CS 380: Artificial Intelligence

Lecture 3: Problem Solving & Uninformed Search

Problem Solving

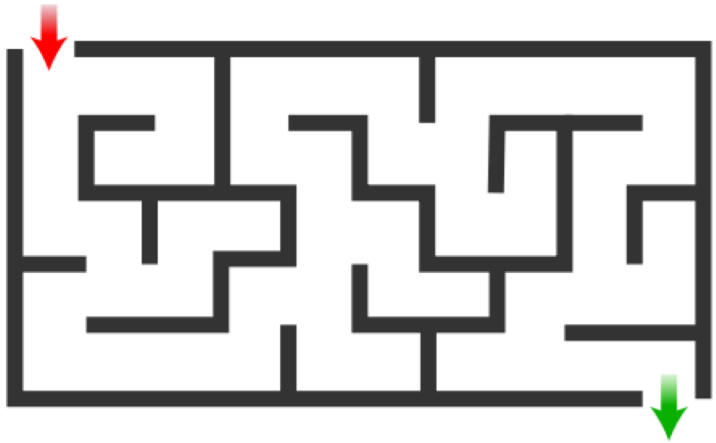
- What is a “problem”?
  - In the context of CS/AI: a question that requires an answer
- Many types of problems
  - **Decision problems**
  - **Search problems**
  - **Counting problems**
  - **Optimization problems**

Problem Solving

- There are many problems for which we know specialized ways of solving.
- For example:
  - Problem: Find the roots of the polynomial $ax^2 + bx + c = 0$
  - We have mechanical procedures to solve this problem in an exact way
  - However, those procedures can only be applied to this problem, but not to any other
- Problem Solving in AI:
  - Finding **general procedures** to solve general classes of problems
  - Is there an algorithm that can be used to solve **any** computationally solvable problem?
    - We humans seem to be able to!

Example: Maze

- A robot is at the entrance of a maze.
- Problem: How does the robot get to the exit?
- Step 1: Formulate goal
  - Be in the exit
- Step 2: Formulate problem
  - States, actions, etc.
- Step 3: Find solution
  - Specific sequence of actions



Problem Formulation

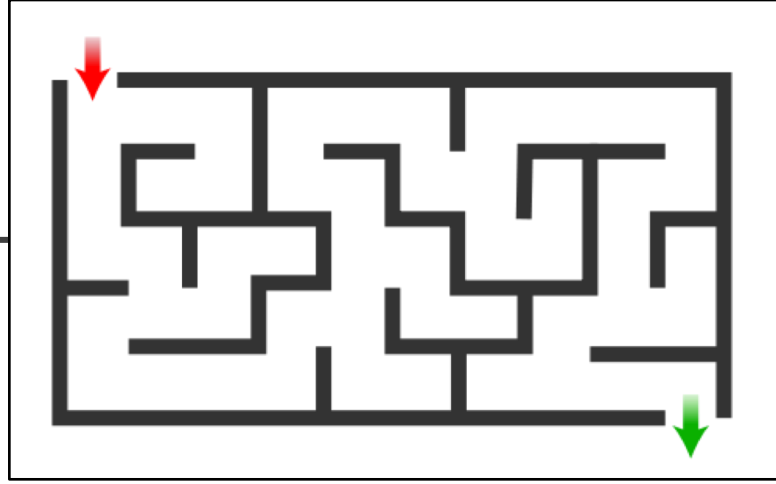
- Initial state: S_0
  - Initial configuration of the problem (e.g. starting position in a maze)
- Actions: A
  - The different ways in which the agent can change the state (e.g. moving to an adjacent position in the maze)
- Goal condition: G
  - A function that determines whether a state reached by a given sequence of actions is a solution to the problem or not.
- Cost function: c
  - A function that assigns a numeric cost to a given sequence of actions.

Problem Formulation

- Initial state: S_0
  - Initial configuration
    maze)
- Actions: A
  - The different ways i
    (e.g. moving to an a
- Goal condition: G
  - A function that determines whether a state reached by a given sequence of actions is a solution to the problem or not.
- Cost function: c
  - A function that assigns a numeric cost to a given sequence of actions.

The initial state, together with the set of actions, define the State Space or Search Space (the set of all the possible states that are reachable from the initial state by some sequence of actions).

Problem Types



- Deterministic, fully observable:
  - Solution: [down, down, down, right, down, …]
- Non-deterministic or partially observable:
  - Agent might now know its exact coordinates, percepts only tell them about the wall configuration around
  - Solution, policy (mapping from percepts, or states, to actions):
    - If no wall on left, then turn left
    - If wall on left, no wall ahead, then advance
    - If wall on left, wall ahead, then turn right
  - For example: the “follow the right/left wall” strategy

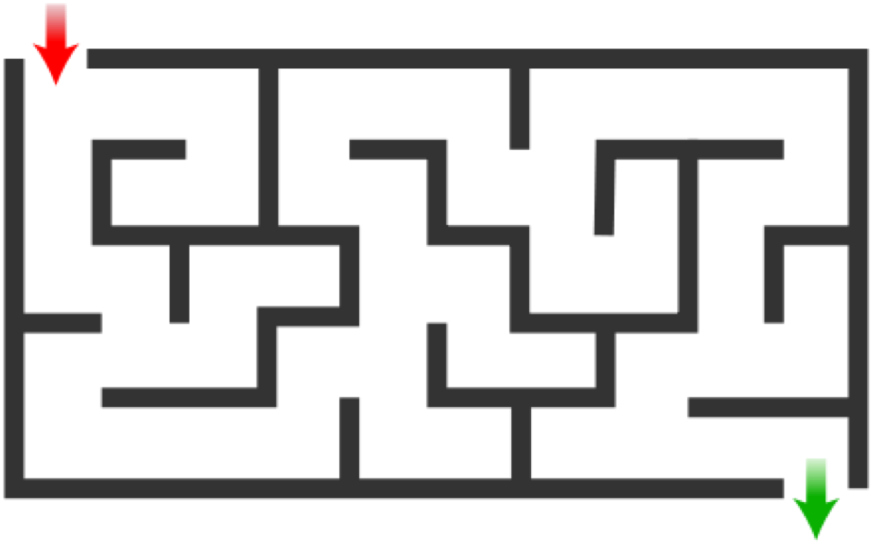
Plan vs. Policy

- Plan:
  - Specific sequence of actions:
    - left,
    - right,
    - up,
    - up
- Contingent plan:
  - Sequence of actions with conditionals:
    - left,
    - right,
    - If wall ahead then up else right

- Policy:
  - Mapping from percepts to actions:

Percept	Action
No walls	advance
Wall left	advance
Wall right	left
…	…
Walls in all directions	No-op

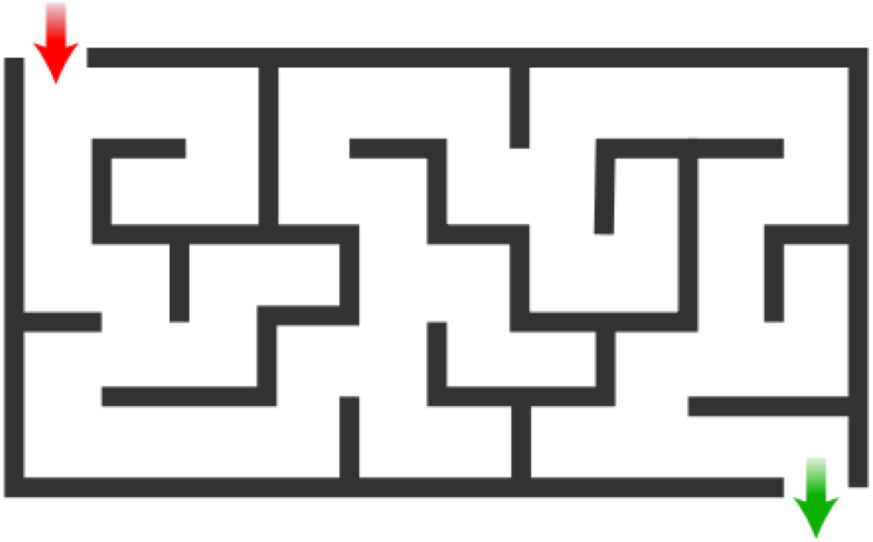
Example: Maze



- **States**:
- **Actions**:
- **Goal**:
- **Cost**:

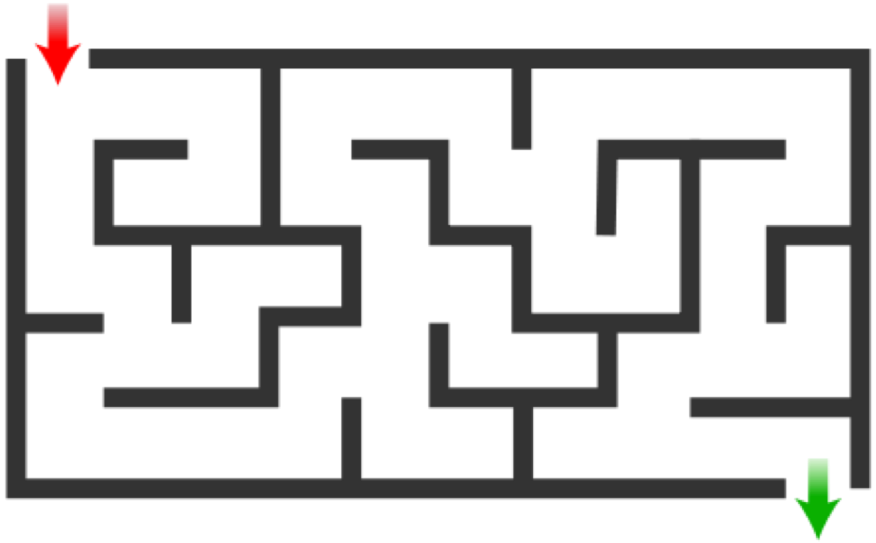
Assuming we have a robot with a direction and partial information…

Example: Maze



- **States**:
  - (x,y) of robot + facing direction
  - Initial state: ((0,0),down)
- **Actions**: advance, left, right
- **Goal**: reach state ((9,4),*)
- **Cost**: 1 per action executed

Example: Maze

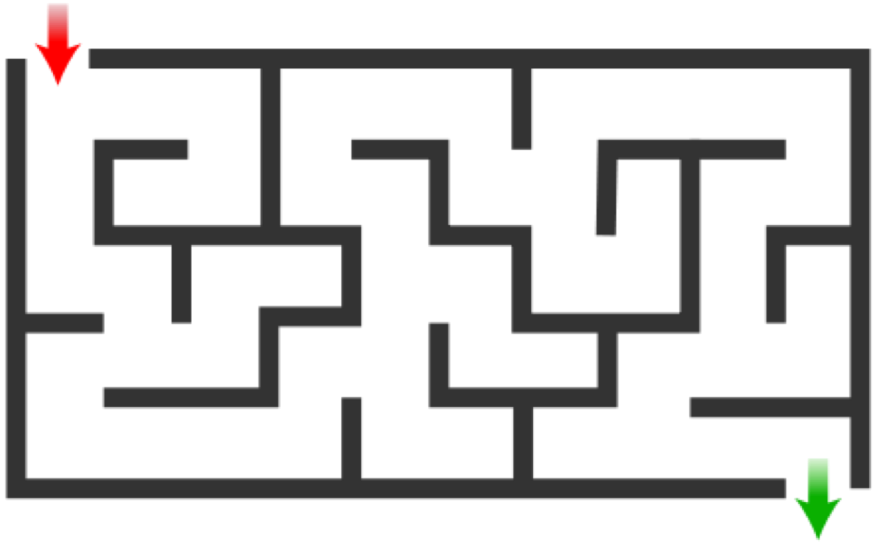


- **States**:
  - (x,y) of robot + facing direction
  - Initial state: ((0,0),down)
- **Actions**: advance, left, right
- **Goal**: reach state ((9,4),*)
- **Cost**: 1 per action executed

How difficult is this problem?

Let's say, compared to a larger maze…
or chess…

Example: Maze



- **States**:
  - (x,y) of robot + facing direction
  - Initial state: ((0,0),down)
- **Actions**: advance, left, right
- **Goal**: reach state ((9,4),*)
- **Cost**: 1 per action executed

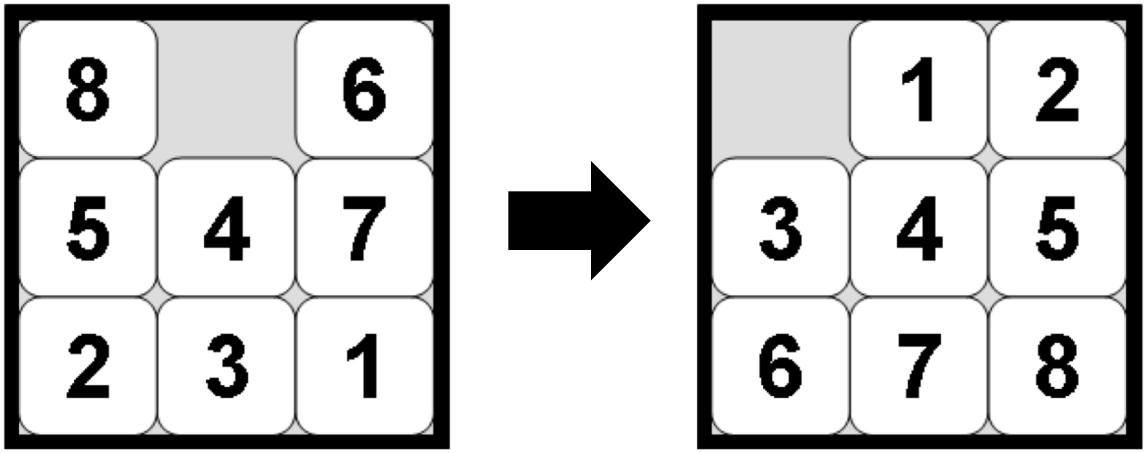
How difficult is a problem?

A typical measure of "difficulty" is the size of the state space: if it's small, it's very easy to find the solution.

For this maze here, the state space has 200 states (50 positions * 4 directions)

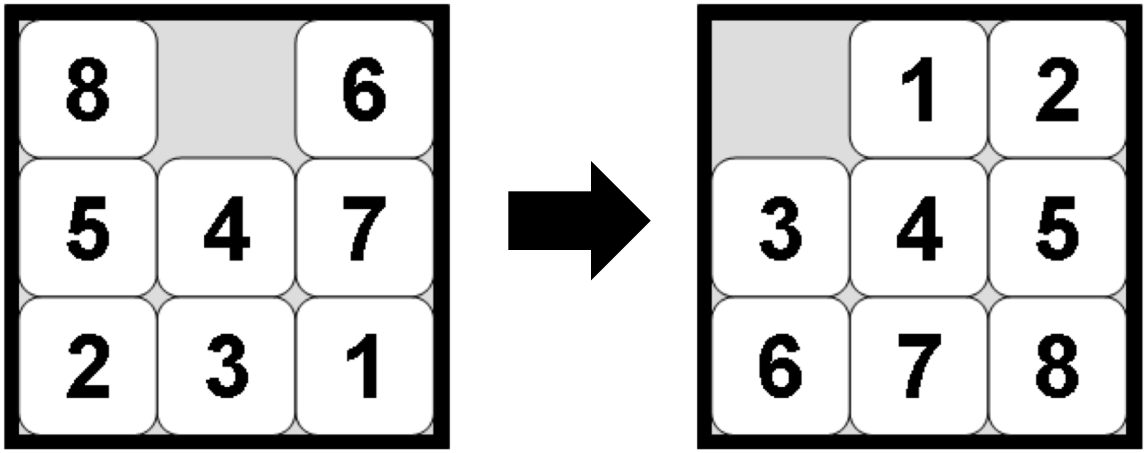
Chess has about 10^{50} states

Example: 8-puzzle



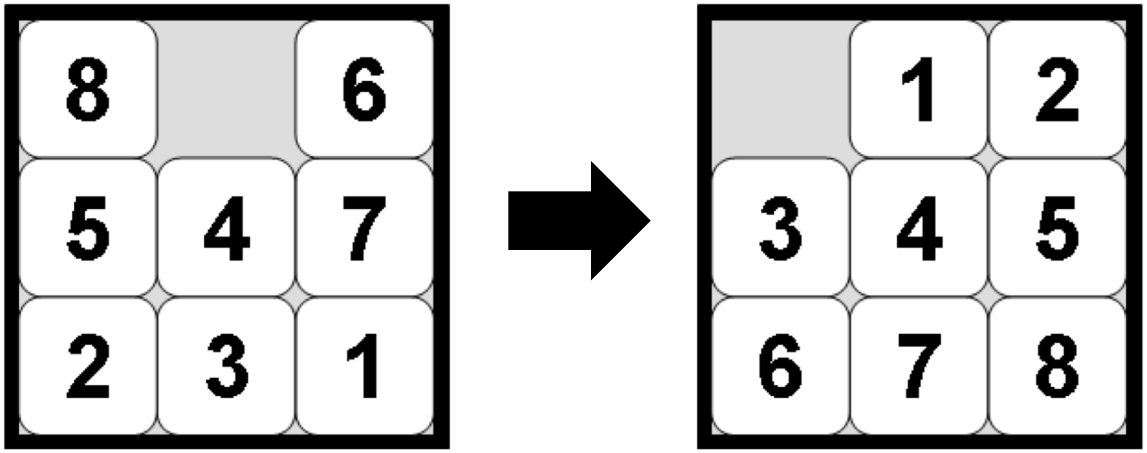
- **States:**
- **Actions:**
- **Goal:**
- **Cost:**

Example: 8-puzzle



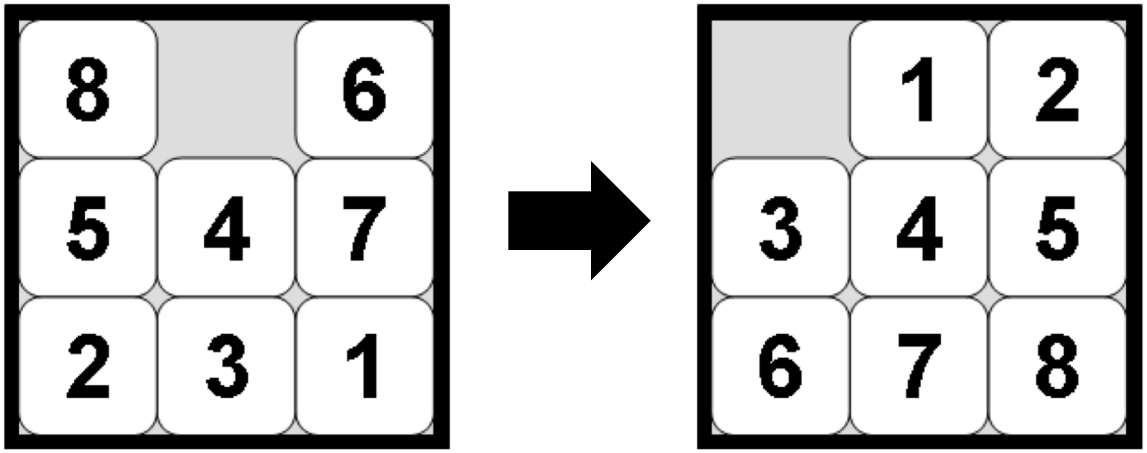
- **States:**
  - Board configuration, represented as a vector of 9 positions (each position represents one of the cells in the board)
  - Initial state: (8,blank,6,5,4,7,2,3,1)
- **Actions:**
- **Goal:**
- **Cost:**

Example: 8-puzzle



- **States:**
  - Board configuration, represented as a vector of 9 positions (each position represents one of the cells in the board)
  - Initial state: (8,blank,6,5,4,7,2,3,1)
- **Actions:**
  - move blank up, down, left, right
- **Goal:**
- **Cost:**

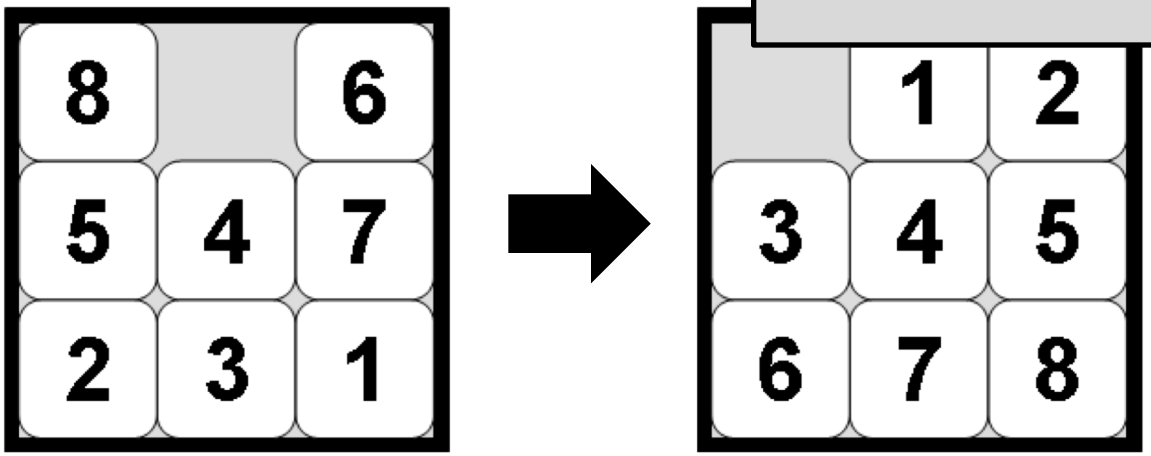
Example: 8-puzzle



- **States:**
  - Board configuration, represented as a vector of 9 positions (each position represents one of the cells in the board)
  - Initial state: (8,blank,6,5,4,7,2,3,1)
- **Actions:**
  - move blank up, down, left, right
- **Goal:**
  - Reach state (blank,1,2,3,4,5,6,7,8)
- **Cost:**
  - 1 per move

Example: 8-puzzle

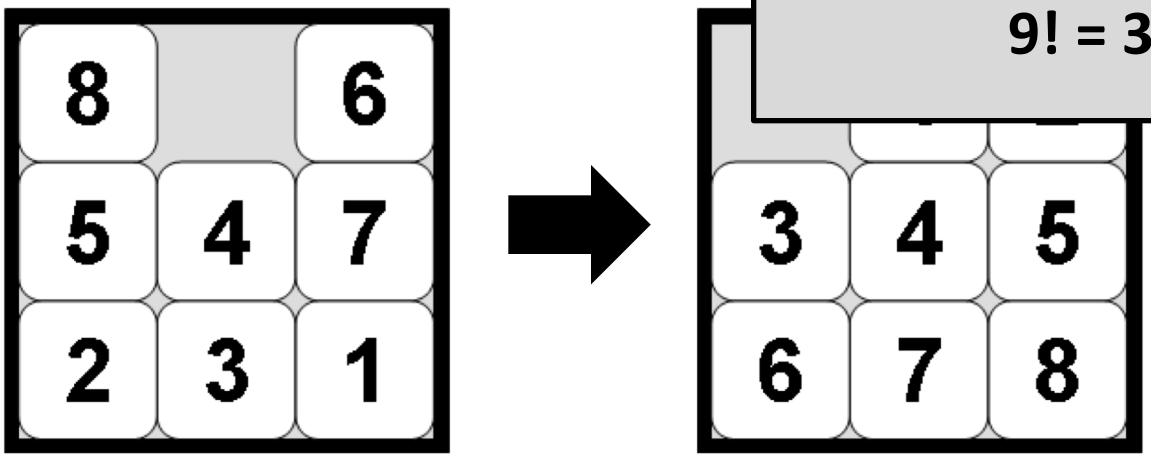
What's the size of the state space for the 8-puzzle?



- **States**:
  - Board configuration, represented as a vector of 9 positions (each position represents one of the cells in the board)
  - Initial state: (8,blank,6,5,4,7,2,3,1)
- **Actions**:
  - move blank up, down, left, right
- **Goal**:
  - Reach state (blank,1,2,3,4,5,6,7,8)
- **Cost**:
  - 1 per move

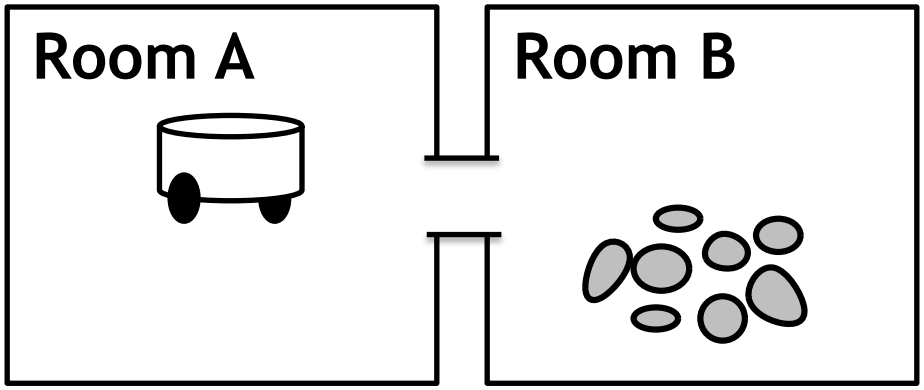
Example: 8-puzzle

What's the size of the state space for the 8-puzzle?
9! = 362880



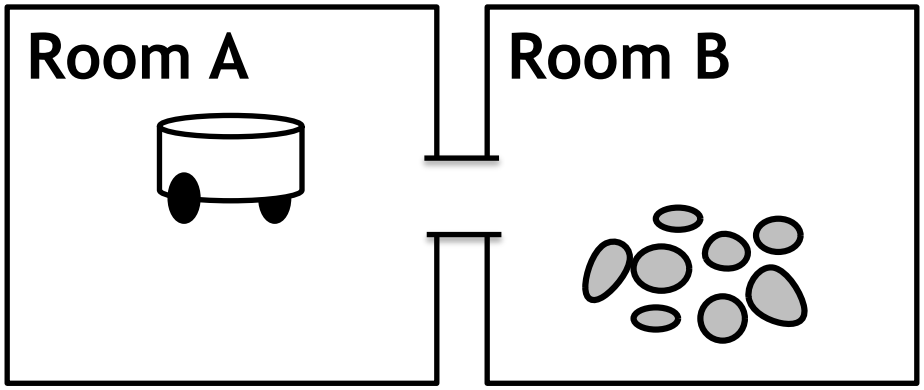
- **States:**
  - Board configuration, represented as a vector of 9 positions (each position represents one of the cells in the board)
  - Initial state: (8,blank,6,5,4,7,2,3,1)
- **Actions:**
  - move blank up, down, left, right
- **Goal:**
  - Reach state (blank,1,2,3,4,5,6,7,8)
- **Cost:**
  - 1 per move

Example: Vacuum World



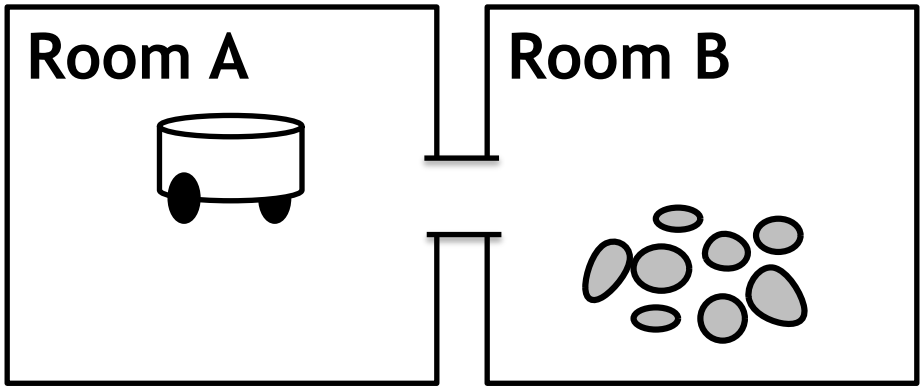
- States:
- Actions:
- Goal:
- Cost:

Example: Vacuum World



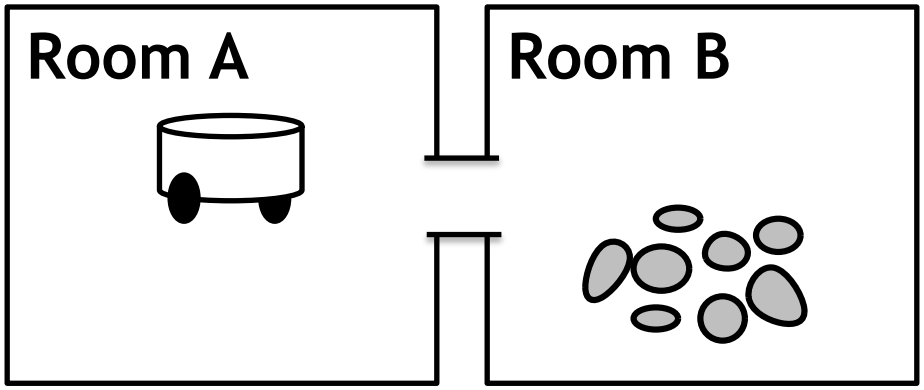
- **States:**
  - Position of robot + dirt in A? + dirt in B?
  - Initial state: (A, no, yes)
- **Actions:**
- **Goal:**
- **Cost:**

Example: Vacuum World



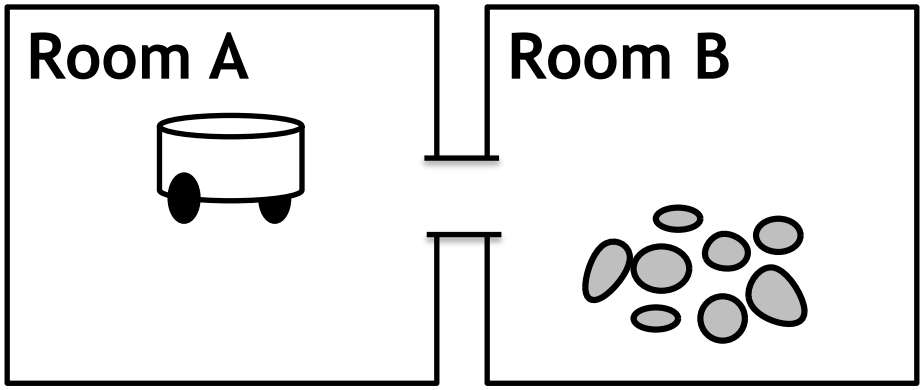
- **States:**
  - Position of robot + dirt in A? + dirt in B?
  - Initial state: (A, no, yes)
- **Actions:**
  - Left, right, suck, no-op
- **Goal:**
- **Cost:**

Example: Vacuum World



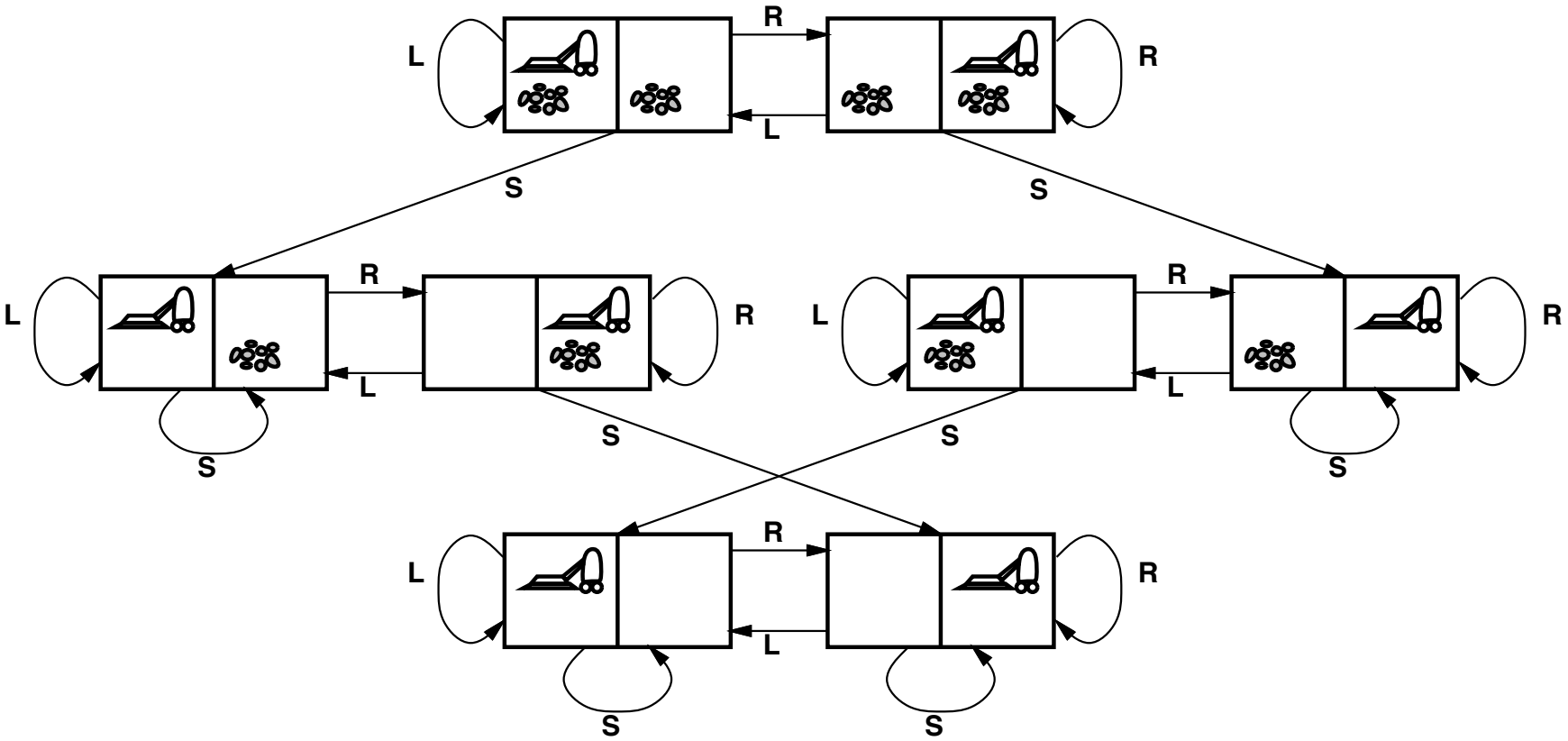
- **States:**
  - Position of robot + dirt in A? + dirt in B?
  - Initial state: (A, no, yes)
- **Actions:**
  - Left, right, suck, no-op
- **Goal:**
  - (-,no,no)
- **Cost:**

Example: Vacuum World

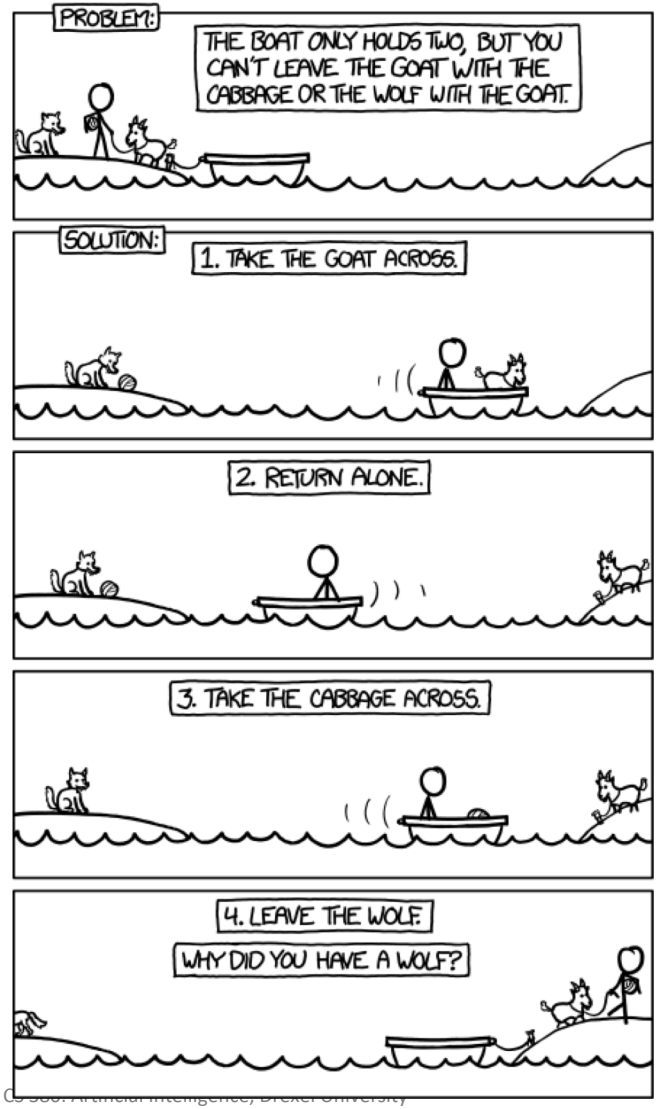


- **States:**
  - Position of robot + dirt in A? + dirt in B?
  - Initial state: (A, no, yes)
- **Actions:**
  - Left, right, suck
- **Goal:**
  - (-,no,no)
- **Cost:**
  - 1 per move, 1 per suck (??)

Example: Vacuum World

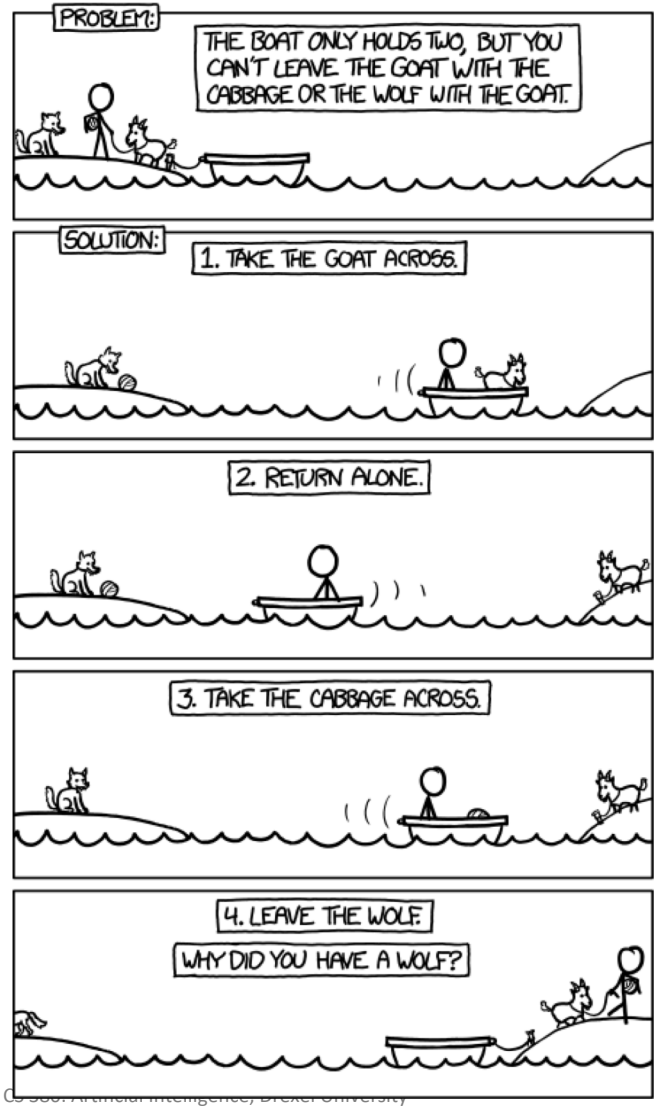


Example: Wolf, Goat, Cabbage



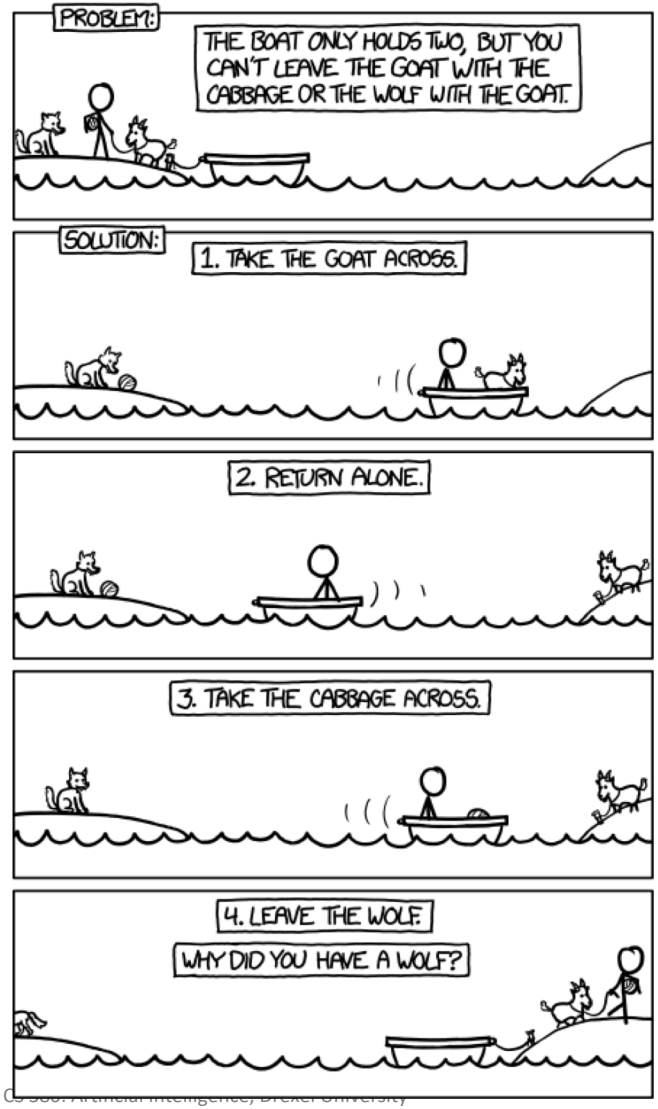
- States
- Actions
- Goal
- Cost

Example: Wolf, Goat, Cabbage



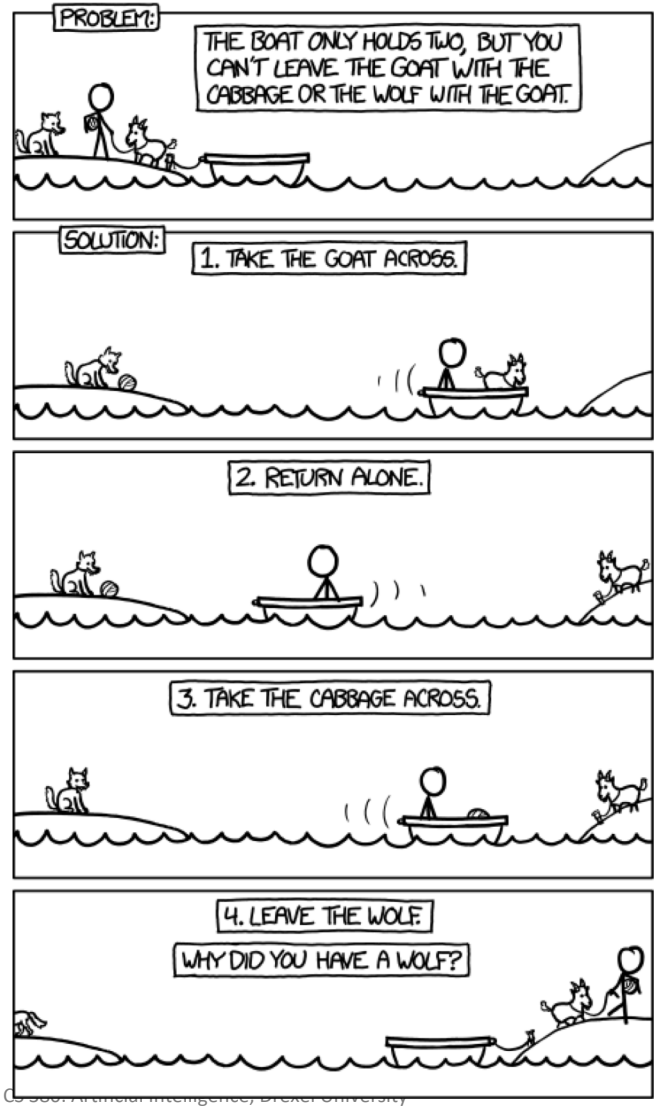
- States:
  - Location of wolf, goat, cabbage (left bank, boat, right bank) and boat (left, right)
  - Initial state: (left, left, left, left)
- Actions
- Goal
- Cost

Example: Wolf, Goat, Cabbage



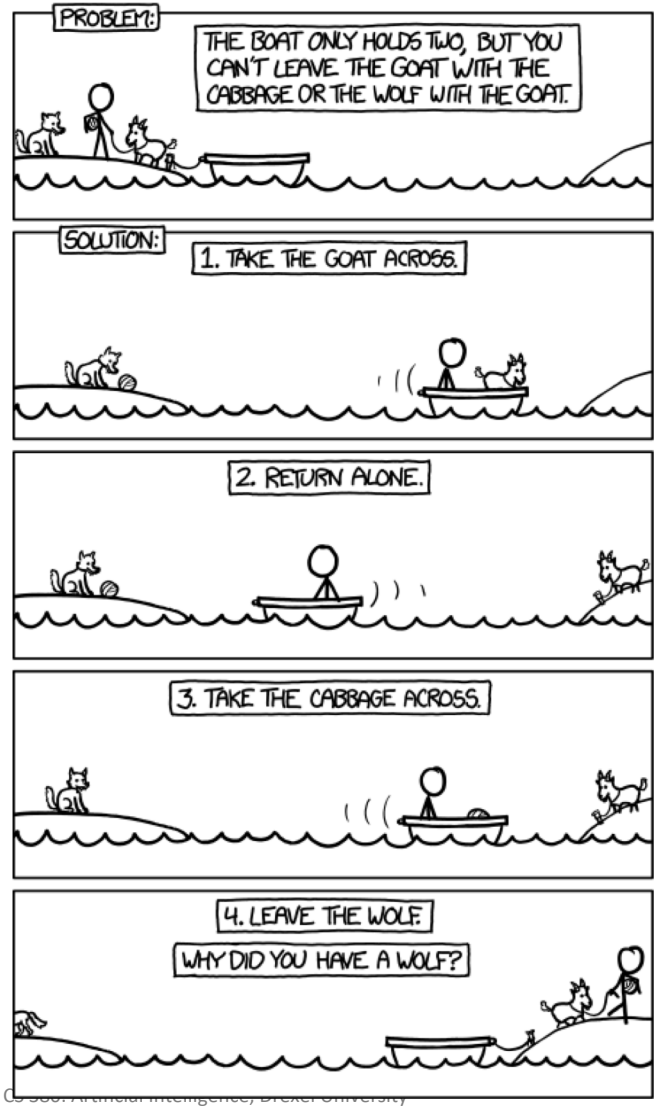
- States:
  - Location of wolf, goat, cabbage (left bank, boat, right bank) and boat (left, right)
  - Initial state: (left, left, left, left)
- Actions:
  - Boat left, boat right
  - Load wolf, unload wolf
  - Load goat, unload goat
  - Load cabbage, unload cabbage
  - (never leaving wolf with goat, goat with cabbage alone)
- Goal:
- Cost:

Example: Wolf, Goat, Cabbage



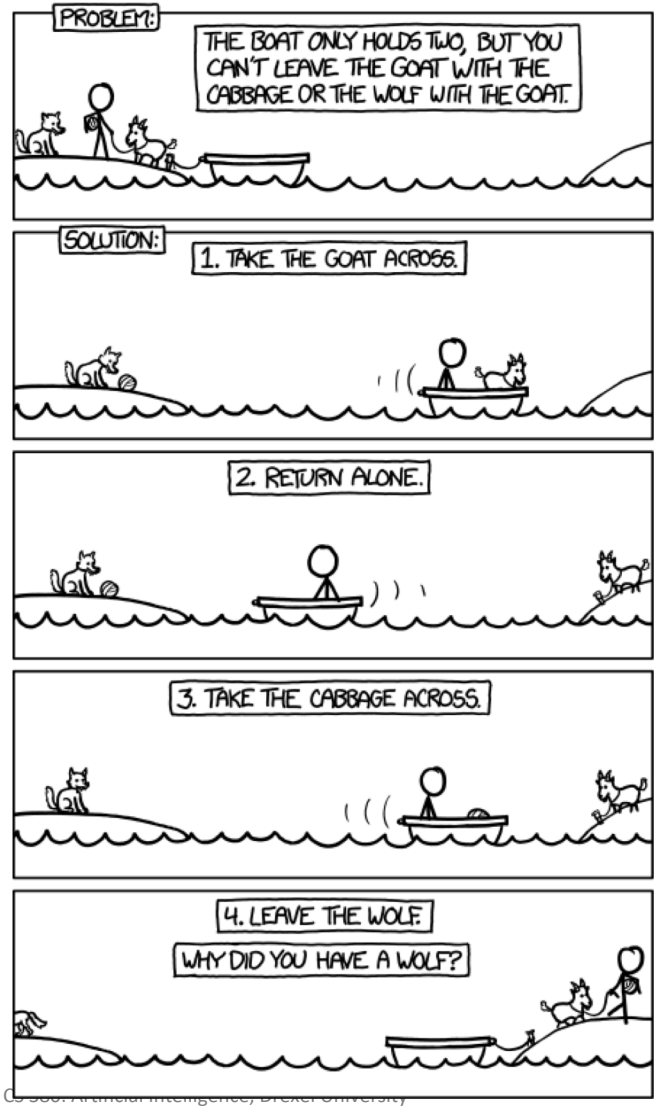
- States:
  - Location of wolf, goat, cabbage (left bank, boat, right bank) and boat (left, right)
  - Initial state: (left, left, left, left)
- Actions:
  - Boat left, boat right
  - Load wolf, unload wolf
  - Load goat, unload goat
  - Load cabbage, unload cabbage
  - (never leaving wolf with goat, goat with cabbage alone)
- Goal:
  - (right, right, right, right)
- Cost:

Example: Wolf, Goat, Cabbage



- States:
  - Location of wolf, goat, cabbage (*left bank, boat, right bank*) and boat (*left, right*)
  - Initial state: (left, left, left, left)
- Actions:
  - Boat left, boat right
  - Load wolf, unload wolf
  - Load goat, unload goat
  - Load cabbage, unload cabbage
  - (never leaving wolf with goat, goat with cabbage alone)
- Goal:
  - (right, right, right, right)
- Cost:
  - 1 per action

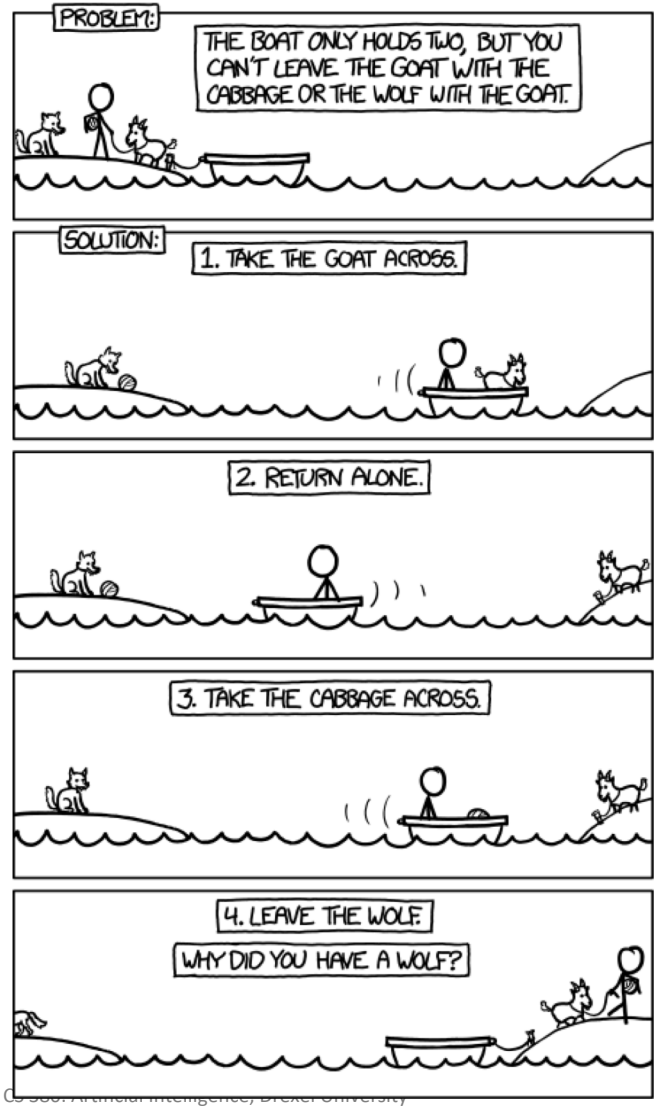
Example: Wolf, Goat, Cabbage



- States:
  - Locati… bank, … right)
  - Initial …
- Actions:
  - Boat left, boat right
  - Load wolf, unload wolf
  - Load goat, unload goat
  - Load cabbage, unload cabbage
  - (never leaving wolf with goat, goat with cabbage alone)
- Goal:
  - (right, right, right, right)
- Cost:
  - 1 per action

What's the size of the state space?

Example: Wolf, Goat, Cabbage



- States:
  - Locati
    bank,
    right)
  - Initial
- Actions:
  - Boat l
  - Load w
  - Load g
  - Load cabbage, unload cabbage
  - (never leaving wolf with goat, goat with cabbage alone)
- Goal:
  - (right, right, right, right)
- Cost:
  - 1 per action

What's the size of the state space?

At most 3*3*3*2 = 54
(though less when we apply the action constraints)

More Examples

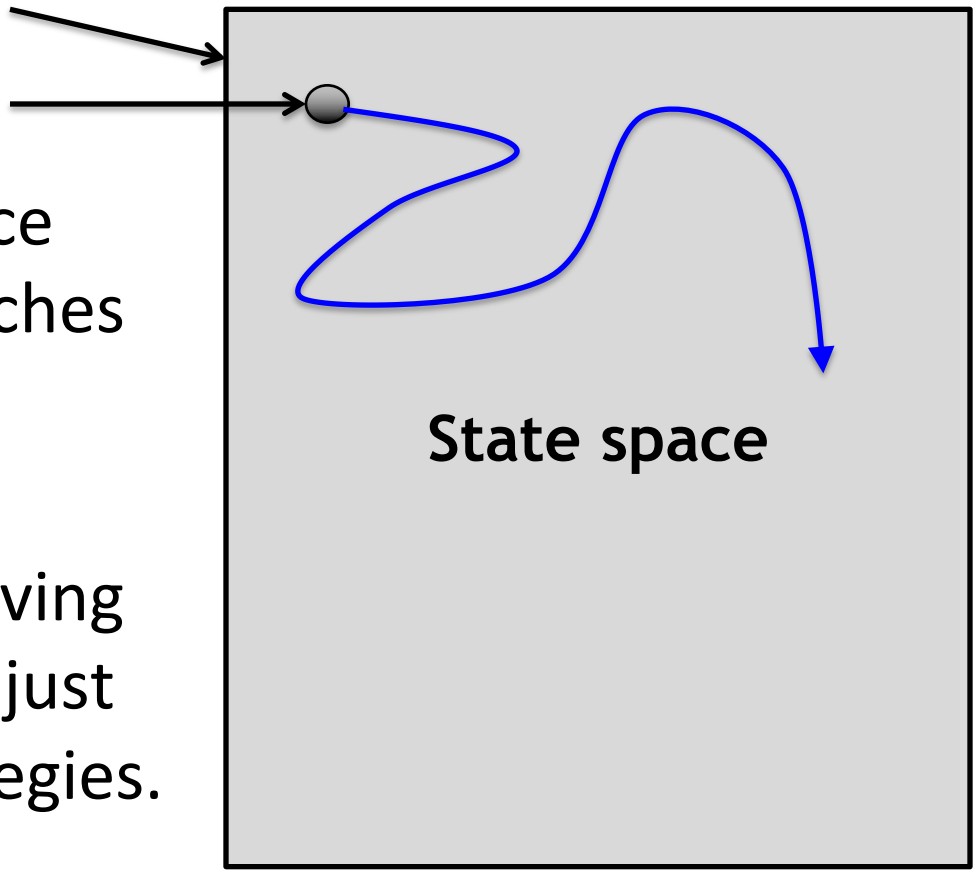
- Search Problems are far more general that they might seem
- For example, we can formulate any of the following as a search problem:
  - Proving a theorem
  - Solving a physics problem
  - Finding your way around a city (path finding)
  - Playing individual sports (multiplayer sports and games belong to another category of problems, we will discuss in a few weeks).
  - Playing a computer game
  - “Where are my keys?!?”
  - Etc.

Solving Problems

- Let us only consider deterministic, fully-observable problems
- Simple algorithm:
  - Enumerate every possible sequence of actions:
    - [left]
    - [right]
    - [advance]
    - [left,left]
    - [left,right]
    - [left,advance]
    - …
  - Try one by one, until one reaches the solution

Solving Problems

- General idea:
  - Given the state space
  - Given the initial state
- Explore the state space until a state that matches the goal condition is found.
- Different problem solving methods are actually just different search strategies.



Random Walk

- Algorithm:
  - Initial state: S
  - (1) Determine set of applicable actions *A* = validActions(*S*)
  - (2) Choose an action *a* in *A* at random
  - (3) Apply a to the current state: *S* = apply(*S*,*a*)
  - (4) If we have not reached the goal (G(*S*) == false), go to (1)
  - (5) Done

- Is this algorithm…
  - guaranteed to find a solution?
  - guaranteed to find the optimal solution? (minimal cost)

Random Walk

- Algorithm:
  - Initial state: S
  - (1) Determine set of applicable actions *A* = validActions(*S*)
  - (2) Choose an action *a* in *A* at random
  - (3) Apply a to the current state: *S* = apply(*S*,*a*)
  - (4) If we have not reached the goal (G(*S*) == false), go to (1)
  - (5) Done

- This algorithm…
  - Is not guaranteed to find a solution
  - Is not guaranteed to find the optimal solution (minimal cost)

Search

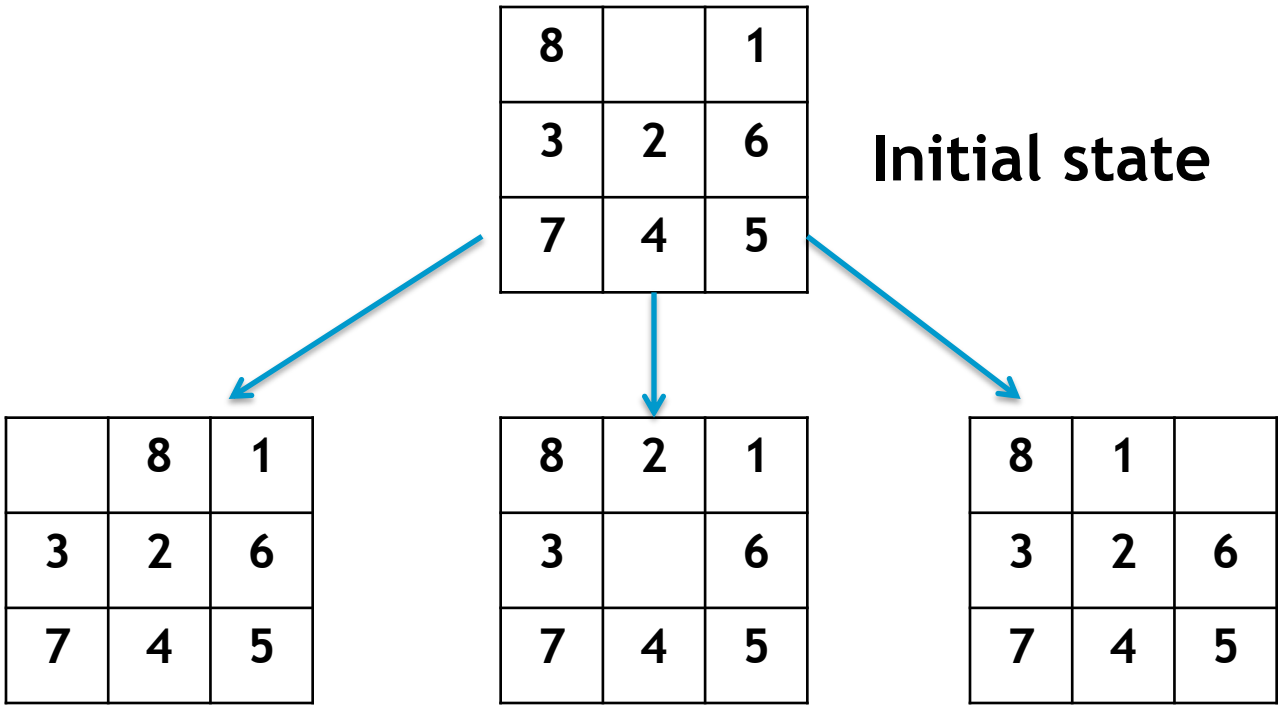
- Summary:
  - **Initial state**: initial configuration of the problem
  - **Actions**: set of actions the agent can perform to solve the problem
  - **State space**: set of possible states that are reachable form the initial space by any sequence of actions
- **Search**
  - Family of strategies to find solutions
  - Based on systematically exploring the state space until finding a sequence of actions that achieves the goal

Illustration

8		1
3	2	6
7	4	5

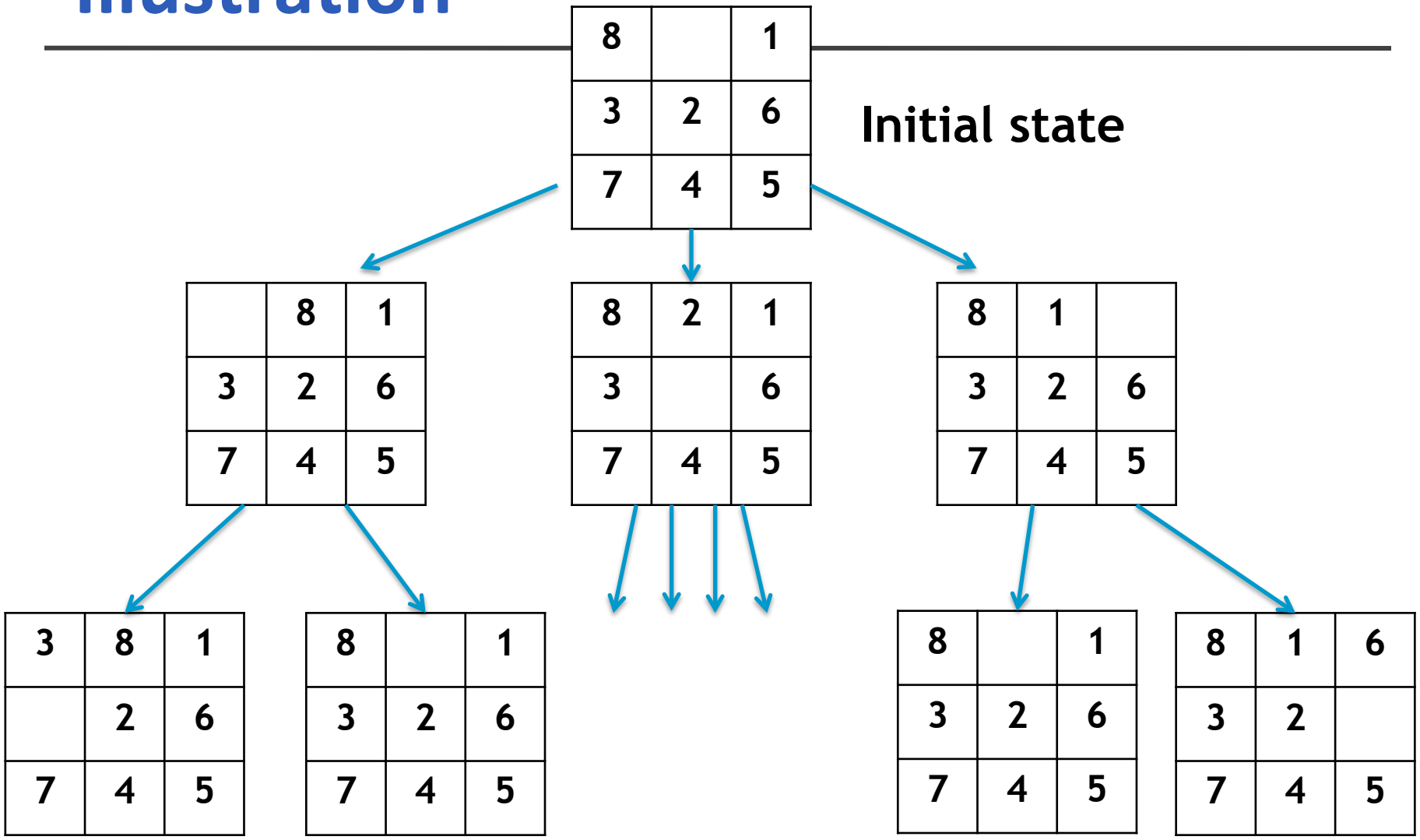
Initial state

Illustration



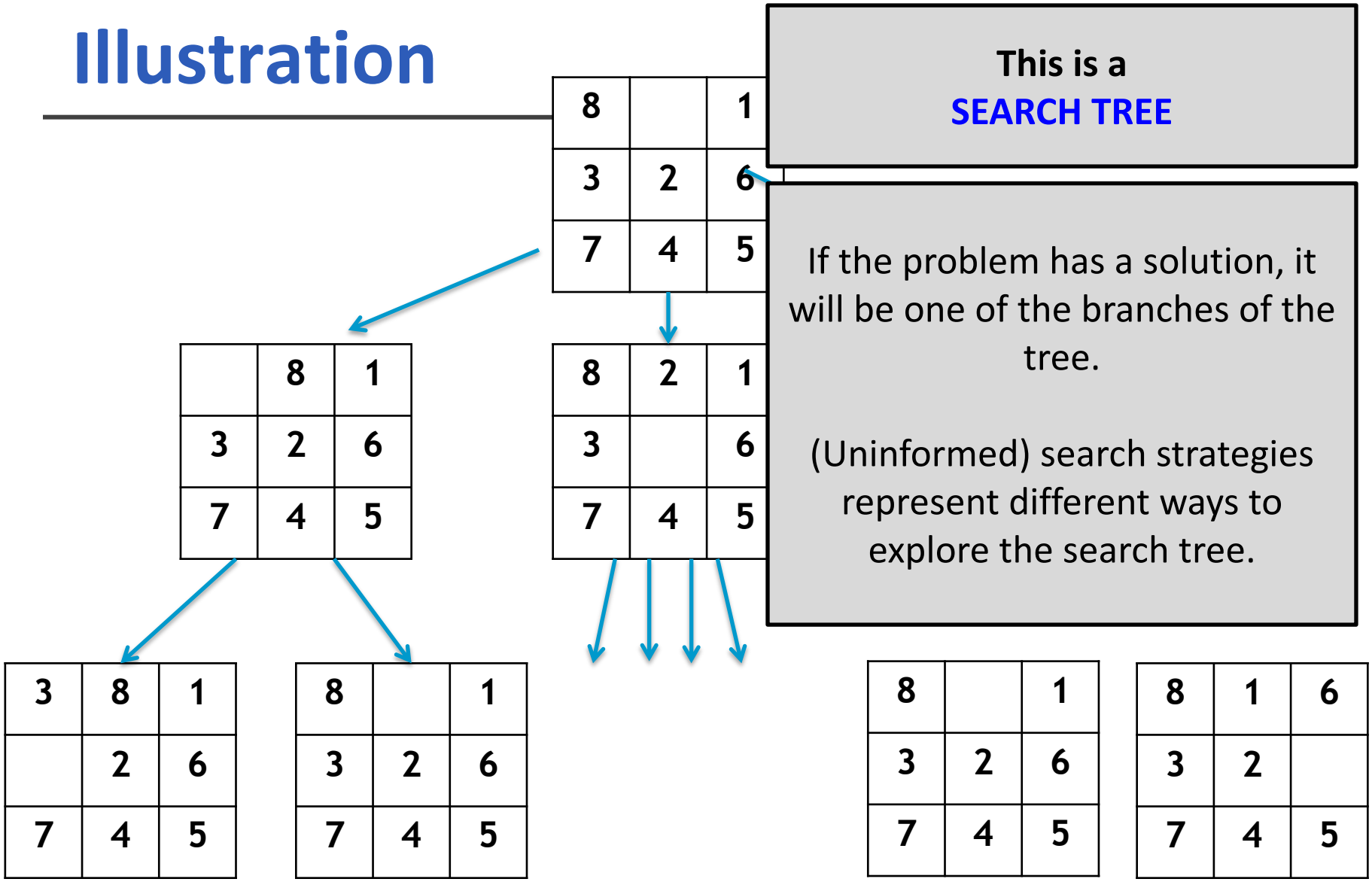
Set of states reachable by executing only one action

Illustration



Set of states reachable by executing only two actions

Illustration



Set of states reachable by executing only two actions

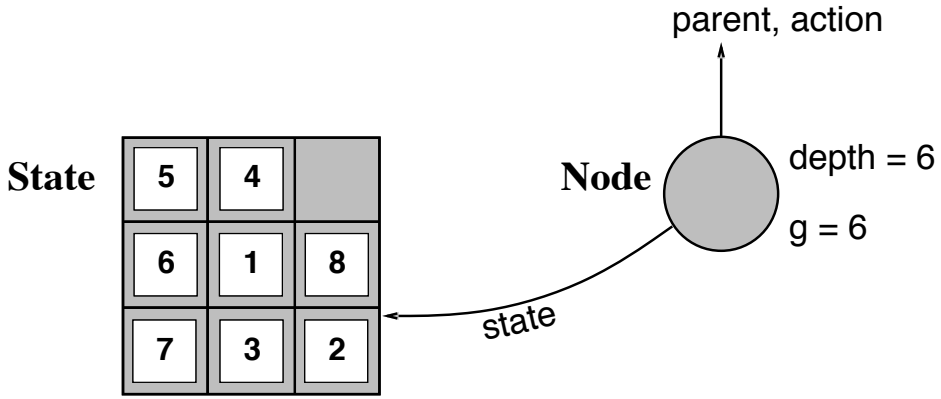
Implementation: states vs. nodes

A state is a (representation of) a physical configuration

A node is a data structure constituting part of a search tree

includes parent, children, depth, path cost $g(x)$

States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

Tree search algorithms

Basic idea:

offline, simulated exploration of state space

by generating successors of already-explored states

(a.k.a. expanding states)

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

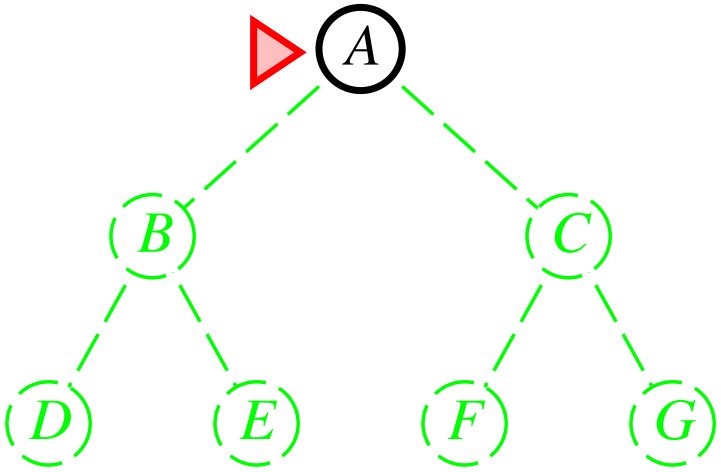
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

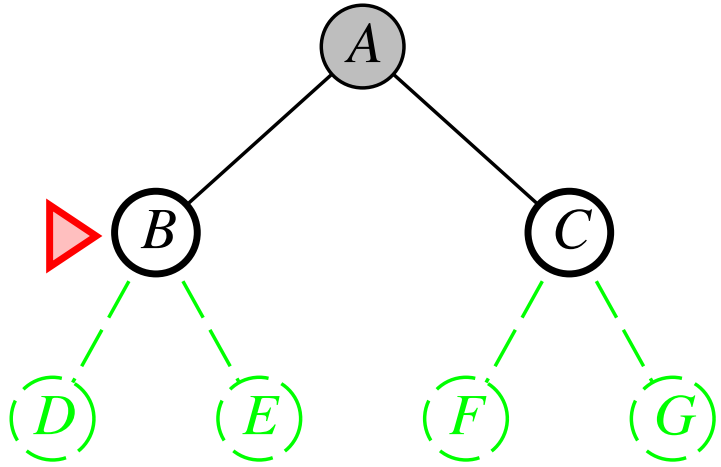


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

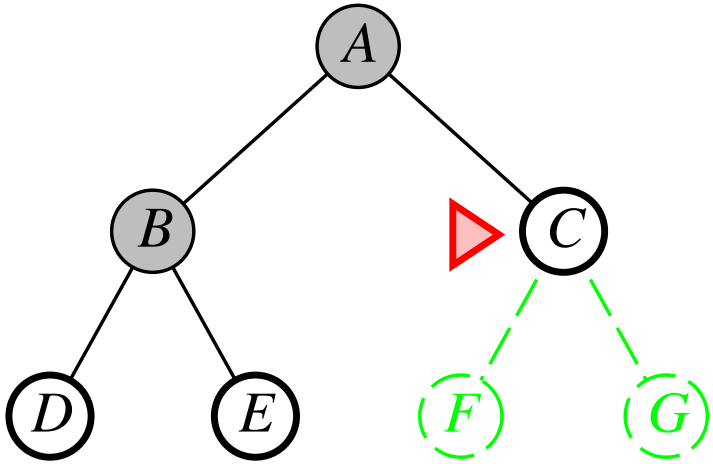


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

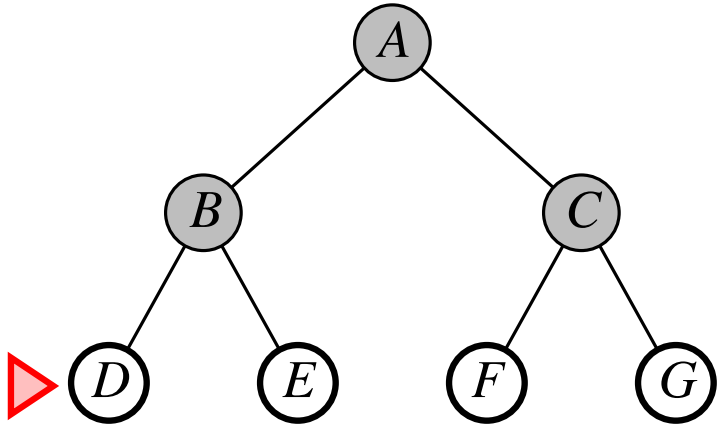


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

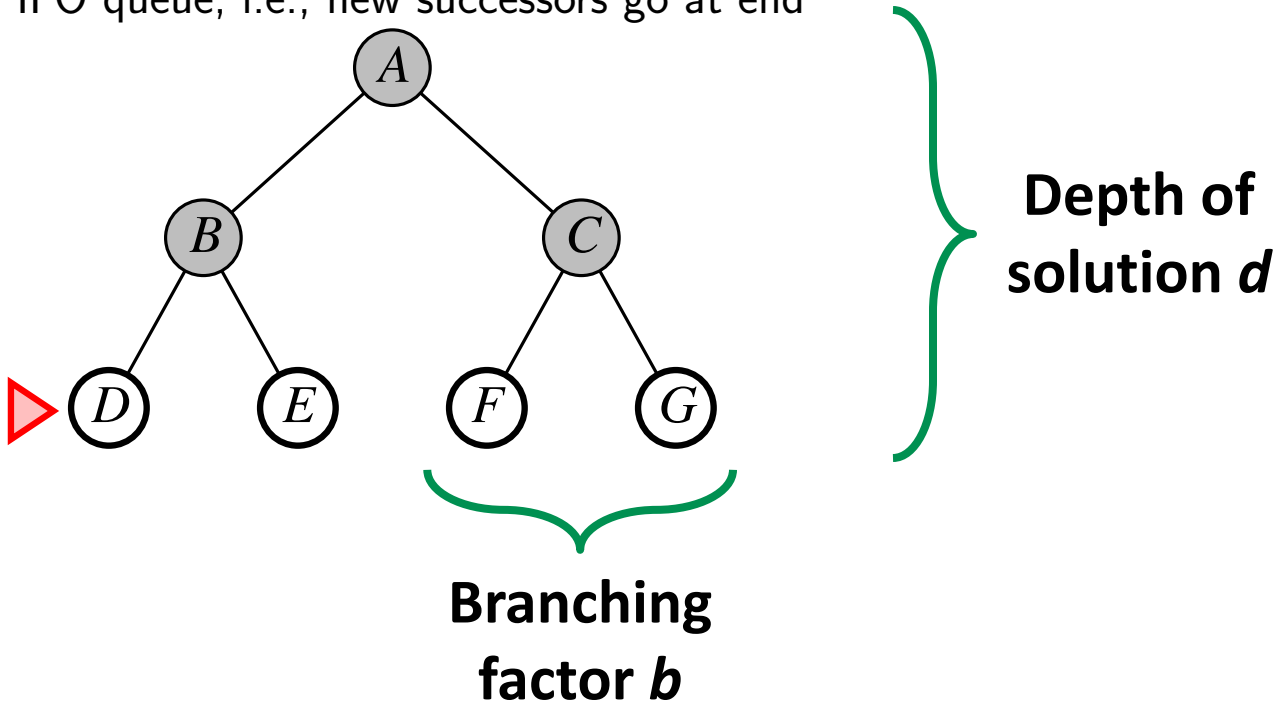


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs $=$ 8640GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

$fringe$ = queue ordered by path cost, lowest first

with path cost function g

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

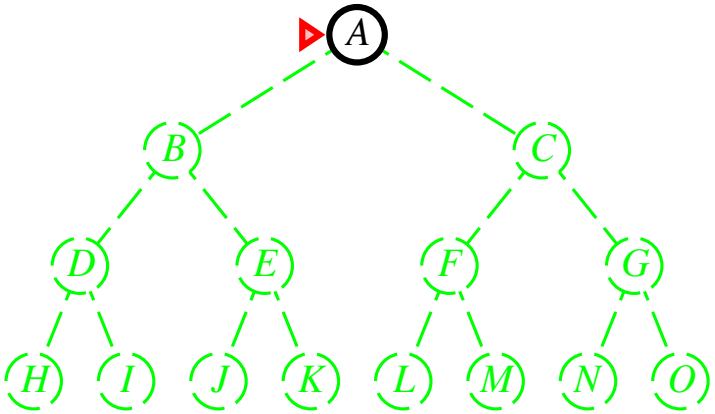
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

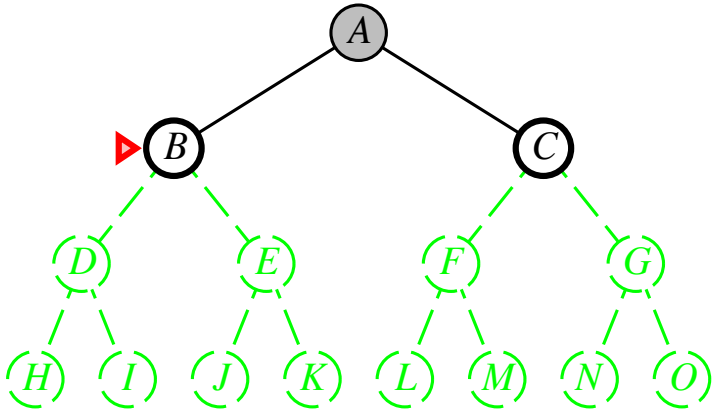


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

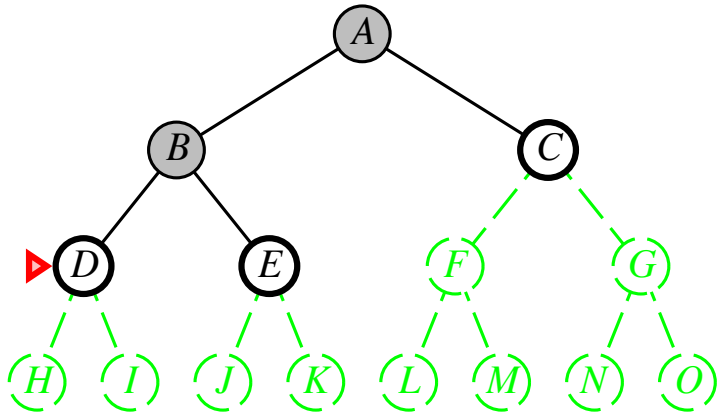


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

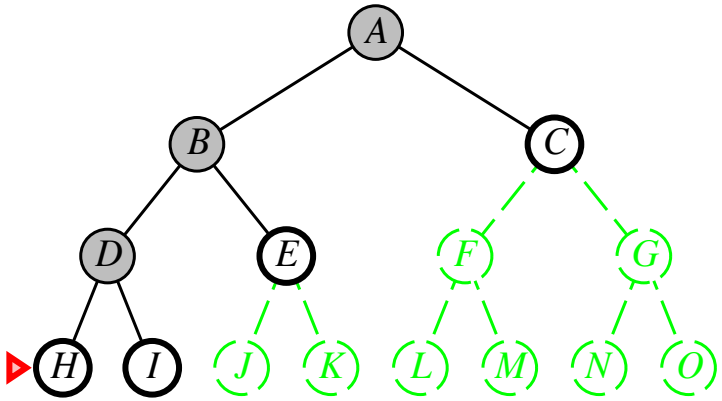


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

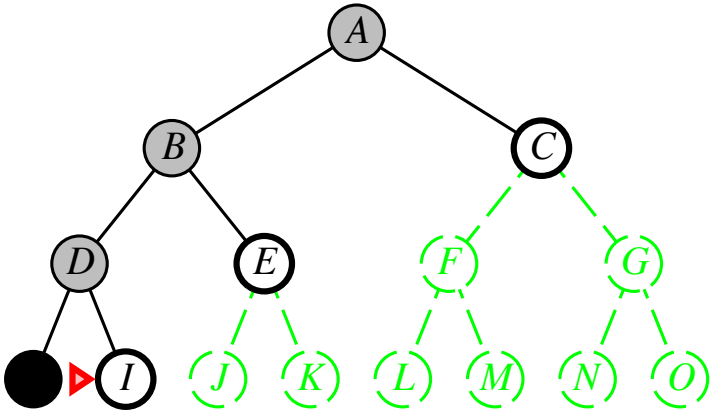


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

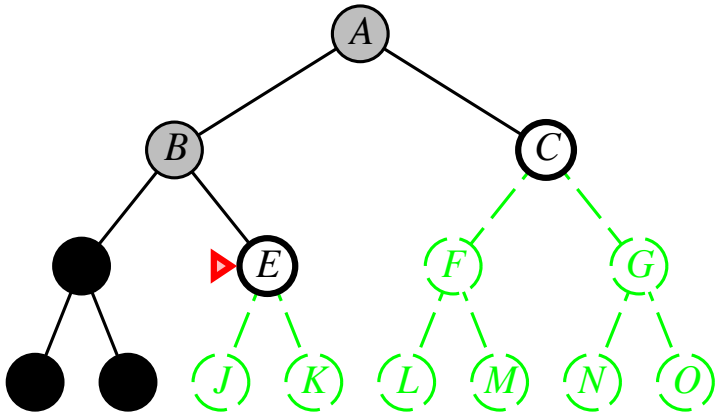


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

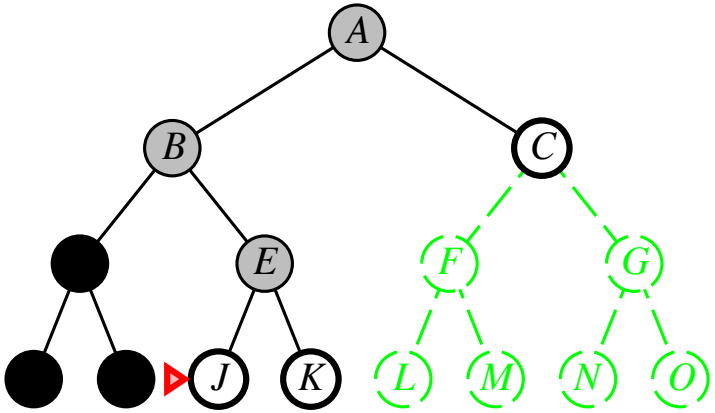


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

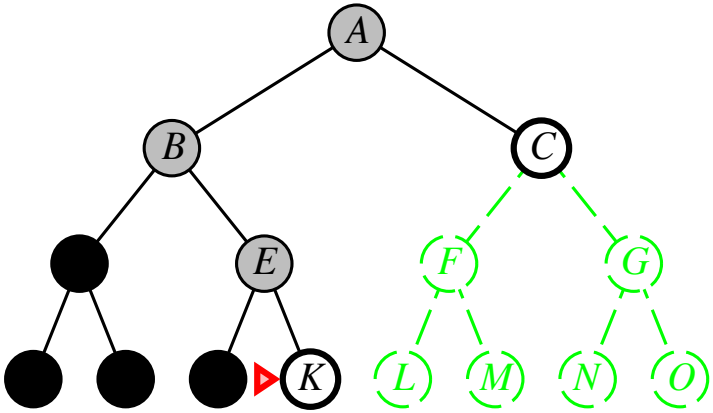


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

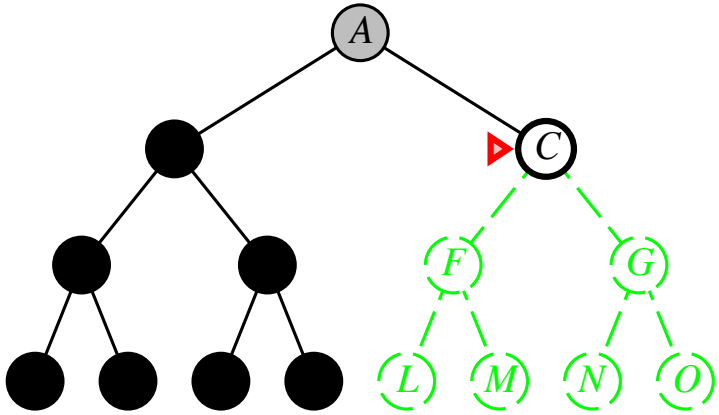


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

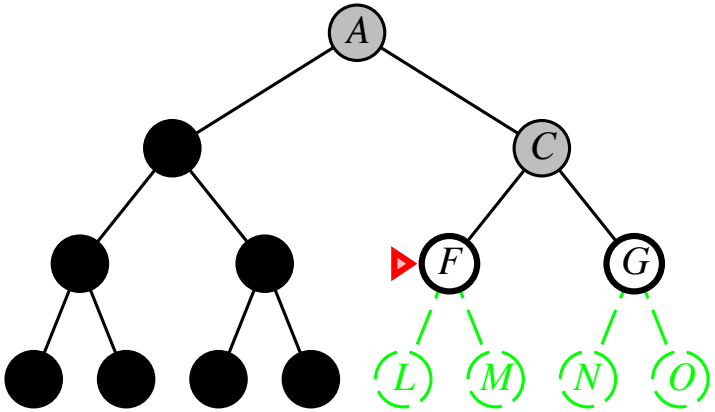


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

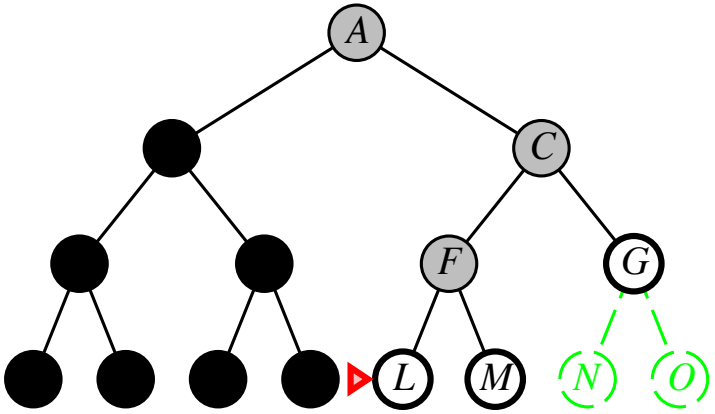


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

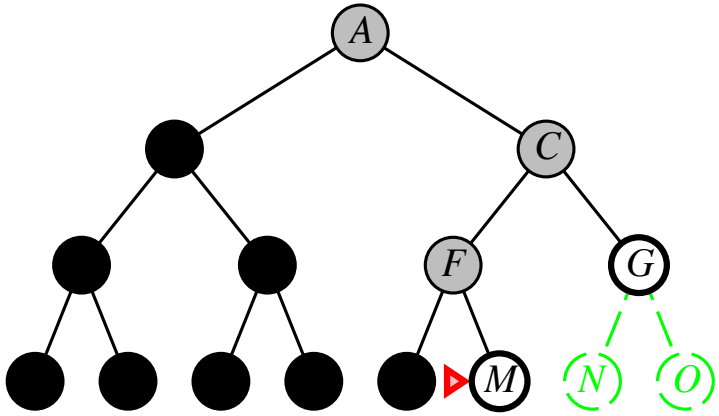


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



Properties of depth-first search

Complete??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time??

Consider

m = maximum depth

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space??

m = maximum depth;
bad if *m* >> *d* !

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

(might be a good idea
if d is well understood)

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Iterative deepening search $l = 0$

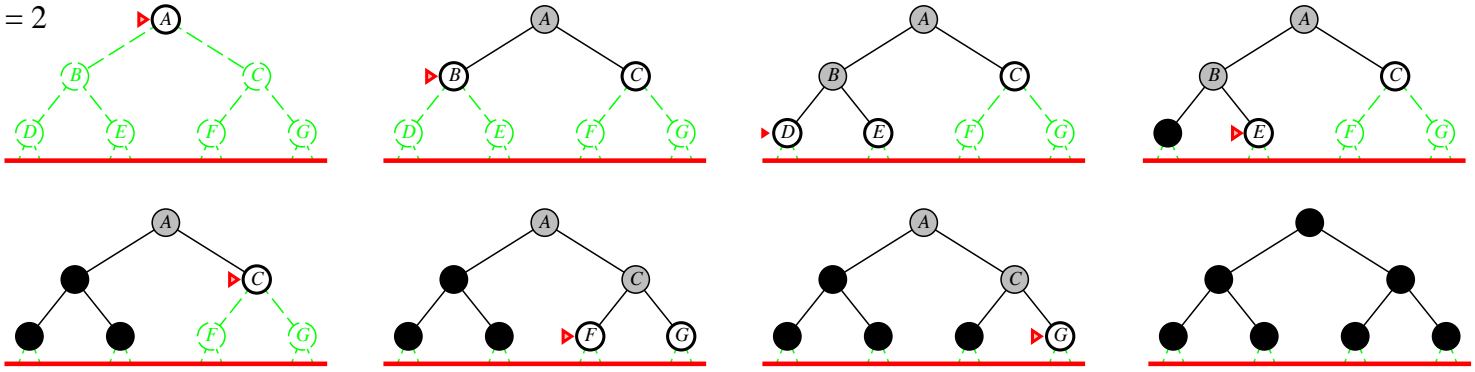
Limit = 0

Iterative deepening search $l = 1$

Limit = 1

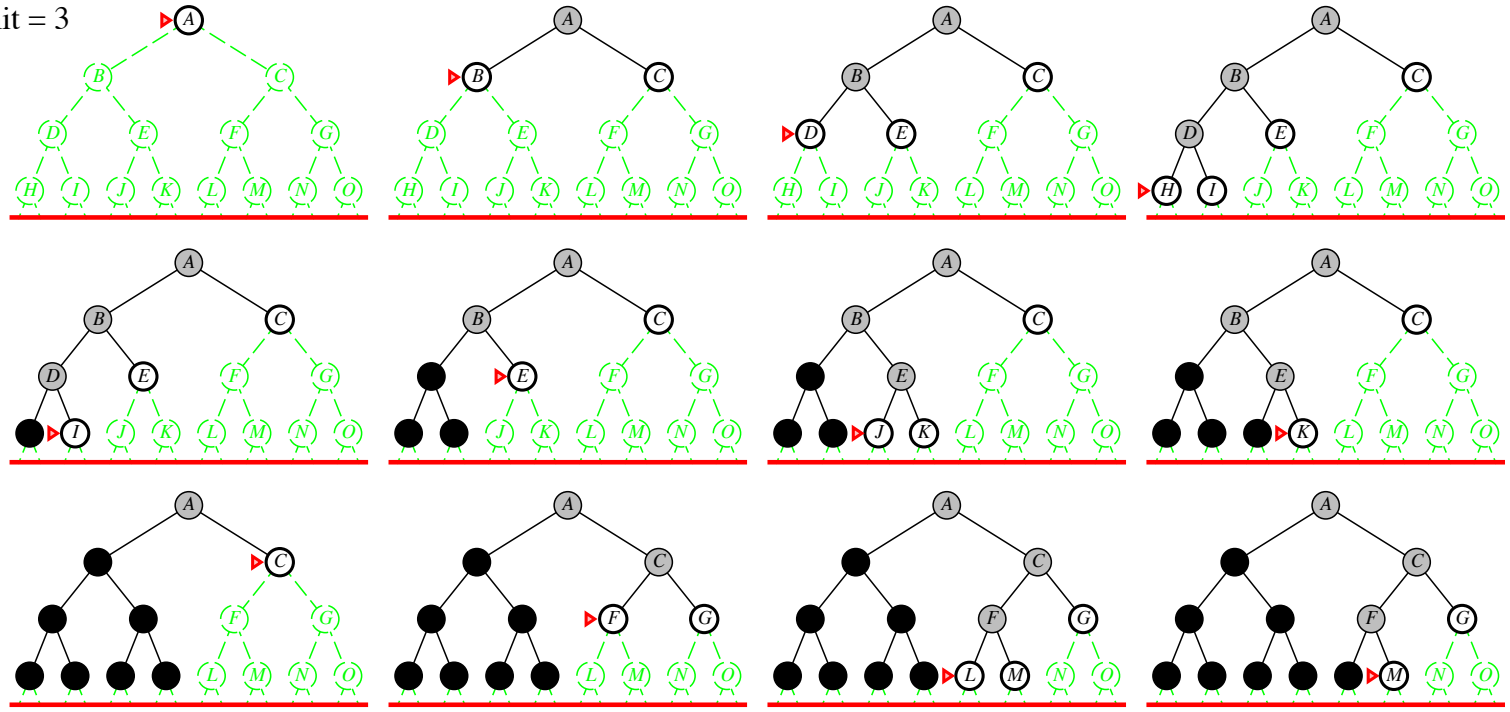
Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$

Limit = 3



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is **generated**

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Coming up…

- Informed search strategies