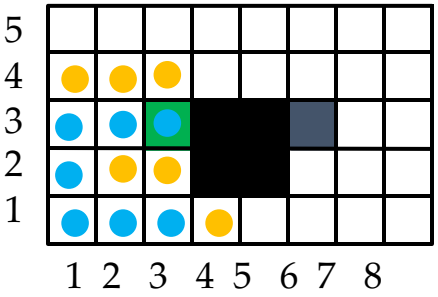




Neighbors = N,E,S,W (in that order)

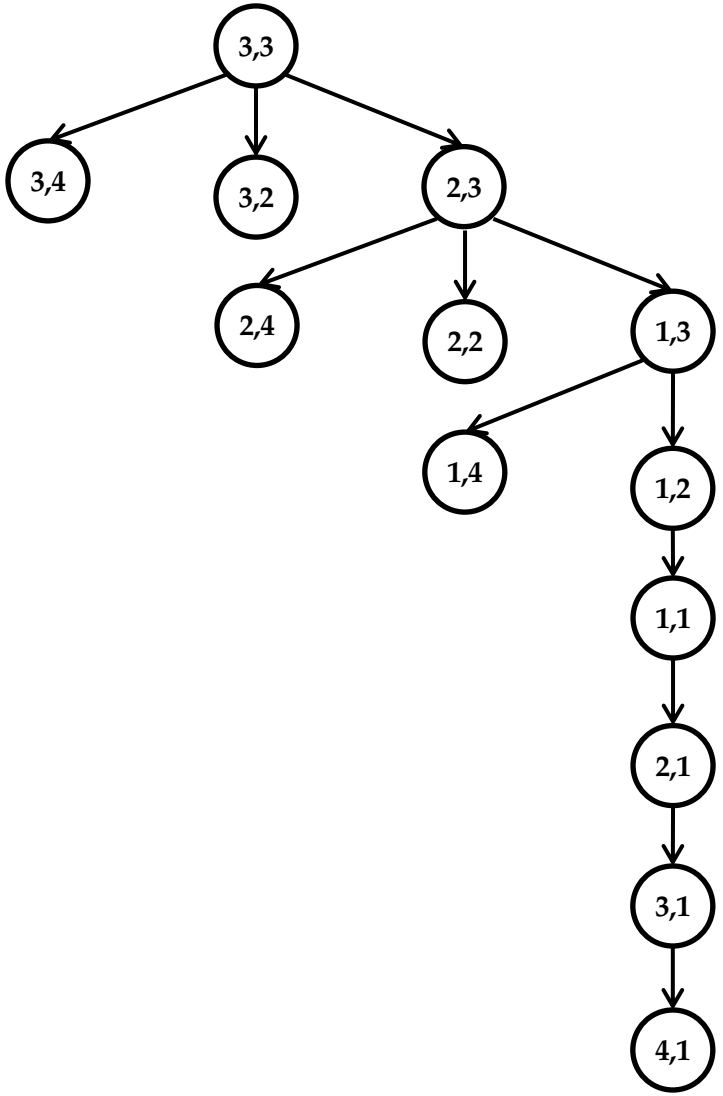
Search Tree: Depth-first Search (no duplicates in fringe)

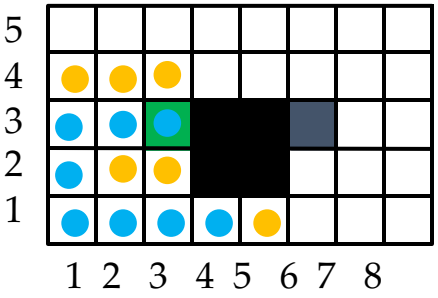




Neighbors = N,E,S,W (in that order)

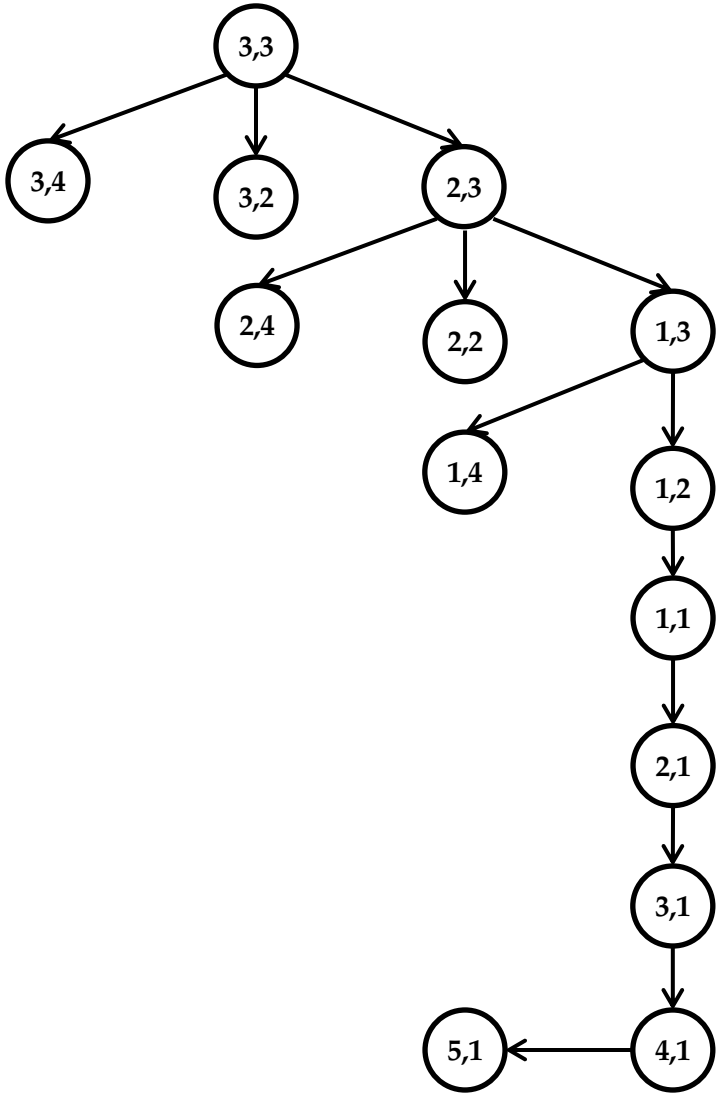
Search Tree: Depth-first Search (no duplicates in fringe)

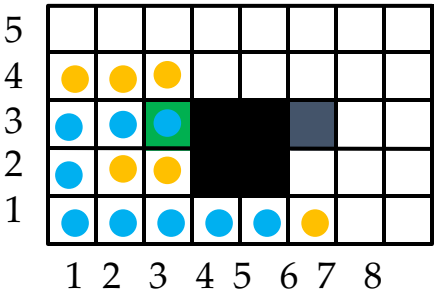




Neighbors = N,E,S,W (in that order)

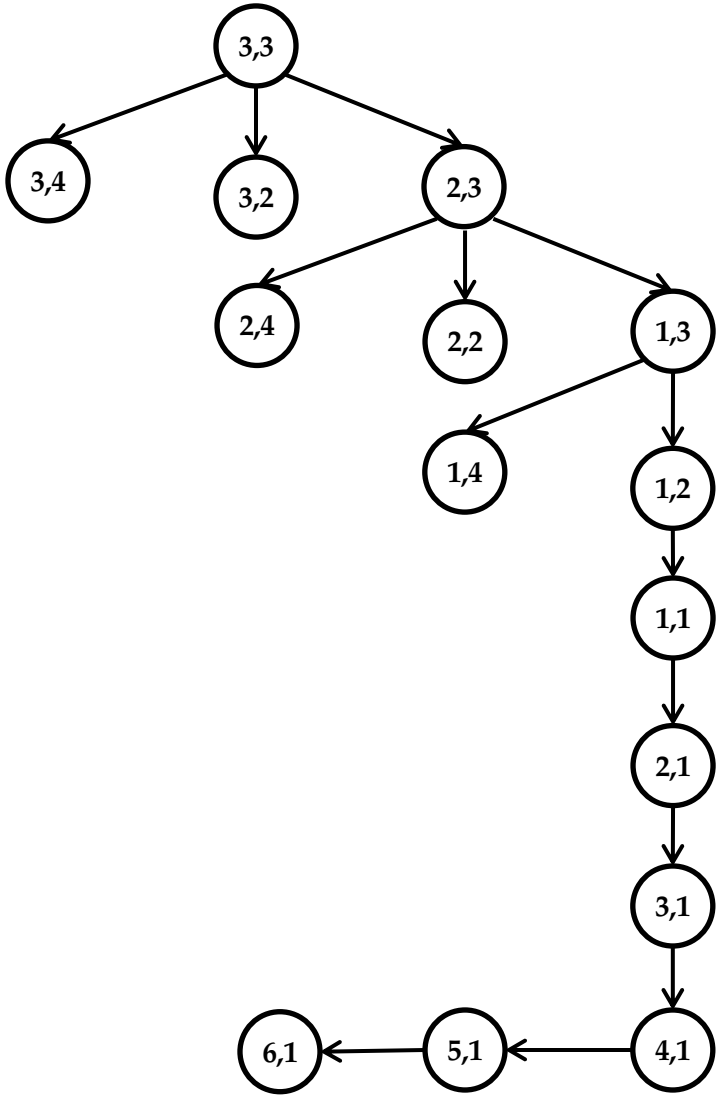
Search Tree: Depth-first Search (no duplicates in fringe)

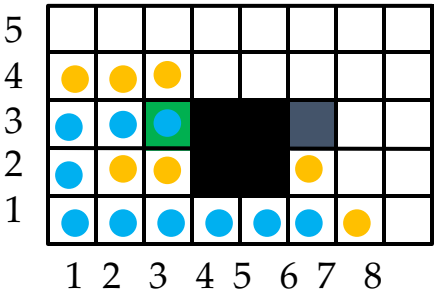




Neighbors = N,E,S,W (in that order)

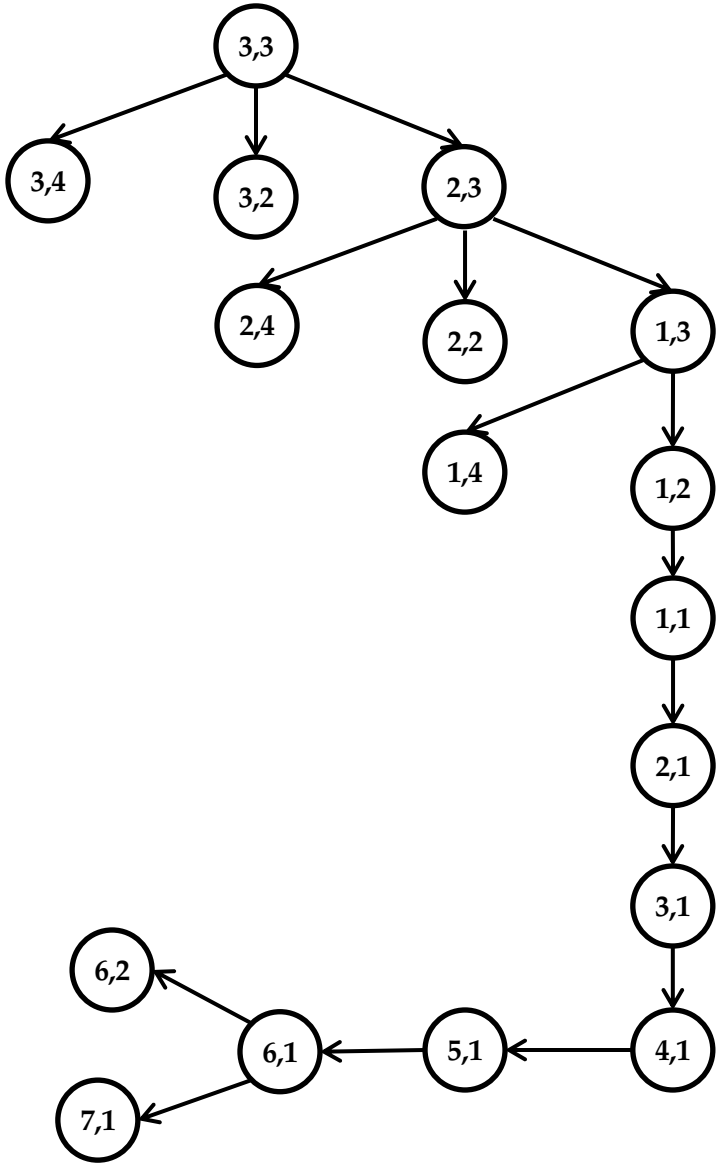
Search Tree: Depth-first Search (no duplicates in fringe)

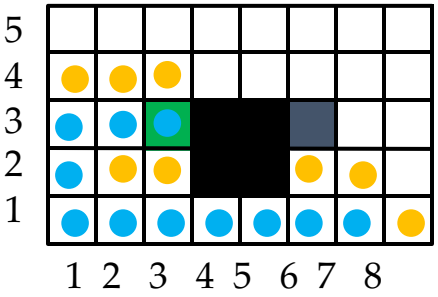




Neighbors = N,E,S,W (in that order)

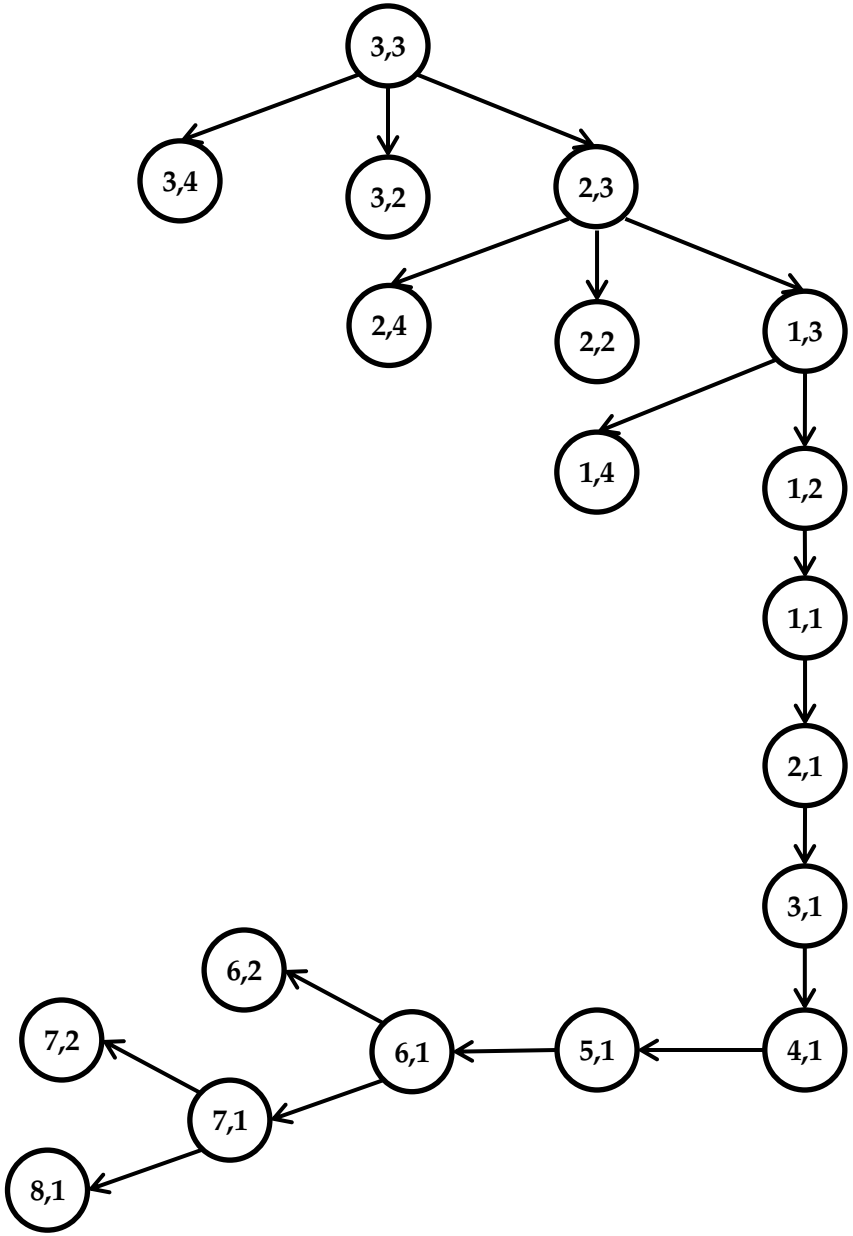
Search Tree: Depth-first Search (no duplicates in fringe)

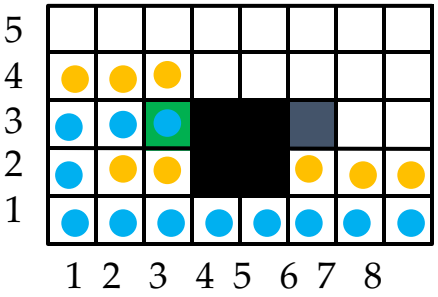




Neighbors = N,E,S,W (in that order)

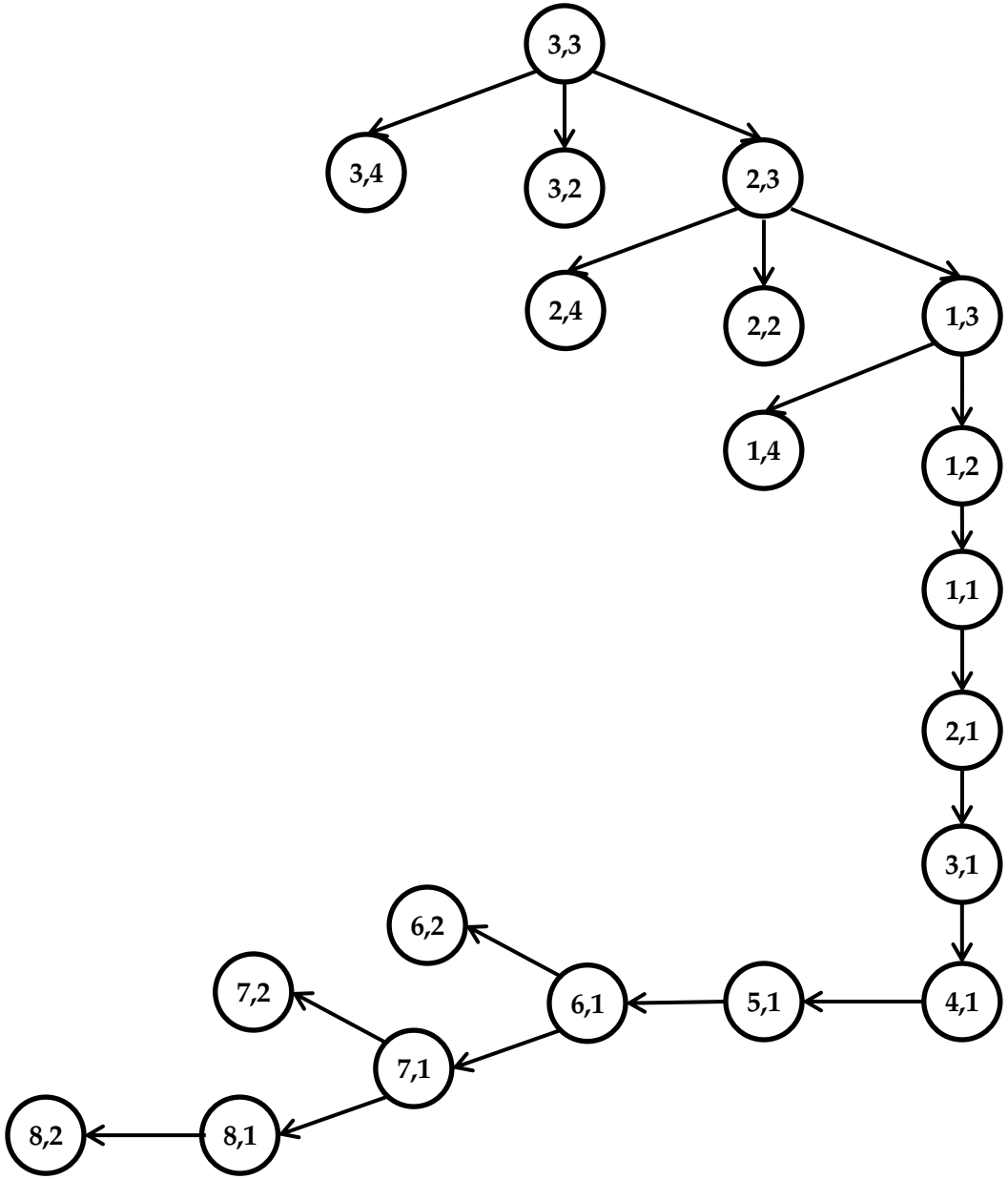
Search Tree: Depth-first Search (no duplicates in fringe)

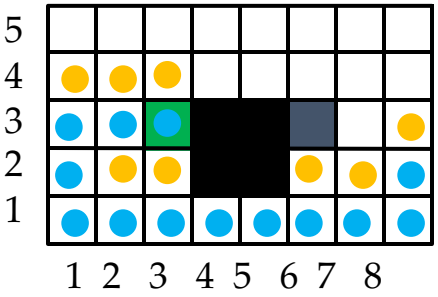




Neighbors = N,E,S,W (in that order)

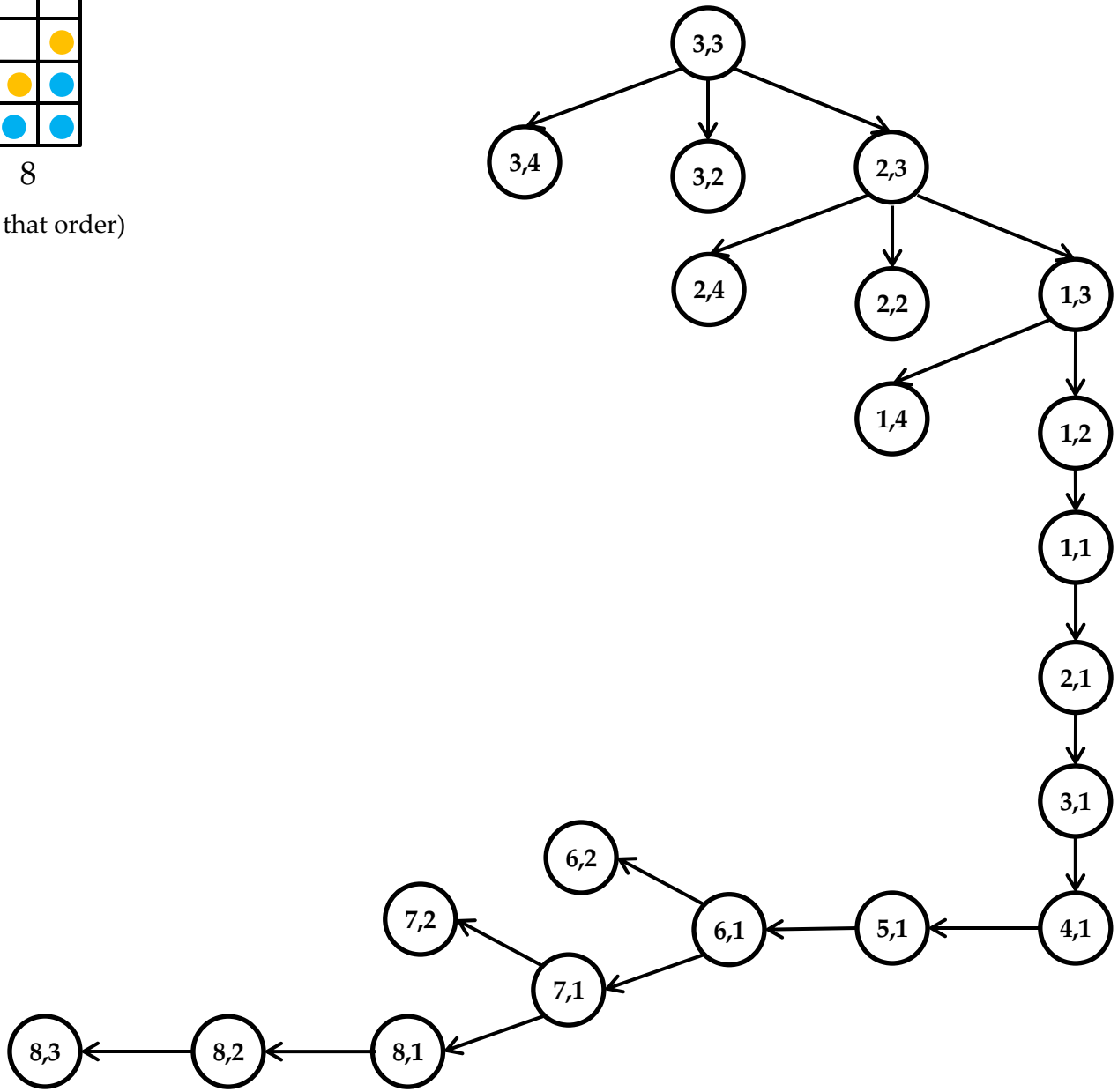
Search Tree: Depth-first Search (no duplicates in fringe)

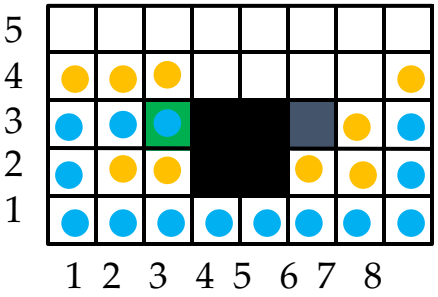




Neighbors = N,E,S,W (in that order)

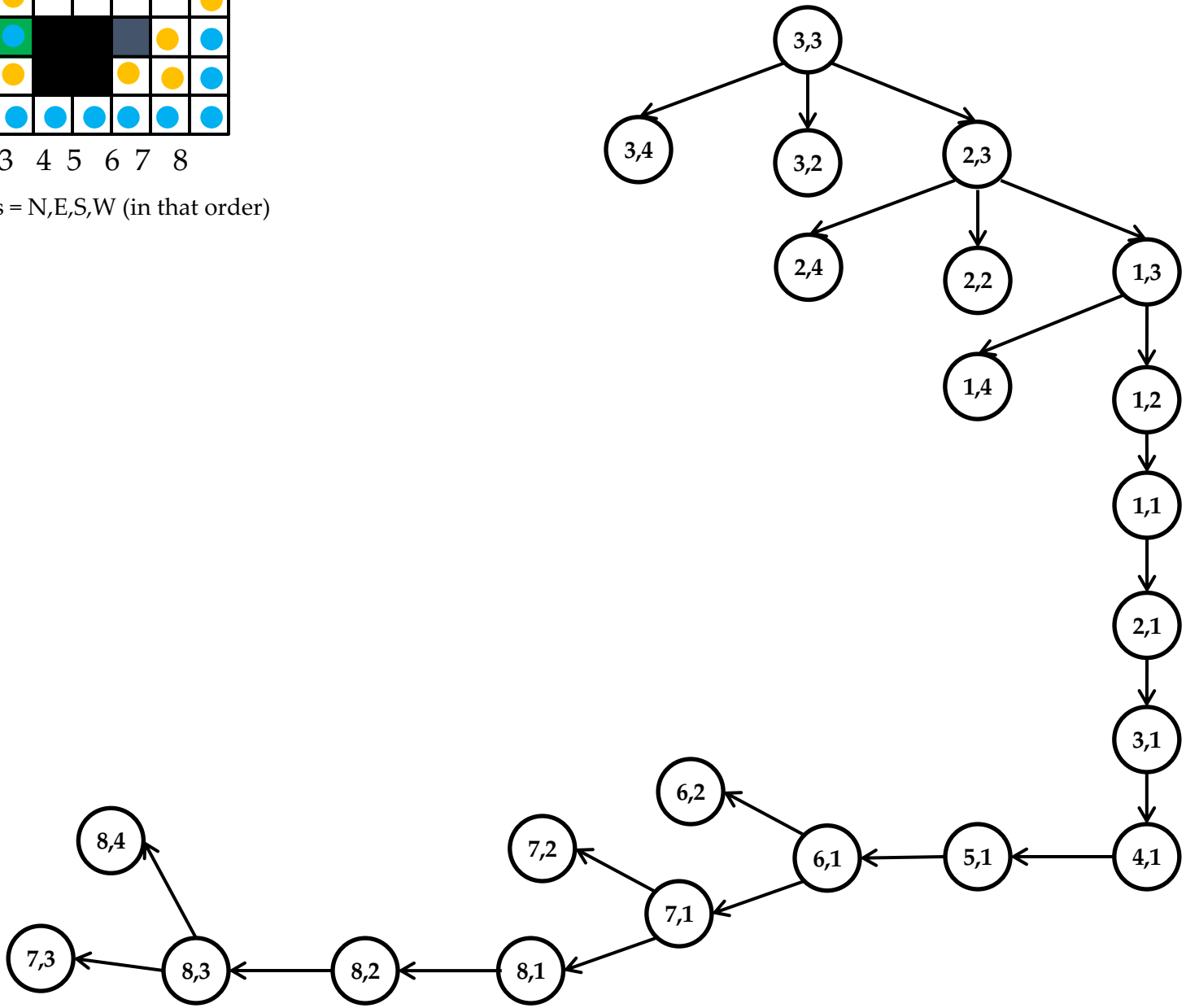
Search Tree: Depth-first Search (no duplicates in fringe)

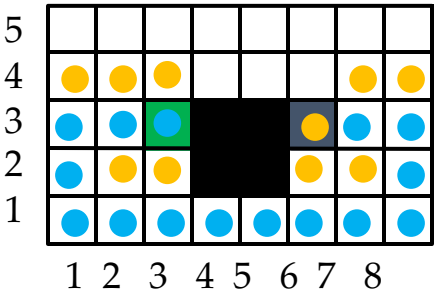




Neighbors = N,E,S,W (in that order)

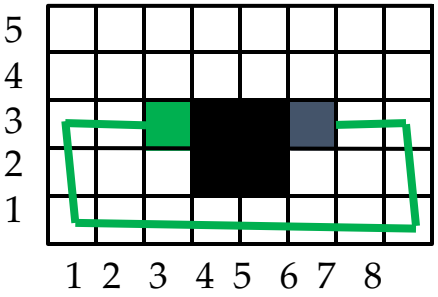
Search Tree: Depth-first Search (no duplicates in fringe)





Neighbors = N,E,S,W (in that order)

Search Tree: Depth-first Search (no duplicates in fringe)



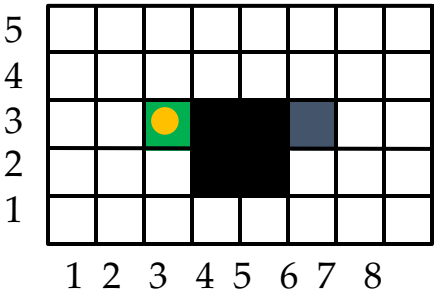
Neighbors = N,E,S,W (in that order)

Search Tree: Depth-first Search (no duplicates in fringe)

Breadth-First Search

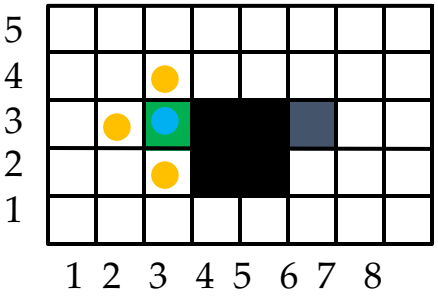
Breadth First Search

```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as explored then
                label w as explored
                Q.enqueue(w)
```



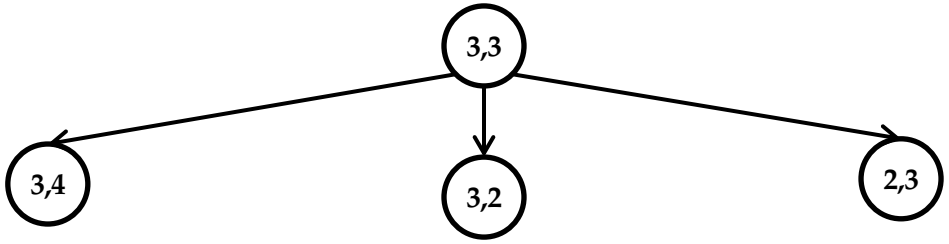
Neighbors = N,E,S,W (in that order)

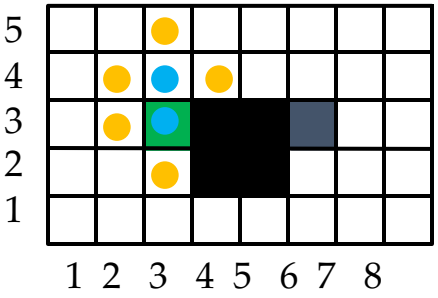
Search Tree: Breadth-first Search (no duplicates in fringe)



Neighbors = N,E,S,W (in that order)

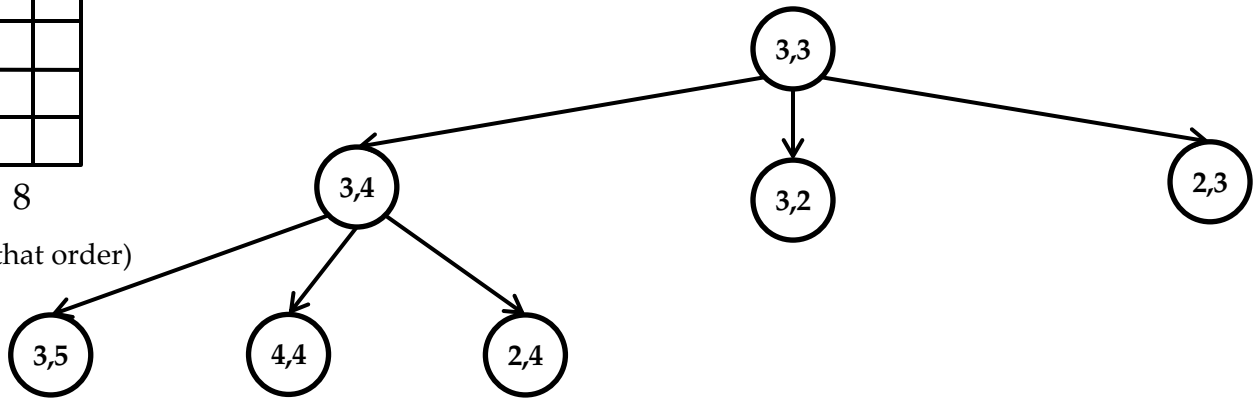
Search Tree: Breadth-first Search (no duplicates in fringe)

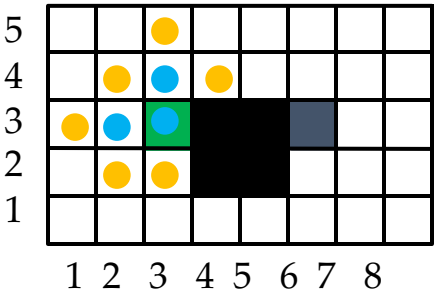




Neighbors = N,E,S,W (in that order)

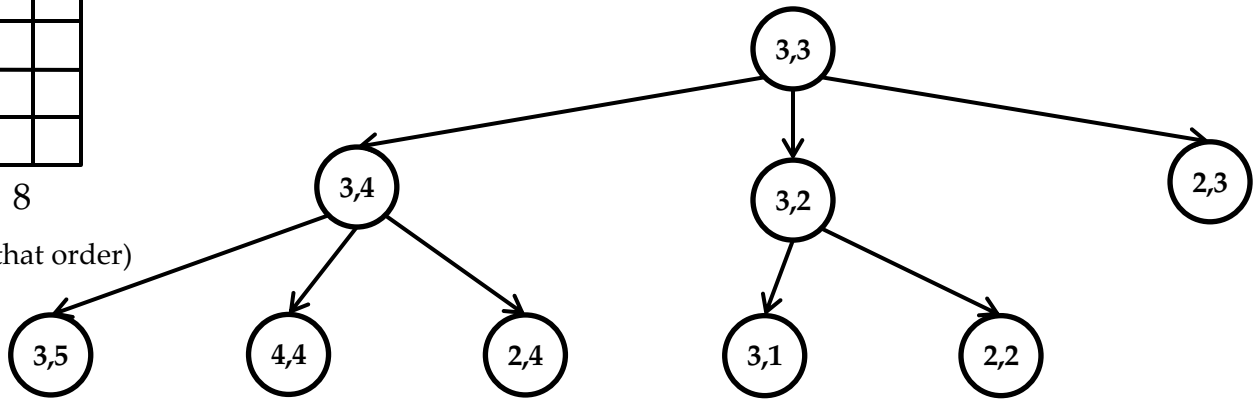
Search Tree: Breadth-first Search (no duplicates in fringe)

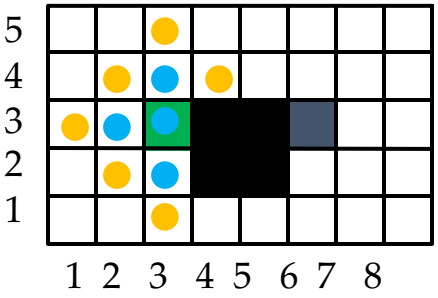




Neighbors = N,E,S,W (in that order)

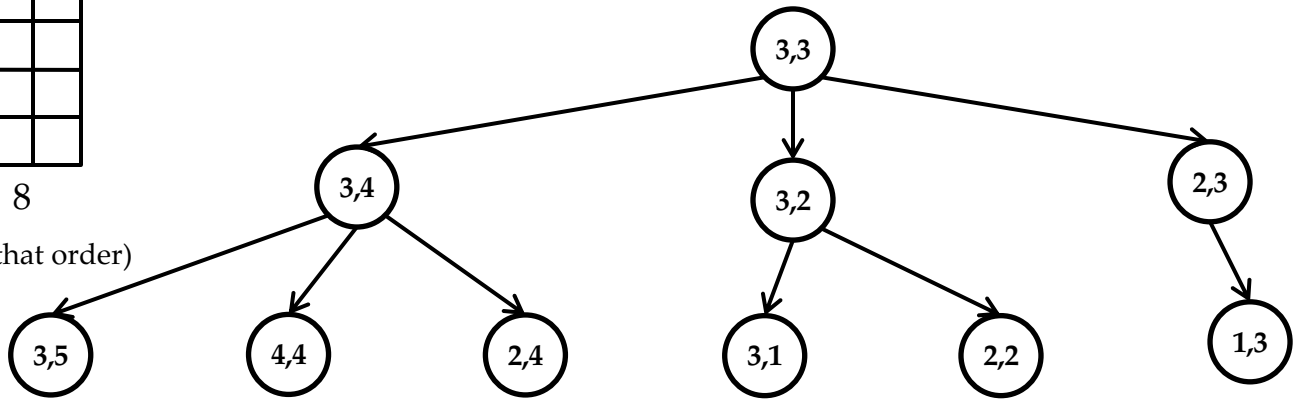
Search Tree: Breadth-first Search (no duplicates in fringe)

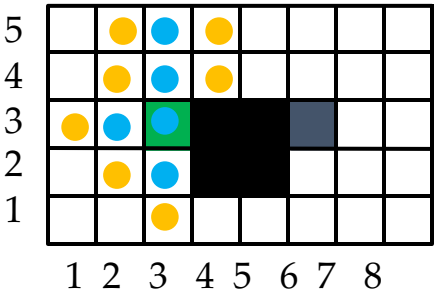




Neighbors = N,E,S,W (in that order)

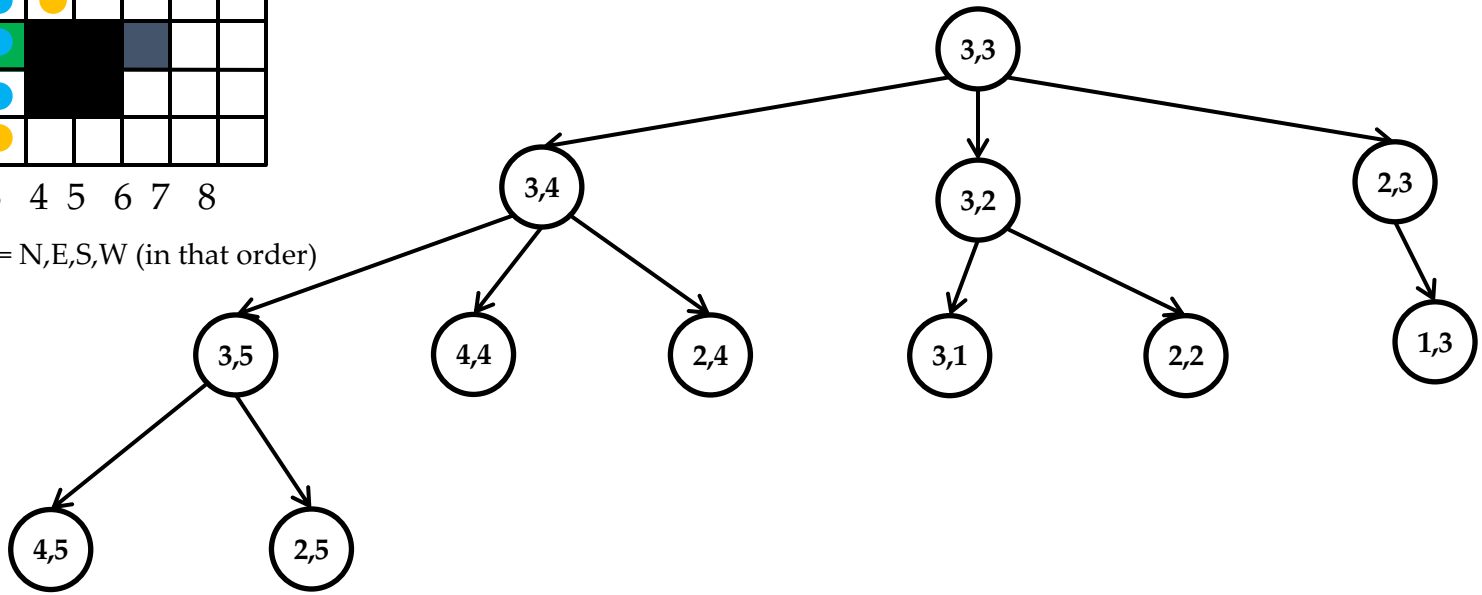
Search Tree: Breadth-first Search (no duplicates in fringe)

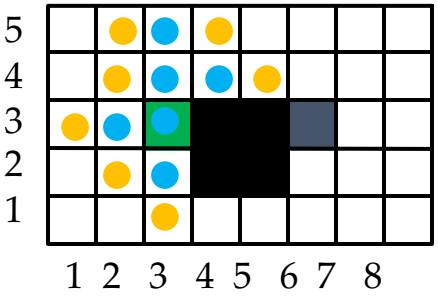




Neighbors = N,E,S,W (in that order)

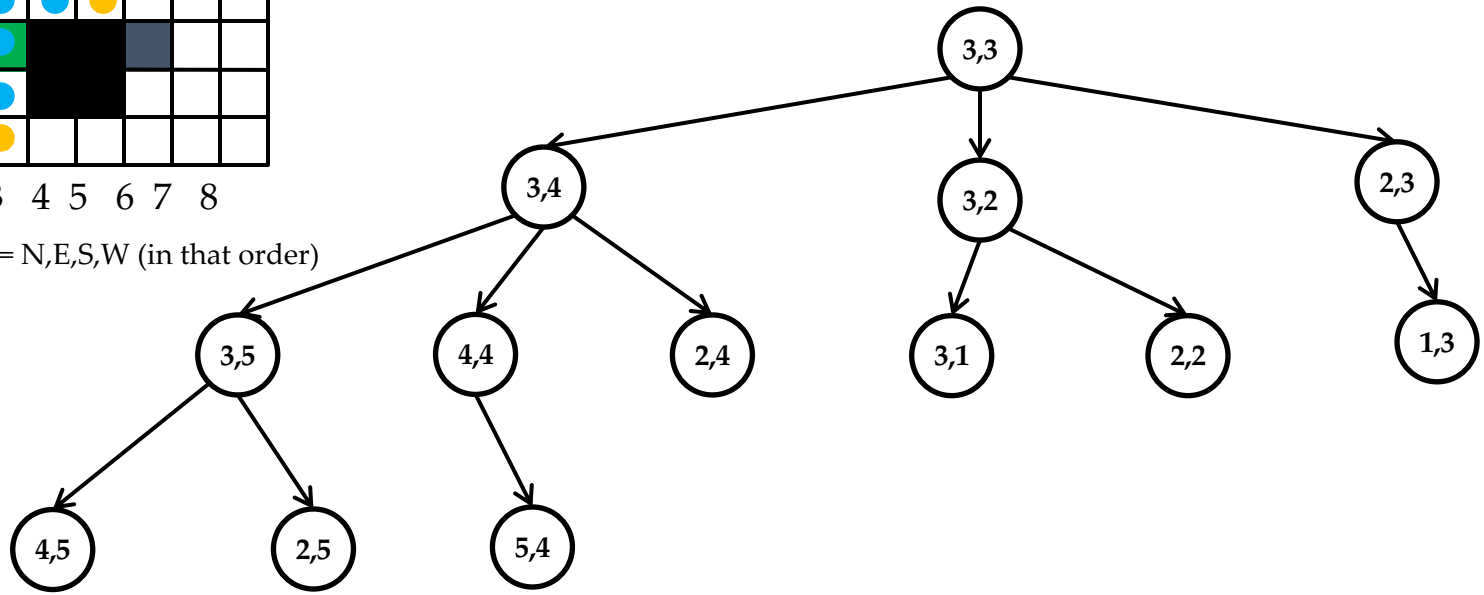
Search Tree: Breadth-first Search (no duplicates in fringe)

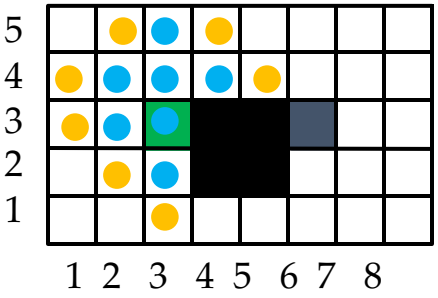




Neighbors = N,E,S,W (in that order)

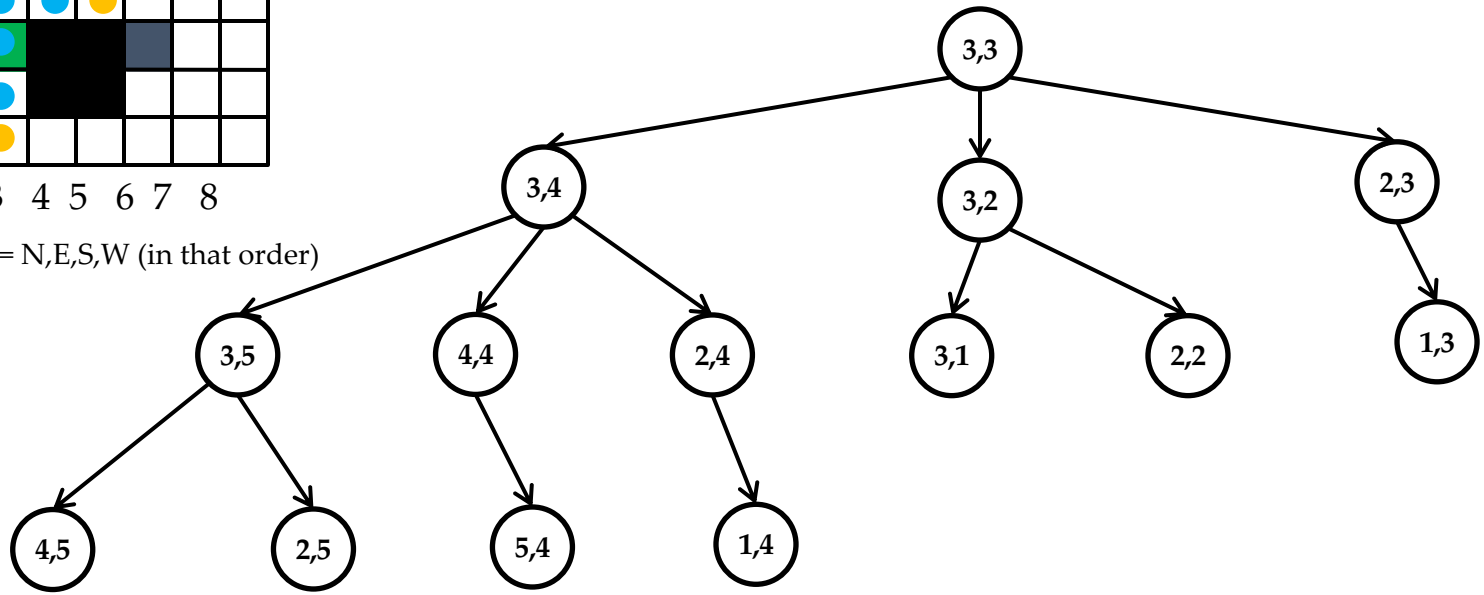
Search Tree: Breadth-first Search (no duplicates in fringe)

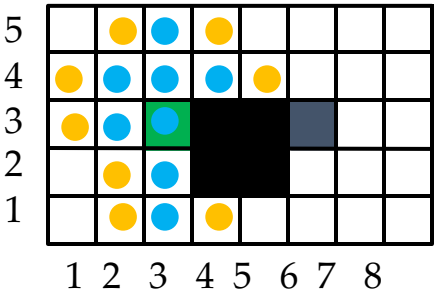




Neighbors = N,E,S,W (in that order)

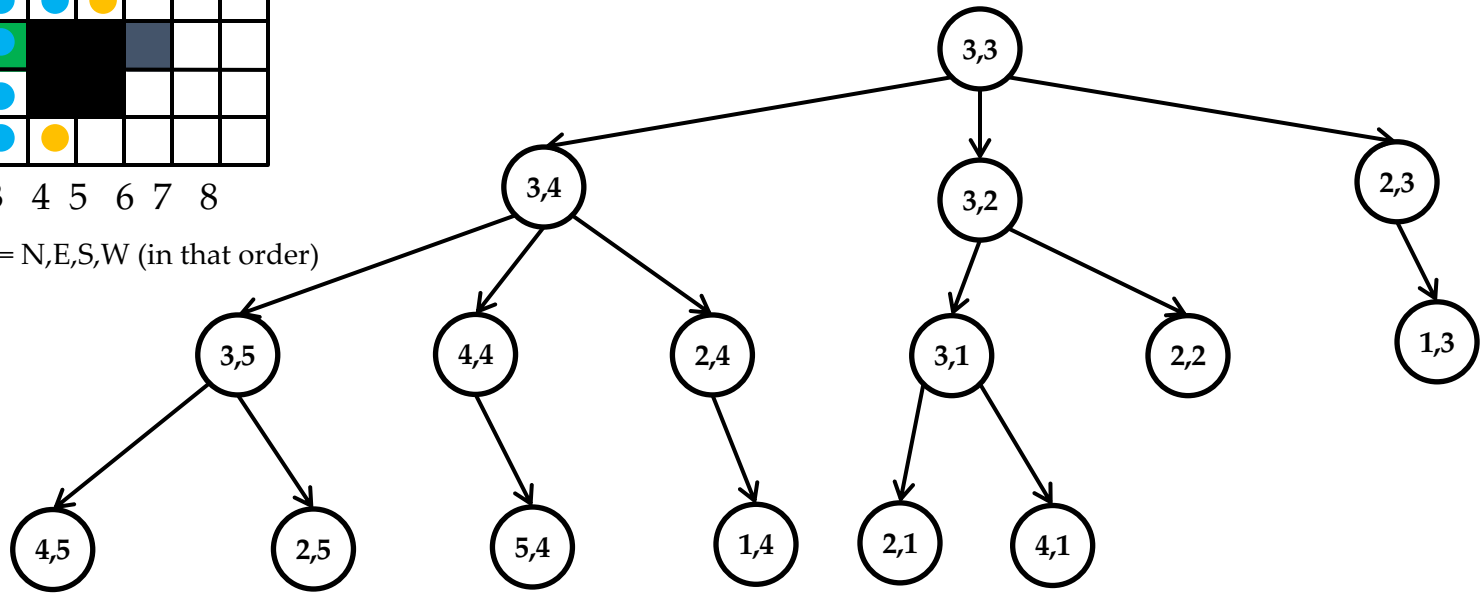
Search Tree: Breadth-first Search (no duplicates in fringe)

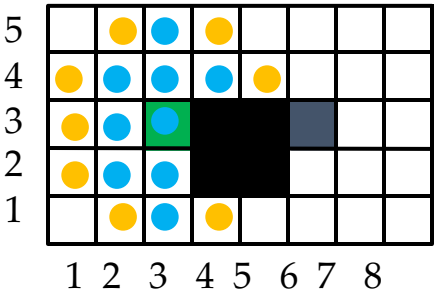




Neighbors = N,E,S,W (in that order)

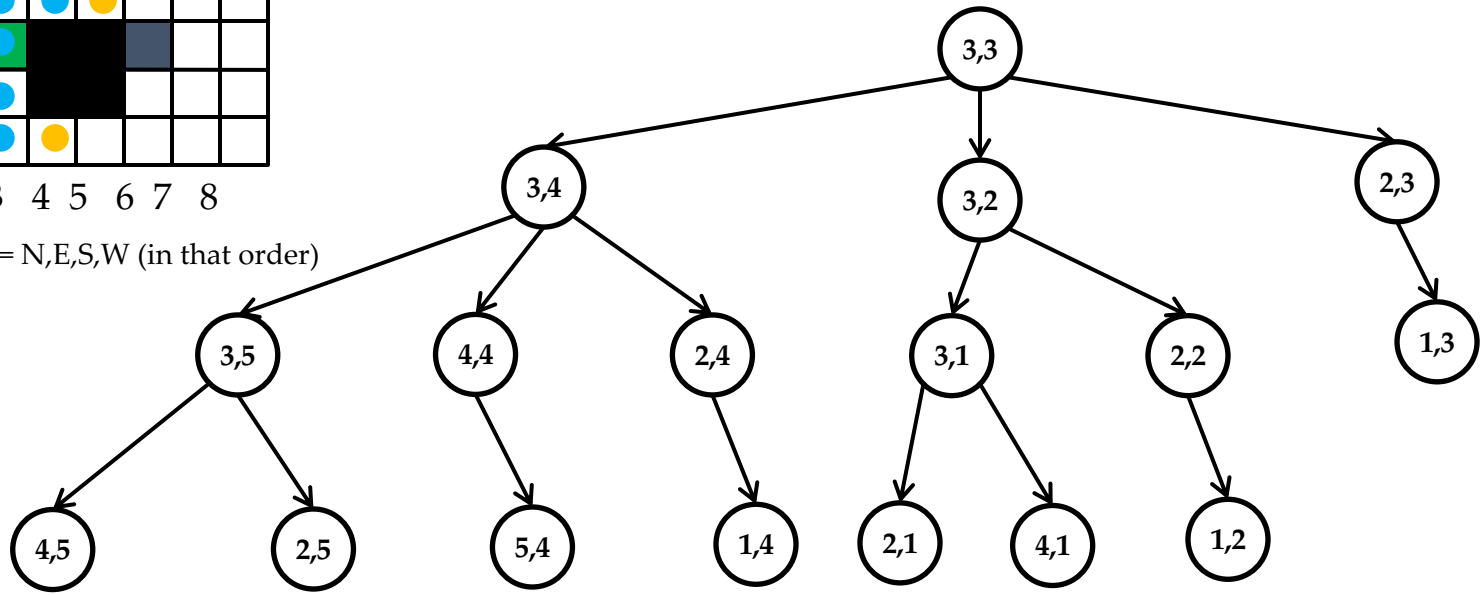
Search Tree: Breadth-first Search (no duplicates in fringe)

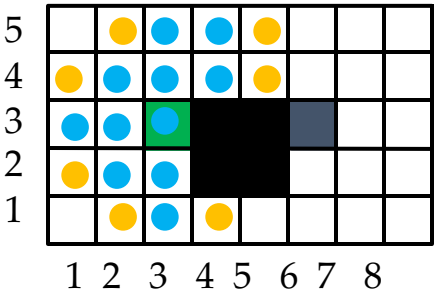




Neighbors = N,E,S,W (in that order)

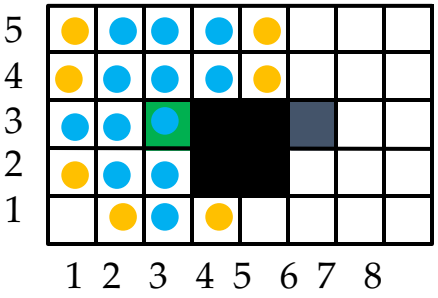
Search Tree: Breadth-first Search (no duplicates in fringe)





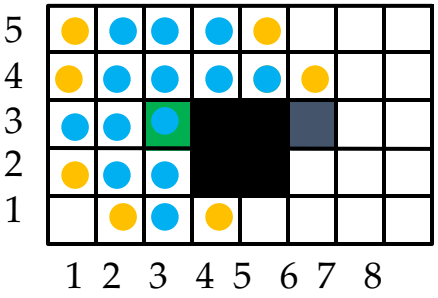
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



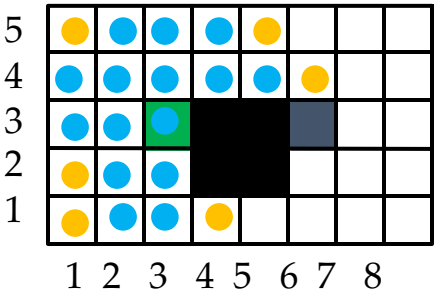
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



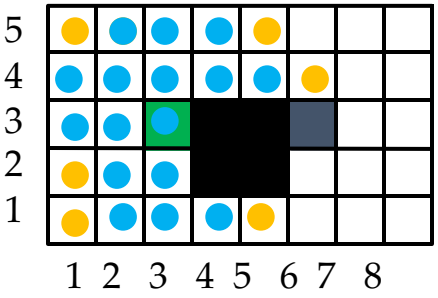
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



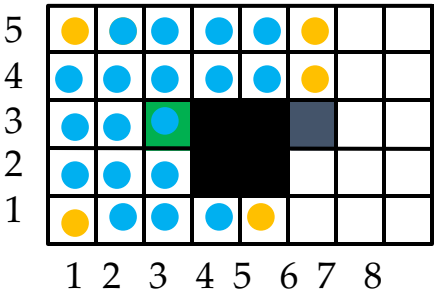
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



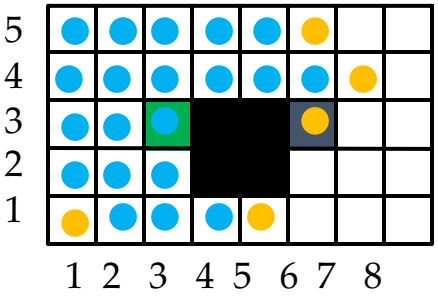
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



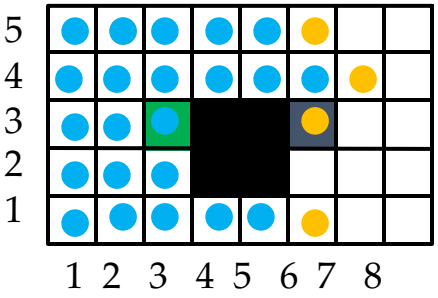
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



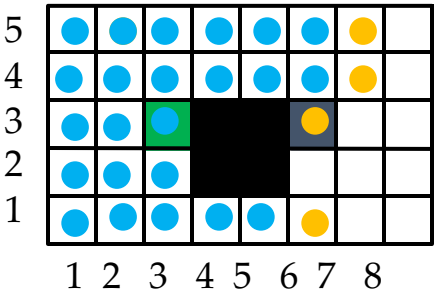
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



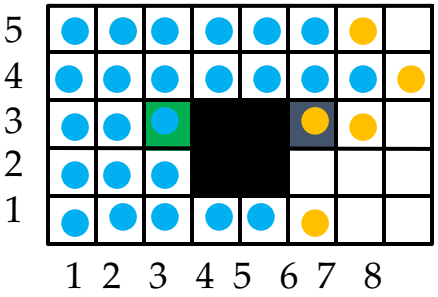
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



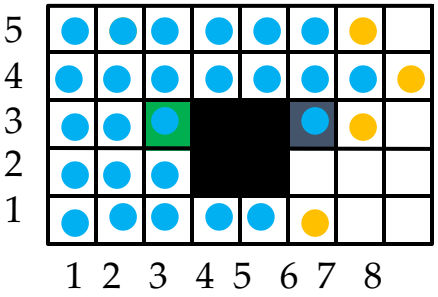
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



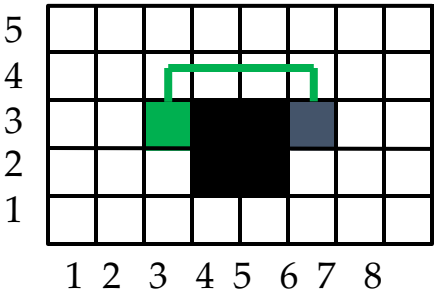
Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



Neighbors = N,E,S,W (in that order)

Search Tree: Breadth-first Search (no duplicates in fringe)



Neighbors = N,E,S,W (in that order)

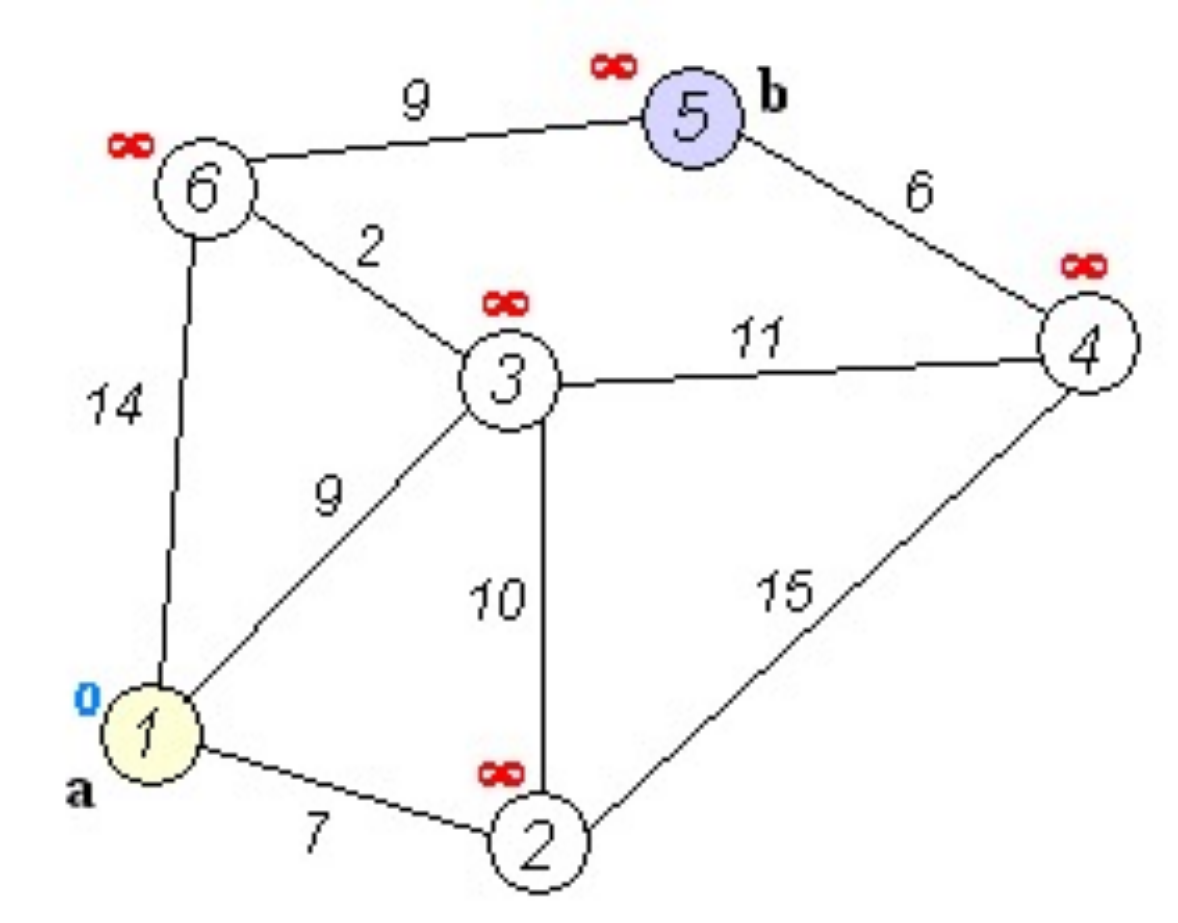
Search Tree: Breadth-first Search (no duplicates in fringe)

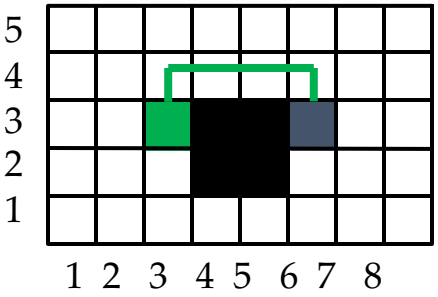
Uniform Cost Search (a.k.a. Dijkstra's Algorithm)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Uniform Cost Search (a.k.a. Dijkstra's Algorithm)





Neighbors = N,E,S,W (in that order)

Search Tree: Uniform-cost Search (no duplicates in fringe)

Uniform-cost search would be exactly the same, since cost per edge is 1.

Informed Search

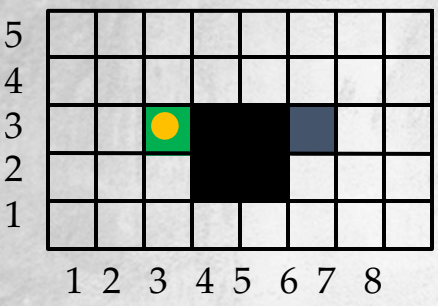
Informed Search

- To bias the search toward the goal we can tell the algorithm how close a state is to the goal, i.e. a *heuristic*
- A *heuristic* returns an estimate of the cost to get to the goal from any state
  - Must return 0 at a goal
- Most common heuristic for robotics – distance to goal (ignoring obstacles)
  - Manhattan distance: distance when only allowed to travel along the x or y axes
  - Euclidean distance: straight line distance

Informed Search

- FRINGE is a priority queue, with priority of each node calculated using the heuristic *h*
- A first idea: expand the node that is closest to the goal!
- **Greedy Best-First Search**
  - Priority = *h(node.state)*
  - Always expands the closest node to the goal

- Let's see how this works!

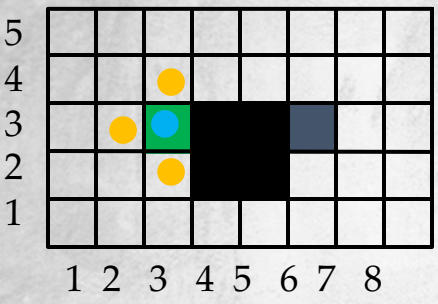


Neighbors = N,E,S,W (in that order)

Search Tree: Greedy Best-First (no duplicates in fringe)

heuristic = manhattan distance

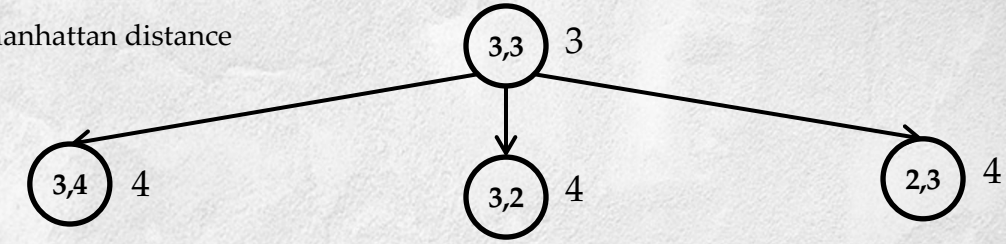
3

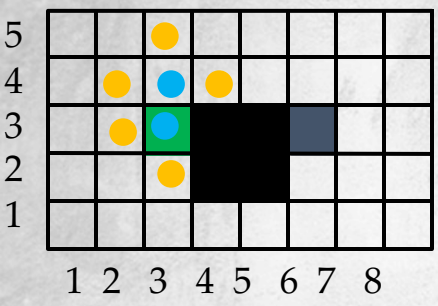


Neighbors = N,E,S,W (in that order)

Search Tree:Greedy Best-First (no duplicates in fringe)

heuristic = manhattan distance

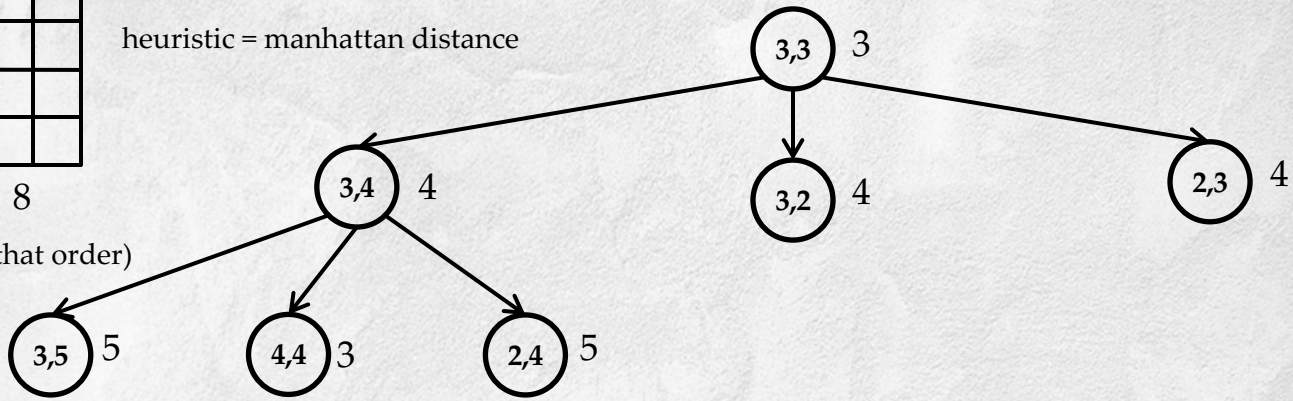


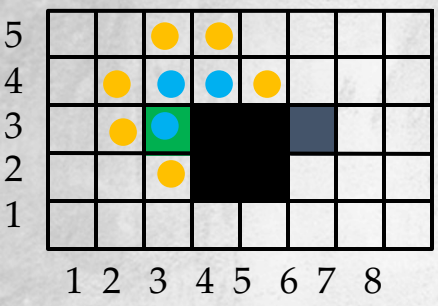


Neighbors = N,E,S,W (in that order)

Search Tree:Greedy Best-First (no duplicates in fringe)

heuristic = manhattan distance

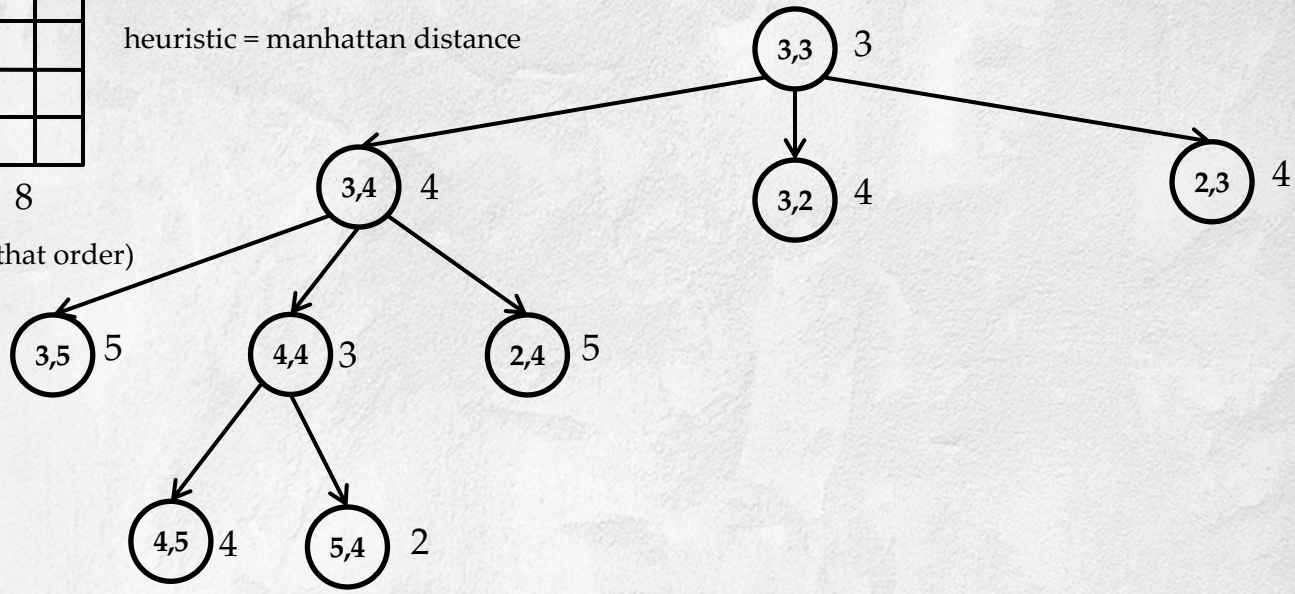


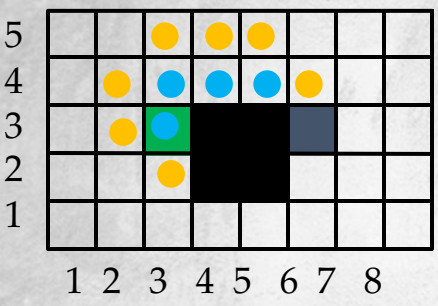


Neighbors = N,E,S,W (in that order)

Search Tree:Greedy Best-First (no duplicates in fringe)

heuristic = manhattan distance

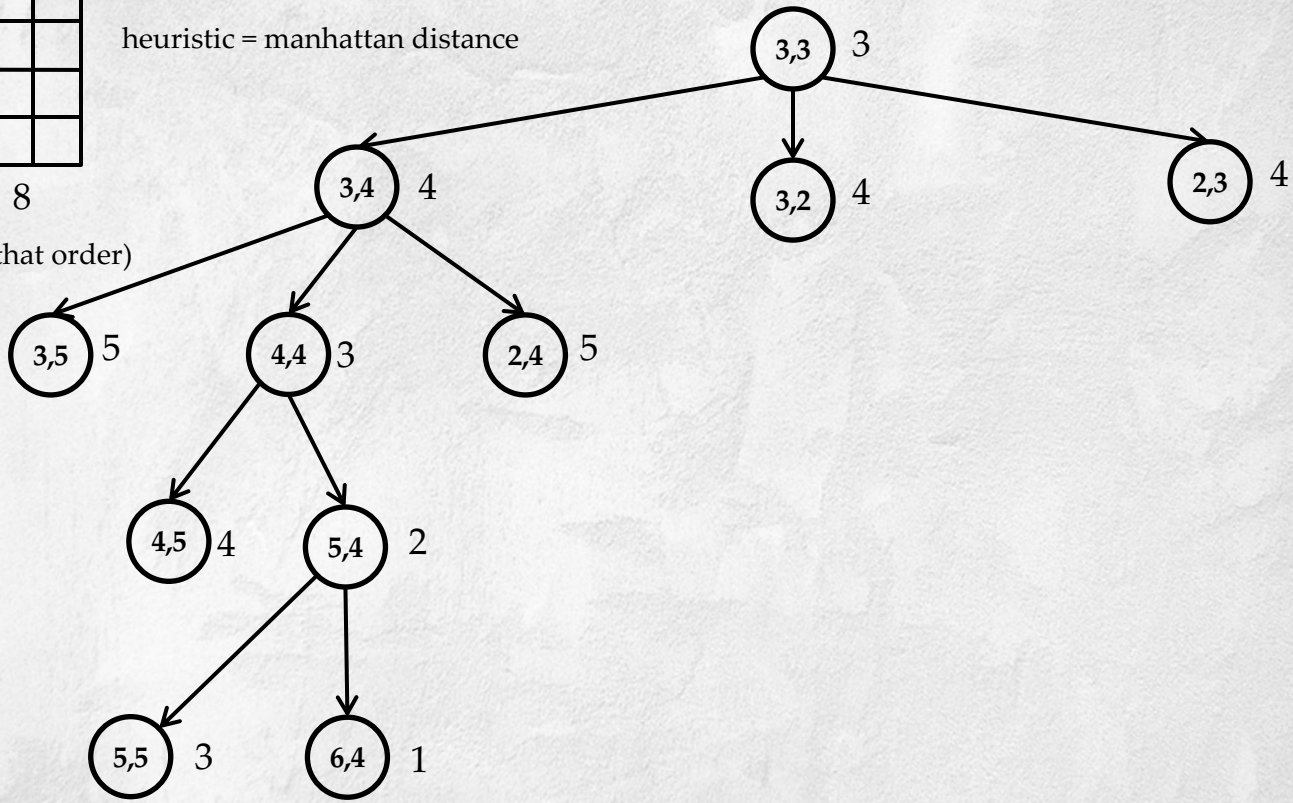




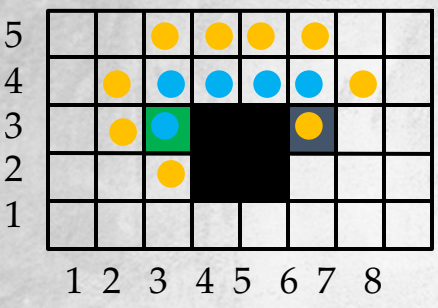
Neighbors = N,E,S,W (in that order)

Search Tree:Greedy Best-First (no duplicates in fringe)

heuristic = manhattan distance

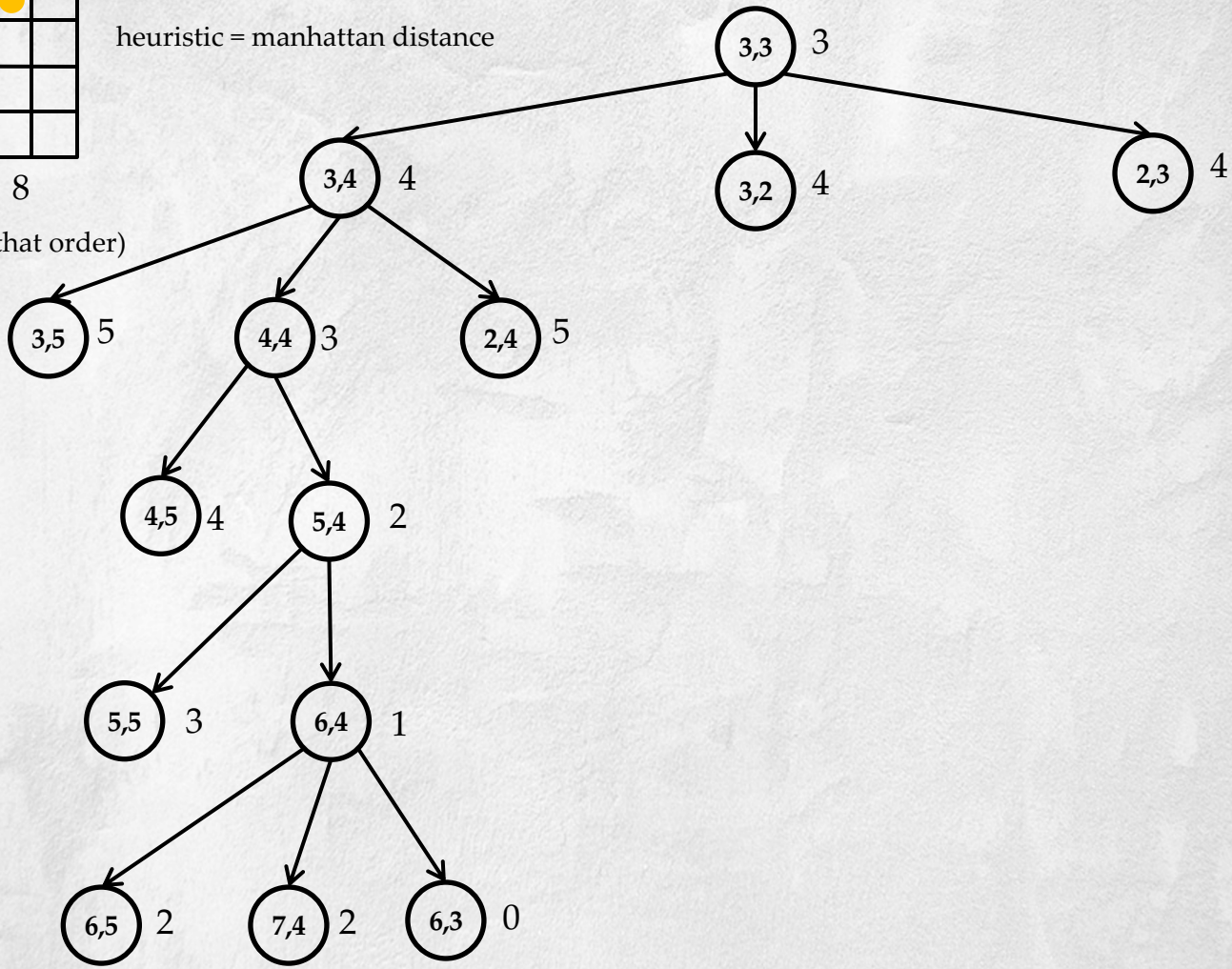


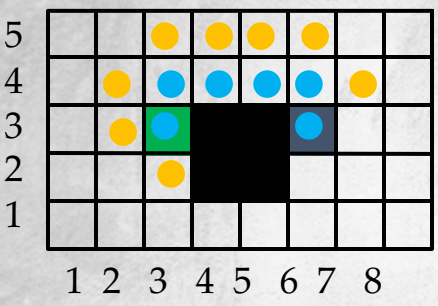
Search Tree:Greedy Best-First (no duplicates in fringe)



Neighbors = N,E,S,W (in that order)

heuristic = manhattan distance

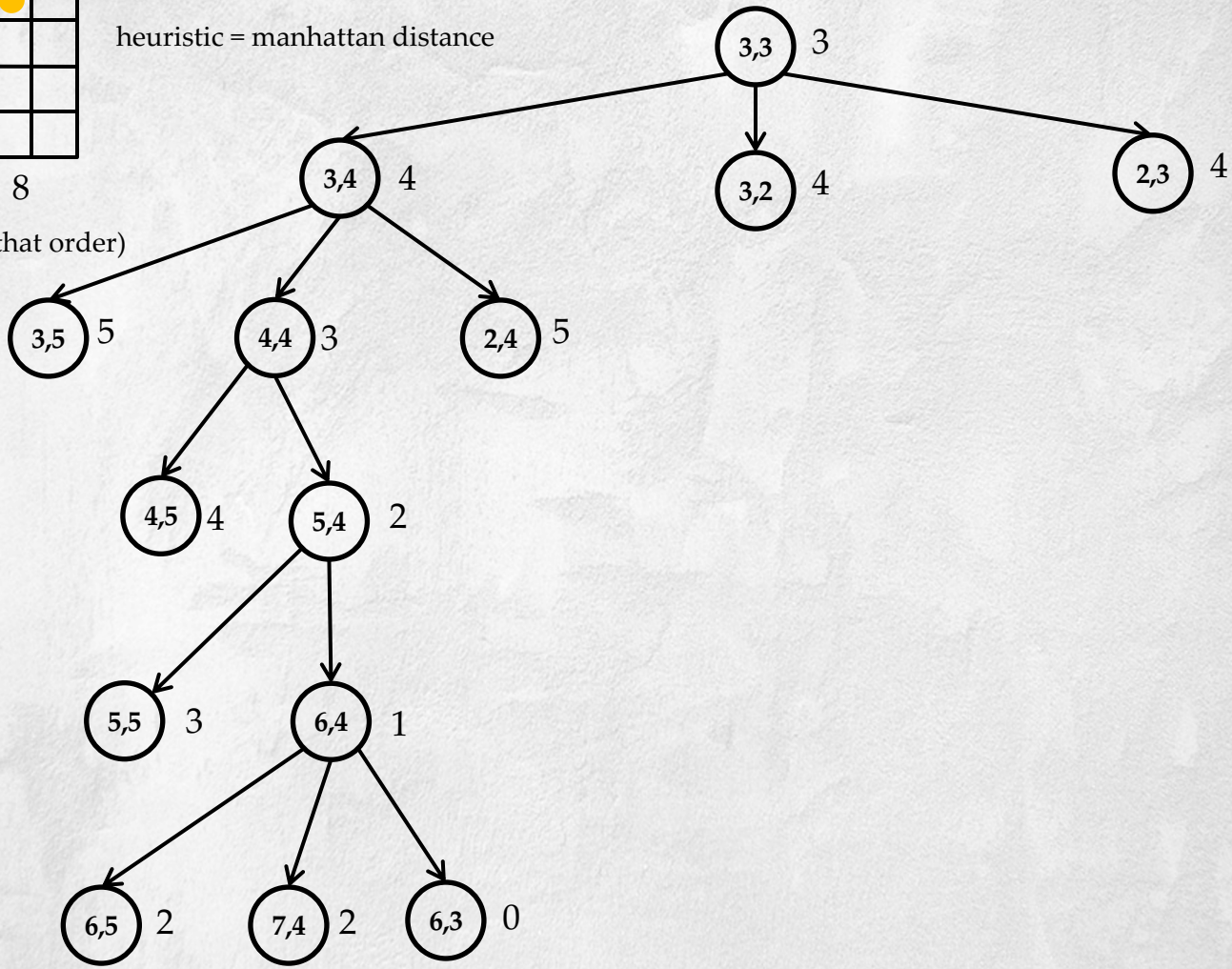




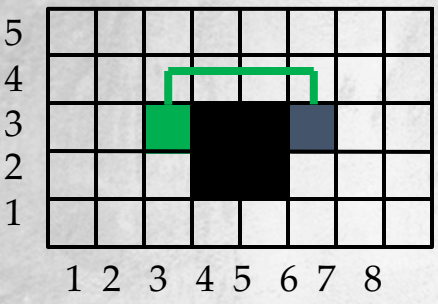
Neighbors = N,E,S,W (in that order)

Search Tree:Greedy Best-First (no duplicates in fringe)

heuristic = manhattan distance

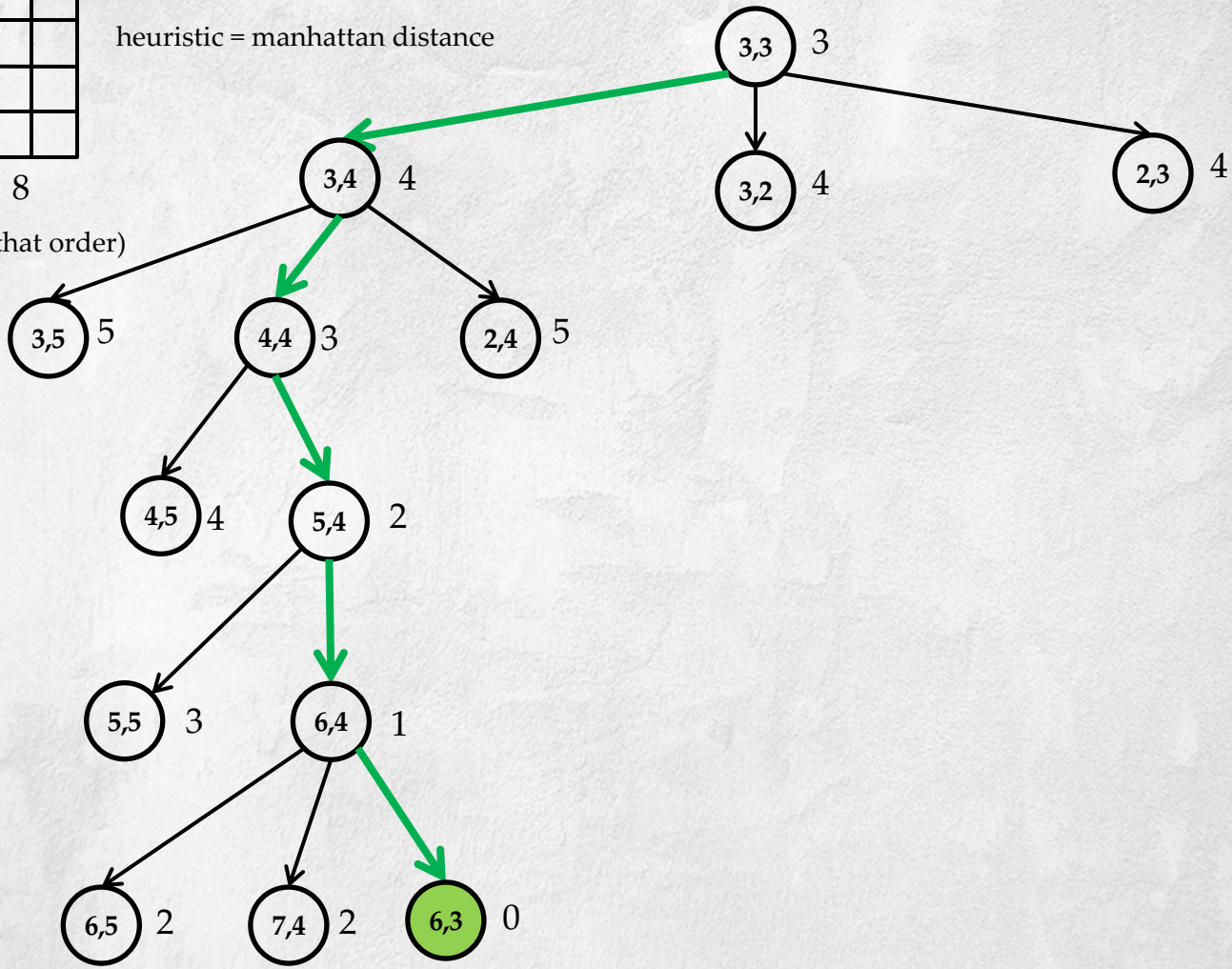


Search Tree:Greedy Best-First (no duplicates in fringe)

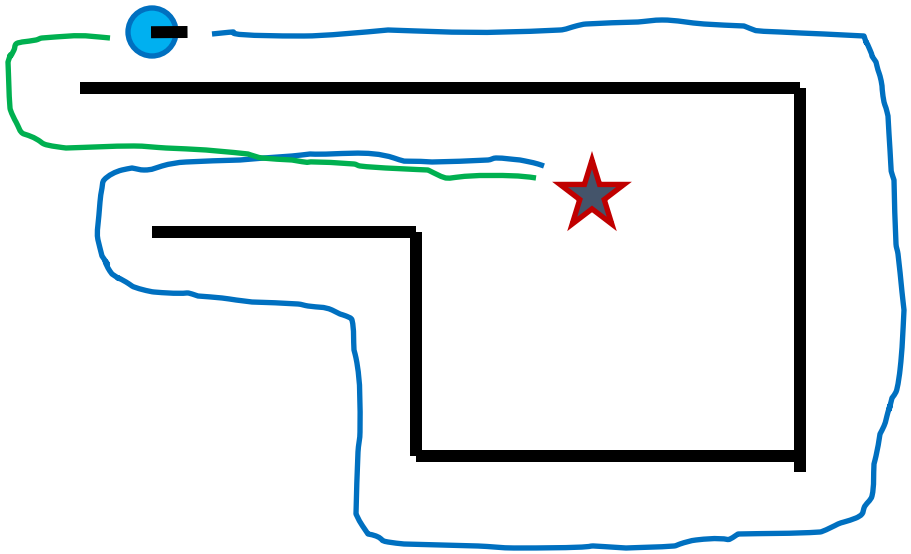


heuristic = manhattan distance

Neighbors = N,E,S,W (in that order)



Another greedy example



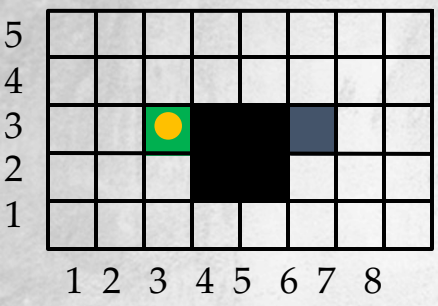
Doesn't find the optimal solution

Informed Search

- FRINGE is a priority queue, with priority of each node calculated using the heuristic *h*
- **Greedy Best-First Search**
  - Priority = *h(node.state)*
  - Always expands the closest node to the goal
  - Optimal?
    - No, ignores cost of path. Could be too much
  - Properties?
    - Usually very fast, small search tree

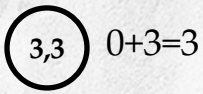
Informed Search

- FRINGE is a priority queue, with priority of each node calculated using the heuristic *h*
- A second idea: look at cost and heuristic
- **A* Search**
  - Priority = *node.cost* + *h(node.state)*
  - Takes into account both cost of path so far and cost to get to goal
  - Estimates *total path cost* for that partial path
  - Optimal?
    - Yes, if used with *admissible heuristic* (never overestimates cost to goal)

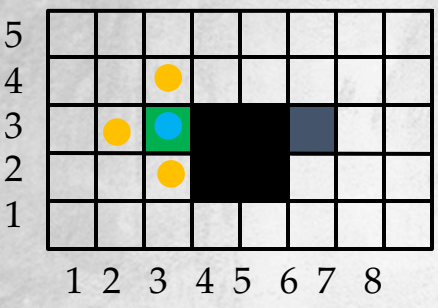


Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance



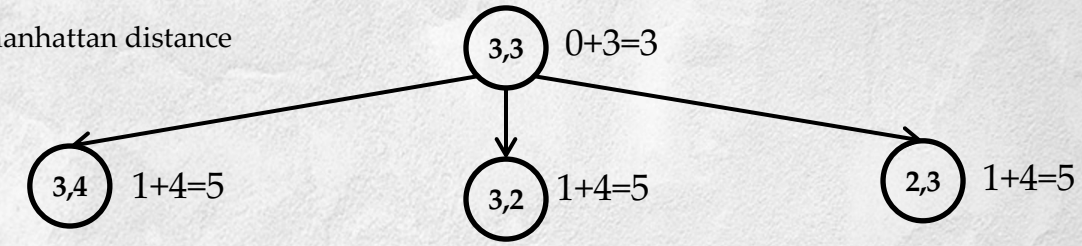
Neighbors = N,E,S,W (in that order)

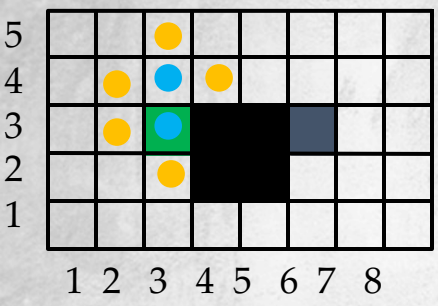


Neighbors = N,E,S,W (in that order)

Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

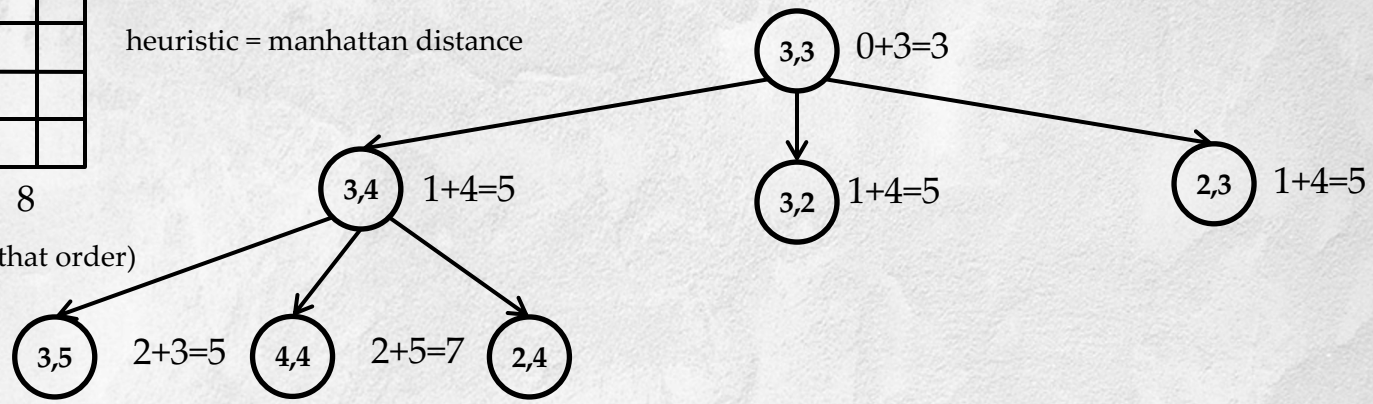
heuristic = manhattan distance



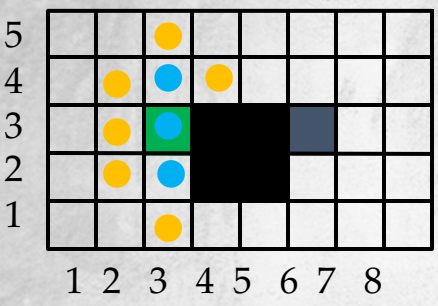


Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance



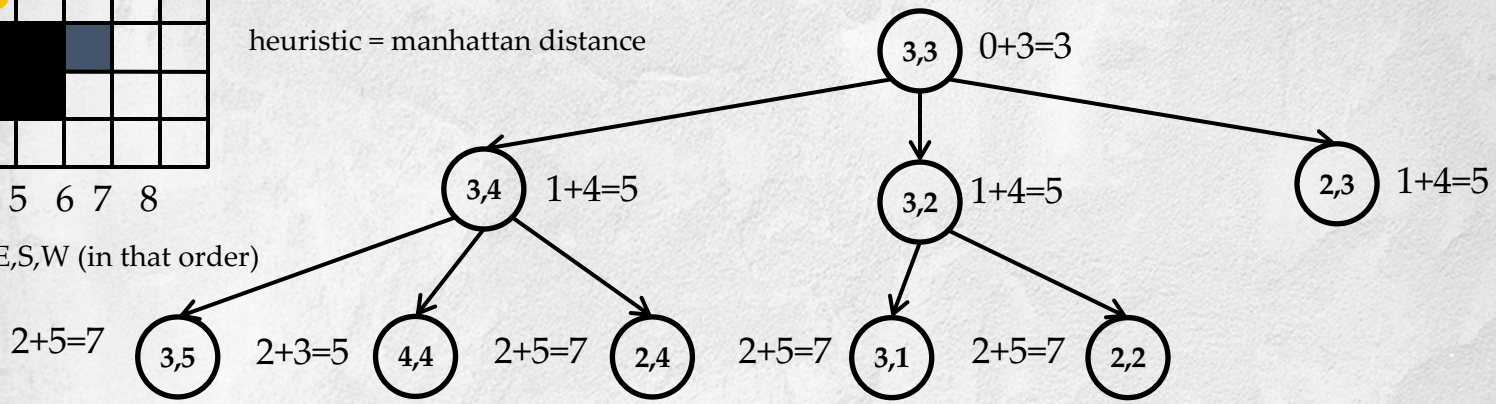
Neighbors = N,E,S,W (in that order)

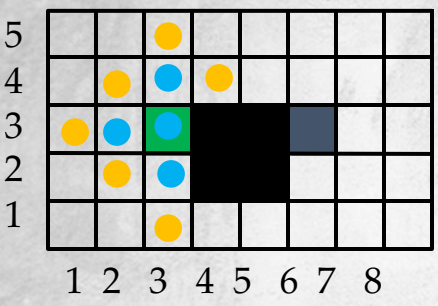


Neighbors = N,E,S,W (in that order)

Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance

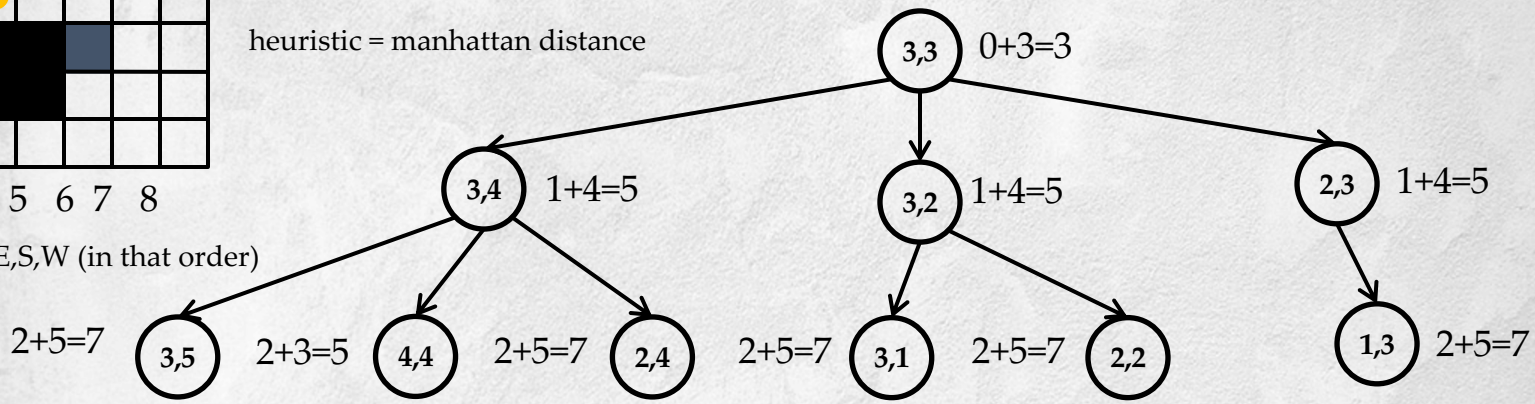


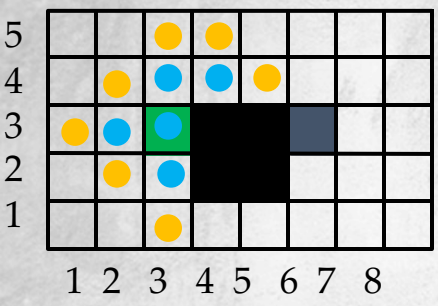


Neighbors = N,E,S,W (in that order)

Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance

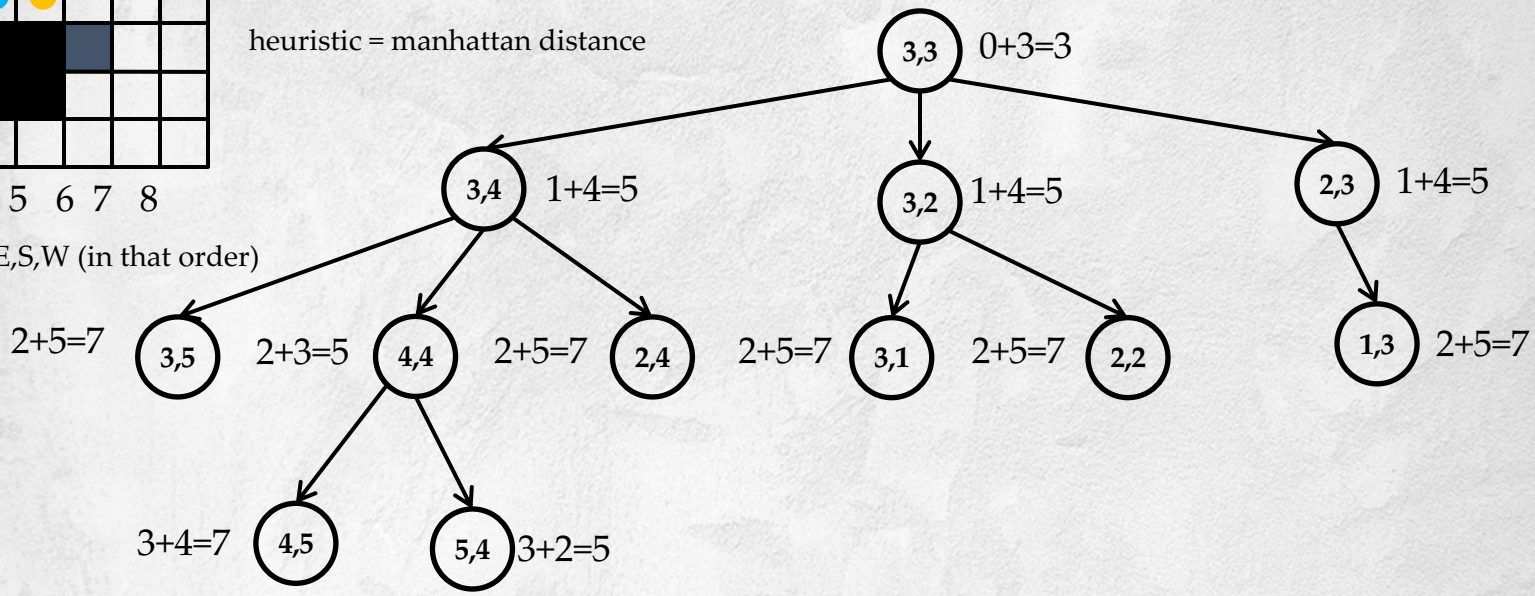


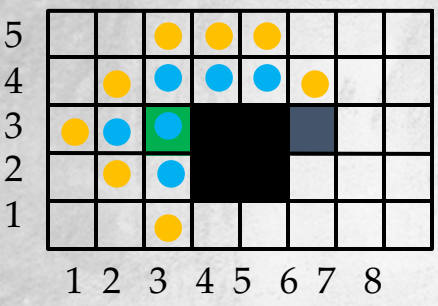


Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance

Neighbors = N,E,S,W (in that order)

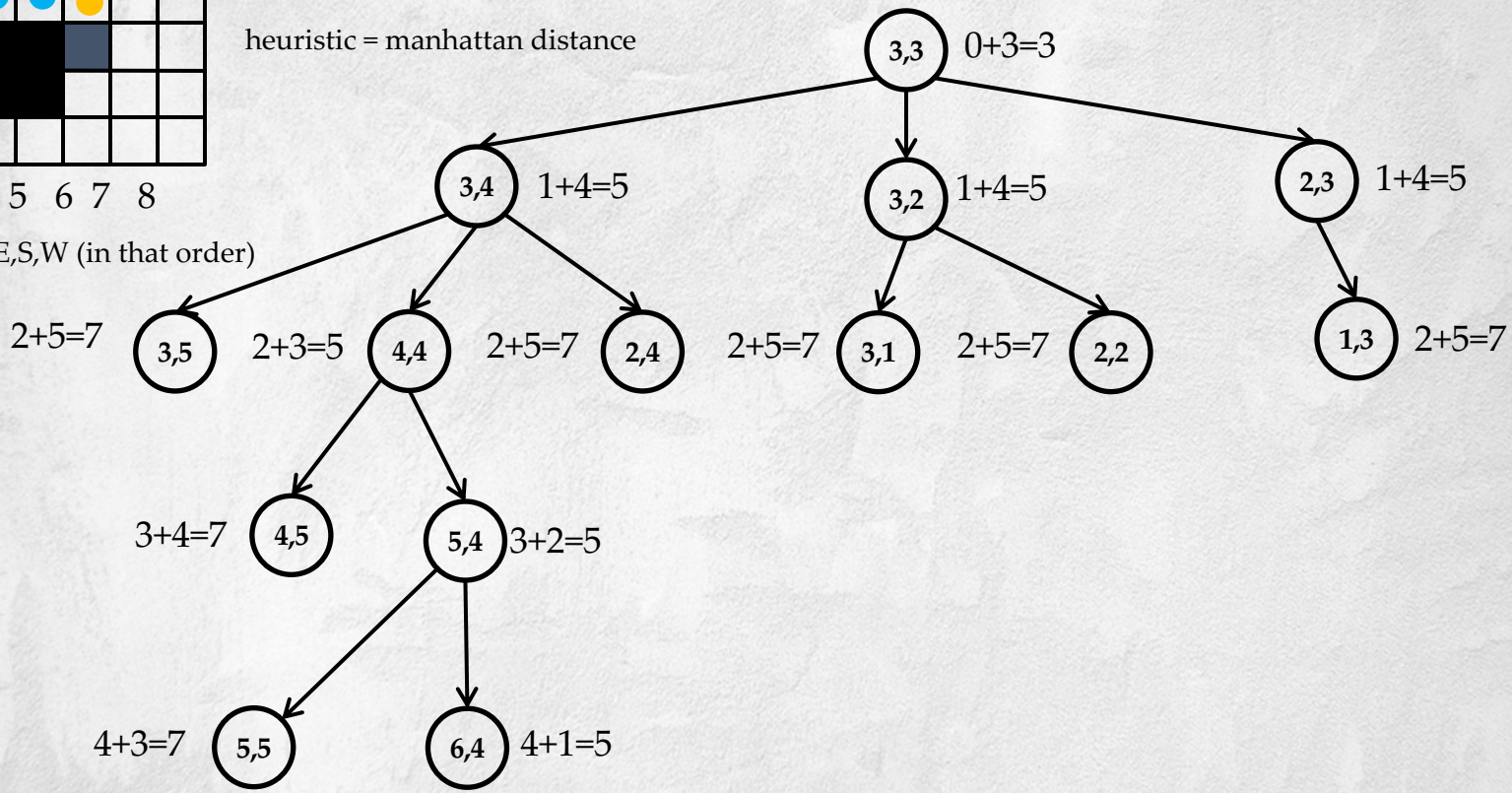


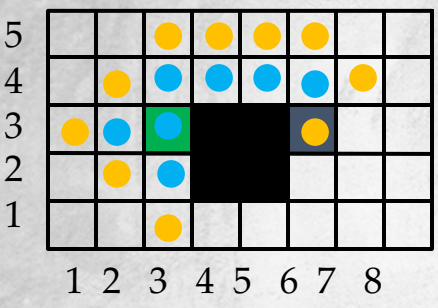


Neighbors = N,E,S,W (in that order)

Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance

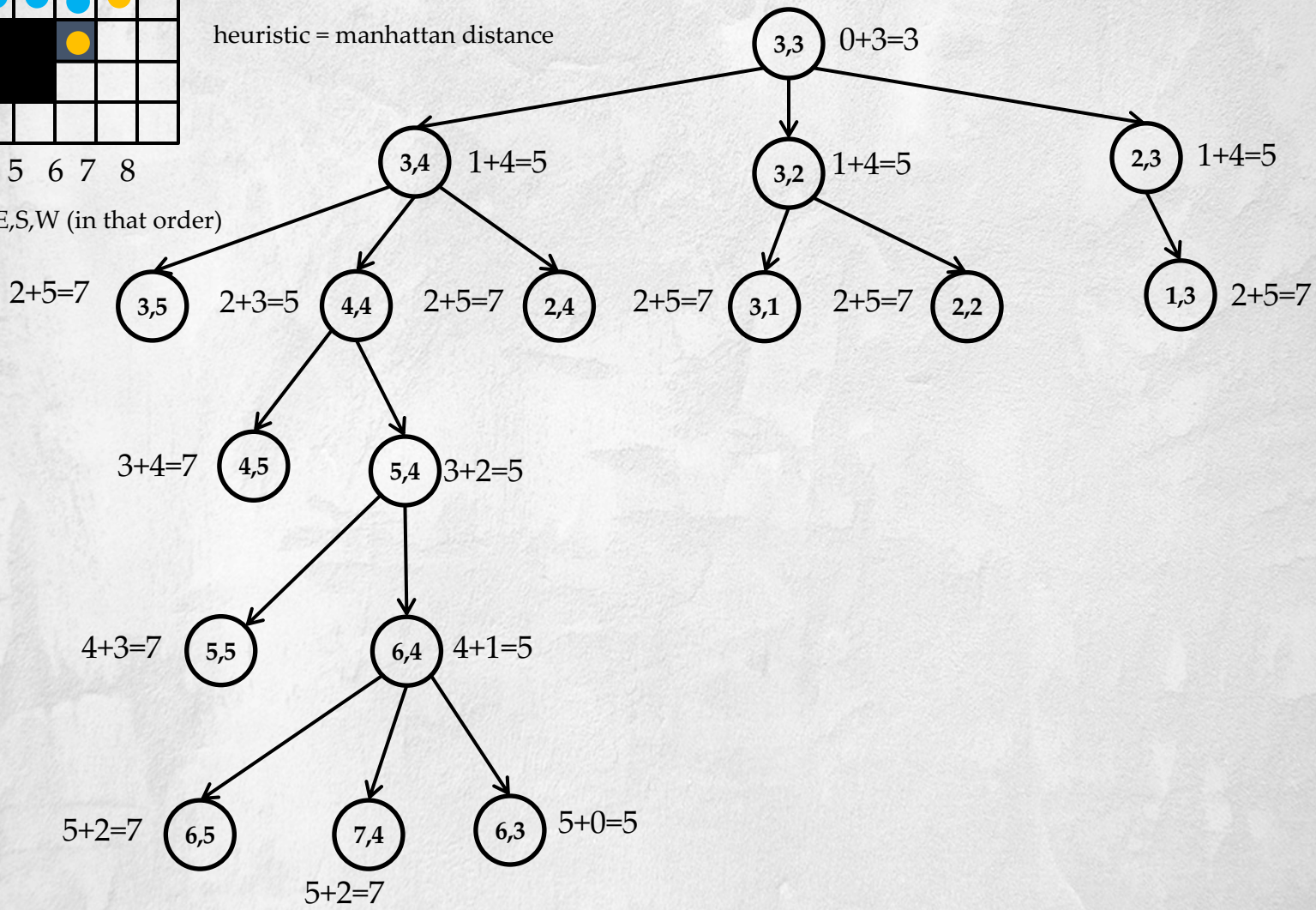


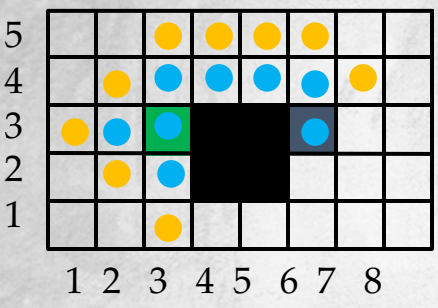


Neighbors = N,E,S,W (in that order)

Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance

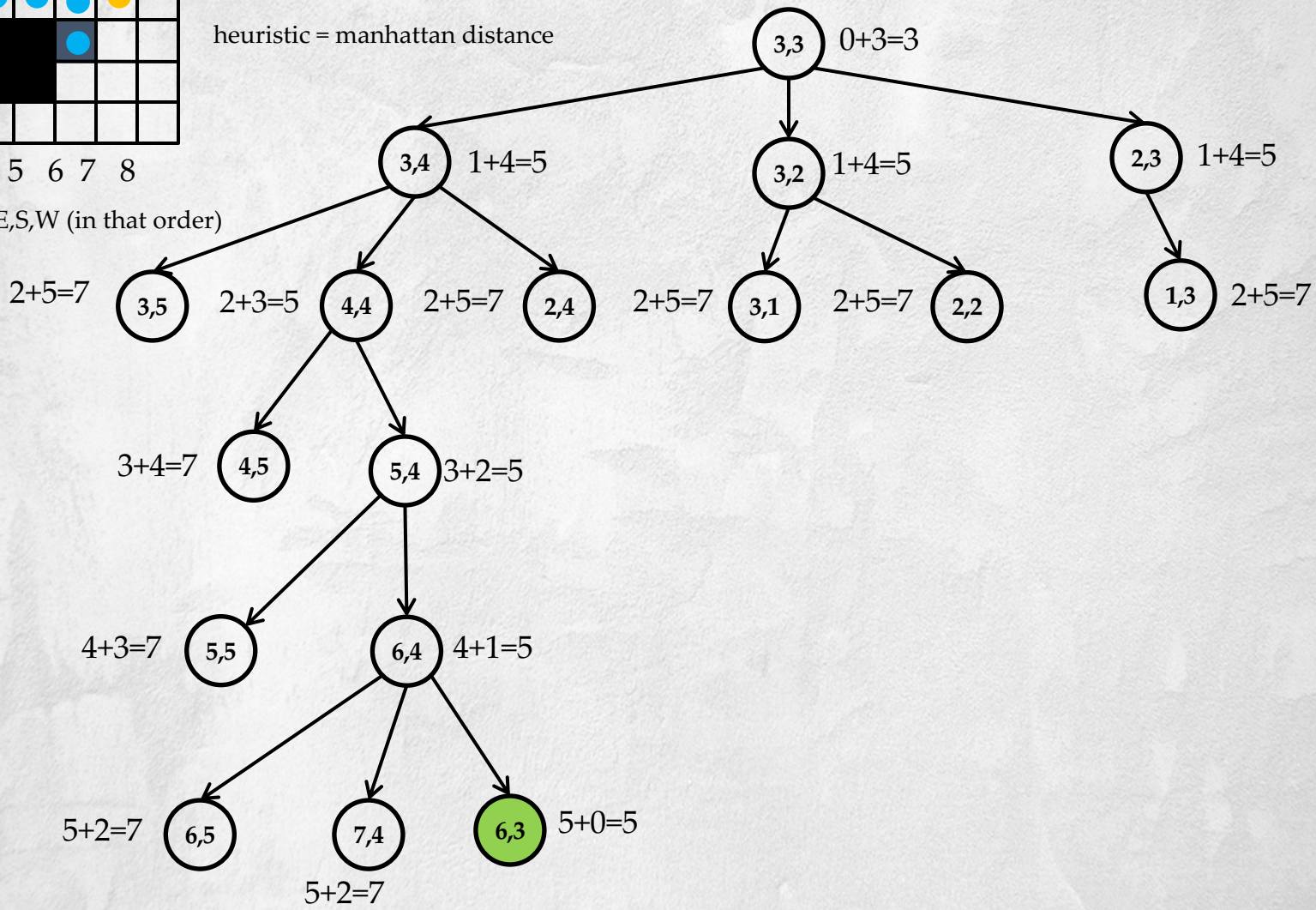


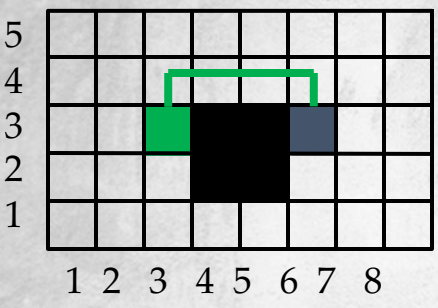


Neighbors = N,E,S,W (in that order)

Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance

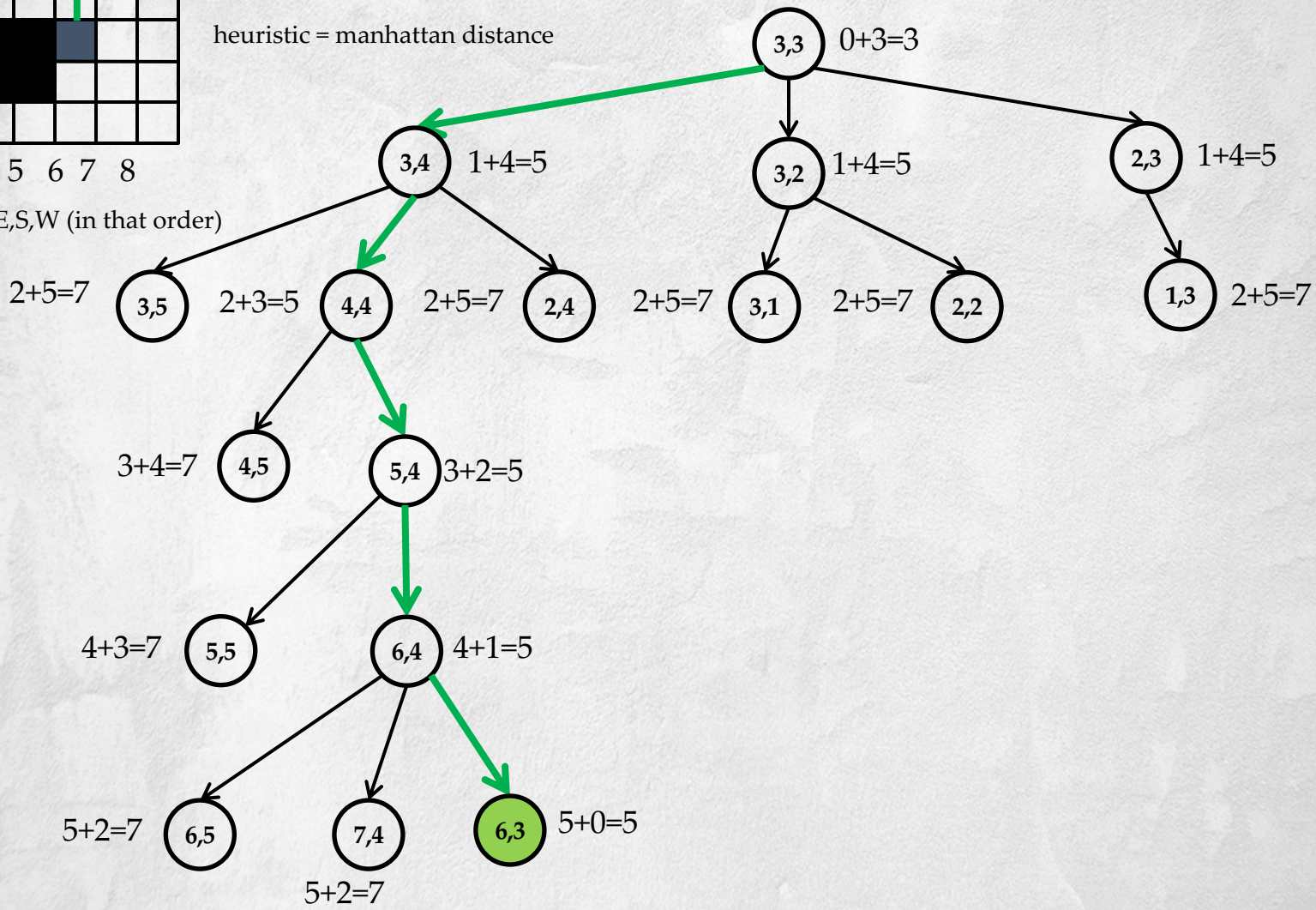




Neighbors = N,E,S,W (in that order)

Search Tree: A* search (no duplicates in fringe) (FIFO tiebreaking)

heuristic = manhattan distance



Why is A* optimal?

- Could there be a shorter path in the tree?
  - Not if *h* doesn't overestimate cost to get to the goal
  - Cost + *h* is the cost of the best possible path to get to the goal along that partial path
  - If that is higher than the cost to get to the goal, then it can't be better

Analysis

Uninformed Search

Criterion	Breadth-First	Uniform-Cost	Depth-First
Complete?	Yes[a]	Yes[a,b]	No
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$
Optimal?	Yes[c]	Yes	No

b: branching factor
d: depth of the shallowest solution
m: maximum depth of search tree

*C**: length of shortest path
ε: minimum edge length

Informed Search

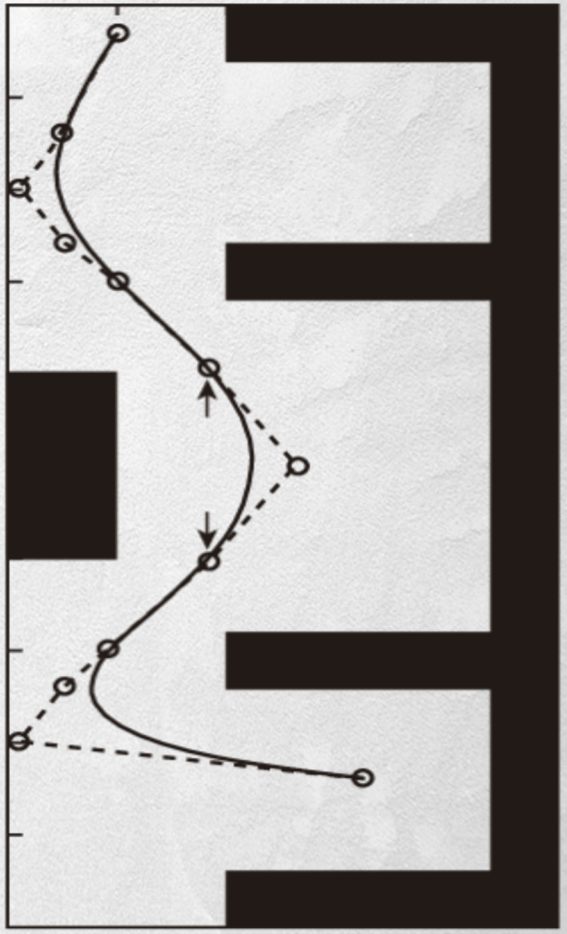
Criterion	**Greedy Best-First Search**	**A* Search**
Complete?	No	Yes
Time	$O(b^m)$	$O(b^d)$
Space	$O(b^m)$	$O(b^d)$
Optimal?	No	Yes

b: branching factor
d: depth of the shallowest solution
m: maximum depth of search tree

*C**: length of shortest path
ε: minimum edge length

Path Smoothing

- We can try to "fit a curve" to the points along the planned path
- Bezier curves work nicely
- *Curve may hit an obstacle where the straight line did not*



Velocity Profiling

- Accelerate as the robot goes along the path, then decelerate near the end
- Limit velocity when curvature is high
- For each segment of the path, calculate distance and curvature:
  - Iterate forwards and set velocity at each point based on the max velocity given acceleration (from kinematics) and curvature
  - Do the same thing in reverse to handle deceleration

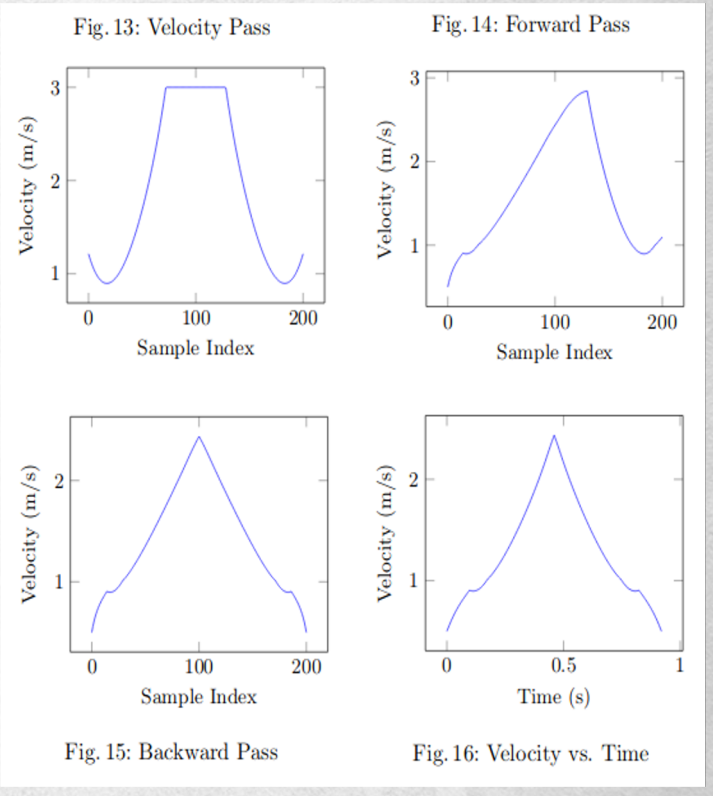


Fig. 13: Velocity Pass
Fig. 14: Forward Pass
Fig. 15: Backward Pass
Fig. 16: Velocity vs. Time

References

1. https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html
2. https://www.geeksforgeeks.org/search-algorithms-in-ai/
3. https://medium.com/kredo-ai-engineering/search-algorithms-part-1-problem-formulation-and-searching-for-solutions-28f722b7a1a6