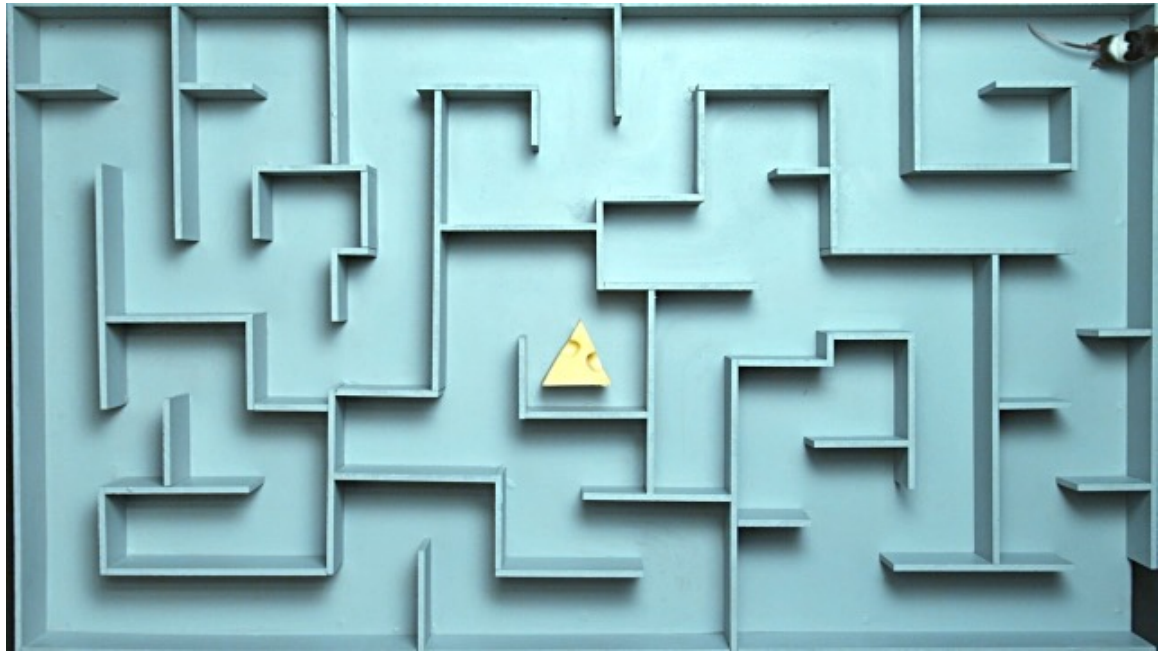
COMP 341: Introduction to AI
Uninformed Search



Asst. Prof. Barış Akgün

Koç University

Recap

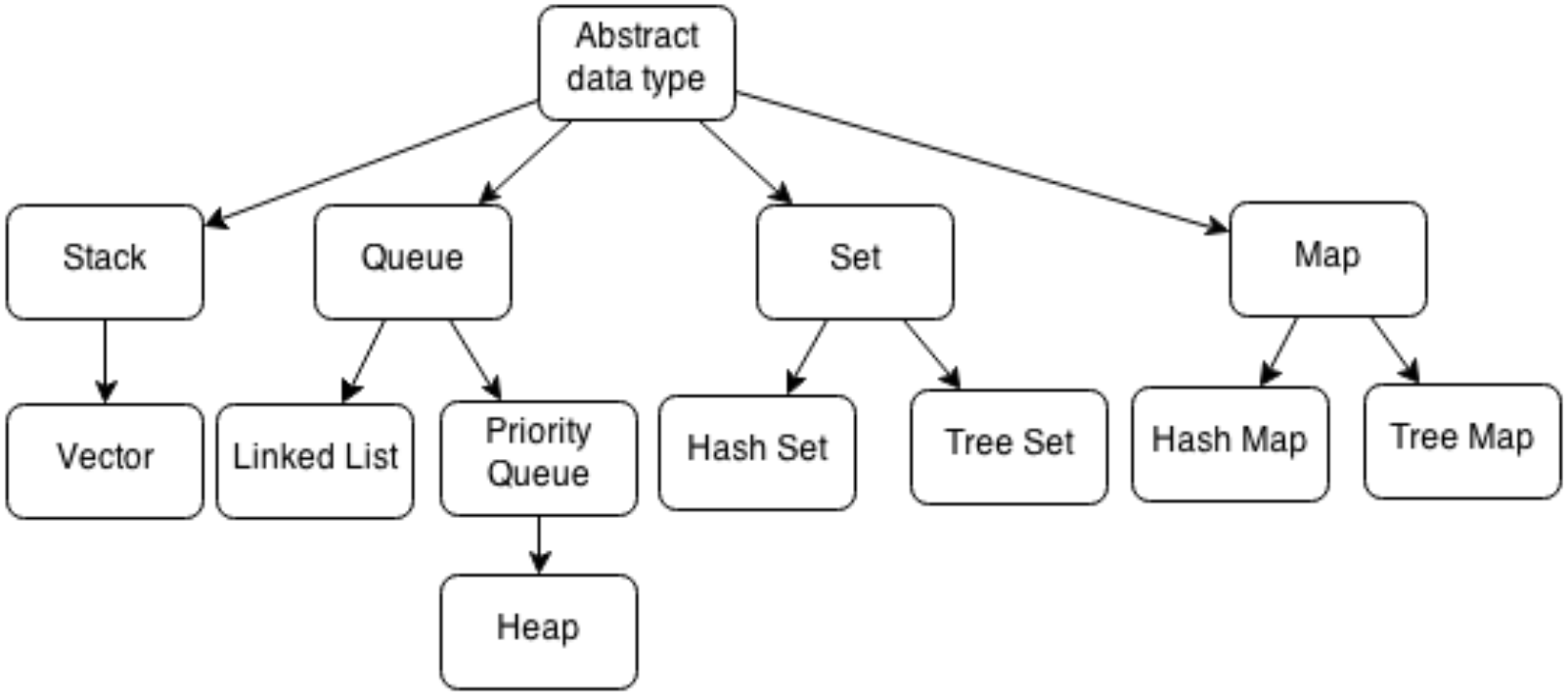
- AI working definition
- Agents
- Rationality
- Agent types (reflex vs planning)
- Problem types

- Selecting an action based on (history of) percepts.
- Solutions based on problem types

- Today: Starting “search”

Detour: Data Structures for Search

"We need to formulate our problem to solve it"



What are data structures?

- Almost all the problems we try to solve involve some sort of information – **data**
- Data structure: A particular way of organizing data so that it can be used efficiently

- Can you think of any examples?
  - Phone contacts
  - Computer storage (file systems)
  - Web search results
  - …

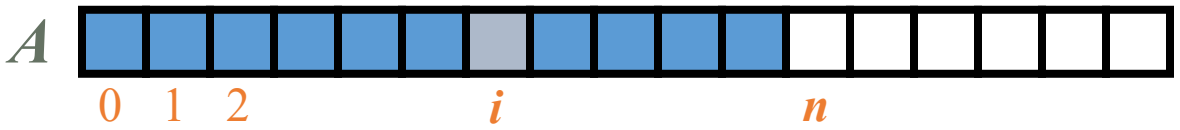
Data Structures and AI

- AI problems are hard, we want the best organization for our data to make our lives easier!

- What could be our data?
  - Sensory info
  - Current State
  - A collection of states (planning)
  - Information about states (e.g. if a state is good or bad)
  - Etc.

Arrays

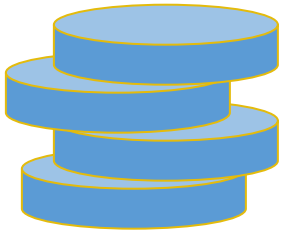
- A sequenced collection of variables, indexed using integers



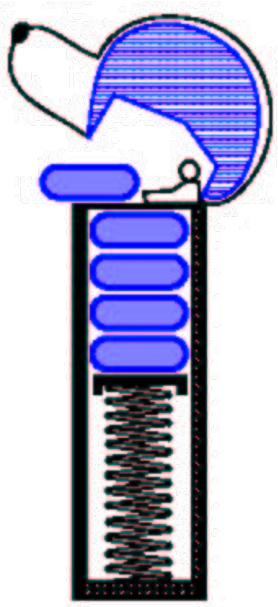
- The **elements** can be reached **randomly** i.e. given an index, we can directly reach that element
- Elements can be anything: integers, real valued numbers, strings (i.e. words), other arrays etc.

Stacks

- A collection of variables
- Accessed as last-in first-out fashion: “Pez dispenser”



- Two operations:
  - Push: Insert an element
  - Pop: Removes and returns an element



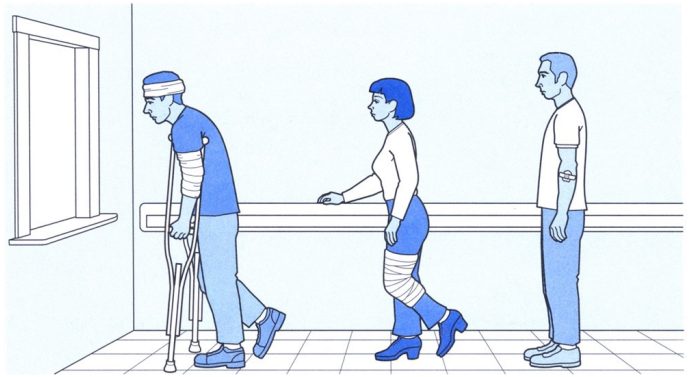
Queues

- A collection of variables
- Accessed as first-in first-out fashion

- Two operations:
  - Push: Insert an element
  - Pop: Removes and returns an element

Priority Queues

- A collection of variables
- Each variable has an associated “priority”
- Variables are accessed by increasing priority

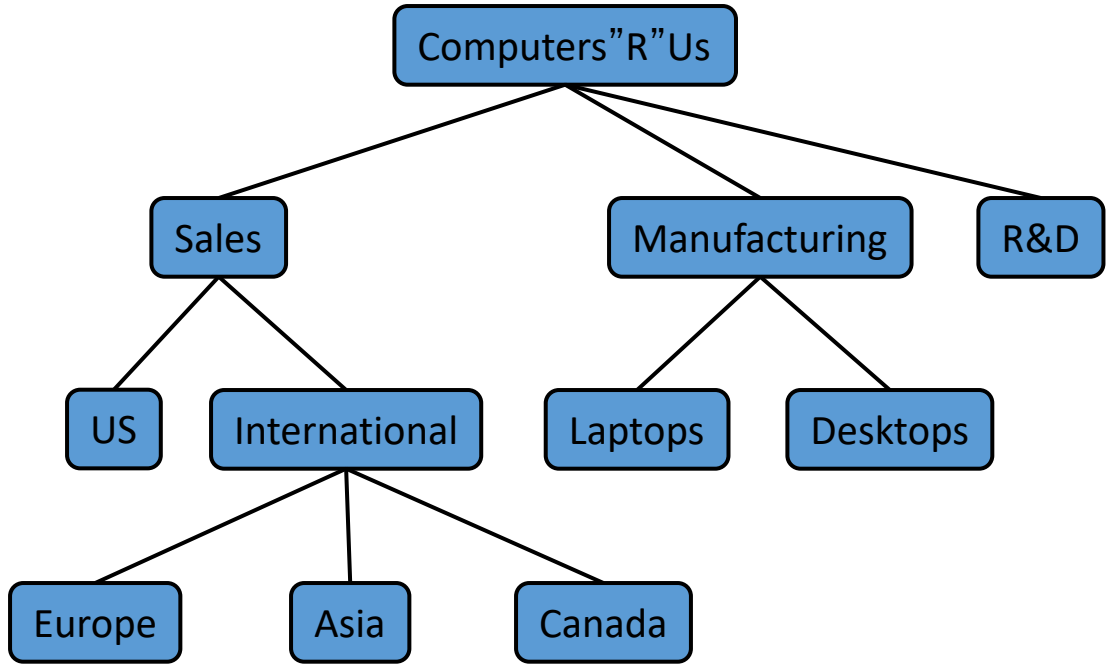


- Two operations:
  - Push: Insert an element with value
  - Pop: Removes and returns the element with the minimum value

Stacks and Simple Queues are a special case of priority queues

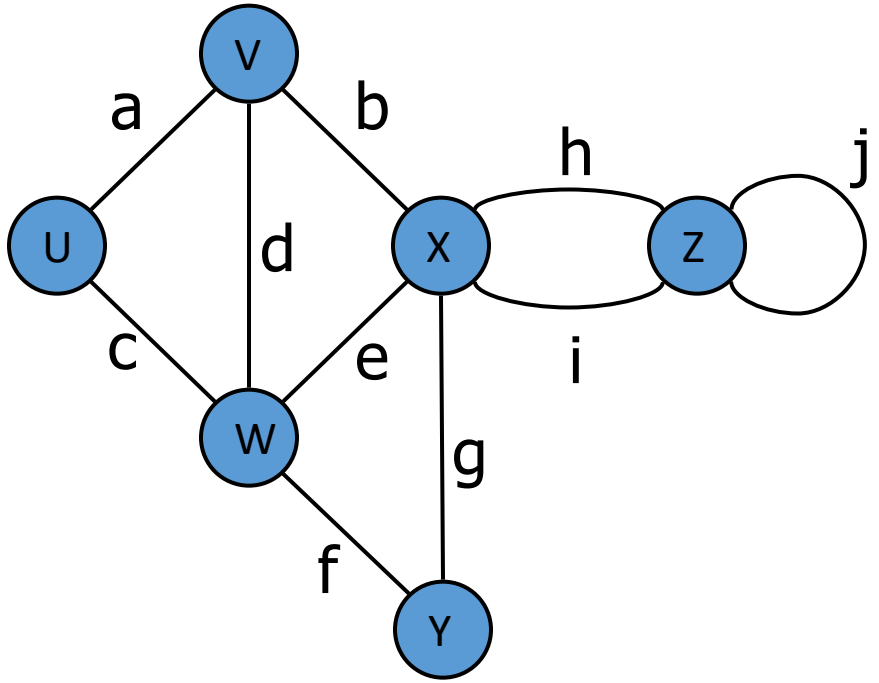
Trees

- An “abstraction” of a **hierarchical** structure
- Parent-child relations
- Each element is called a **node/vertex**
- Each connection is called an **edge**
- Encodes hierarchy

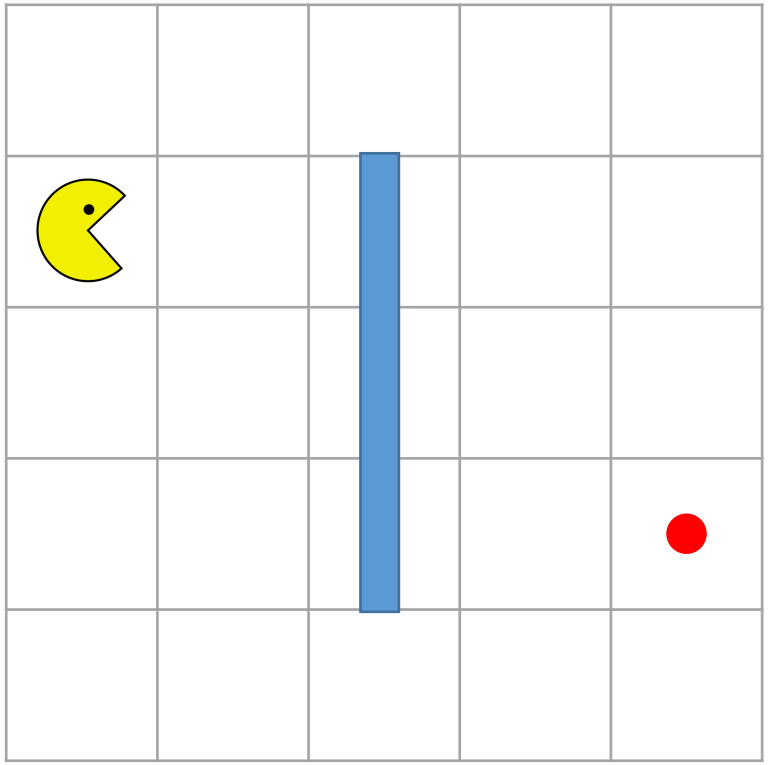


Graphs

- Encodes relations
- Extension of trees: A set of nodes/vertices and edges
- Used in many applications

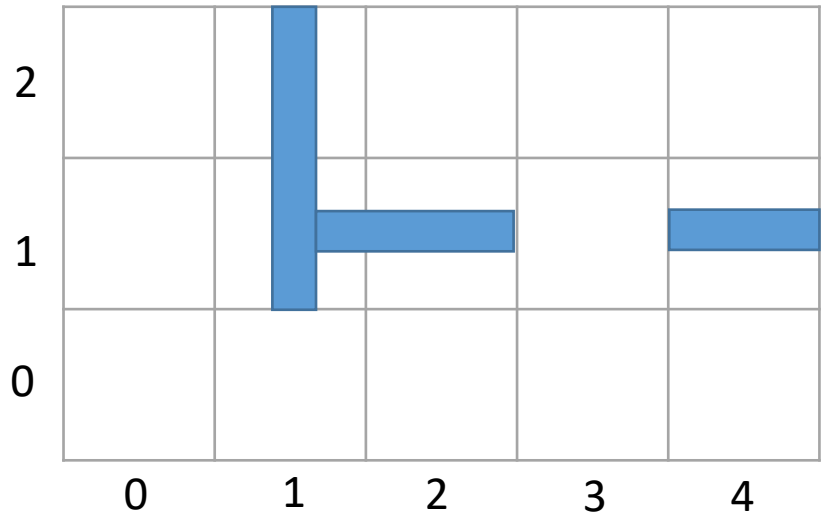


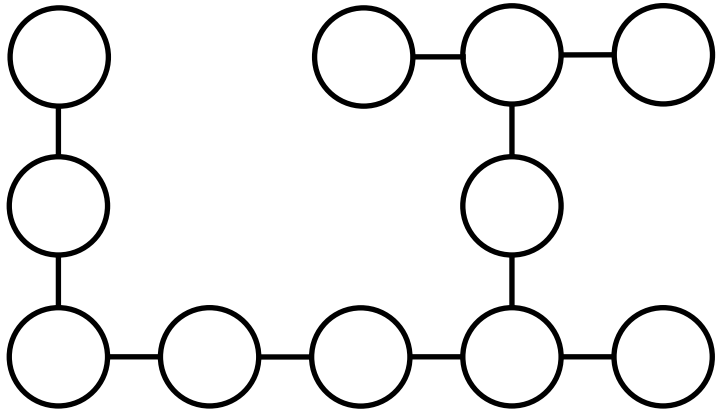
Finding Paths in Mazes



How can the Pacman find out the path to the food?

Representing “Mazes” as Graphs

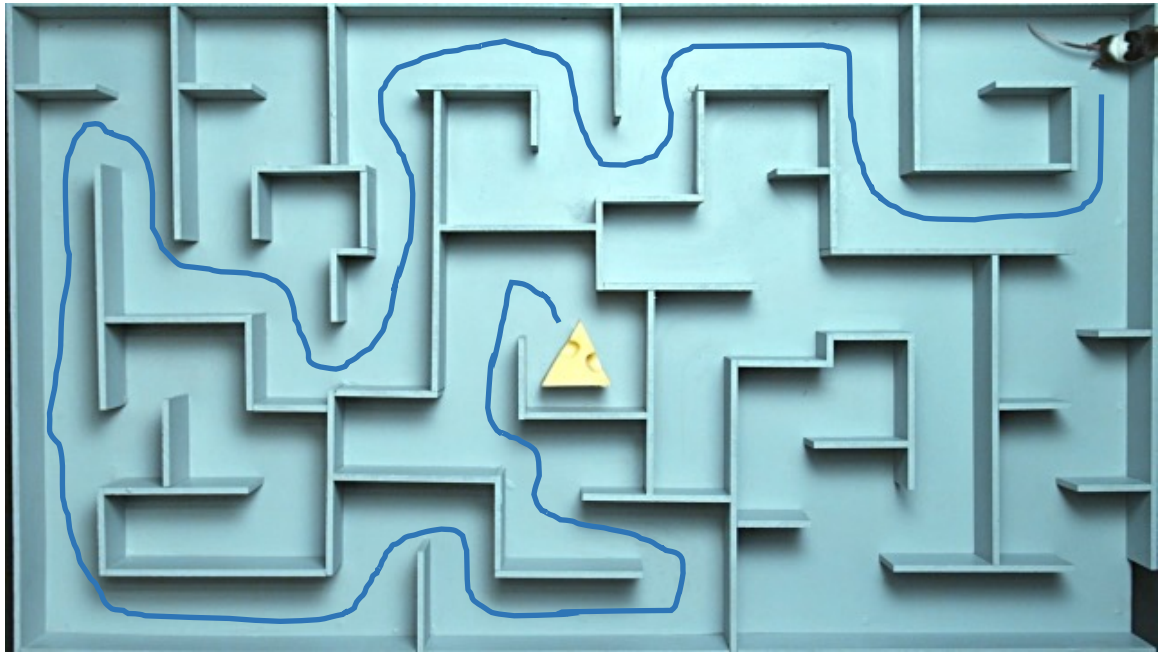




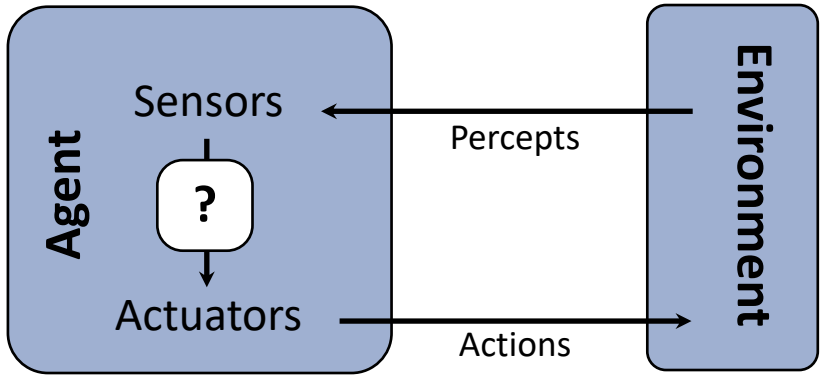
- Each possible location on the gird is represented by a node
- 4-way connectivity edge

Note that finding a path within a Pacman labyrinth has a different state space than the Pacman game itself! – Why?

Search Problems – Planning Agents



Problem Formulation

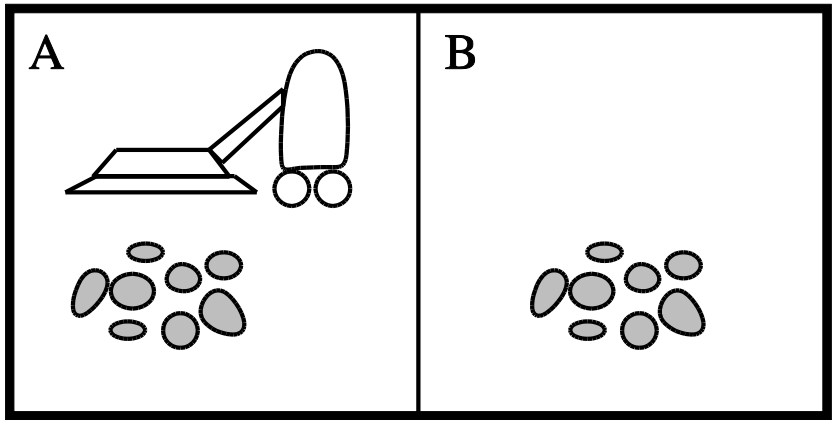


- Problem:
  - A state space: S
  - An action space: A
  - Transition Model/Successor Function: (S x A)→ S
  - Cost of taking an action (scalar): (S x A) → c
  - A start state: s_0
  - Goal Test: g(s) → {true, false}
- Solution:
  - A sequence of actions that transforms the start state to a goal state
  - Solution Cost: Sum of costs of each action along the solution

Assumptions for Now

- Fully Observable
- Static
- Discrete (both states and actions)
- Deterministic

Simple Vacuum Cleaner Robot

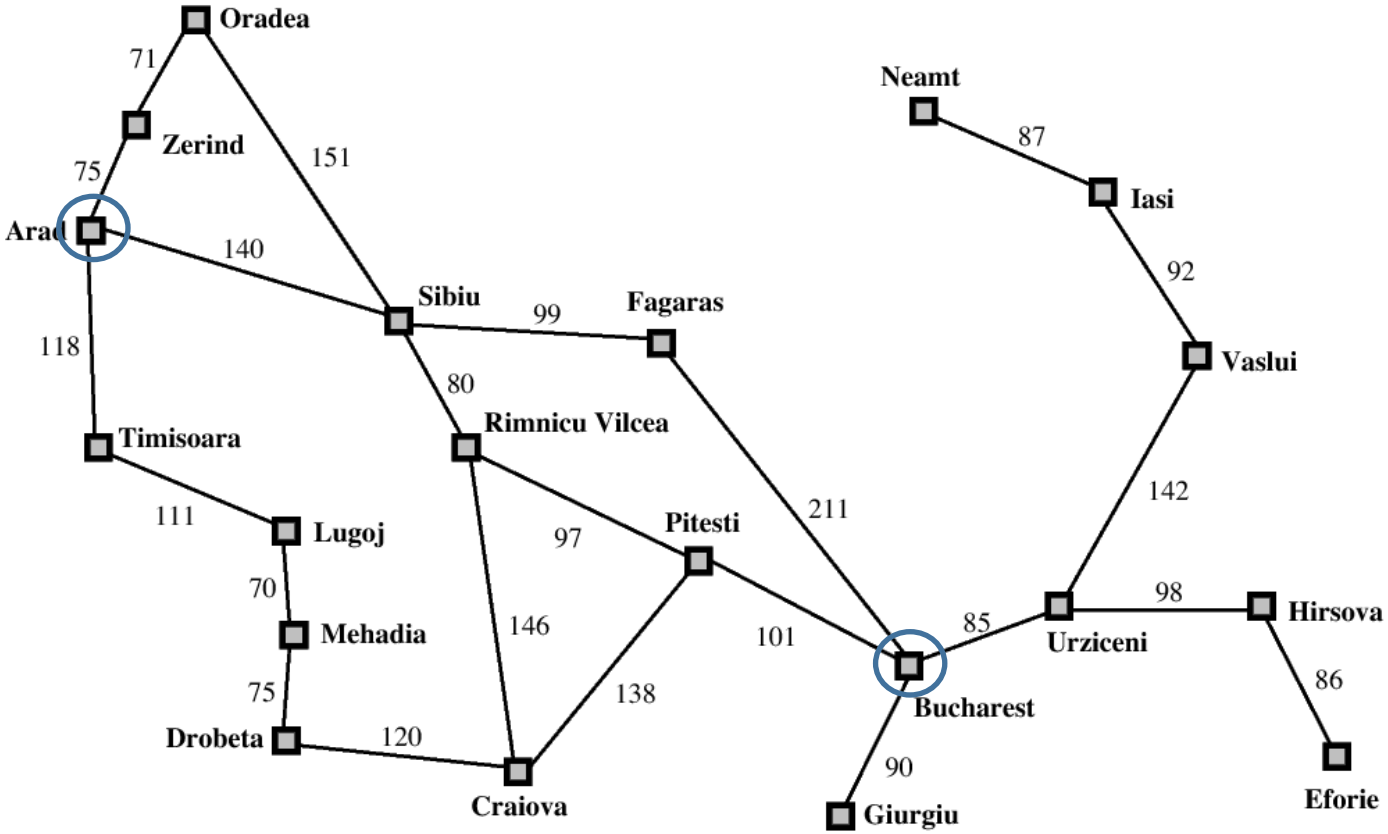


- State: <location, status>
- Action Set: {Move, Suck, NoOp}
- Transitions: Actions always succeed
- Costs
  - NoOp: 0
  - Right/Left: 1
  - Suck: 2
- Goal Test:
  - Is A&B clean?

- State space size: 4
- Action space size: 3

Solution?

Arad to Bucharest



Solution?

- State Space
  - Cities
- Action Space
  - Going to adjacent cities
- Transition Model
  - Changing the state to the city agent goes to
- Cost
  - Distance between cities
- Start State
  - Arad
- Goal Test
  - Current city == Bucharest

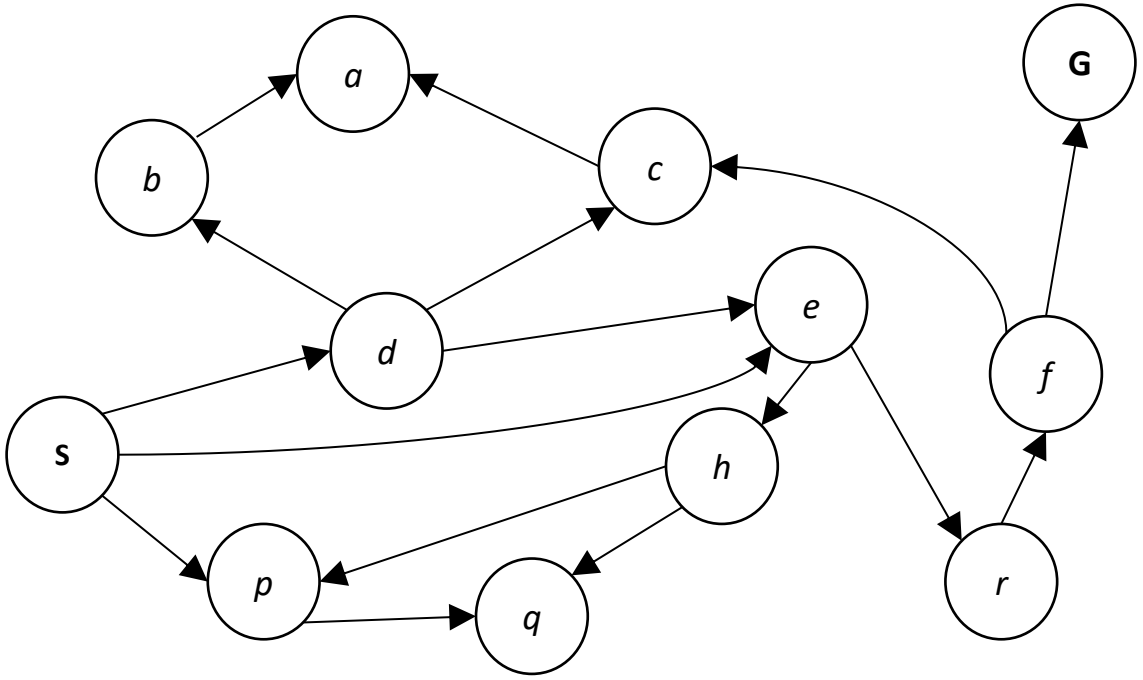
A bit more complex?

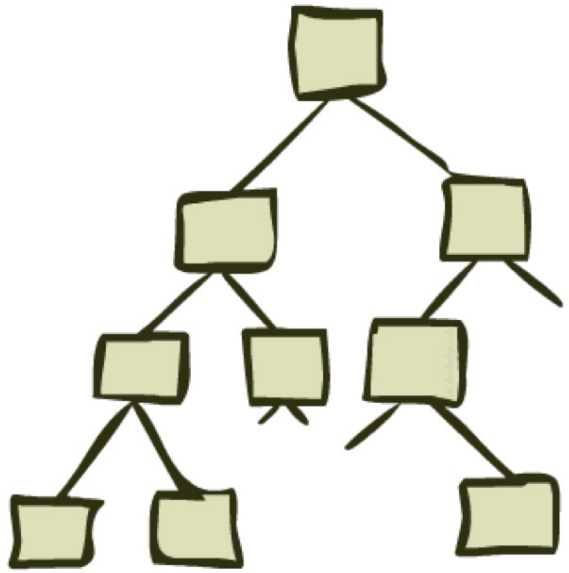
- State of a chessboard: 32 bytes are enough
- State Space Size Upper bound 2^{155}
- Memory needed?
  - Hint: 1TB ~ 2^{40}
- What about 19x19 Go?
  - State space size is around 2^{565}
  - More than the atoms in the visible universe!

- Clearly generating a table or a rule set is prohibitive

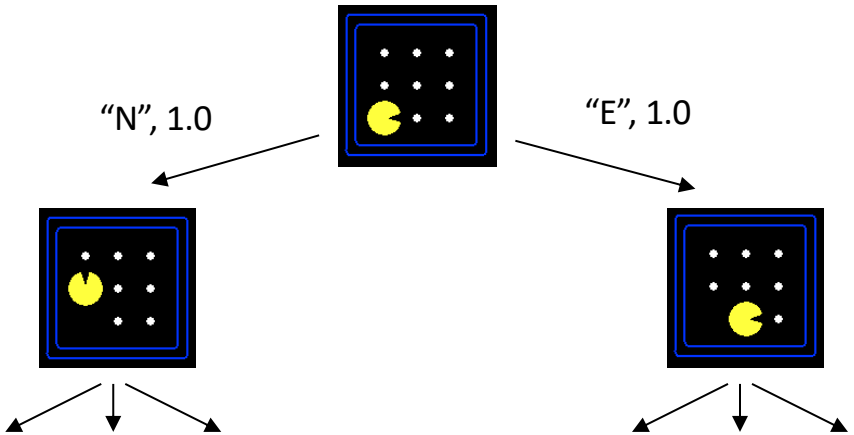
Let’s Search for the Solution!

- What are we searching on?
- State Space Graphs vs Search Trees





Search Trees



This is now / start

Possible futures

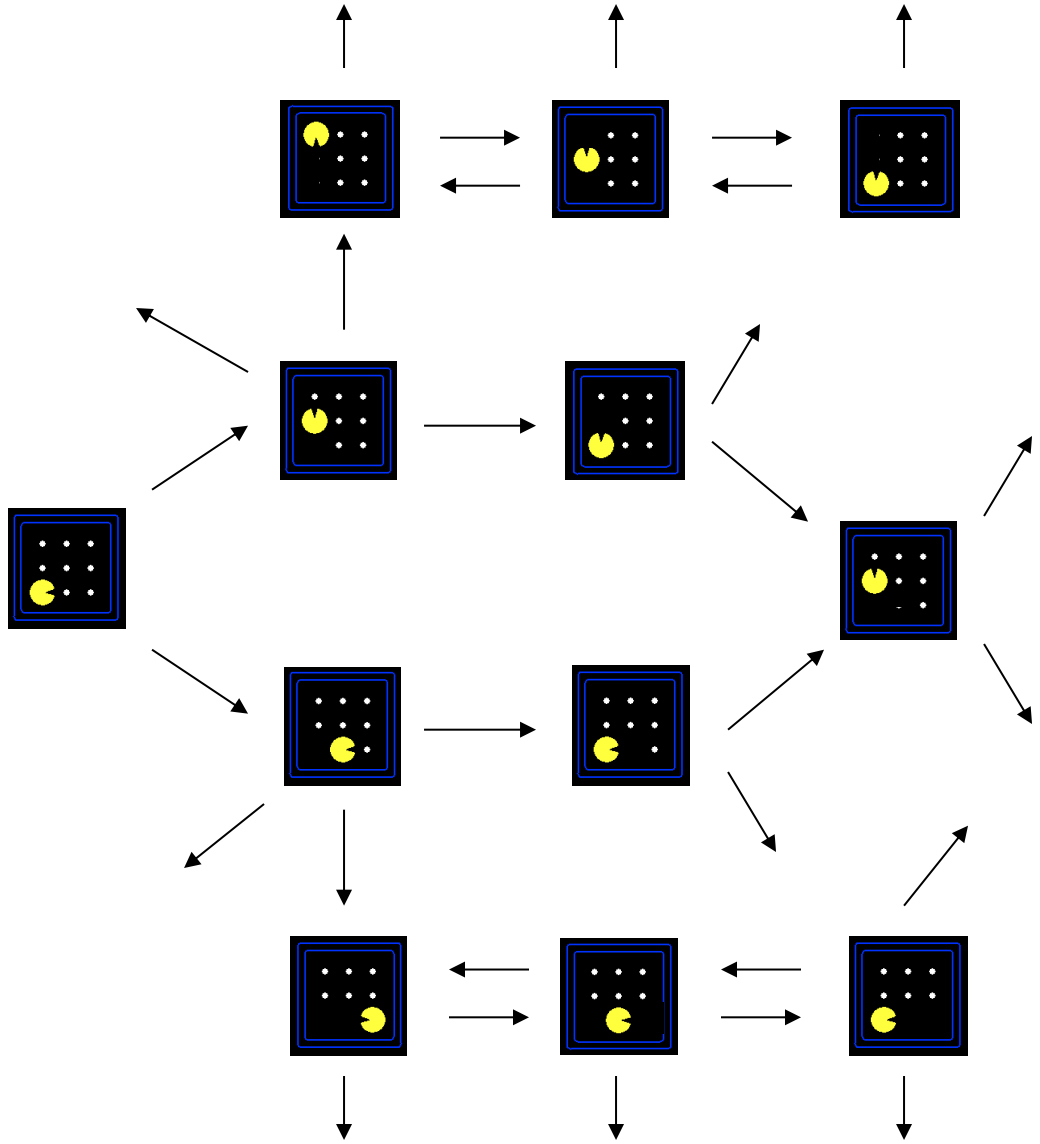
- The start state is the root node
- Children correspond to successors
- Nodes correspond to PLANS that achieve those states (go back up to the root to get the plan)
  - Nodes with same states can occur multiple times within the tree
- This also cannot be built for most problems!
  - But we can still do search on it!

State Space Graphs

- Nodes represent states
- Arcs represent transitions (actions)
- Goal states are a set of nodes

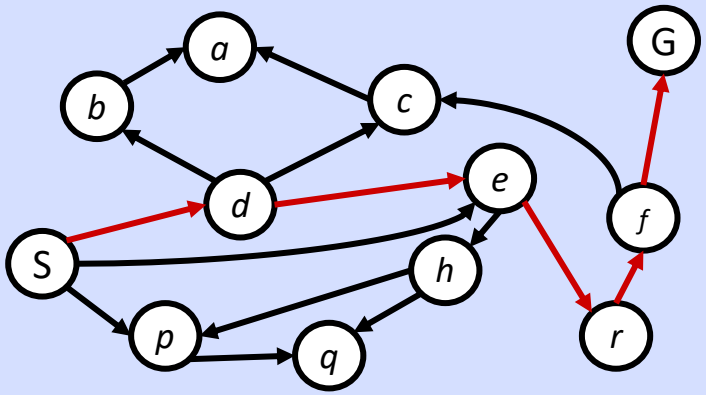
- Each state occurs only once!

- Remember, cannot build this for most problems!
  - But we can still do search on it!



State Space Graphs vs Search Trees

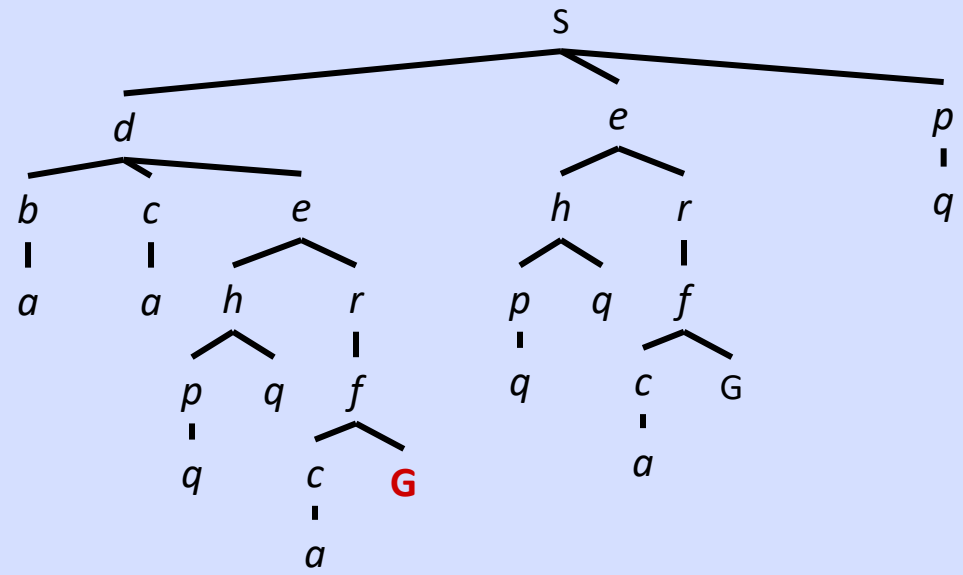
State Space Graph



Each NODE in in the search tree is an entire PATH in the state space graph.

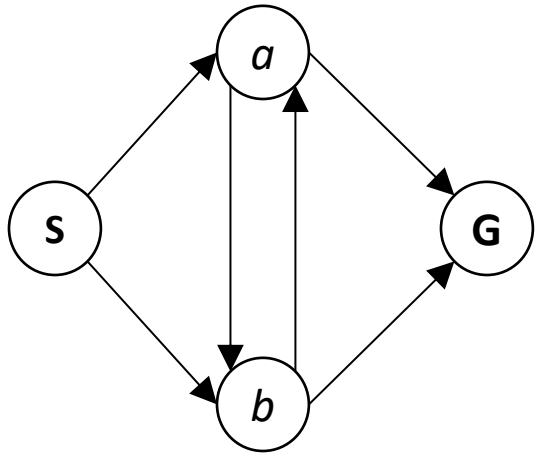
We construct both on demand – and we construct as little as possible.

Search Tree



Extreme Case: State Space Graphs vs. Search Trees

Consider this 4-state graph:



How big is its search tree (from S)?

Important: Lots of repeated structure in the search tree!

Tree Search

function GENERAL-TREE-SEARCH(problem) **returns** a solution, or failure

initialize the frontier using the initial state of problem

loop do

1. **if** the frontier is empty **then return** failure
2. choose a leaf node and remove it from the frontier
3. **if** the node contains a goal state **then return** the corresponding solution
4. expand the chosen node, adding the resulting nodes to the frontier

The way that a leaf node is chosen results in different algorithms

Frontier: A data structure containing the leaf nodes in the search tree, initialized with the start state

Leaf Nodes (Frontier)

function GENERAL-TREE-SEARCH(problem) **returns** a solution, or failure

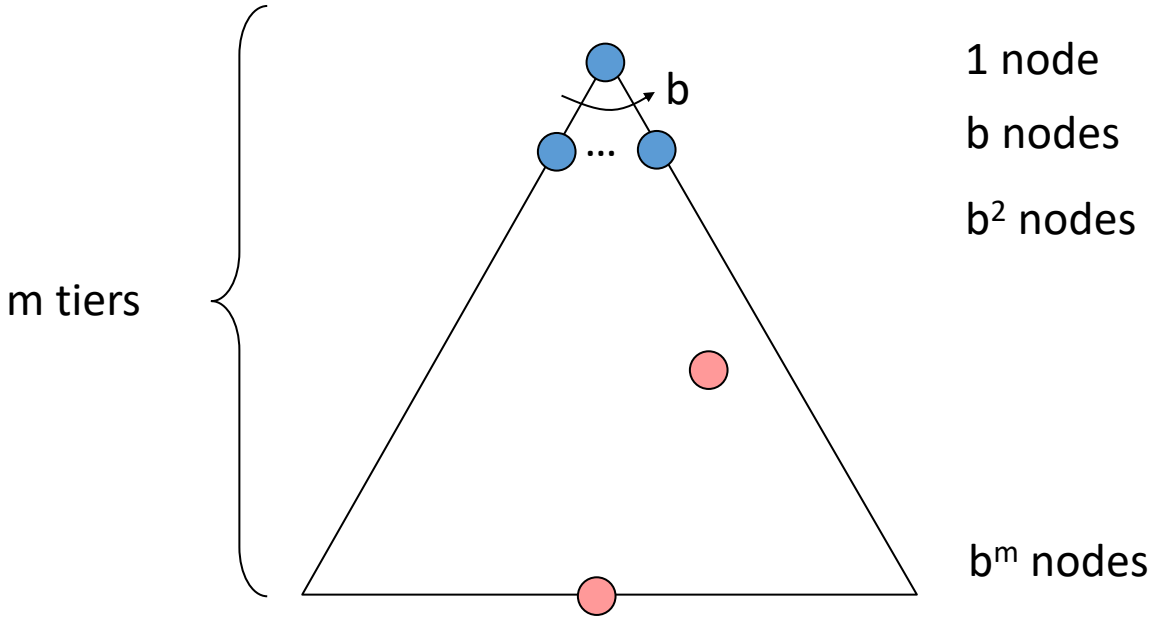
initialize the frontier using the initial state of problem

loop do

1. **if** the frontier is empty **then return** failure
2. choose a leaf node and remove it from the frontier
3. **if** the node is goal **then return** the solution
4. expand the node, add its children to the frontier

Detour: Properties of Search Algorithms

- **Completeness**: Finds a solution if one exists
- **Optimality**: Finds the least cost (or maximum utility!) solution
- **Time Complexity**: Number of nodes generated to find the solution
- **Space Complexity**: Maximum number of nodes in memory (size of the frontier)
- Complexities are mostly measured in:
  - b: branching factor
  - d: depth of the least cost solution
  - m: maximum depth (can be infinite!)
- Total number of nodes in the tree?

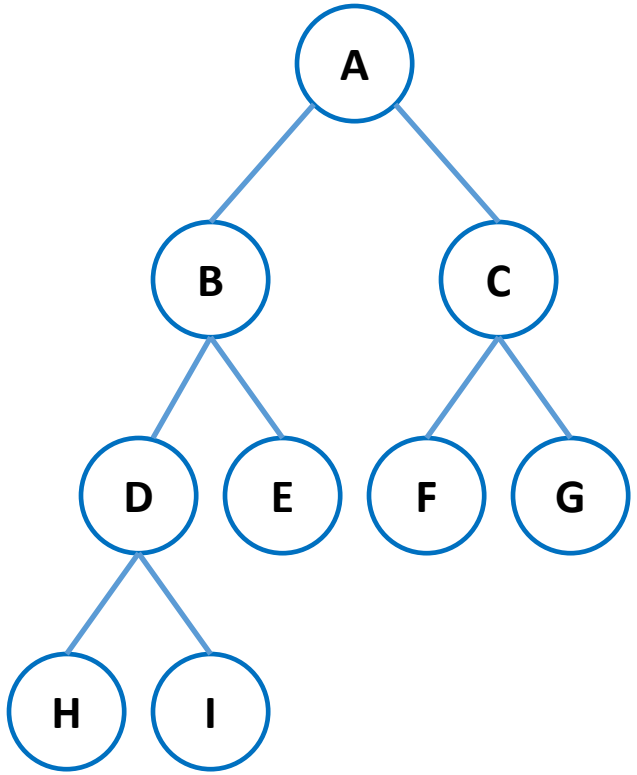


Depth First Search

Idea: expand the deepest node first

Implementation: Frontier is a LIFO stack

Nodes are added based on the alphabetical order

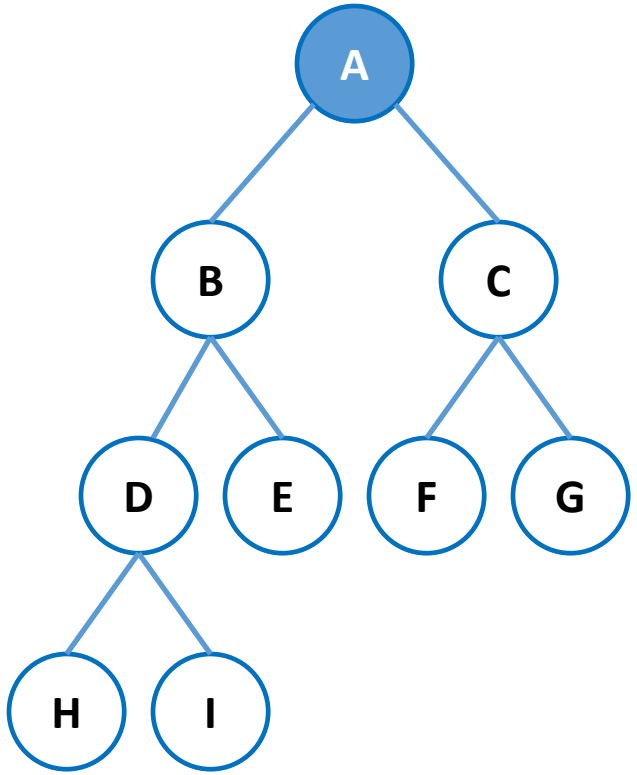


Depth First Search

Idea: expand the deepest node first

Implementation: Frontier is a LIFO stack

Nodes are added based on the alphabetical order

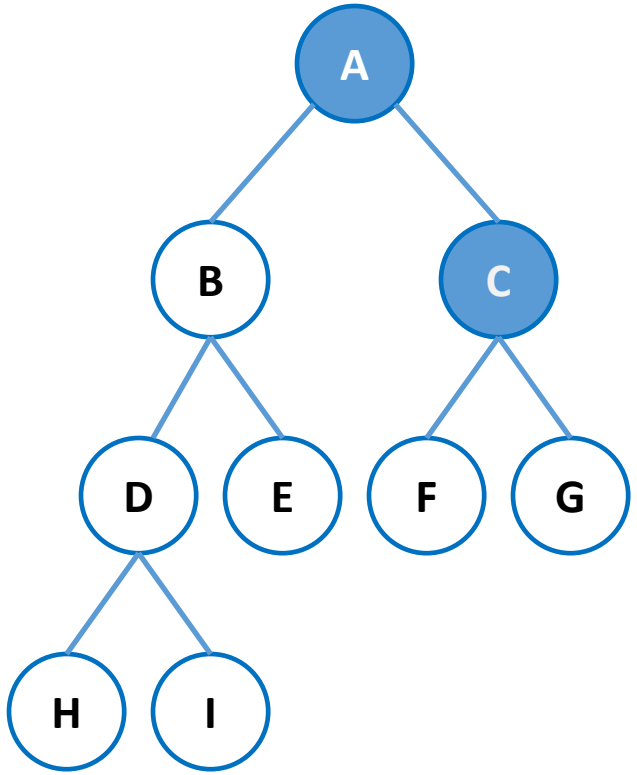


Depth First Search

Idea: expand the deepest node first

Implementation: Frontier is a LIFO stack

Nodes are added based on the alphabetical order

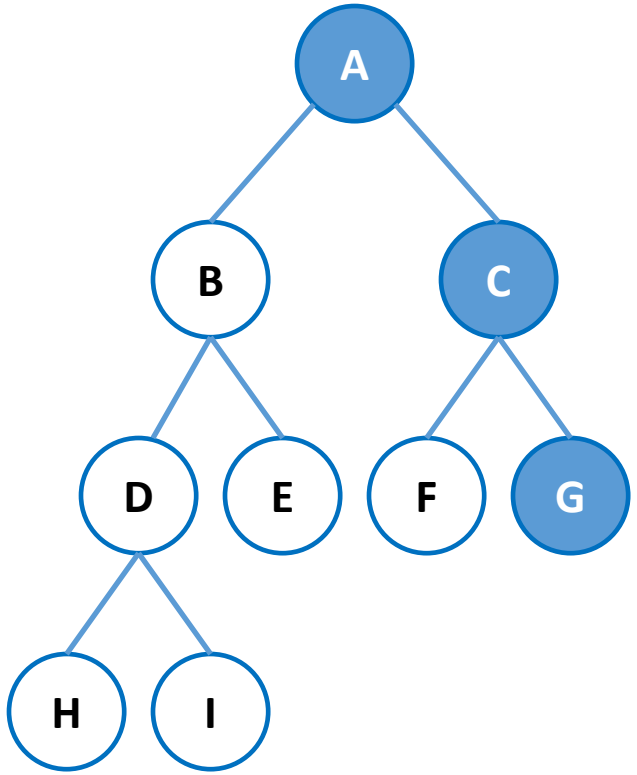


Depth First Search

Idea: expand the deepest node first

Implementation: Frontier is a LIFO stack

Nodes are added based on the alphabetical order

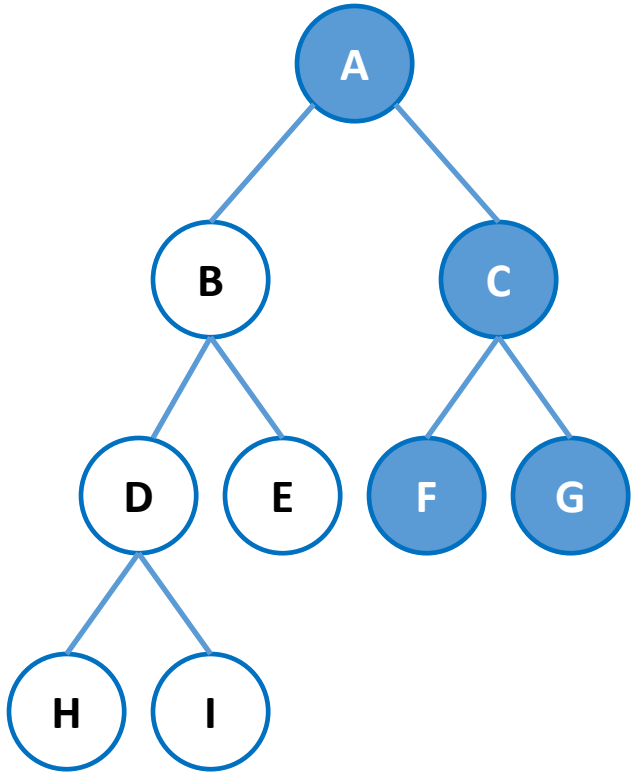


Depth First Search

Idea: expand the deepest node first

Implementation: Frontier is a LIFO stack

Nodes are added based on the alphabetical order

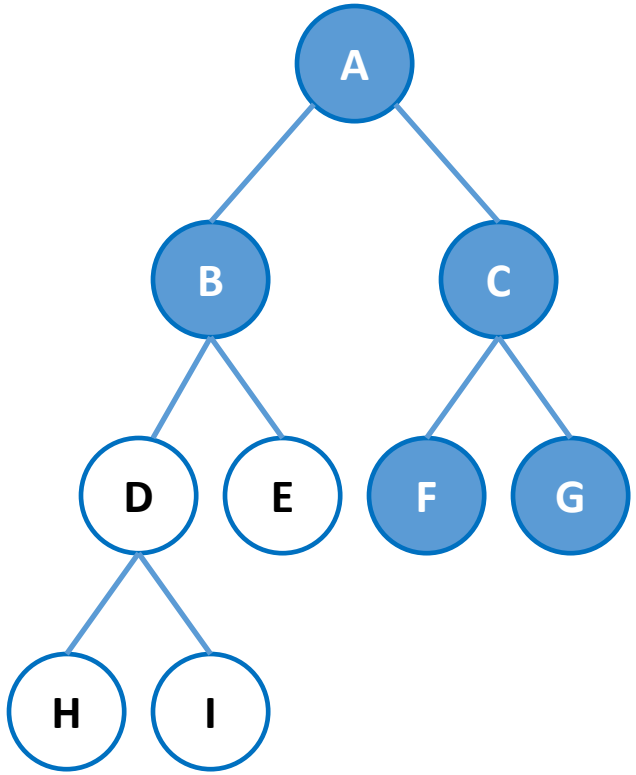


Depth First Search

Idea: expand the deepest node first

Implementation: Frontier is a LIFO stack

Nodes are added based on the alphabetical order

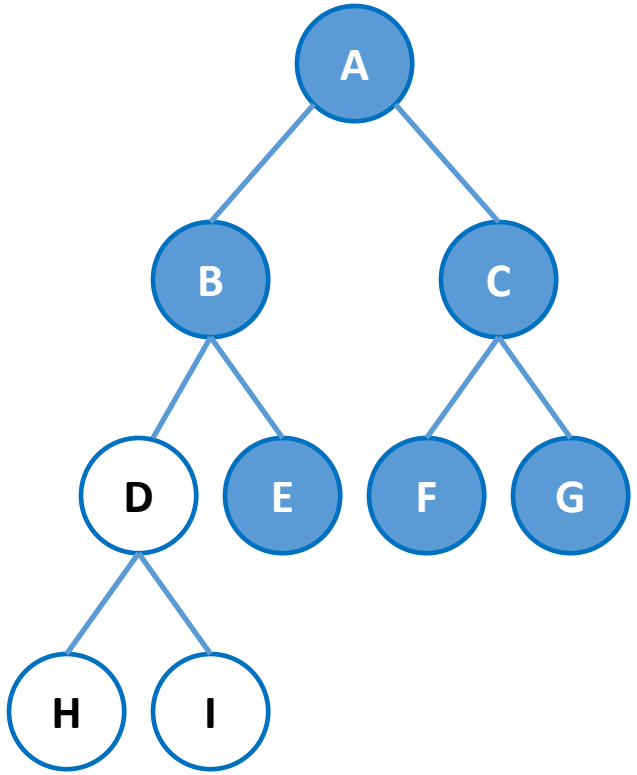


Depth First Search

Idea: expand the deepest node first

Implementation: Frontier is a LIFO stack

Nodes are added based on the alphabetical order



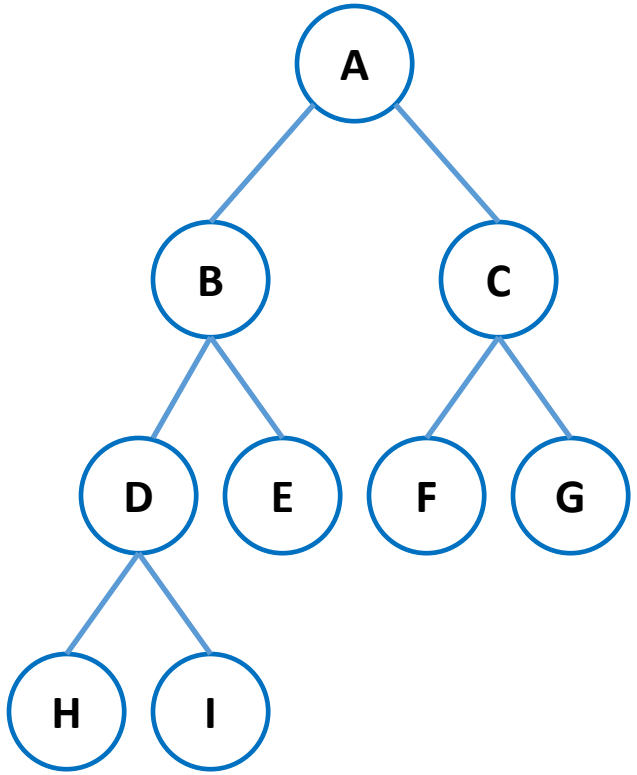
… until you get to the desired node

Depth First Search – Recursive Expand Order?

Idea: expand the deepest node first

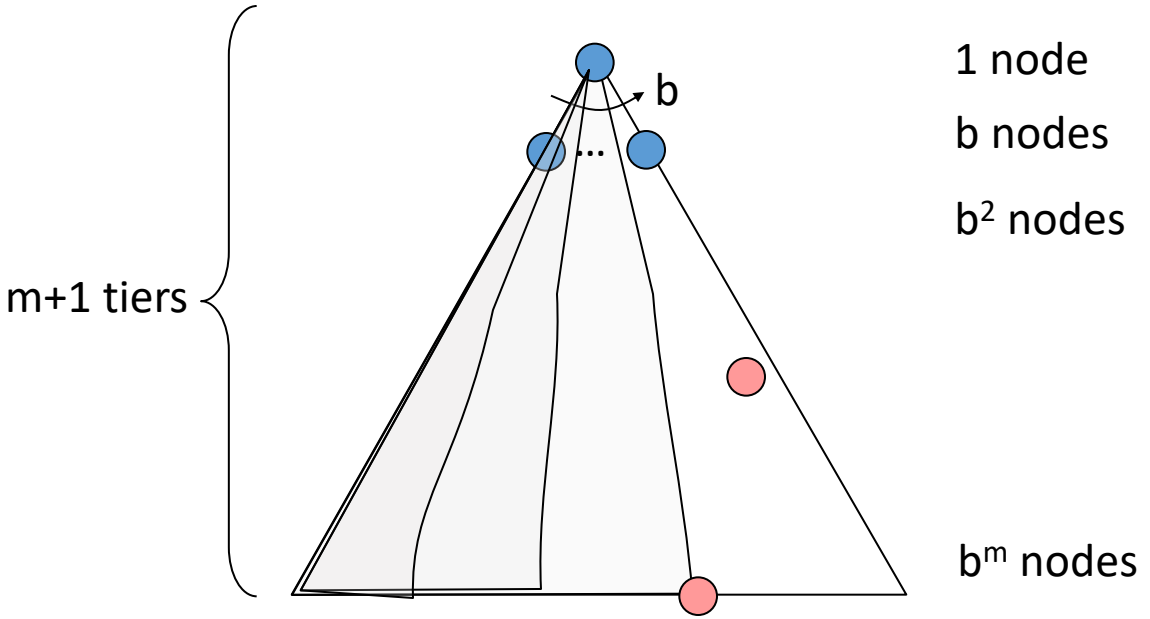
Implementation: Using a recursive DFS

Nodes are traversed based on the alphabetical order

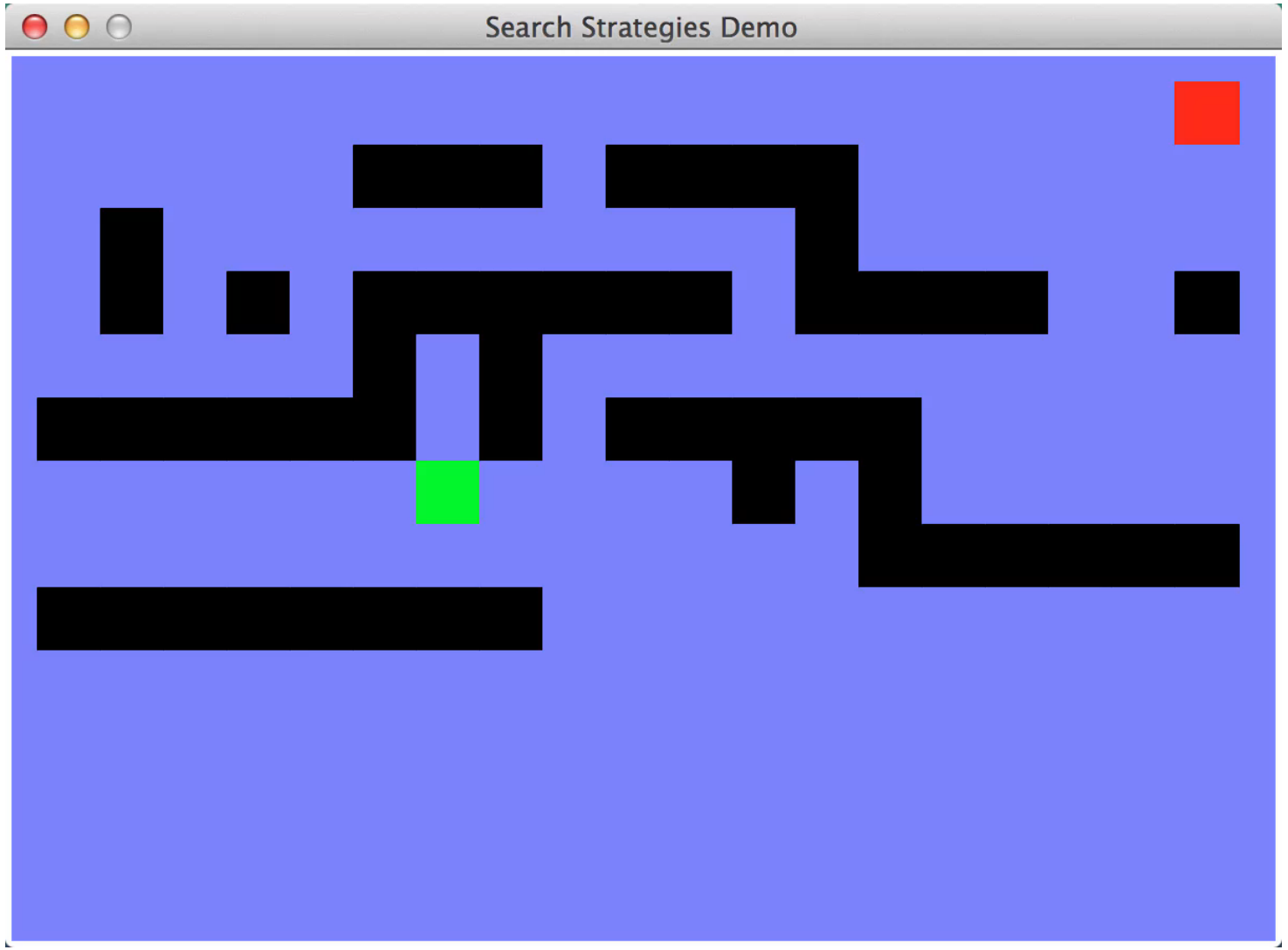


DFS Properties

- **Completeness** ?
  - Only if cycles are prevented!
- **Optimality** ?
  - No!
- **Time Complexity** ?
  - $O(b^m)$, $b^m >> b^{m-1}$
- **Space Complexity** ?
  - O(bm), siblings



$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

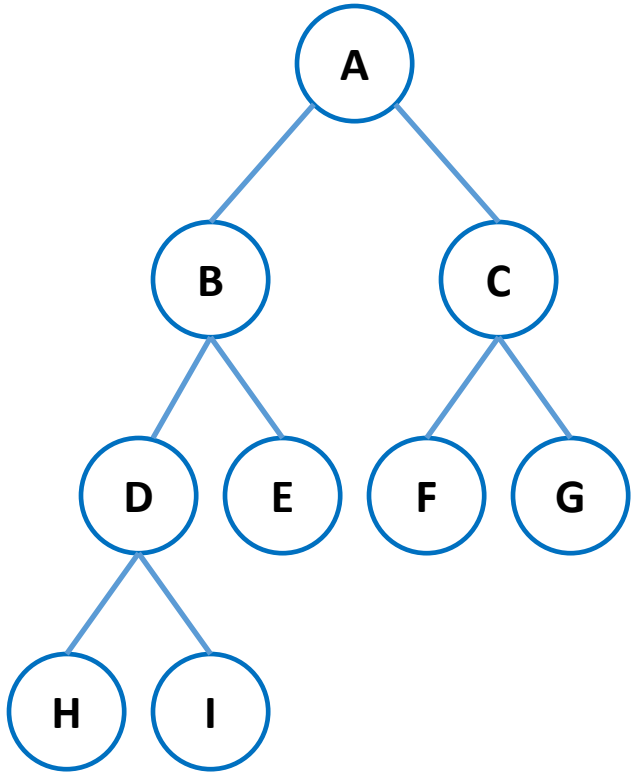

Search Strategies Demo

Breadth First Search

Idea: expand the shallowest node first

Implementation: Frontier is a FIFO queue

Nodes are added based on the alphabetical order

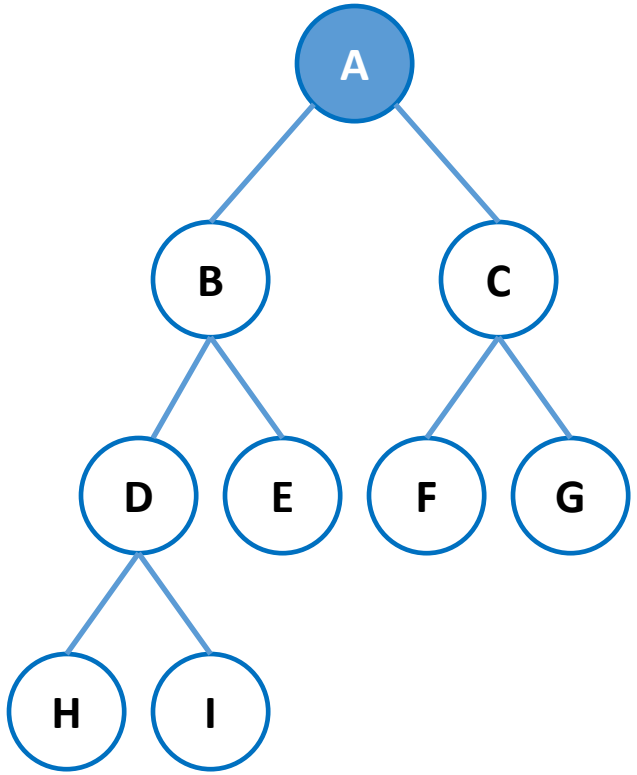


Breadth First Search

Idea: expand the shallowest node first

Implementation: Frontier is a FIFO queue

Nodes are added based on the alphabetical order

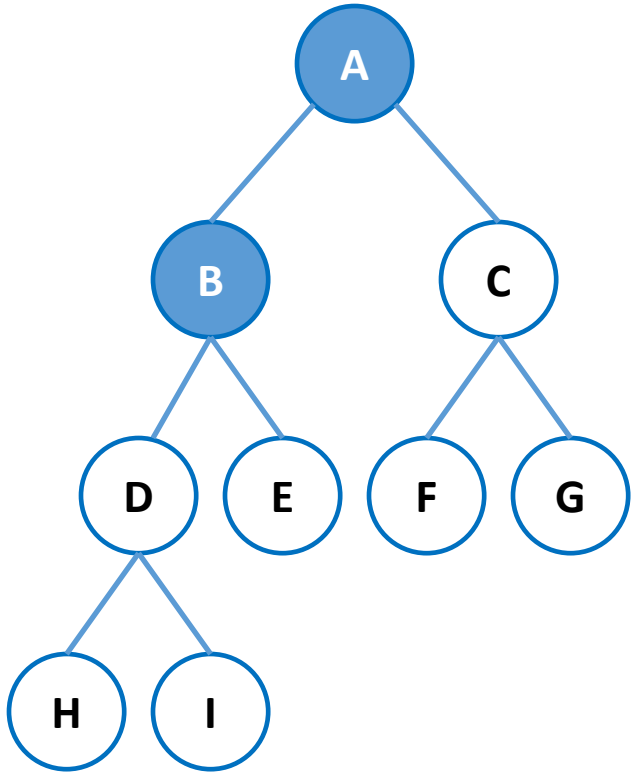


Breadth First Search

Idea: expand the shallowest node first

Implementation: Frontier is a FIFO queue

Nodes are added based on the alphabetical order

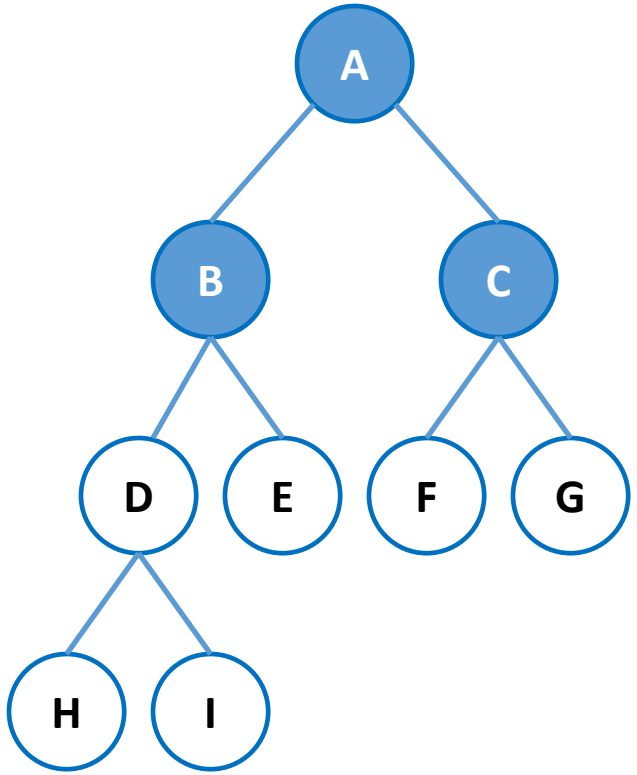


Breadth First Search

Idea: expand the shallowest node first

Implementation: Frontier is a FIFO queue

Nodes are added based on the alphabetical order

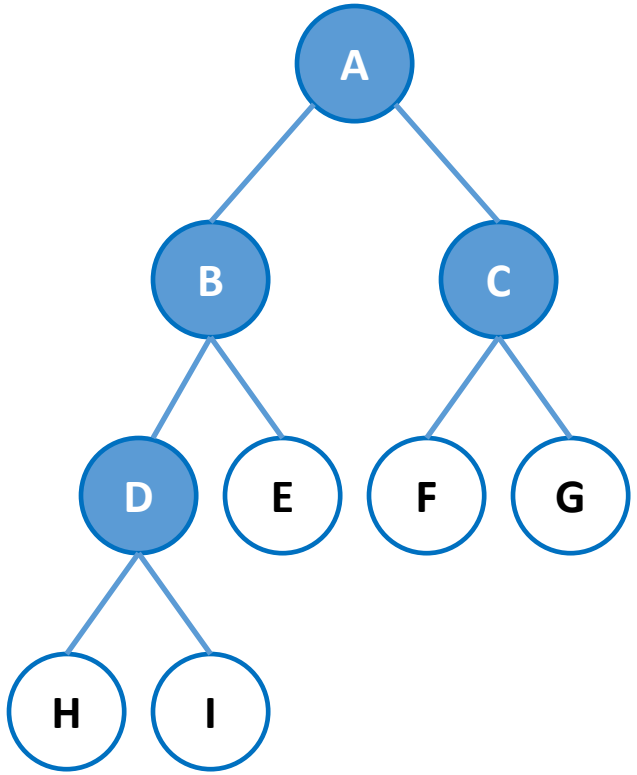


Breadth First Search

Idea: expand the shallowest node first

Implementation: Frontier is a FIFO queue

Nodes are added based on the alphabetical order

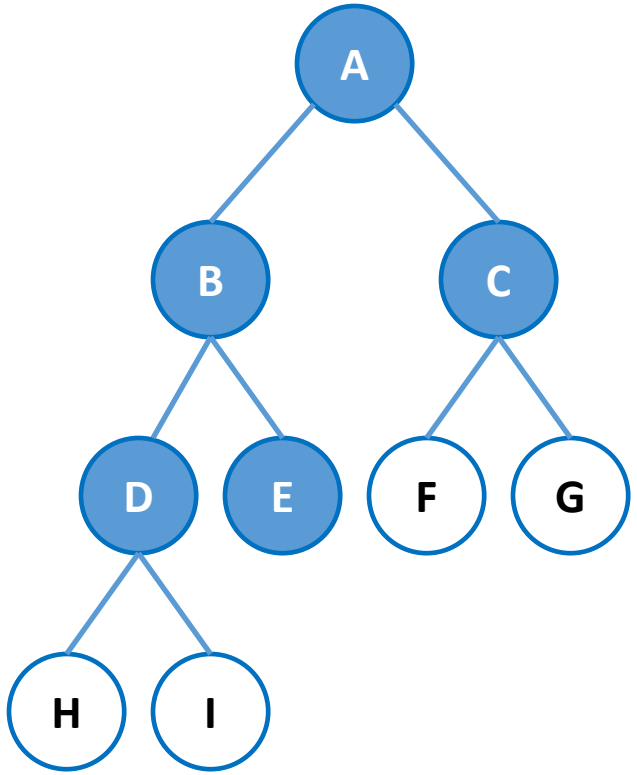


Breadth First Search

Idea: expand the shallowest node first

Implementation: Frontier is a FIFO queue

Nodes are added based on the alphabetical order

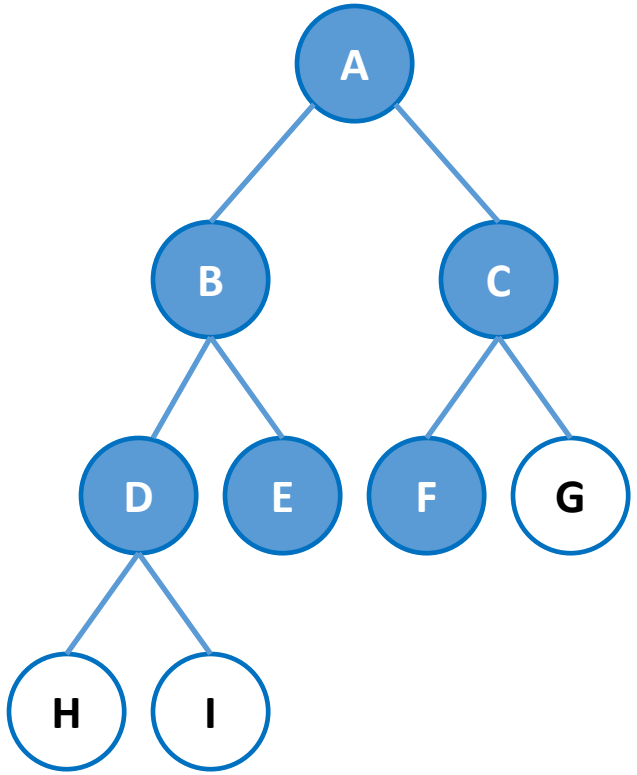


Breadth First Search

Idea: expand the shallowest node first

Implementation:
Frontier is a FIFO queue

Nodes are added based on the alphabetical order

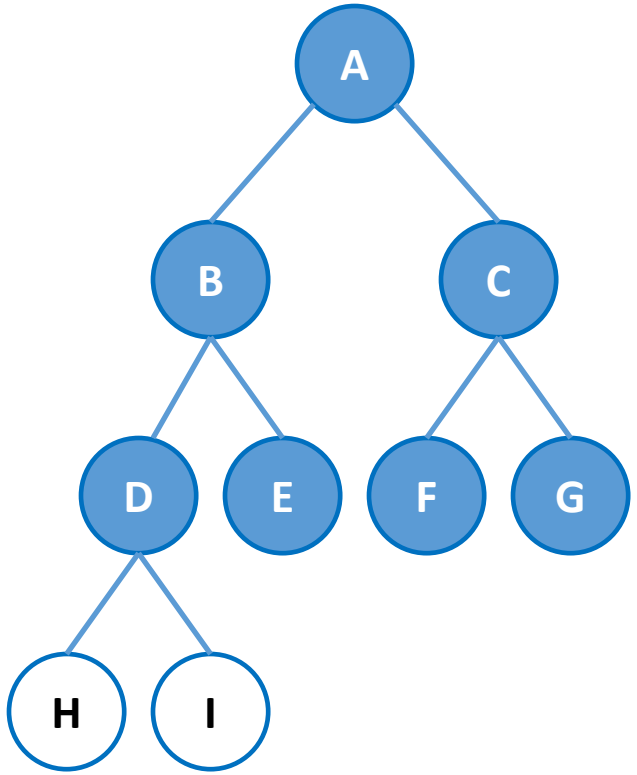


Breadth First Search

Idea: expand the shallowest node first

Implementation: Frontier is a FIFO queue

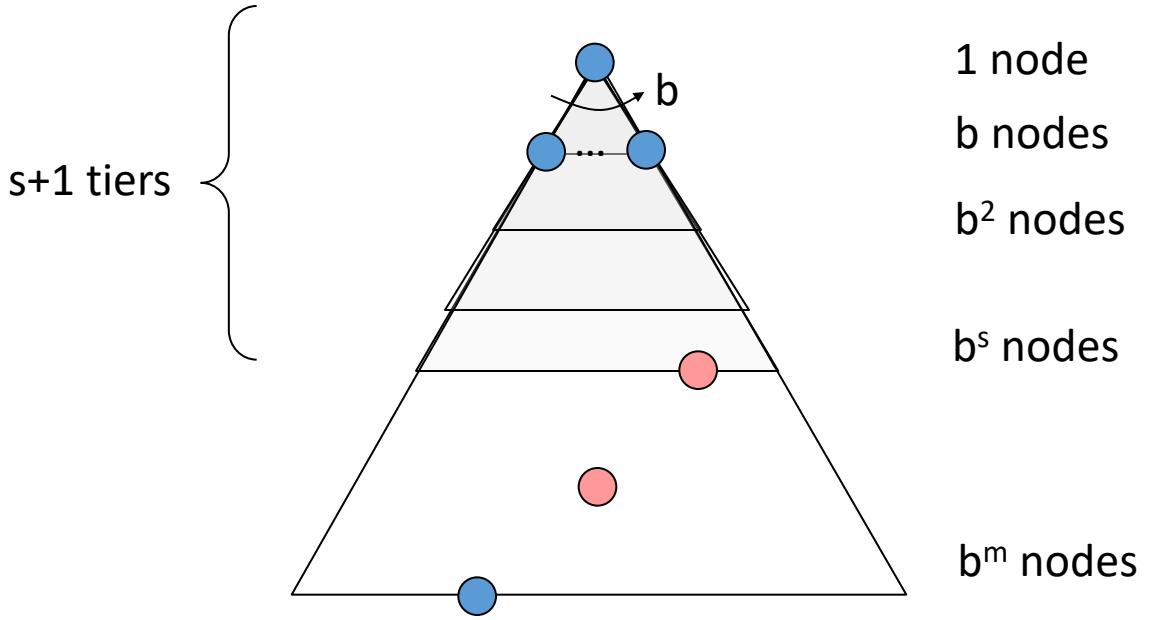
Nodes are added based on the alphabetical order

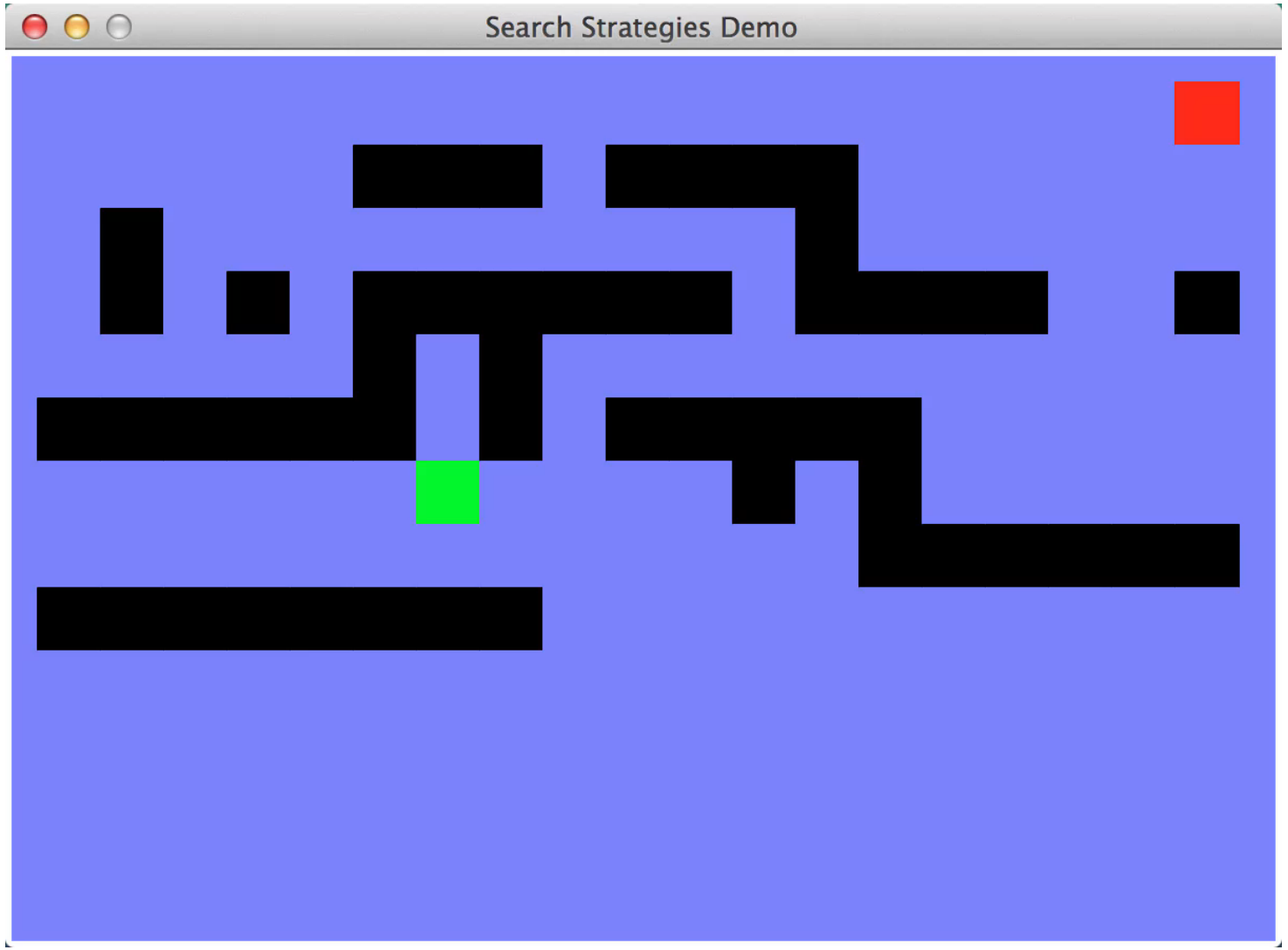


… until you get to the desired node

BFS Properties

- **Completeness** ?
  - Yes (finite s)
- **Optimality** ?
  - Yes, if costs are equal
- **Time Complexity** ?
  - $O(b^s)$
- **Space Complexity** ?
  - $O(b^s)$, last tier




Search Strategies Demo

Depth Limited DFS

- Depth first tree search is not complete. It may not terminate if the tree is infinite
- A common approach is to limit the search depth
  - Keep track of the depth as you search
  - Do not add new states to the “frontier” if the current expanded node is at the depth limit
- Easier to keep track of the depth using a recursive DFS

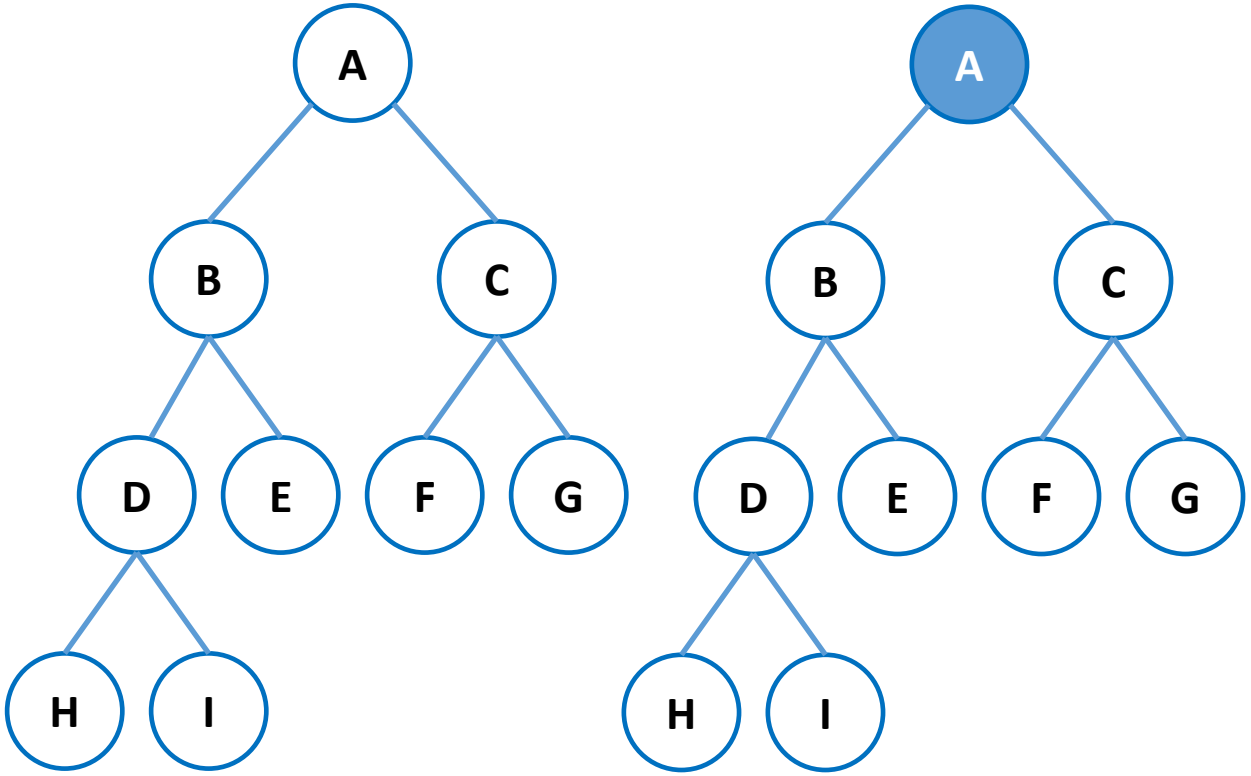
- We will get back to similar ideas during the “game-playing” part of the class

Iterative Deepening Search

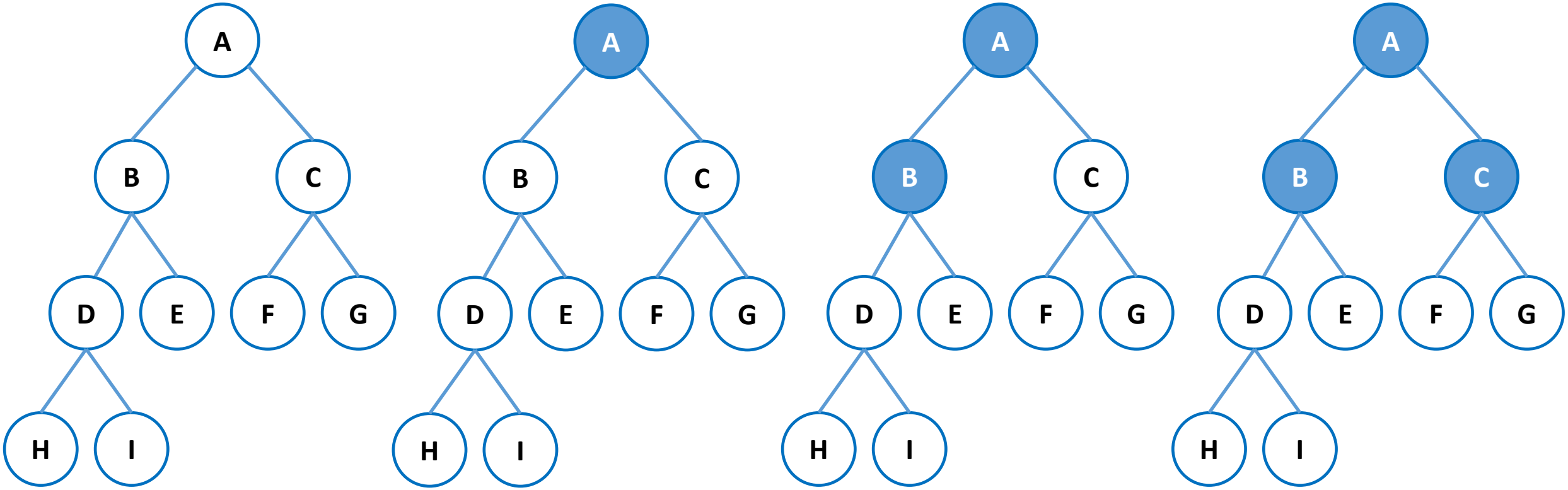
- DFS: Has a space advantage
- BFS: Finds the “shallowest solution” and is complete
- Combine them!

1. Set depth limit as 0
2. Do depth limited DFS
3. If the goal is found return, else increase depth limit by 1
4. Go back to step 2

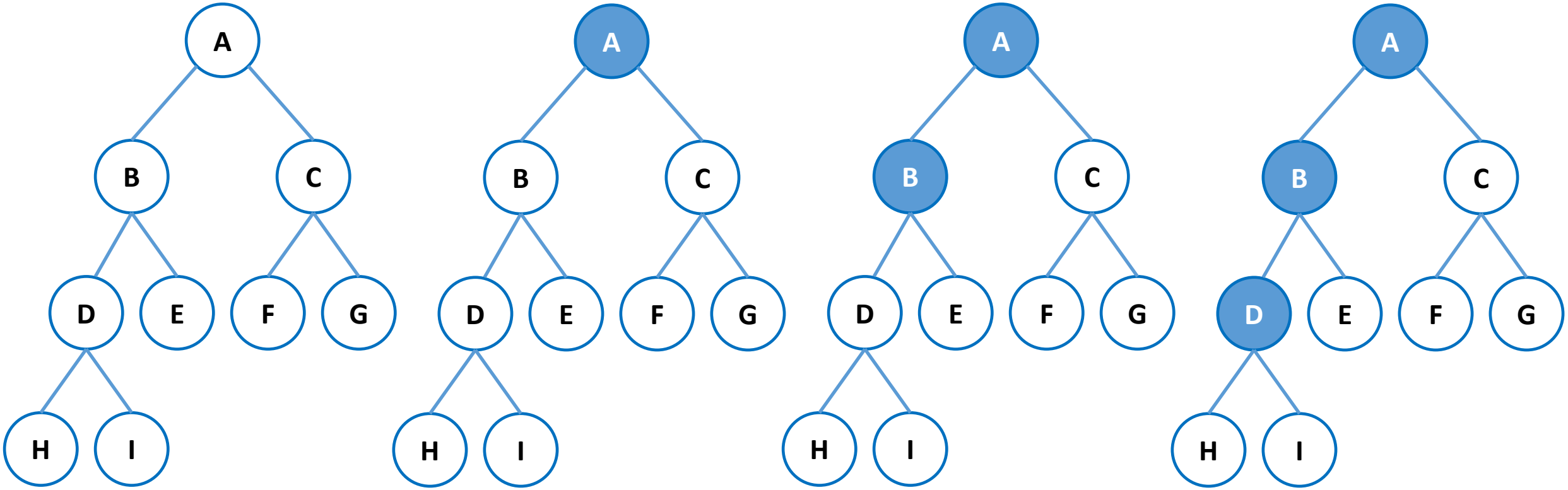
Iterative Deepening Search: Find G, Limit = 0



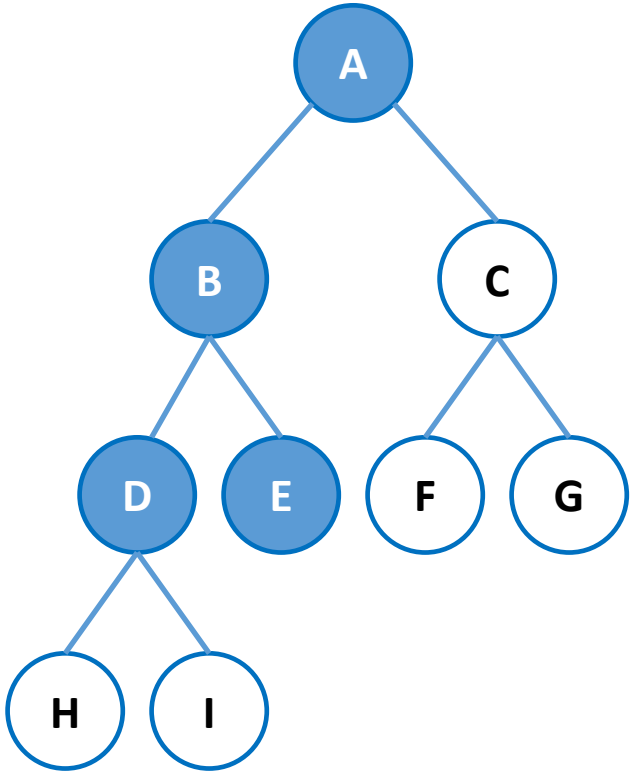
Iterative Deepening Search: Find G, Limit = 1

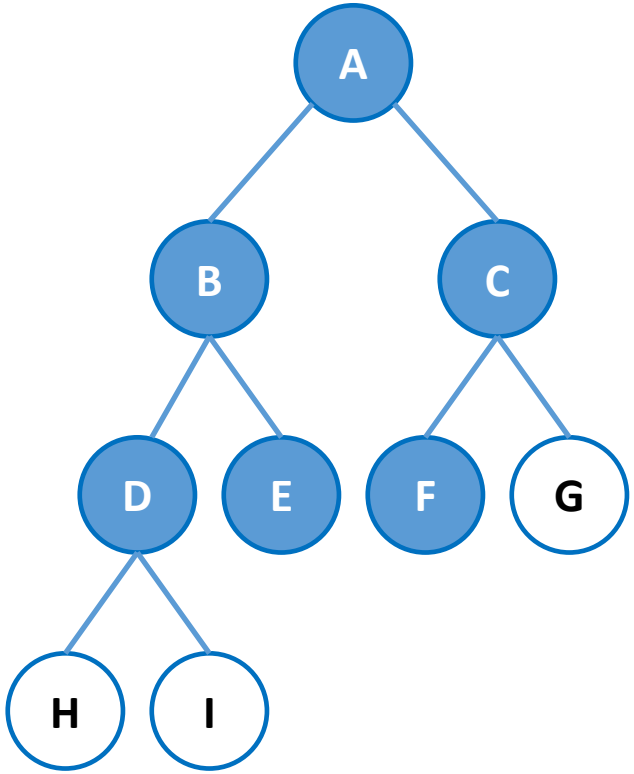


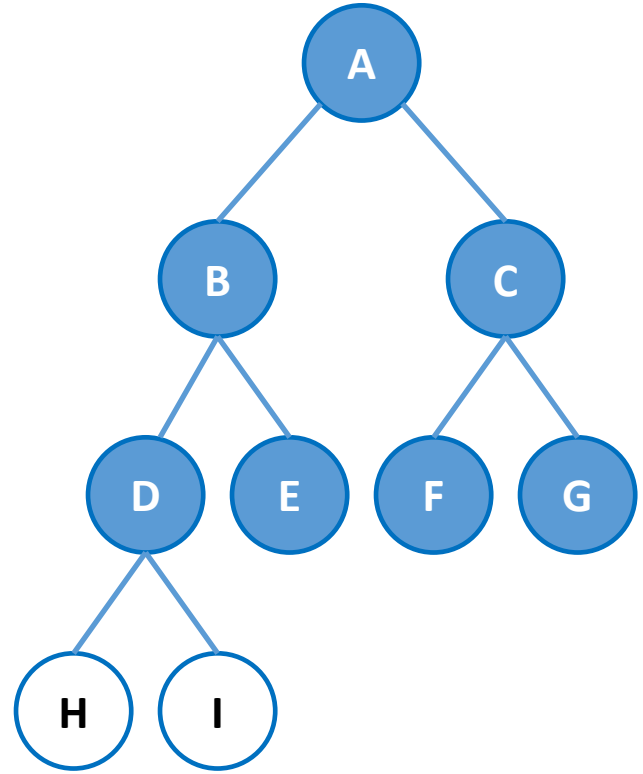
Iterative Deepening Search: Find G, Limit = 2



Iterative Deepening Search: Find G, Limit = 2

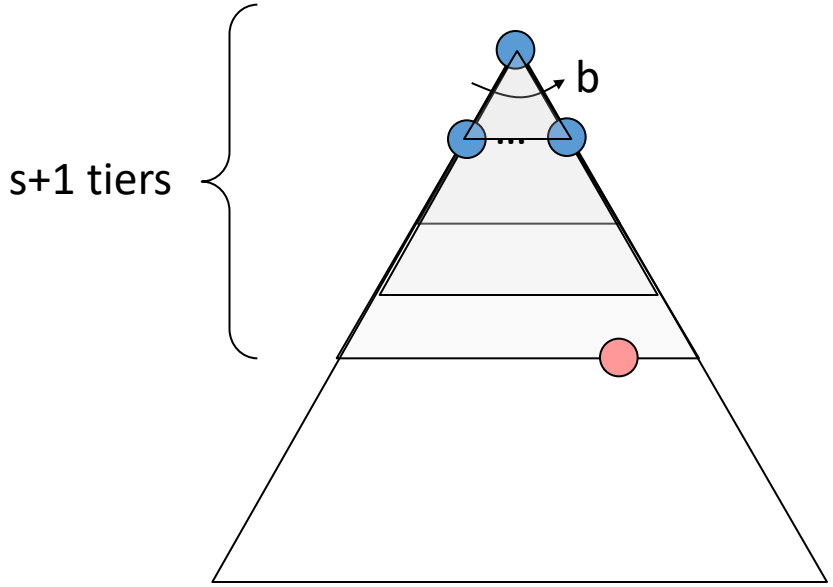






Iterative Deepening Properties

- **Completeness** ?
  - Yes
- **Optimality** ?
  - Yes, if costs are equal
- **Space Complexity** ?
  - O(sb) – same as DFS
- **Time Complexity** ?
  - $O(b^s)$ – main work happens at the lowest level!



Iterative Deepening Time Complexity Proof

Note that $\sum_{i=0}^{\infty}(i+1)x^i = (1 + 2x + 3x^2 + \cdots) = (1-x)^{-2}$, for $|x| < 1$

For iterative deepening DFS with branching factor b , we expand the total of:

1

$1 + b$

$1 + b + b^2$

…

$+\ 1 + b + b^2 + \cdots + b^s$

$(s+1) + (s)b + (s-1)b^2 + \cdots + b^s$ nodes.

Factor out b^s to get: $b^s(1 + 2b^{-1} + 3b^{-2} \ldots + sb^{1-s} + (s+1)b^{-s})$

Let $x = b^{-1}$ (note that b > 1 and thus $|x| < 1$)

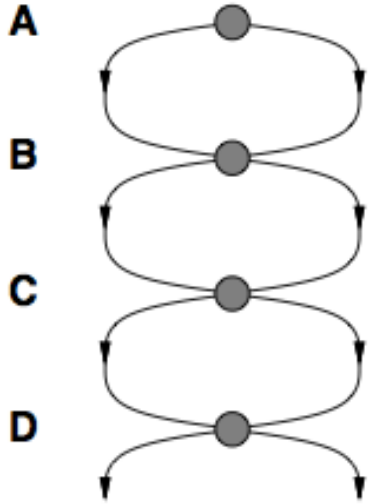
$b^s(1 + 2x + 3x^2 \ldots + sx^{s-1} + (s+1)x^s) < b^s(1 + 2x + 3x^2 + \cdots) = b^s(1-x)^{-2} = b^s c$

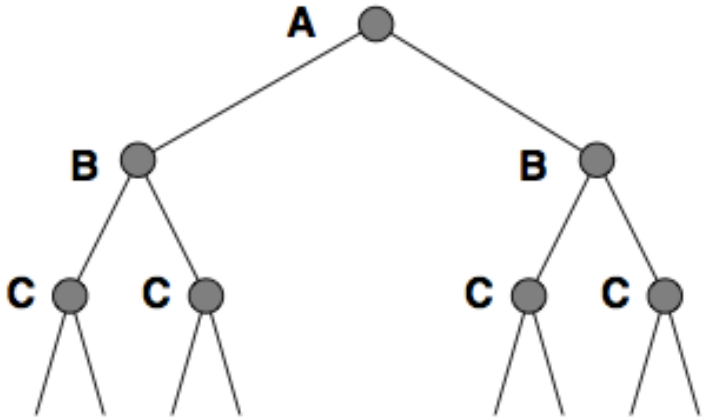
Which is $O(b^s)$ (note that the total is > b^s so the lower bound is also satisfied for the big-Oh)

Graph Search

- Recall that tree can have nodes with the same states
- Don’t want to waste time going somewhere twice
- When does this happen?
  - 2 paths to a state
  - Actions are reversible
- E.g. linear problem made exponentially larger!

- **Idea**: keep track of where you visited





Graph Search

function GENERAL-GRAPH-SEARCH(problem) **returns** a solution, or failure

initialize the frontier using the initial state of problem

initialize the explored set to be empty

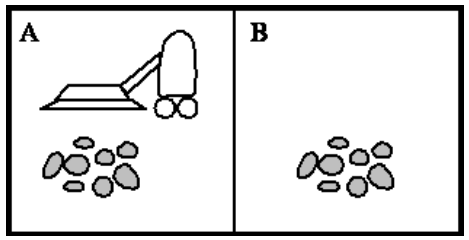
loop do

1. **if** the frontier is empty **then return** failure
2. choose a leaf node and remove it from the frontier
3. **if** the node is in the explored set **then continue**
4. **if** the node contains a goal state **then return** the corresponding solution
5. *add the node to the explored set*
6. expand the chosen node, adding the resulting nodes to the frontier
   only if not in the explored set

All the tree search algorithms can be converted to their graph version!

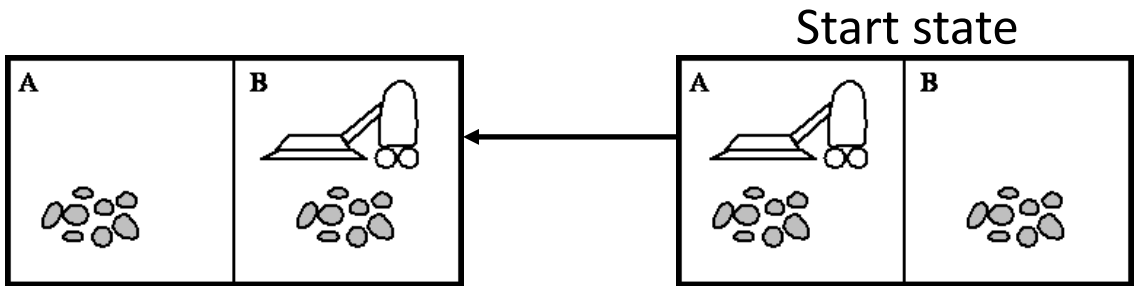
Example: DFS

Start state



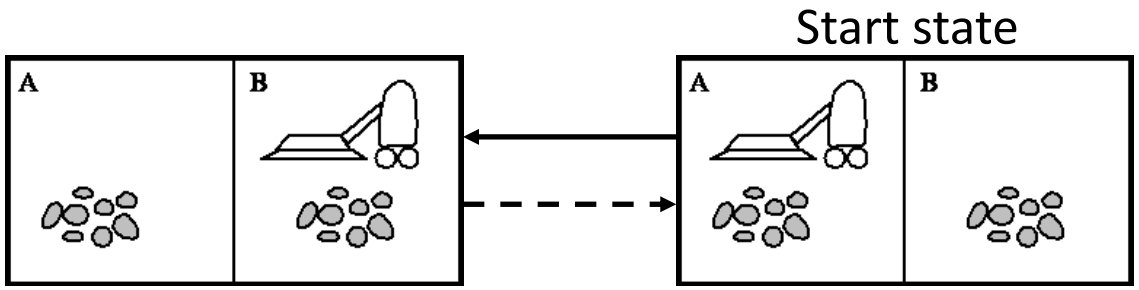
Actions: Move, Suck
Move is explored first

Example: DFS



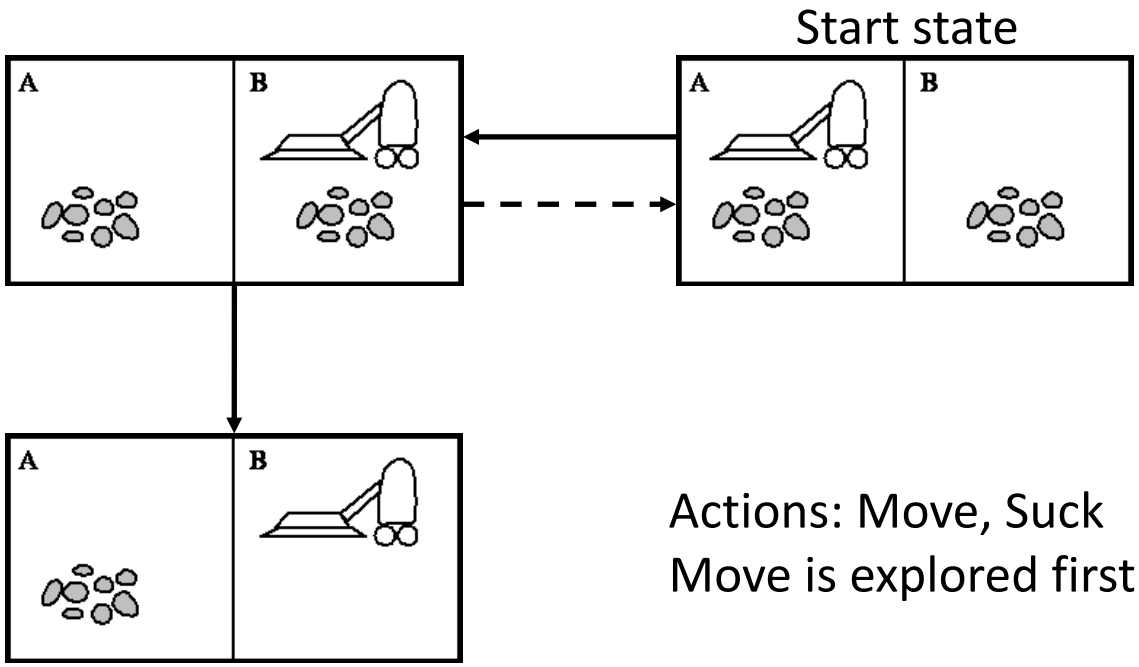
Actions: Move, Suck
Move is explored first

Example: DFS



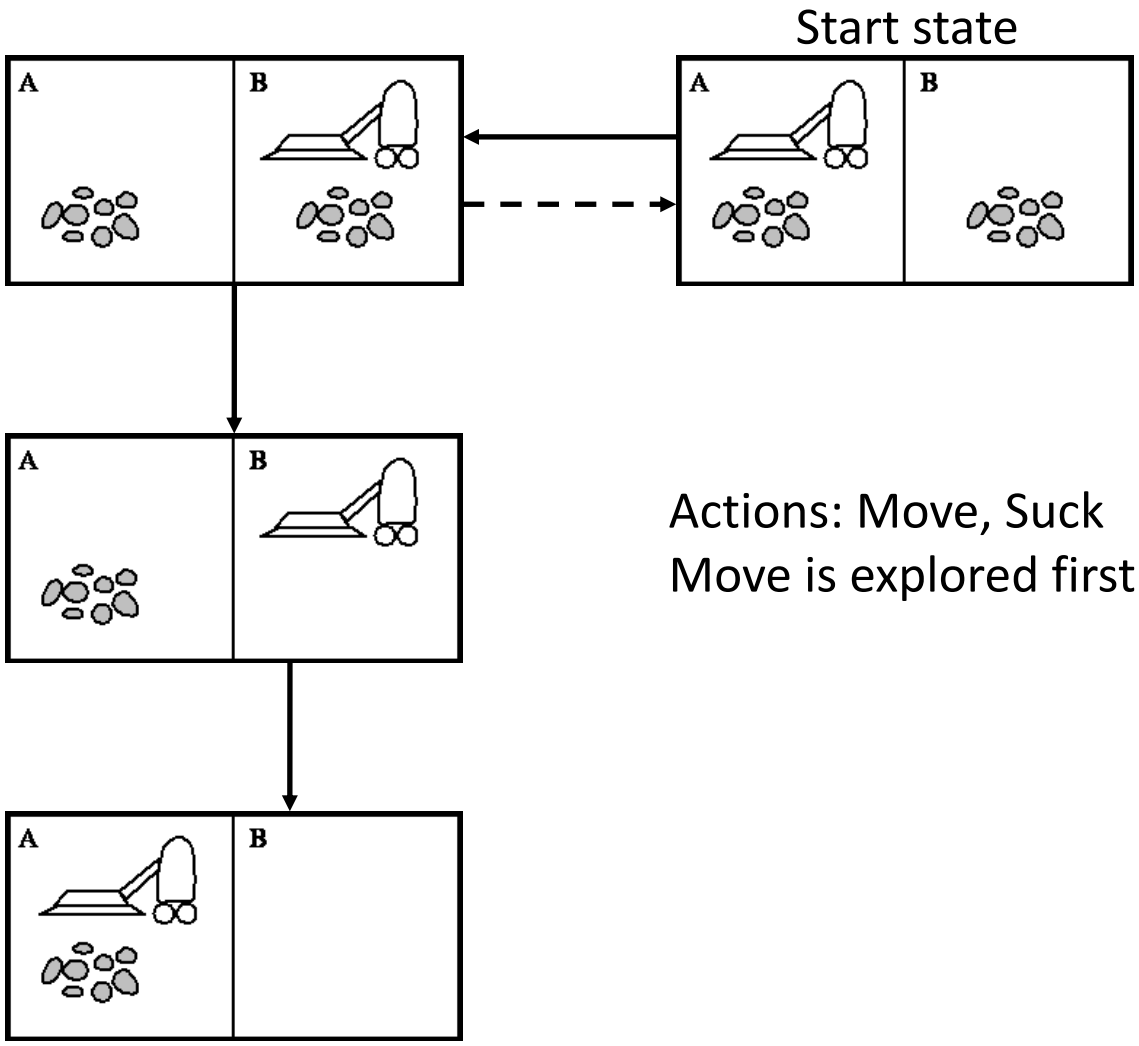
Actions: Move, Suck
Move is explored first

Example: DFS



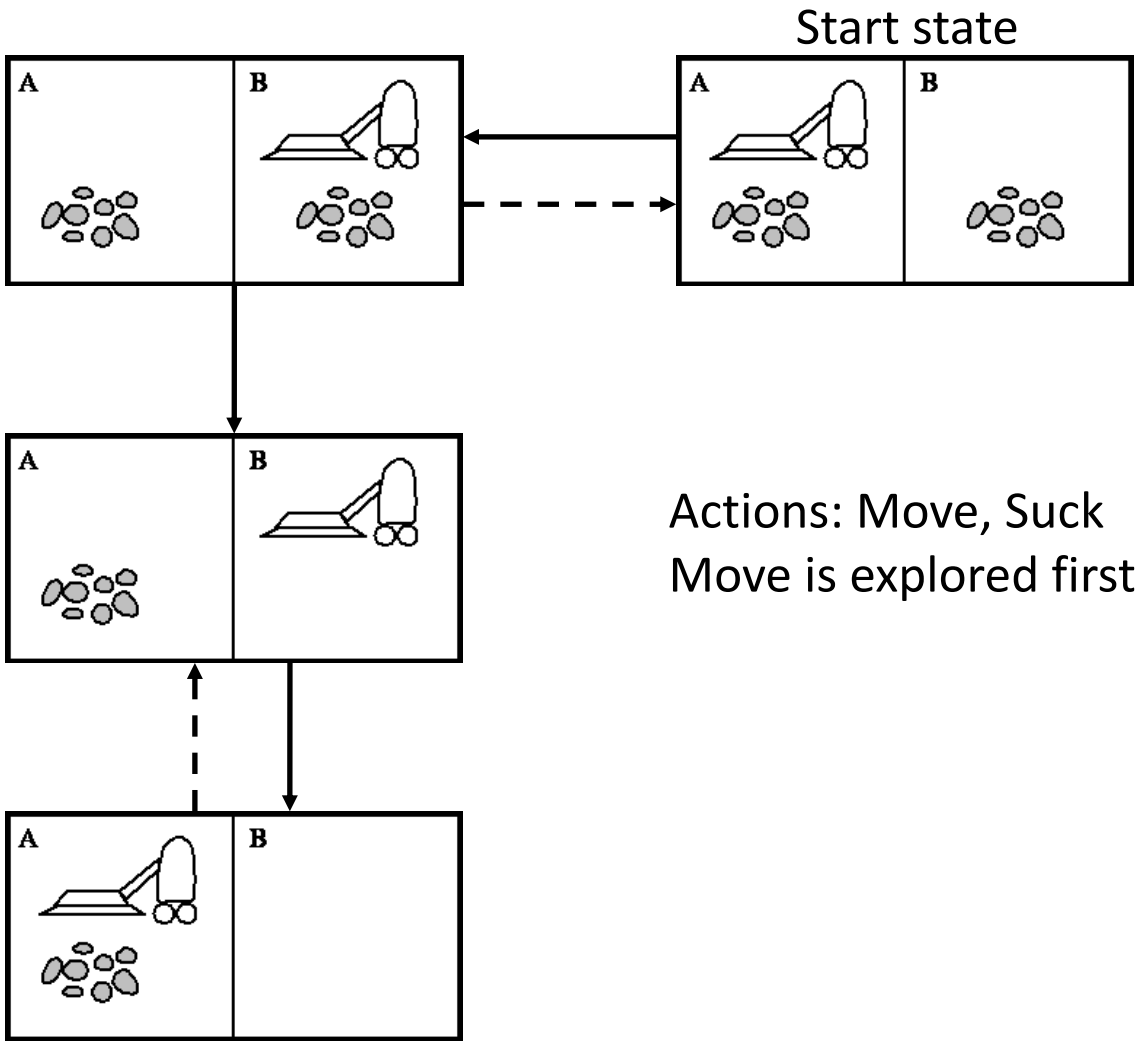
Actions: Move, Suck
Move is explored first

Example: DFS



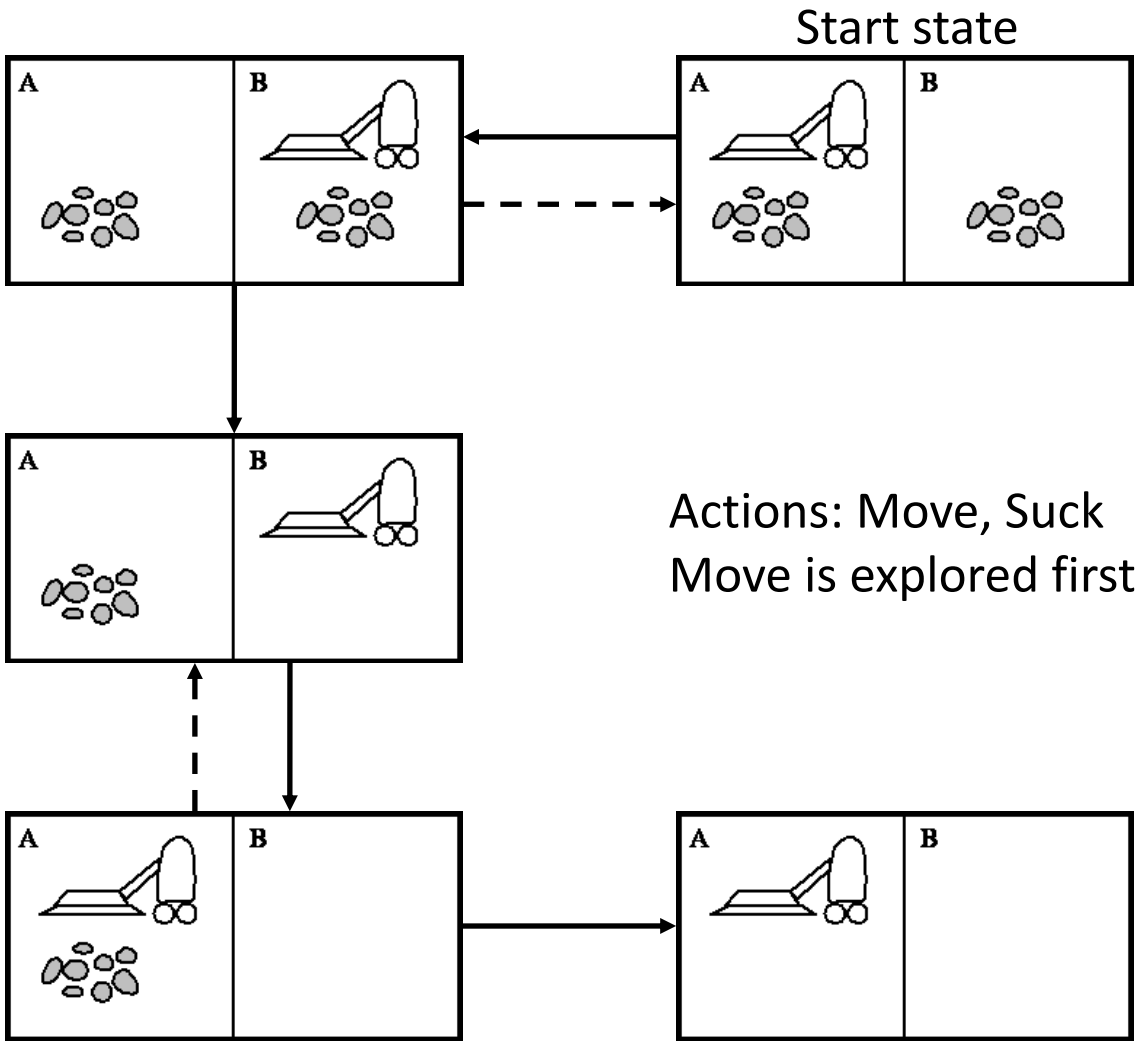
Actions: Move, Suck
Move is explored first

Example: DFS



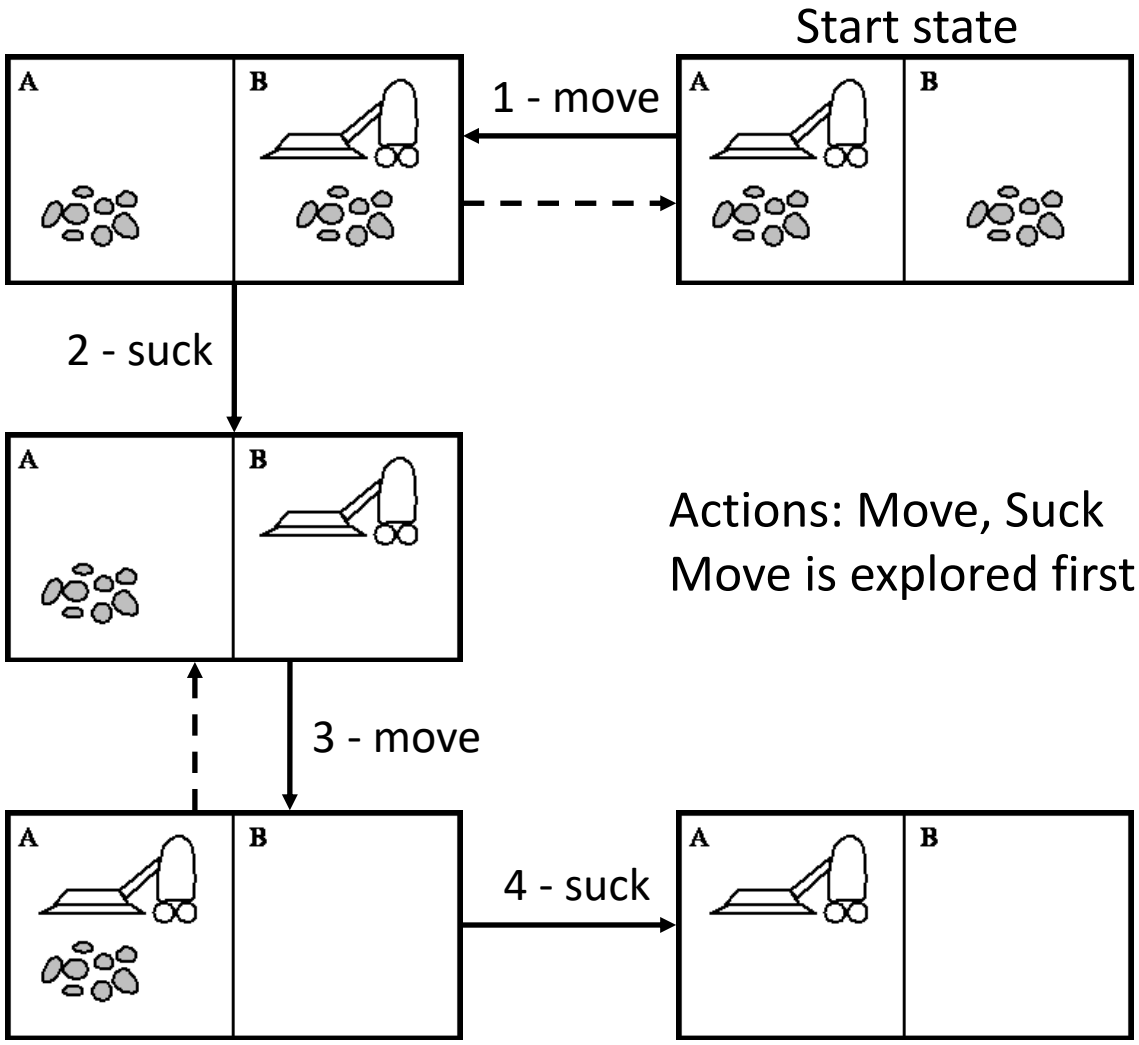
Actions: Move, Suck
Move is explored first

Example: DFS



Actions: Move, Suck
Move is explored first

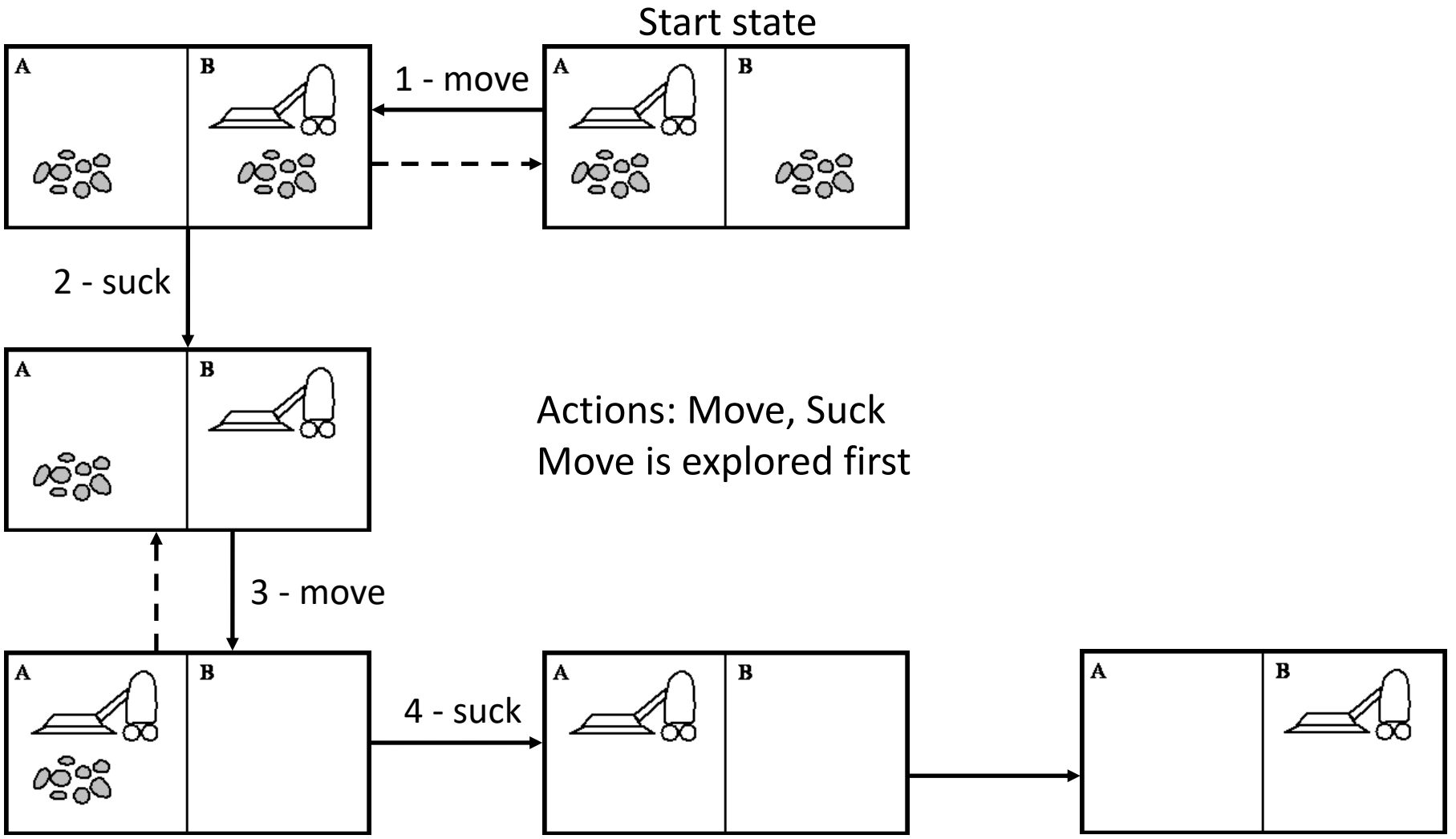
Example: DFS



Actions: Move, Suck
Move is explored first

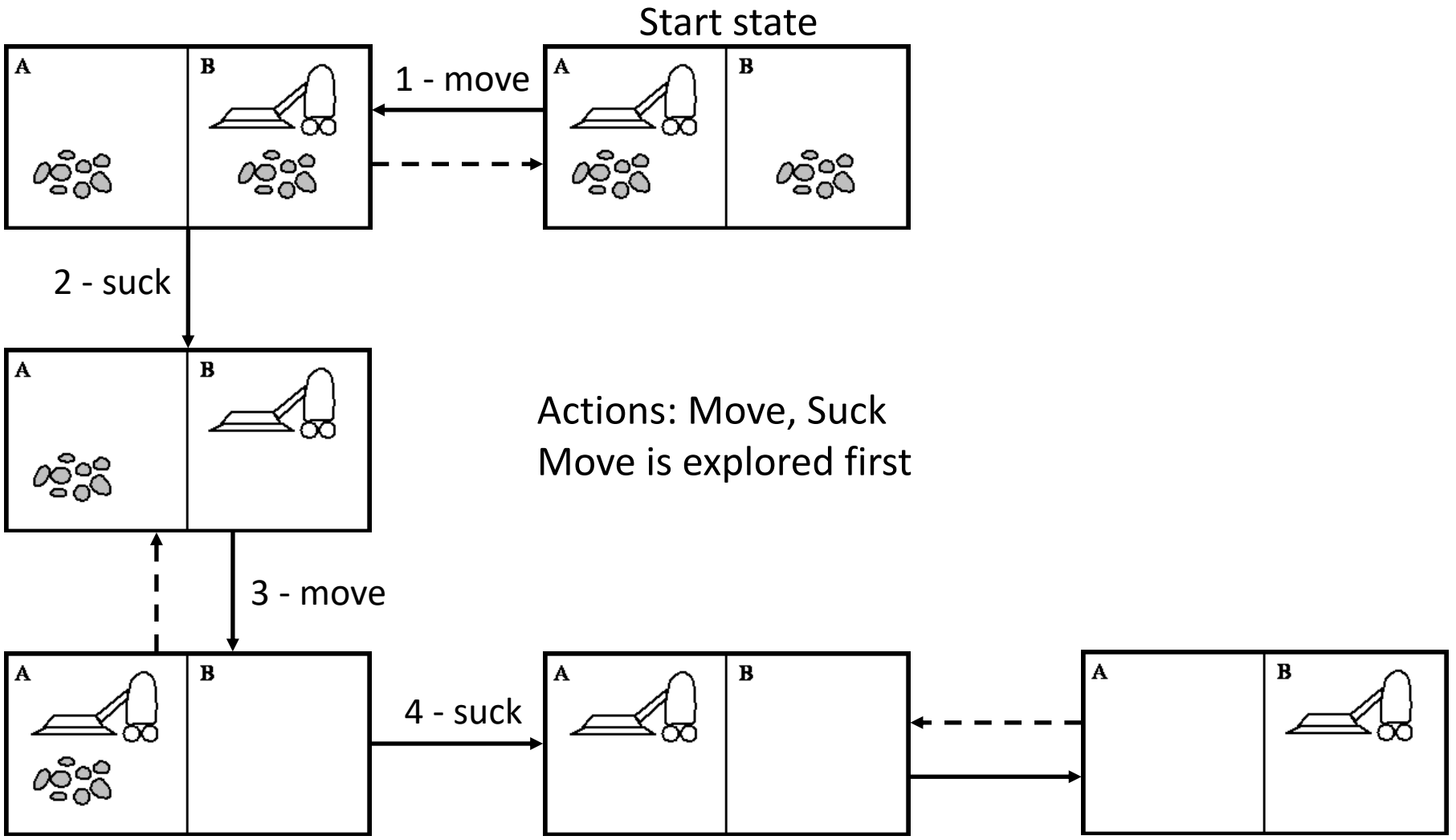
Found it! What if we kept going?

Example: DFS



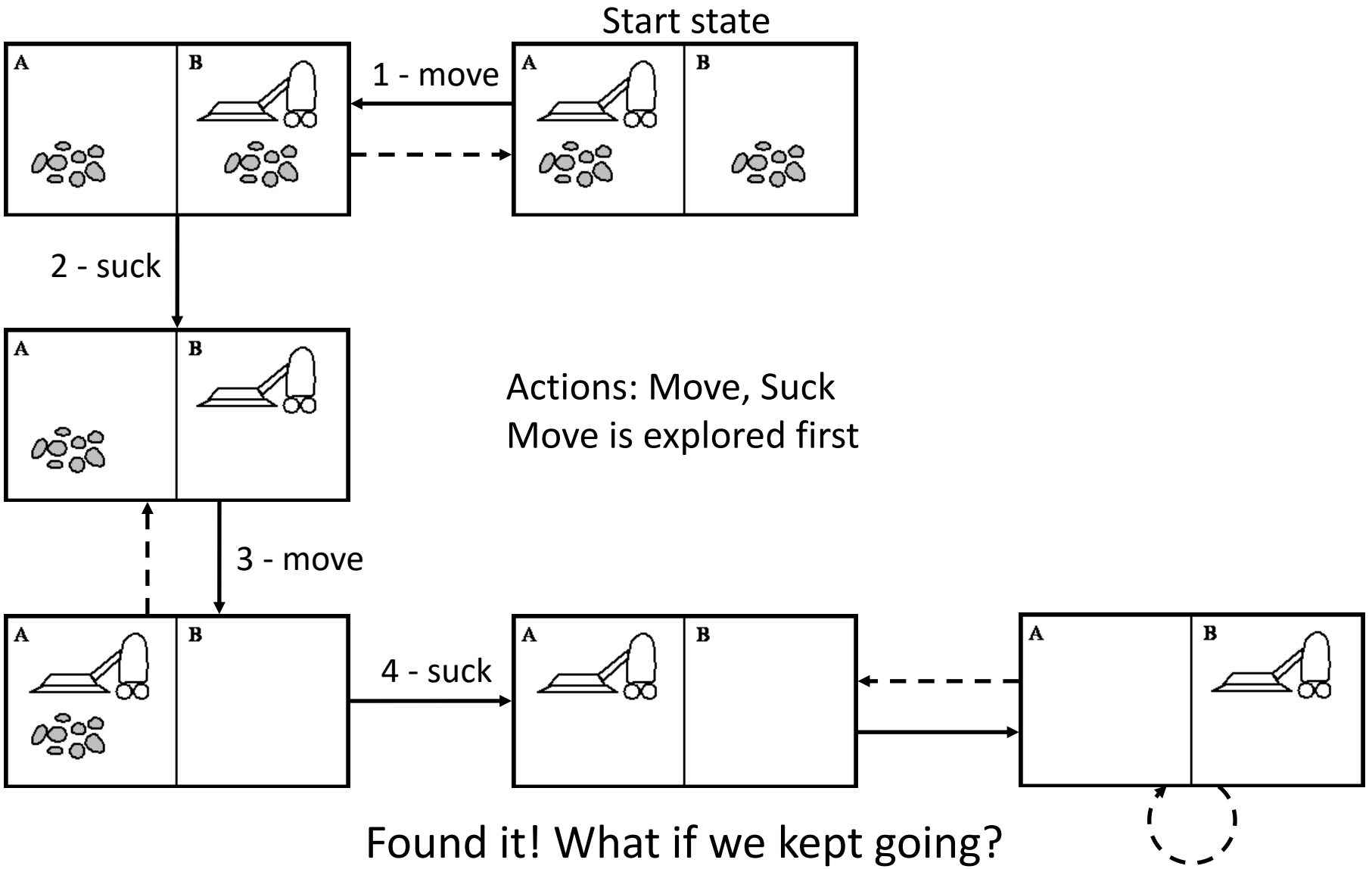
Found it! What if we kept going?

Example: DFS

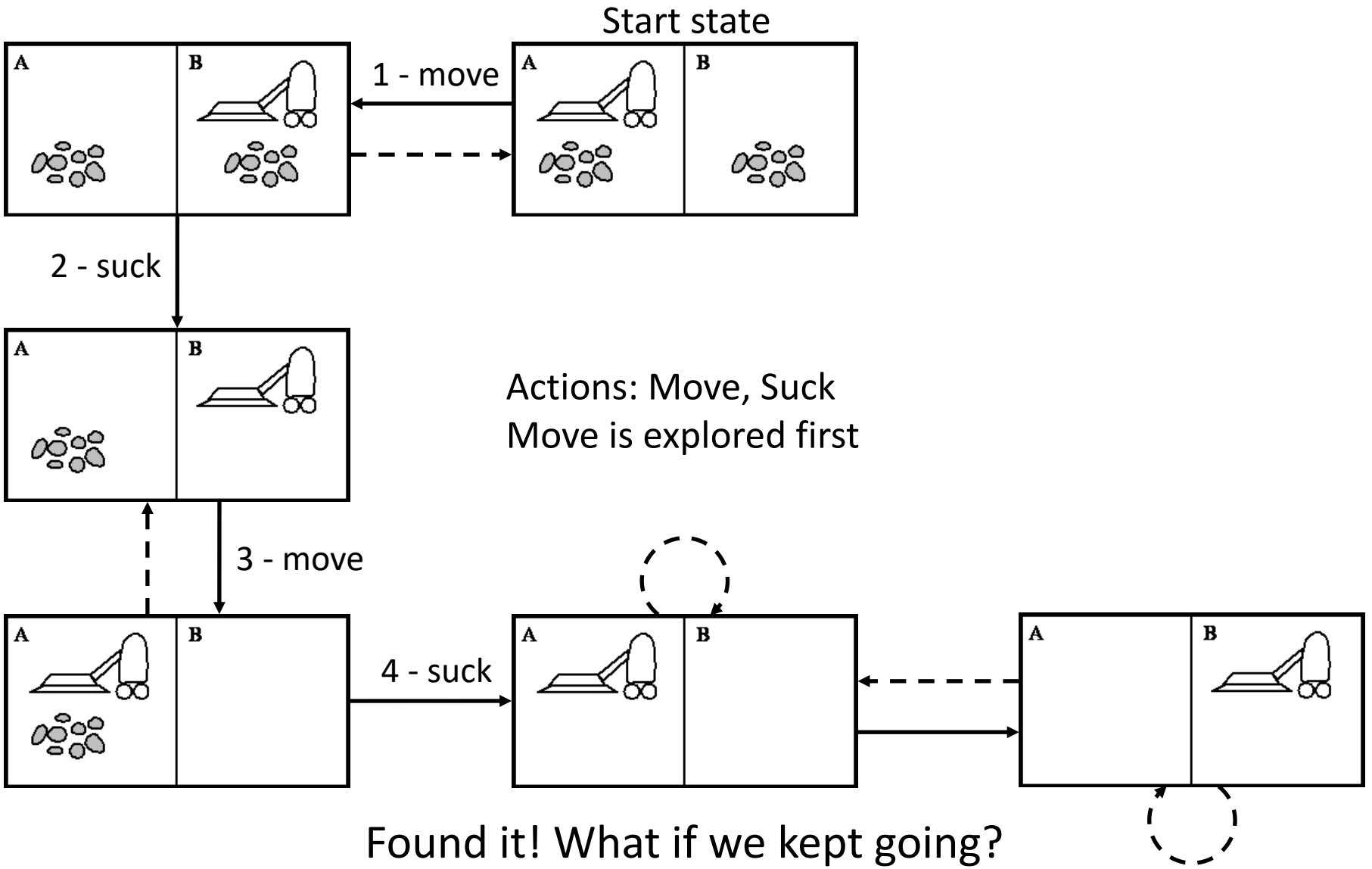


Found it! What if we kept going?

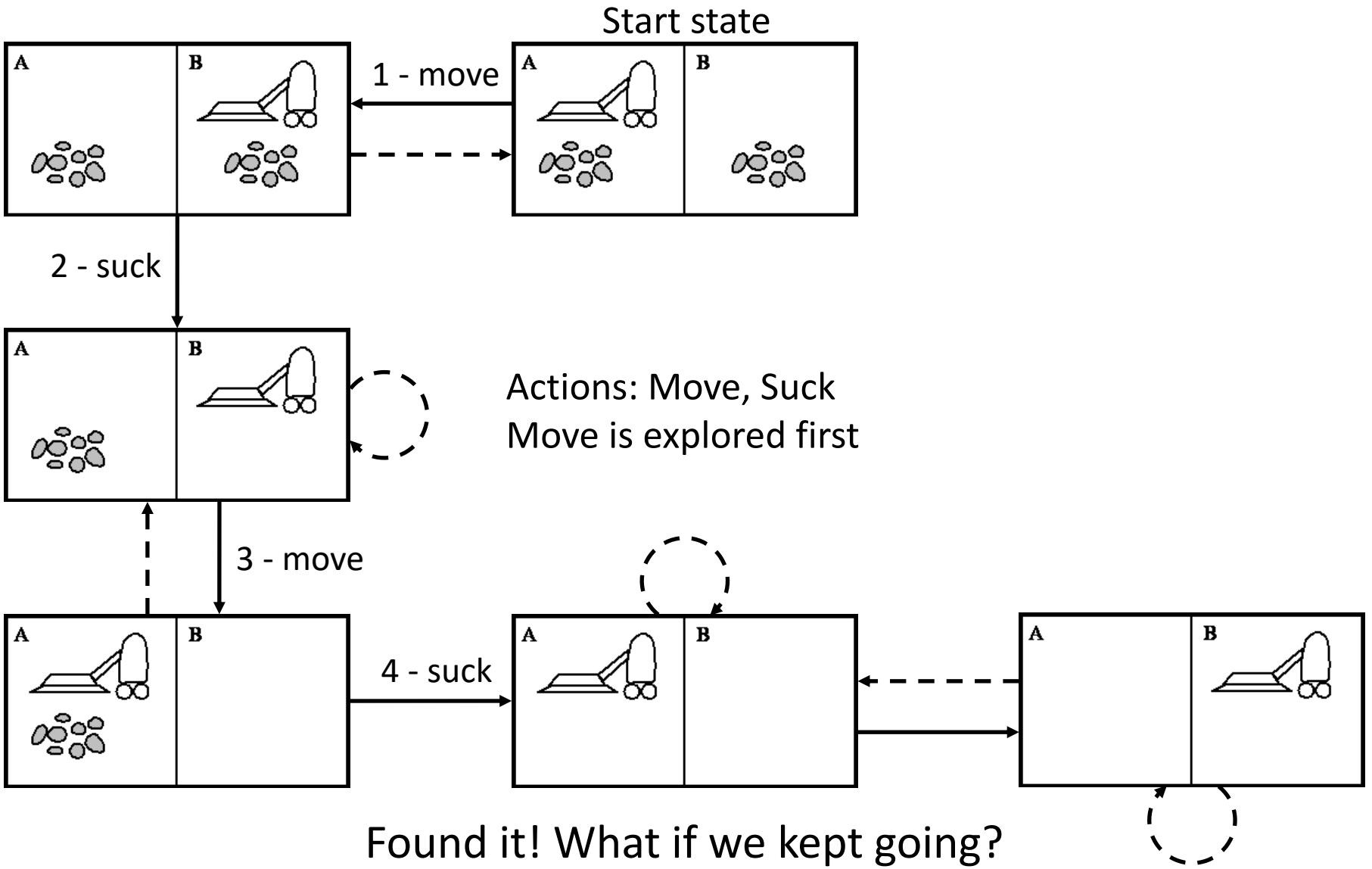
Example: DFS


Start state
A
B
1 - move
2 - suck
3 - move
4 - suck
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

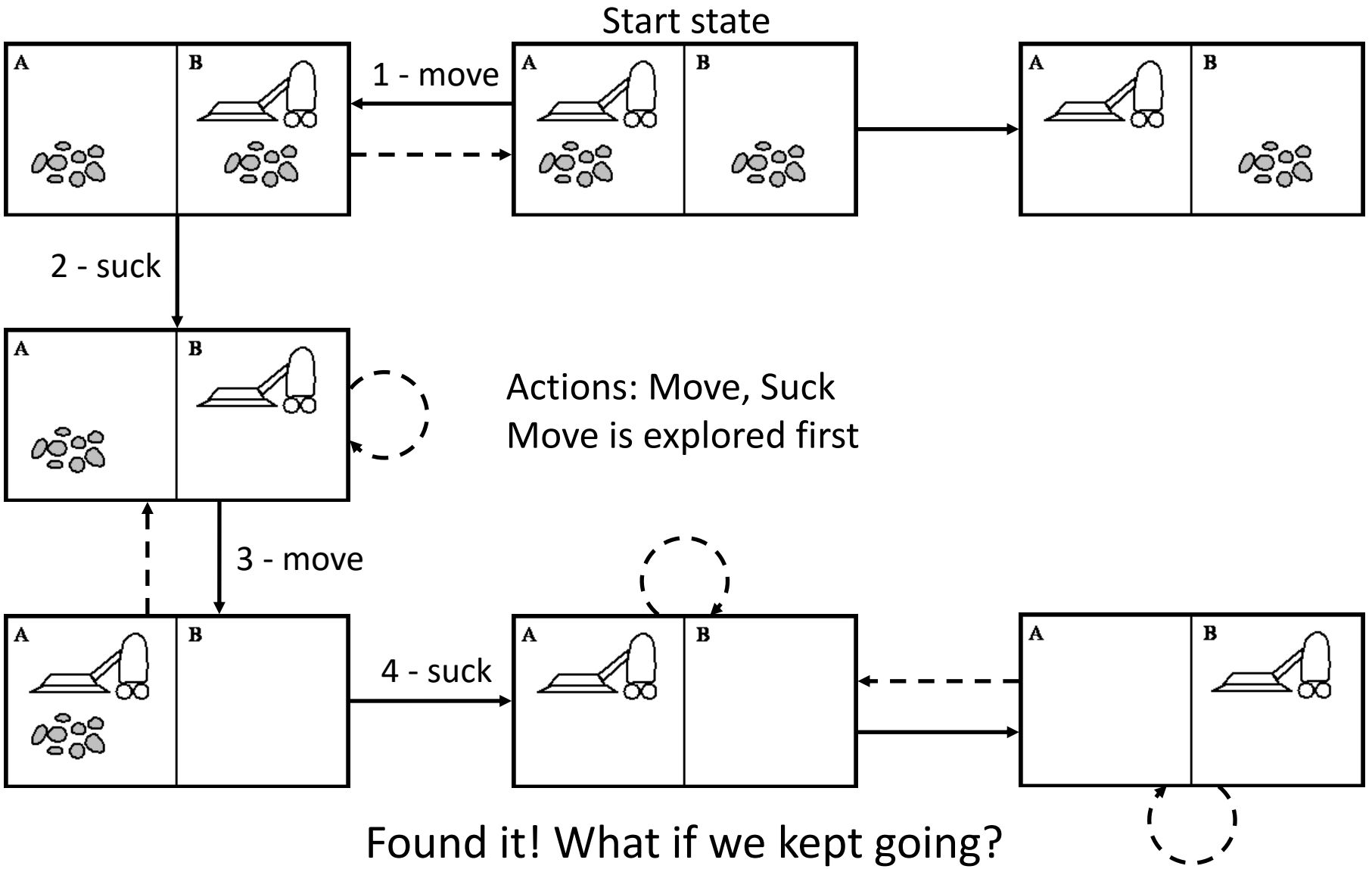
Example: DFS


Start state
A
B
1 - move
2 - suck
3 - move
4 - suck
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

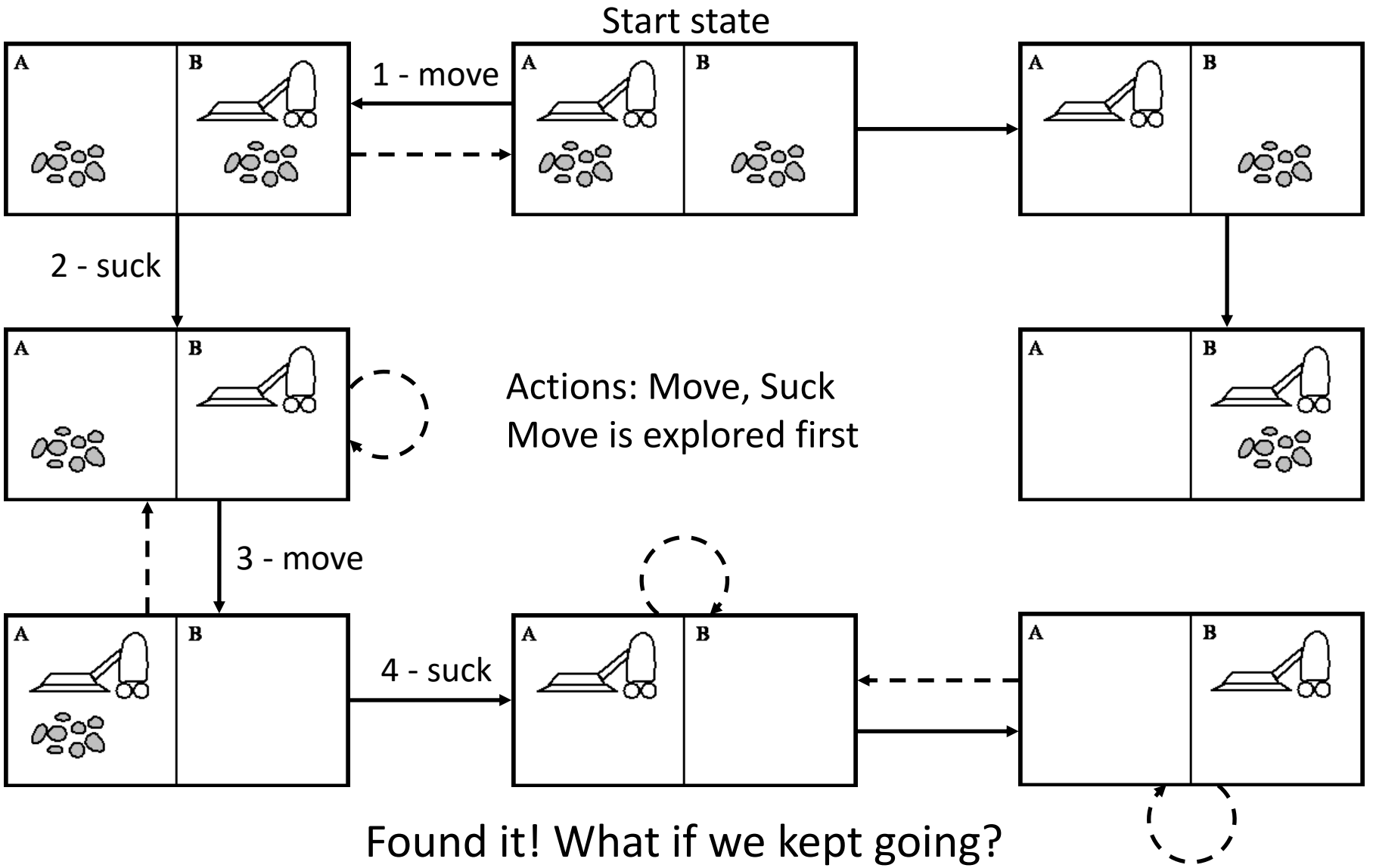
Example: DFS


Start state
A
B
1 - move
2 - suck
3 - move
4 - suck
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

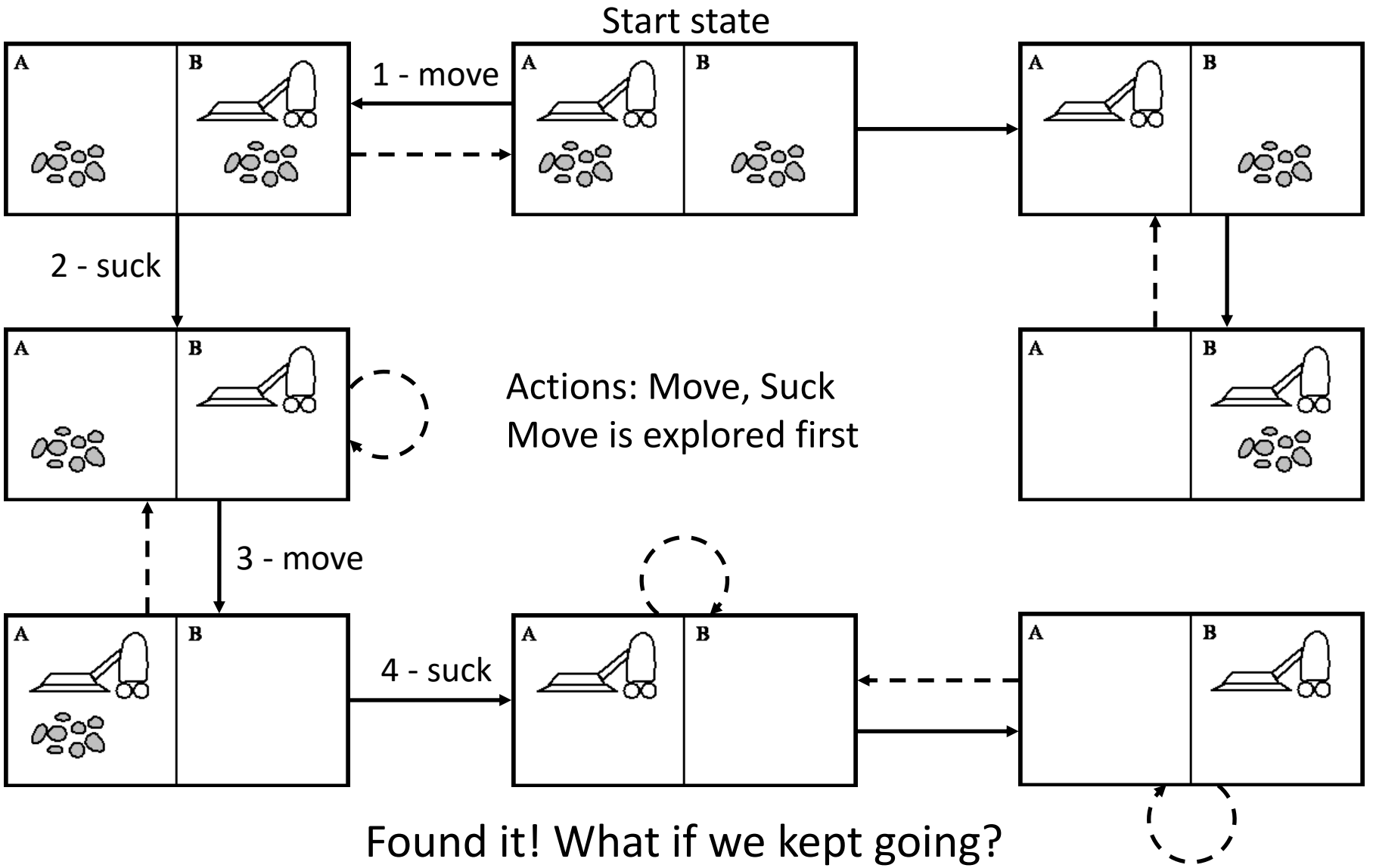
Example: DFS


Start state
A
B
1 - move
2 - suck
3 - move
4 - suck
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

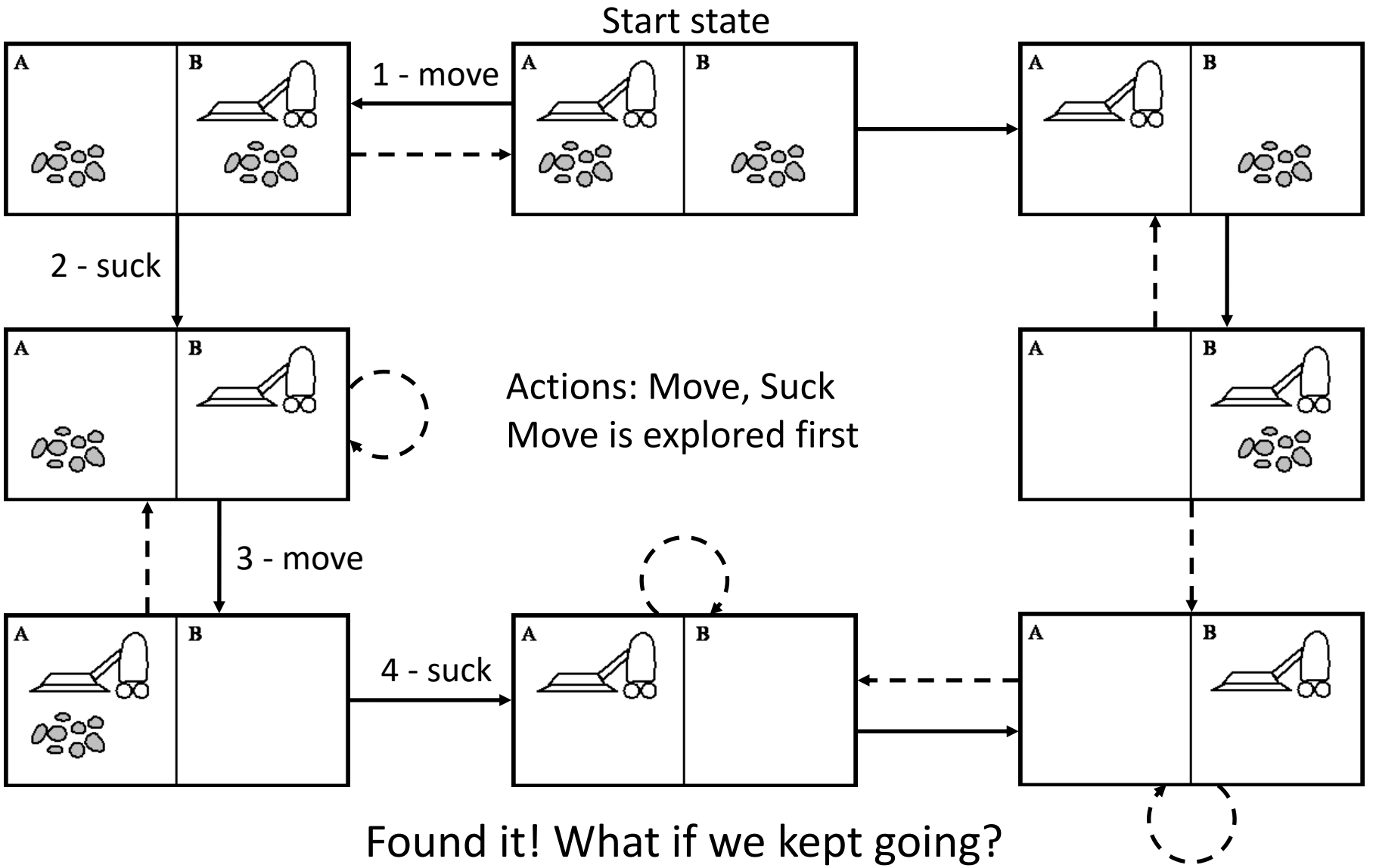
Example: DFS


Start state
1 - move
2 - suck
3 - move
4 - suck
A
B
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

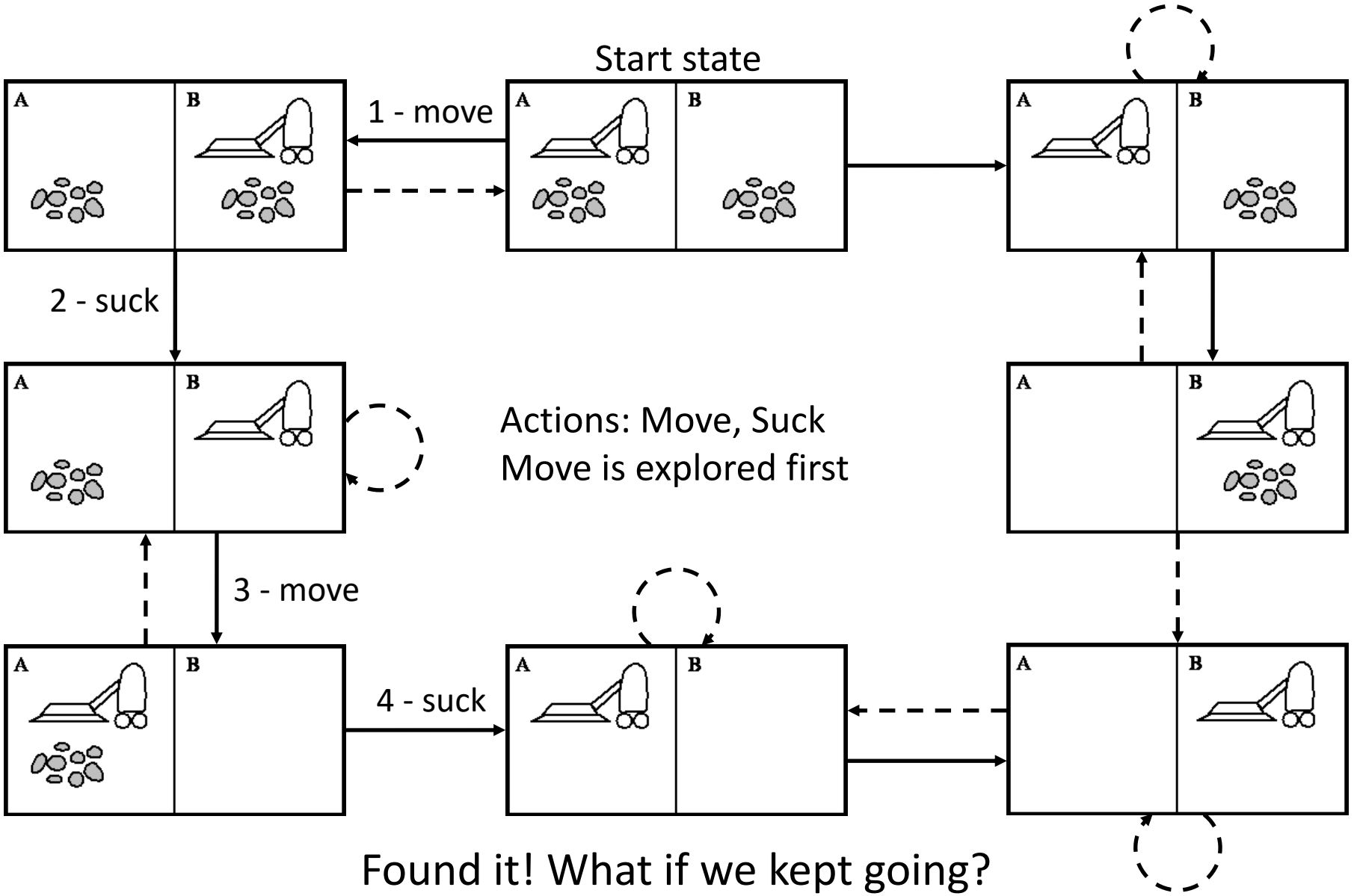
Example: DFS


Start state
1 - move
2 - suck
3 - move
4 - suck
A
B
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

Example: DFS

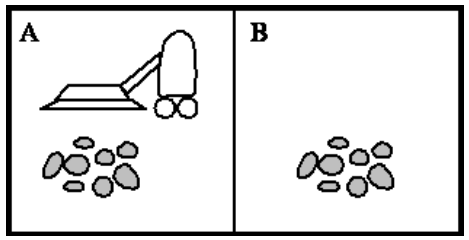

Start state
1 - move
2 - suck
3 - move
4 - suck
A
B
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

Example: DFS


Start state
1 - move
2 - suck
3 - move
4 - suck
A
B
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

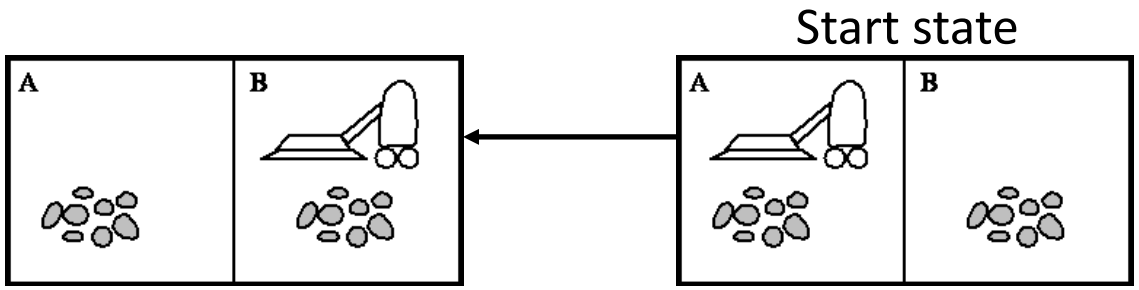
Example: BFS

Start state



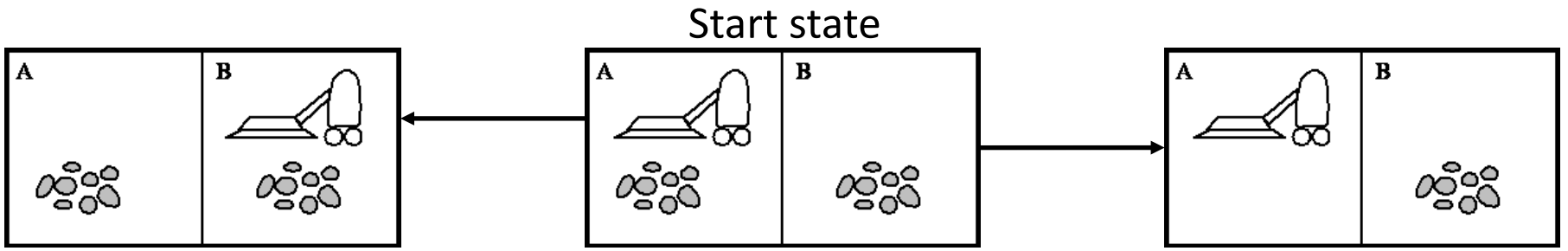
Actions: Move, Suck
Move is explored first

Example: BFS



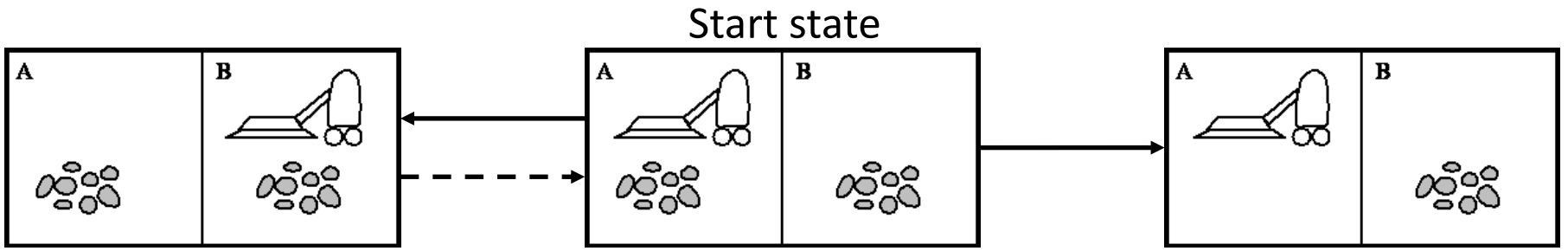
Actions: Move, Suck
Move is explored first

Example: BFS



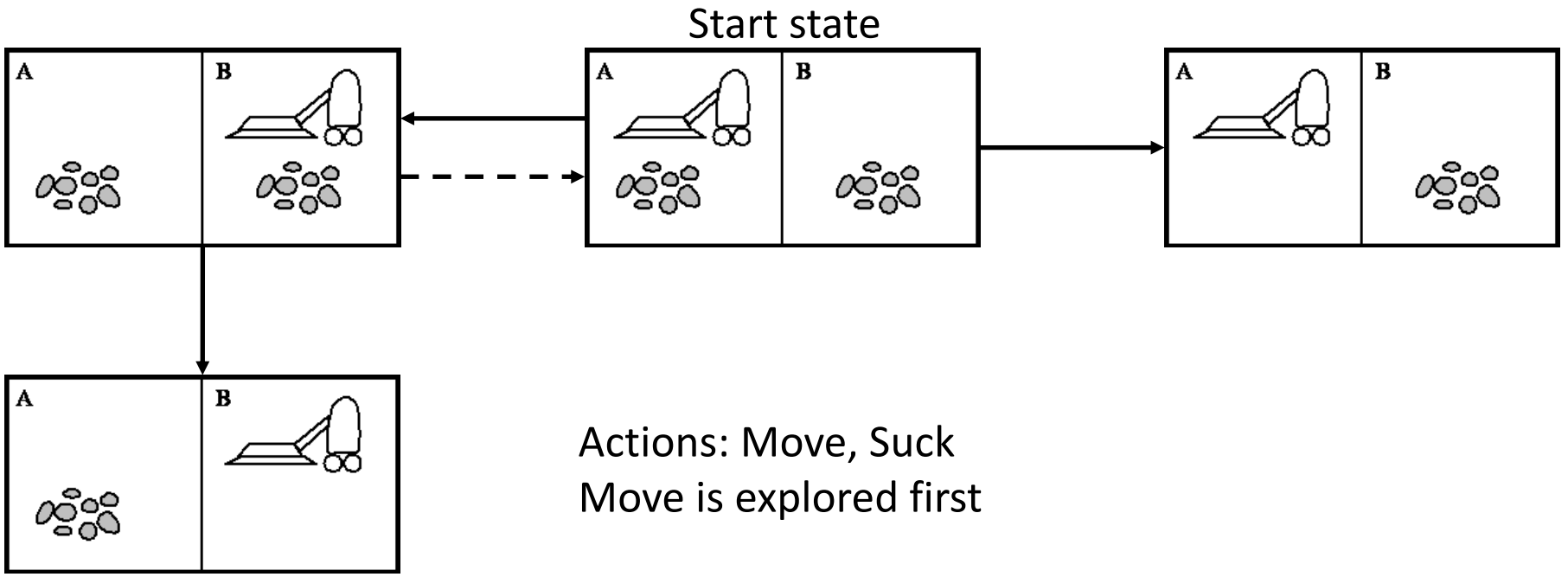
Actions: Move, Suck
Move is explored first

Example: BFS



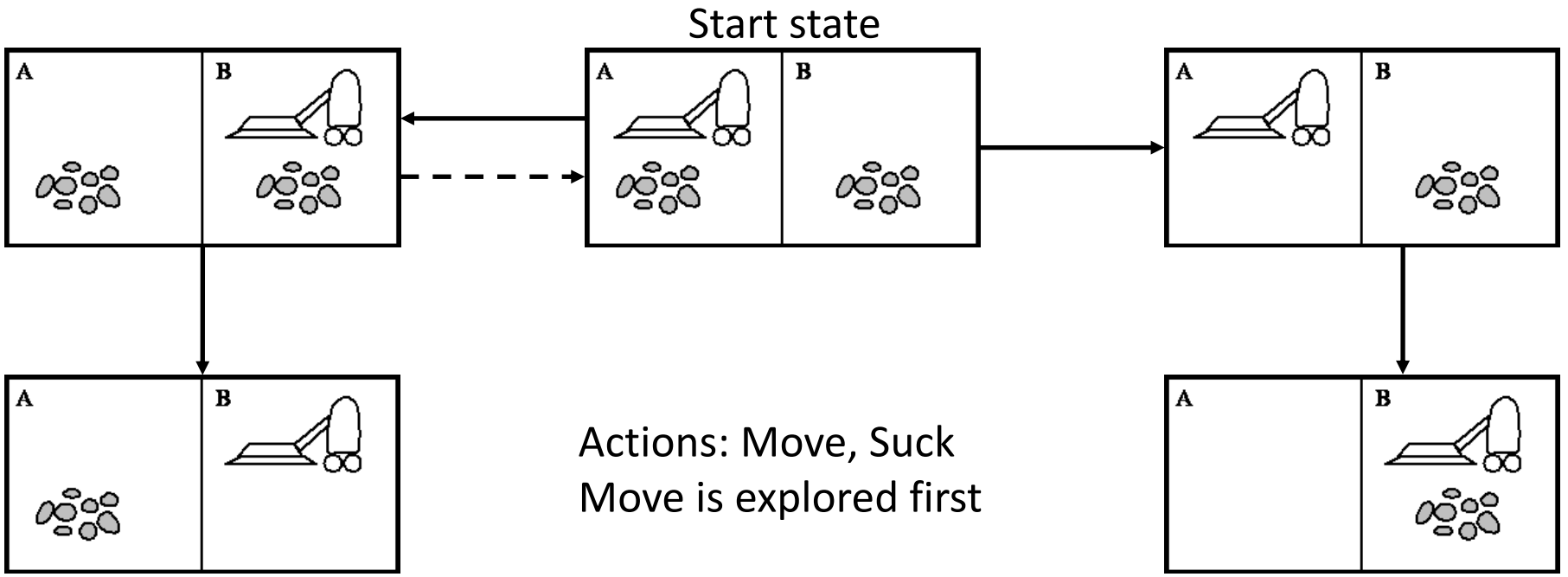
Actions: Move, Suck
Move is explored first

Example: BFS

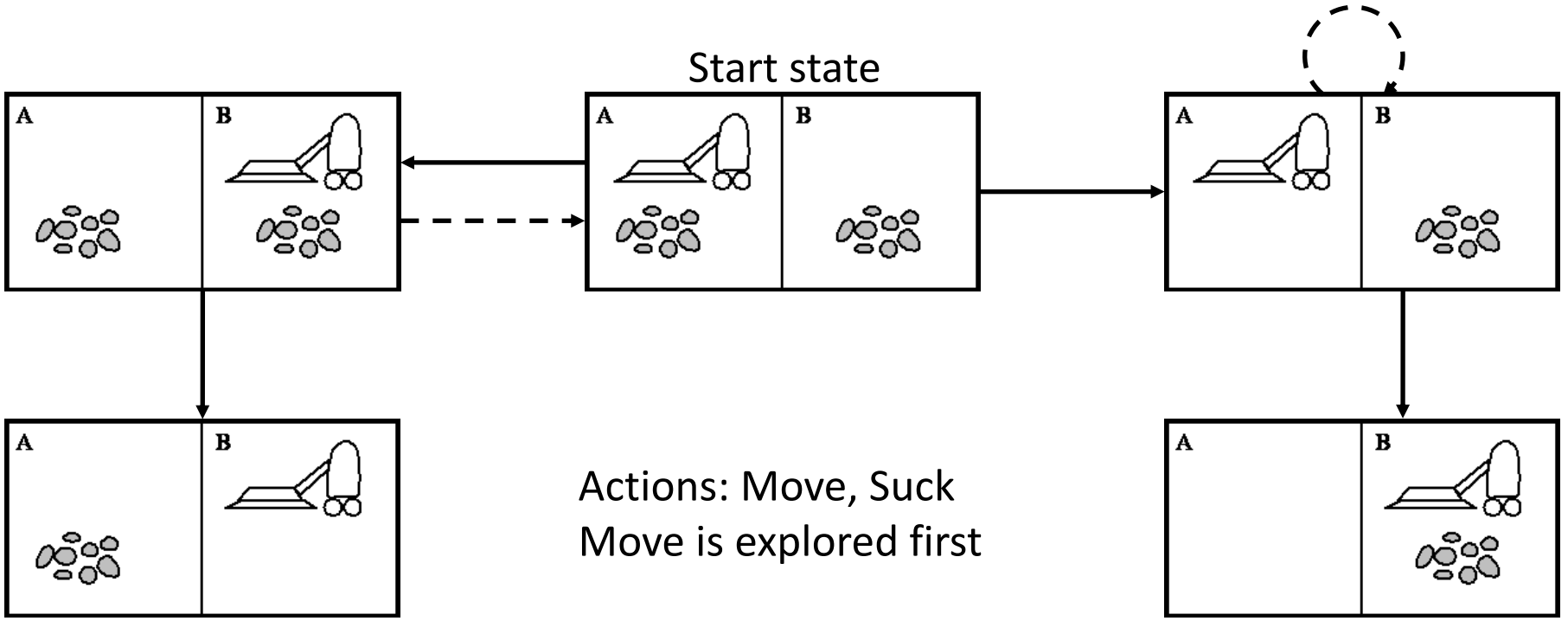


Actions: Move, Suck
Move is explored first

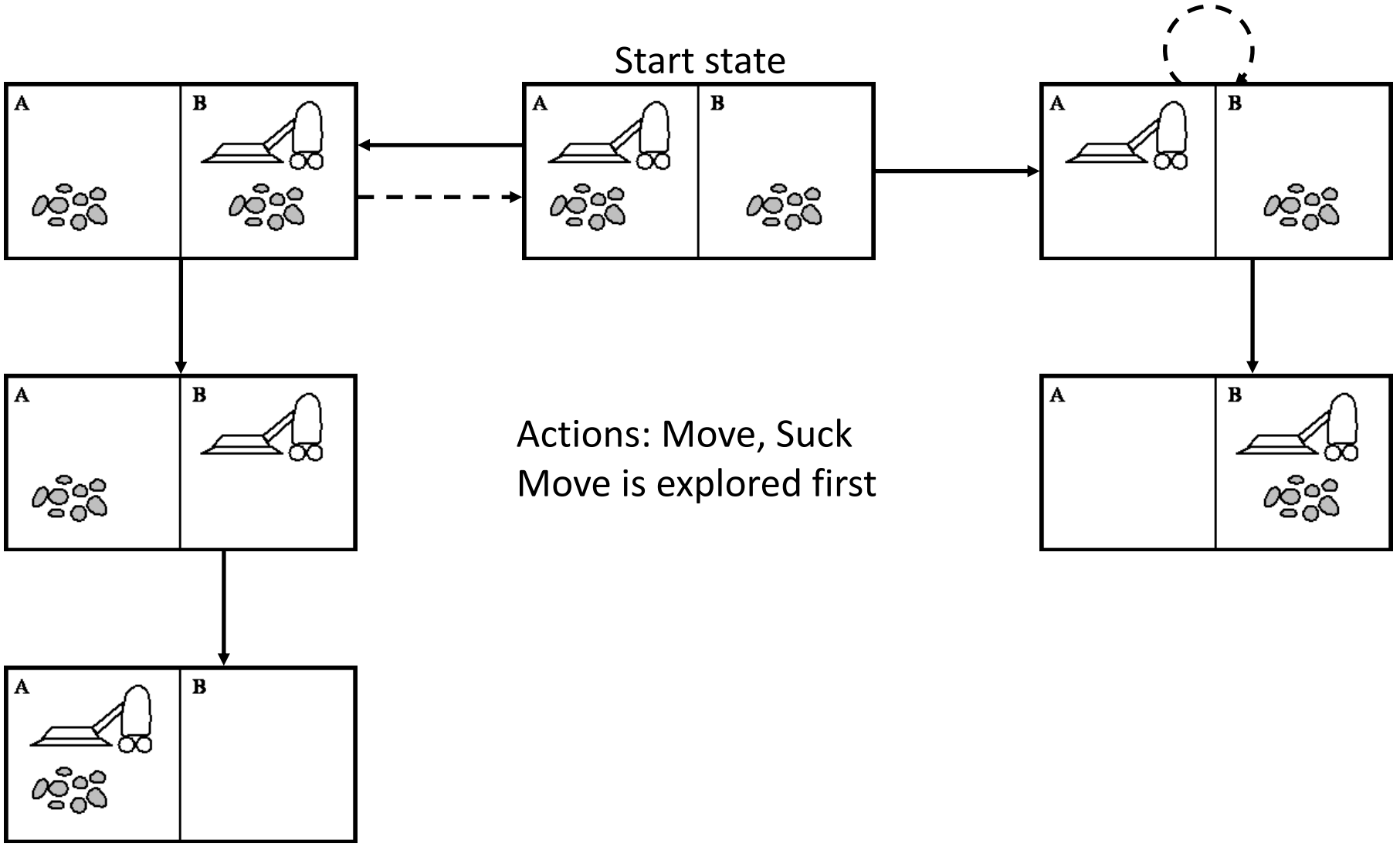
Example: BFS


Start state
A
B
A
B
A
B
A
B
A
B
Actions: Move, Suck
Move is explored first

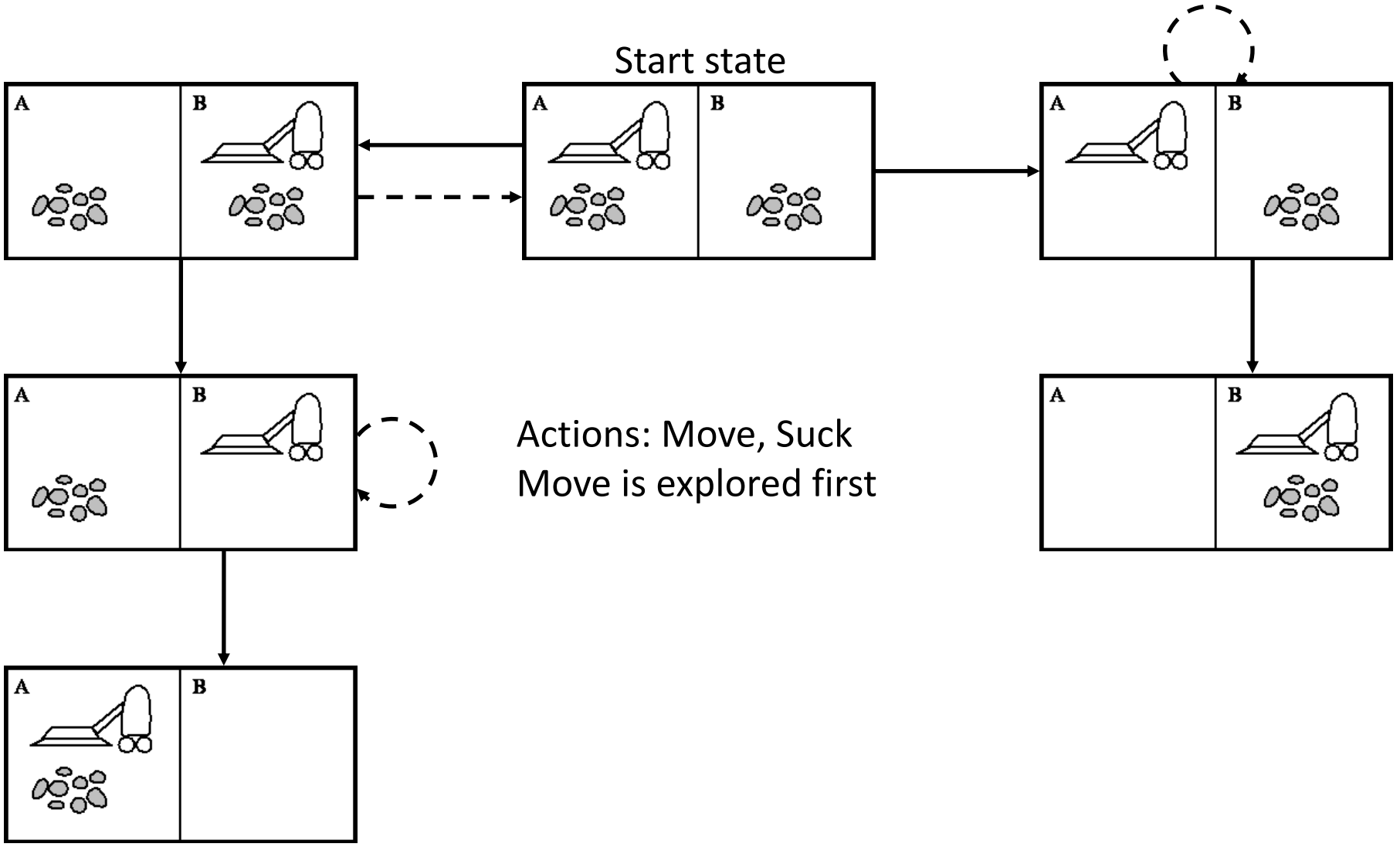
Example: BFS


Start state
A
B
A
B
A
B
A
B
A
B
Actions: Move, Suck
Move is explored first

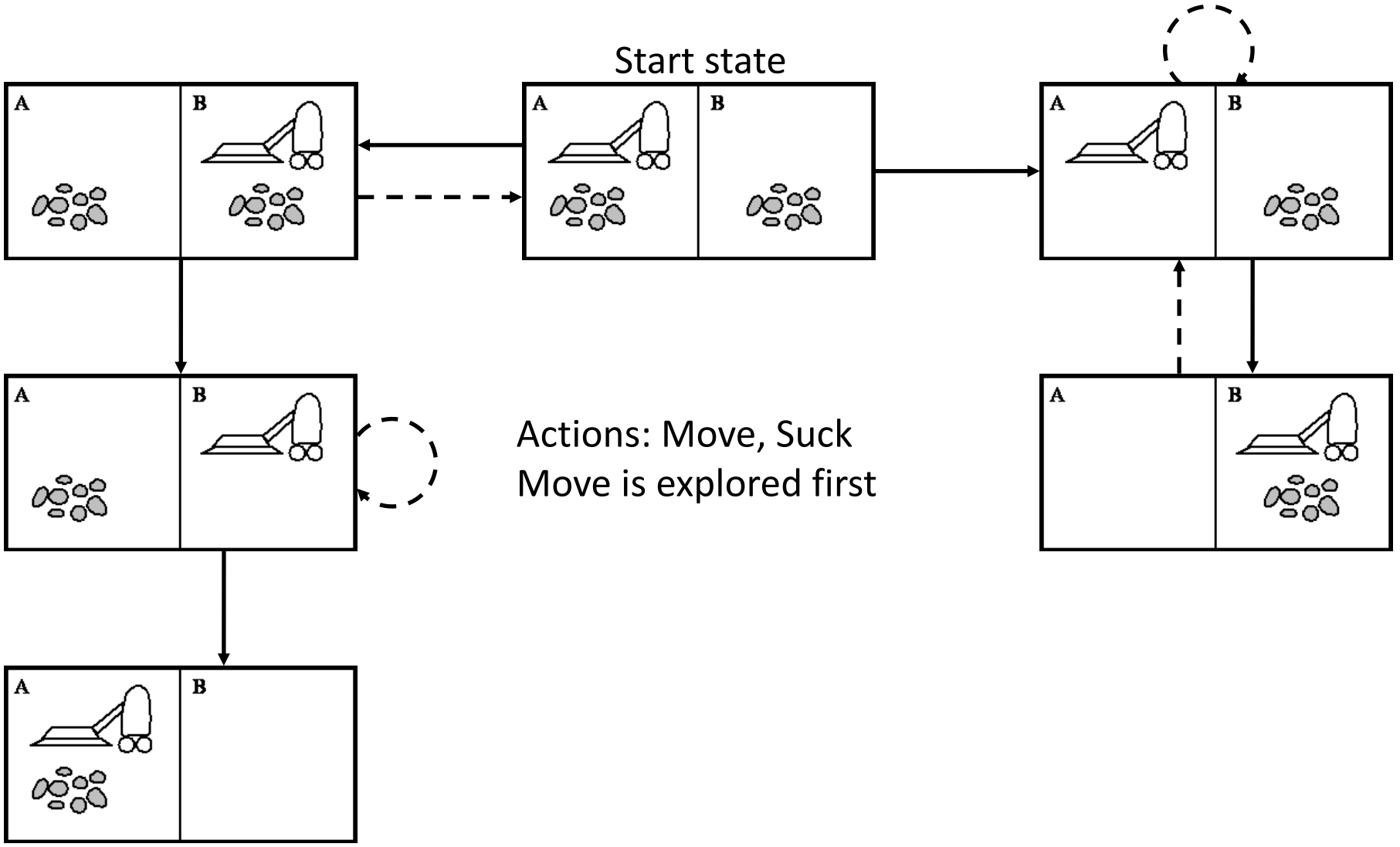
Example: BFS



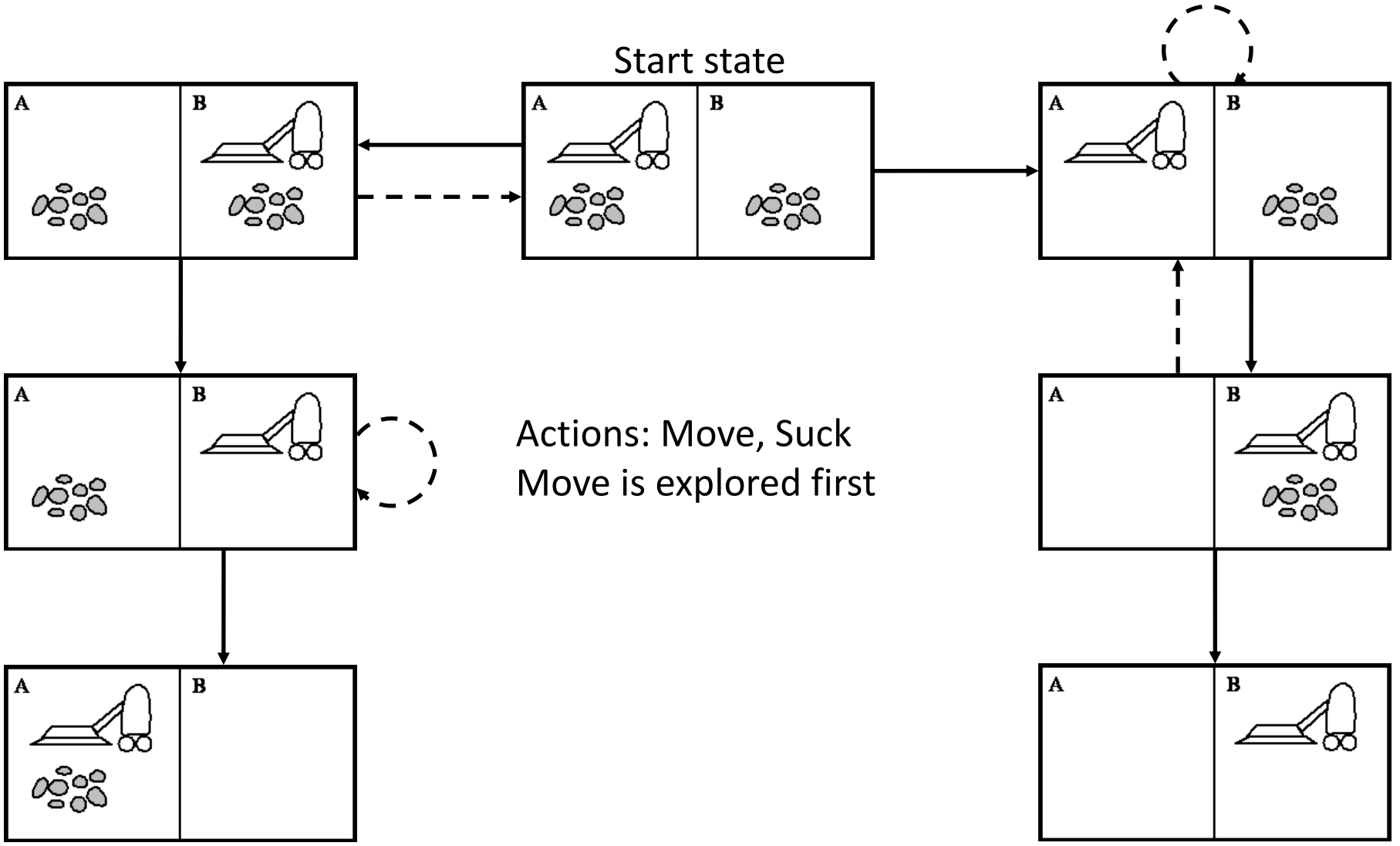
Example: BFS



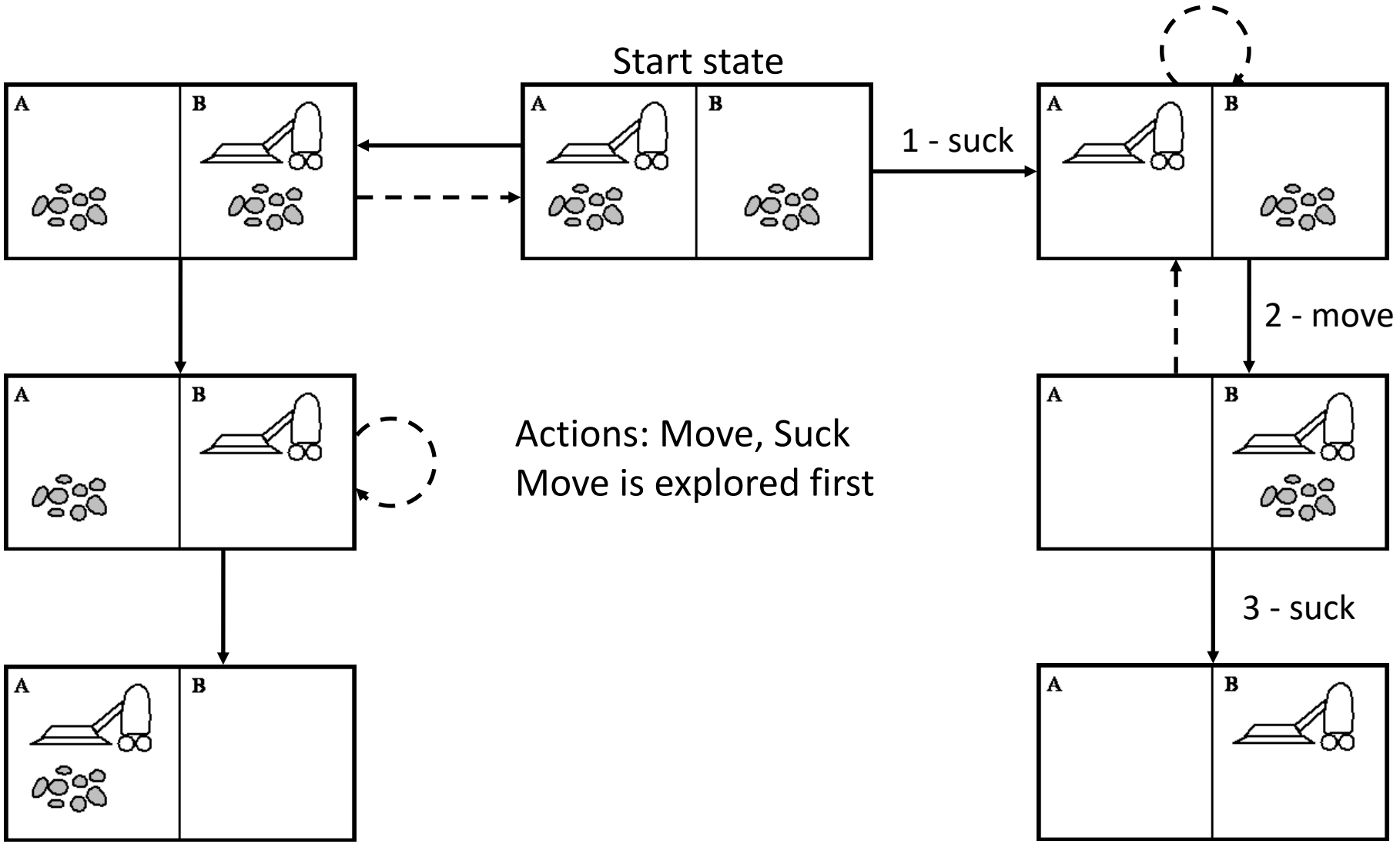
Example: BFS



Example: BFS

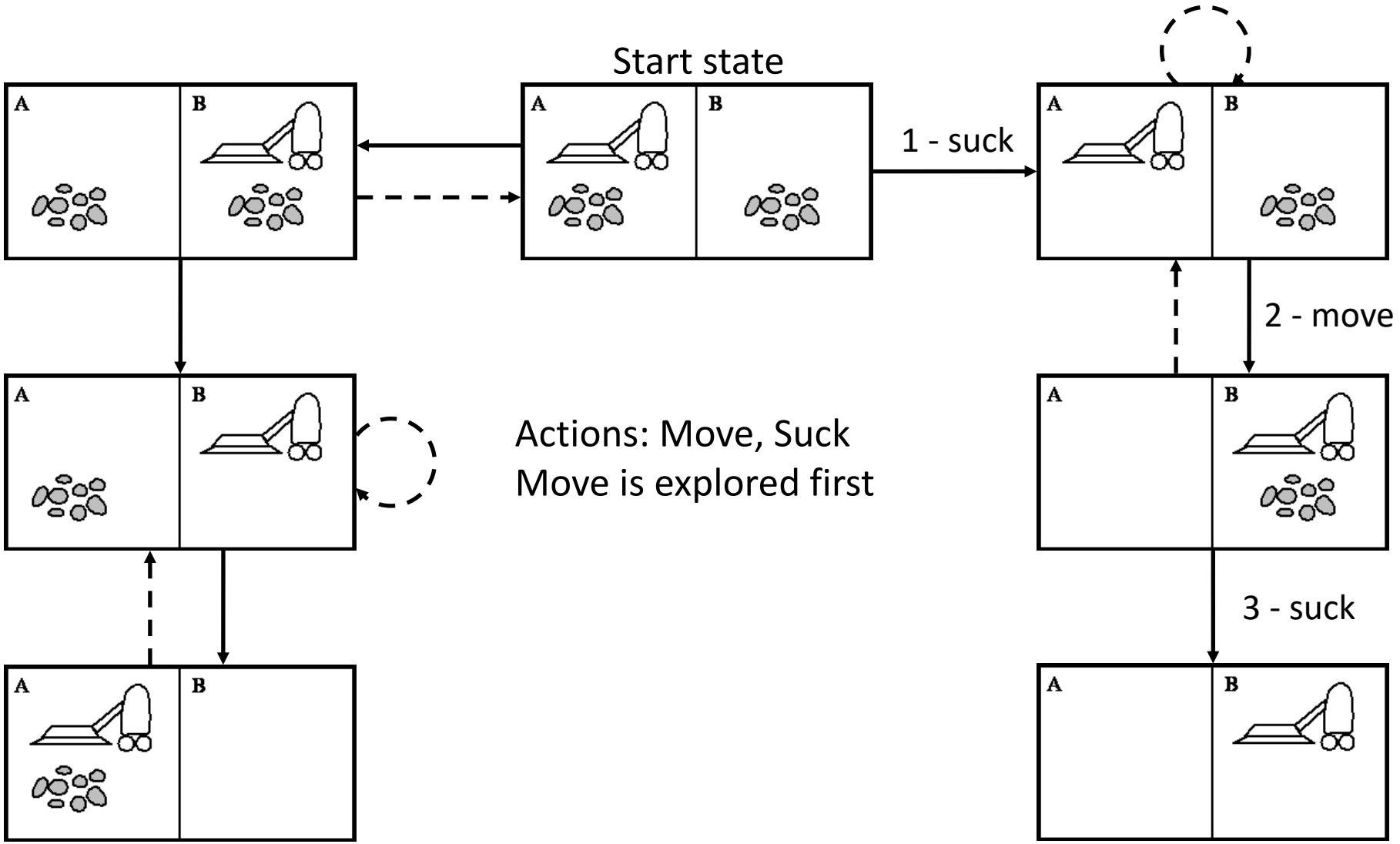


Example: BFS



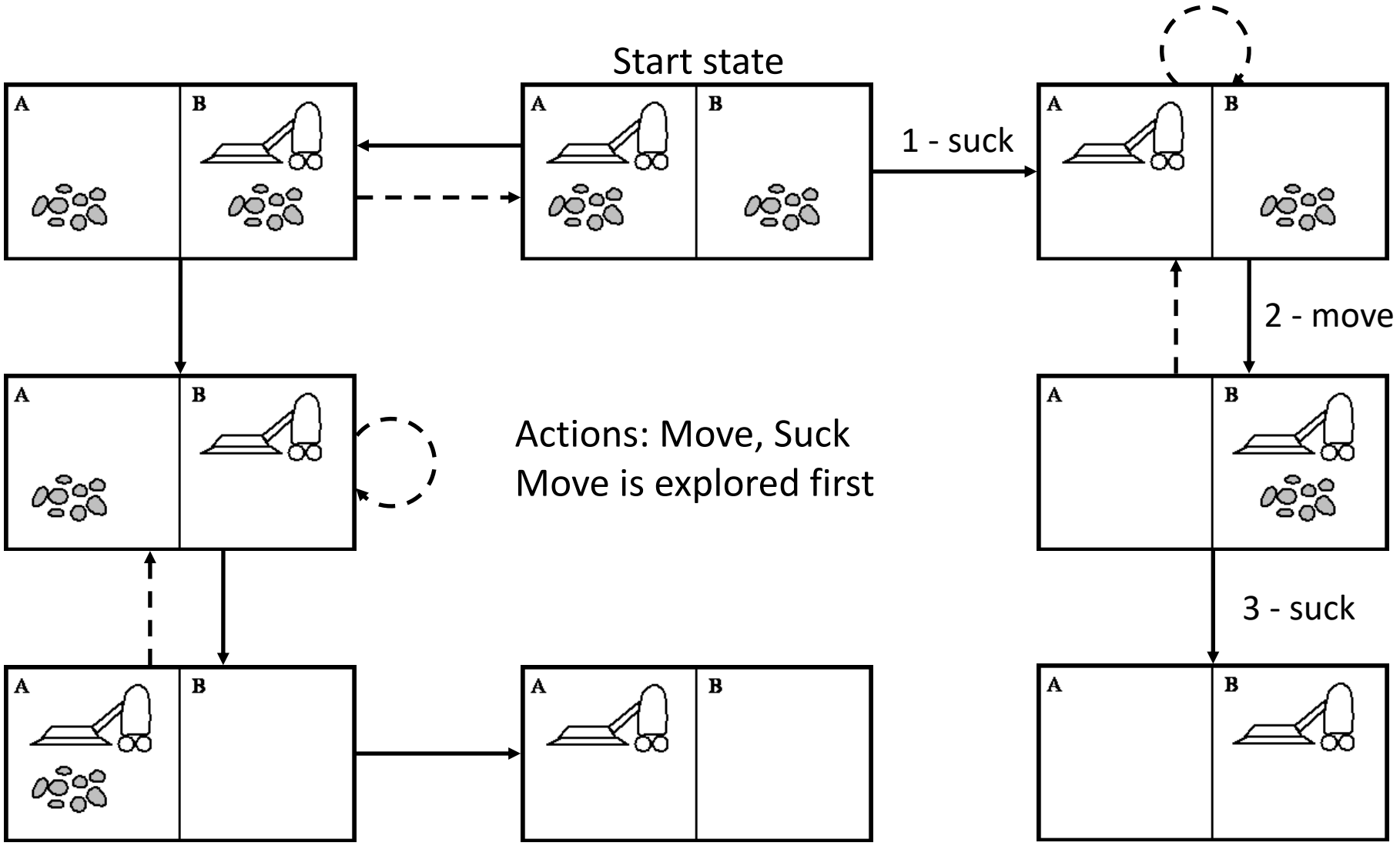
Found it! What if we kept going?

Example: BFS



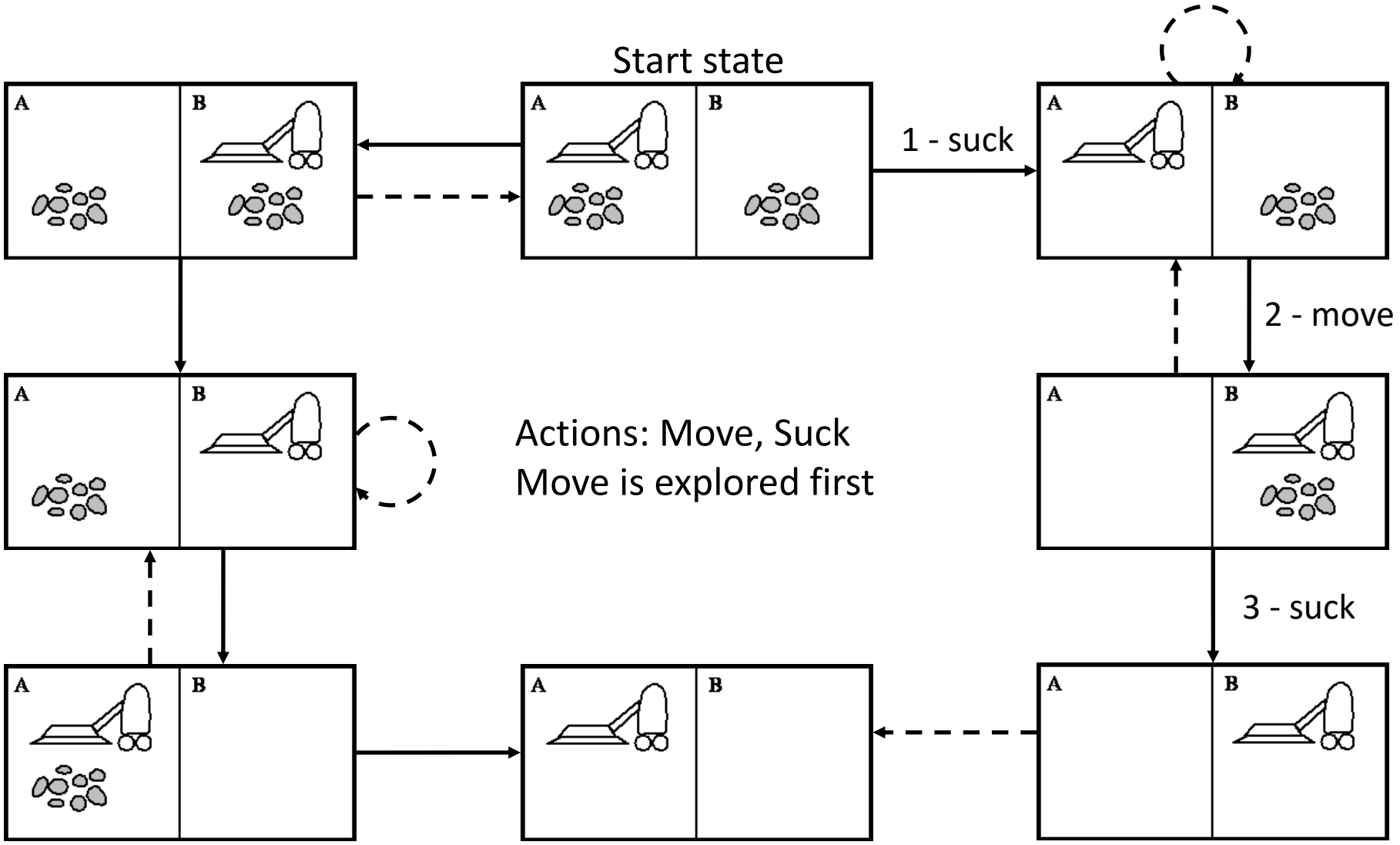
Found it! What if we kept going?

Example: BFS



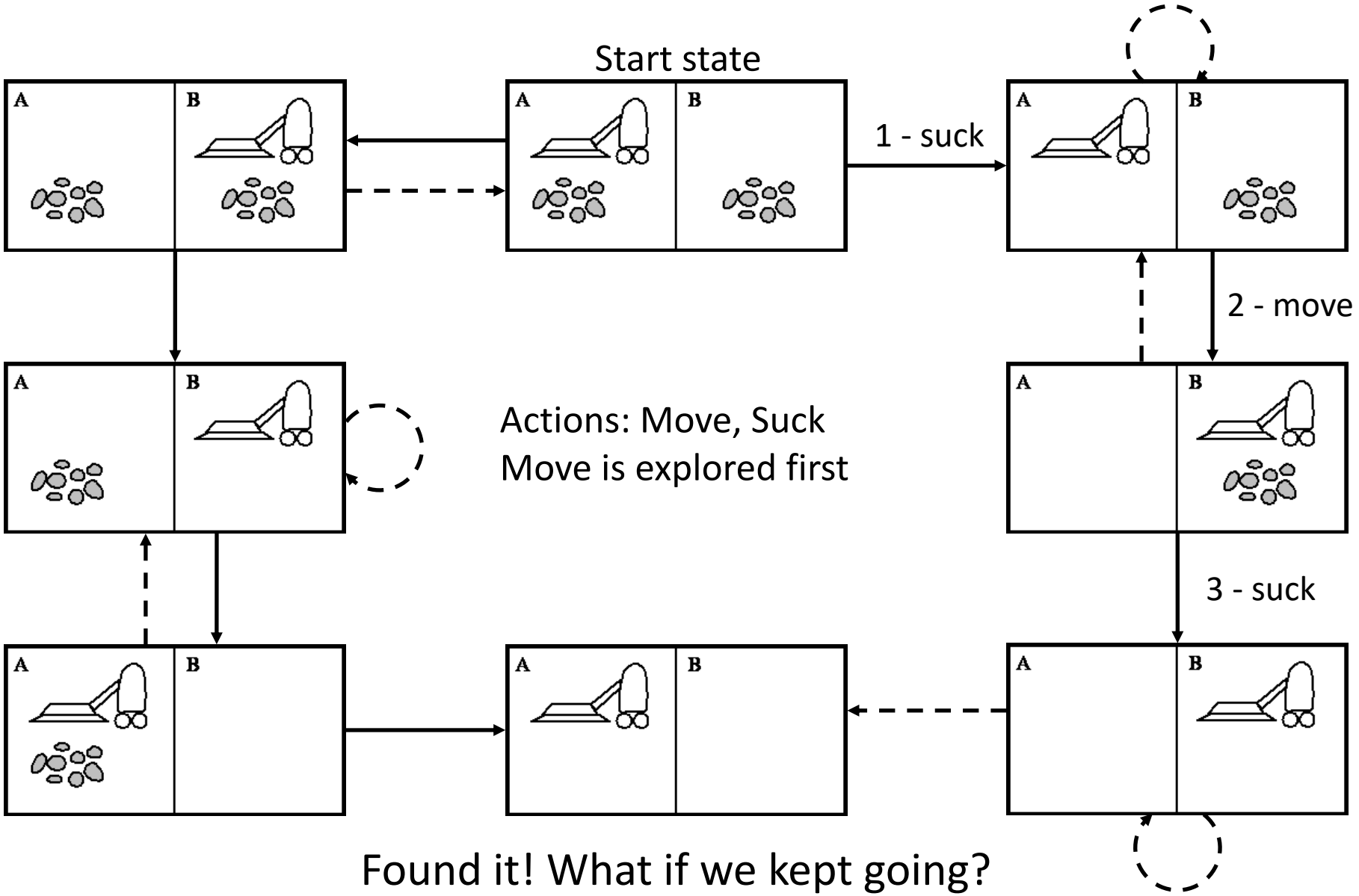
Found it! What if we kept going?

Example: BFS

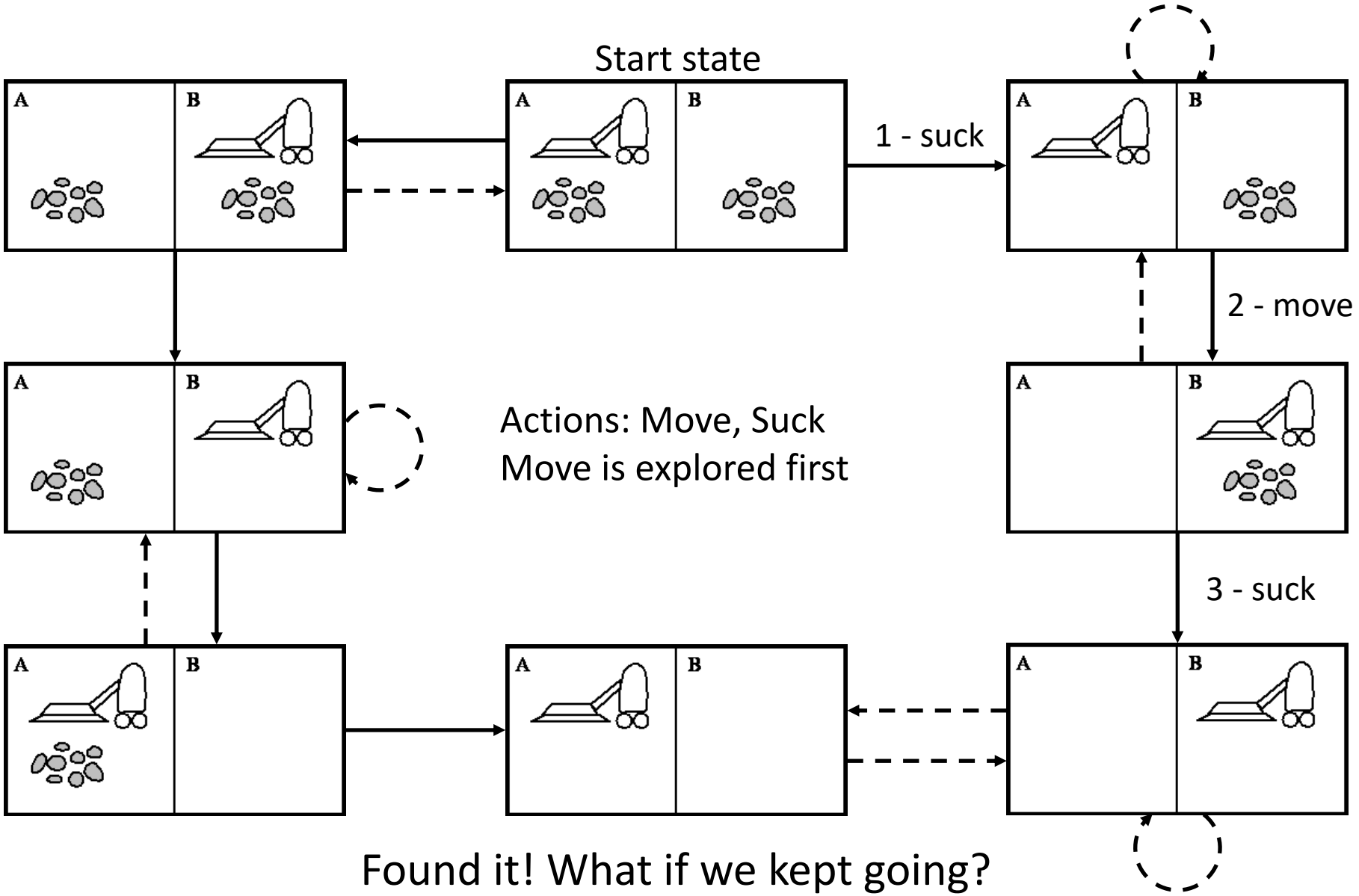


Found it! What if we kept going?

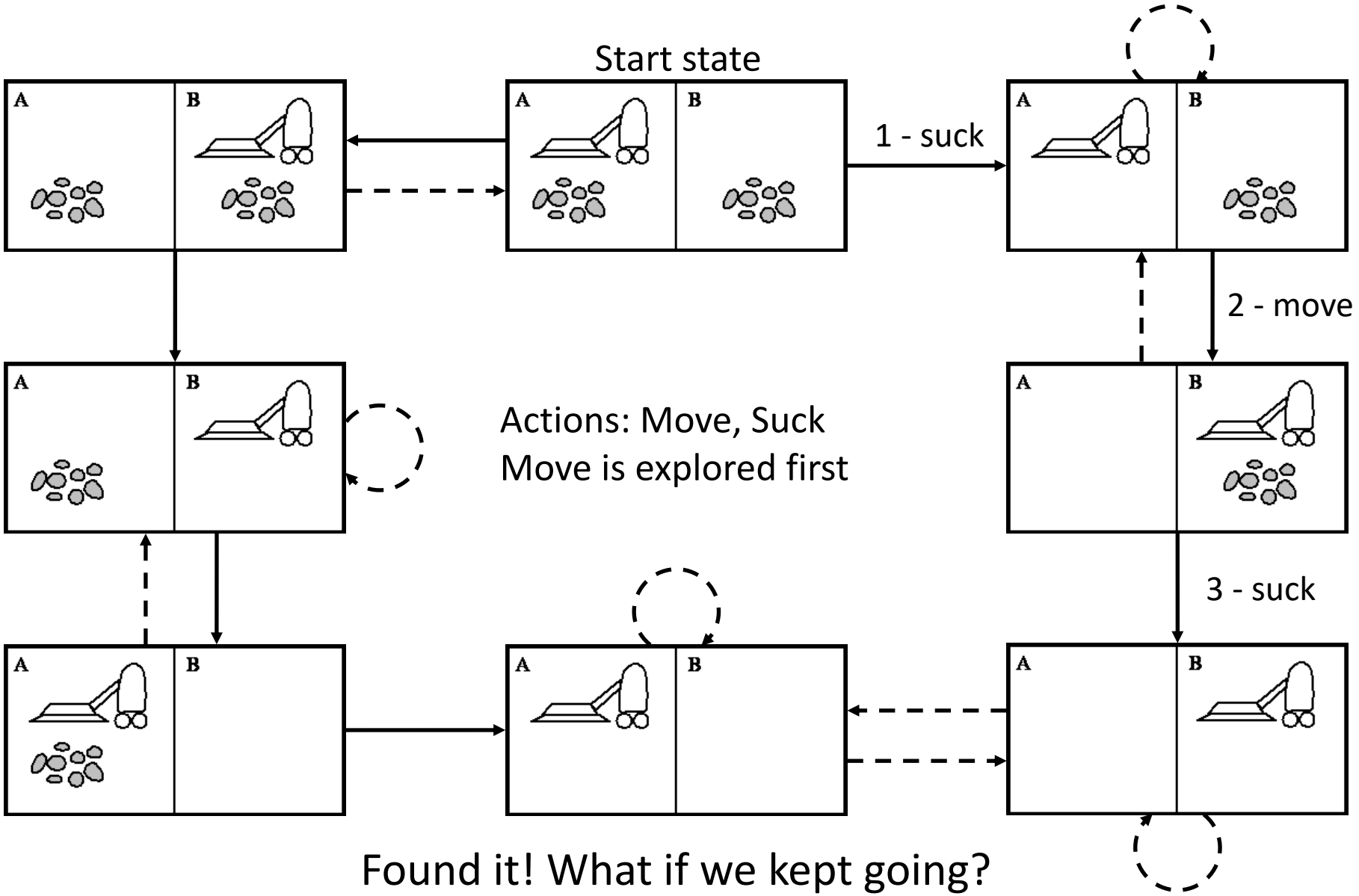
Example: BFS


Start state
1 - suck
2 - move
3 - suck
A
B
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

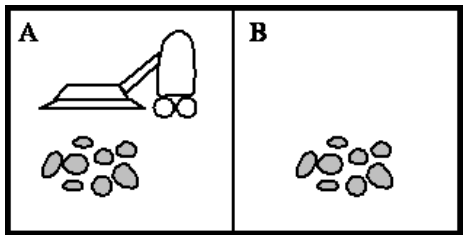
Example: BFS


Start state
1 - suck
2 - move
3 - suck
A
B
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

Example: BFS


Start state
1 - suck
2 - move
3 - suck
A
B
Actions: Move, Suck
Move is explored first
Found it! What if we kept going?

Comparison



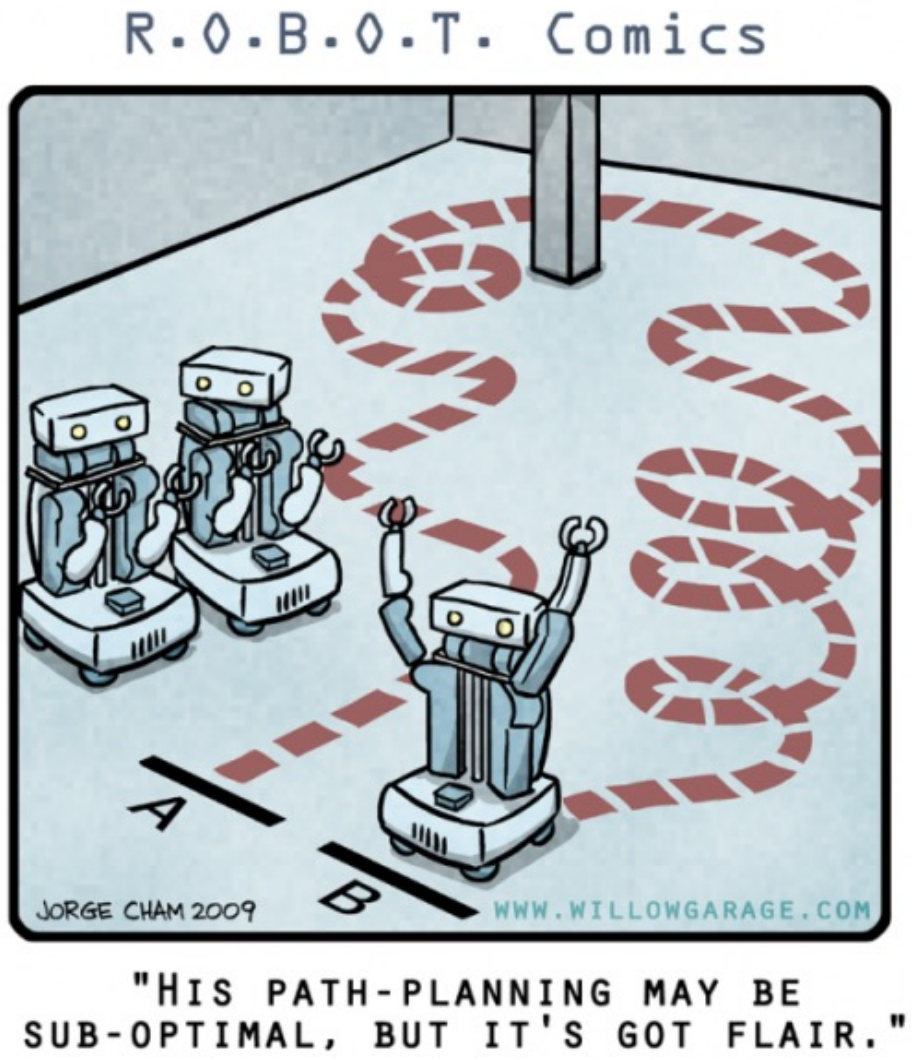
Solutions

- DFS: Move-Suck-Move-Suck – 4 actions
- BFS: Suck-Move-Suck – 3 actions

Number of expanded states during search:

- DFS: 4
- BFS: 7

What about costs?



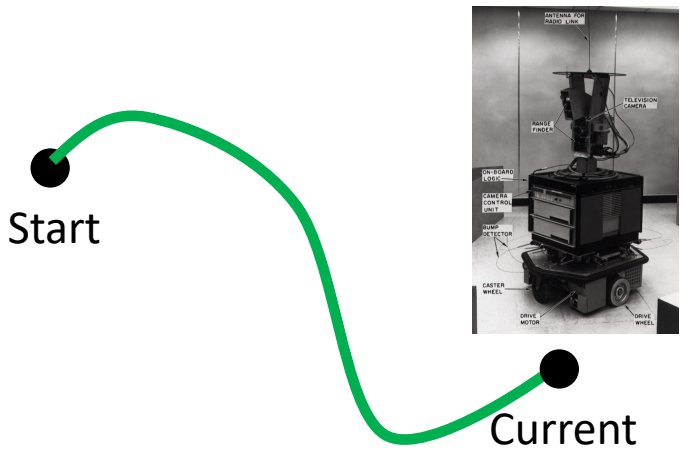
Cost of Actions

- Cost of taking the action a at state s: $d(s, a) = c,\ \ c \in R$
- Let $\sigma = \{(s_0, a_0), (s_1, a_1), \dots, s_n\}$ be a state-action path (no action at the last state)
- Path cost from start to the last state:
  $J(\sigma) = \sum_{i=0}^{n-1} d(s_i, a_i)$
- Let's call it g(s) for succinctness, where s is s_n

Cost of State Path

- Cost of going from state s_i to s_{i+1}: $d(s_i, s_{i+1}) = c, i \geq 0,\ c \in R$
- Let $\sigma = \{s_0, s_1, \ldots, s_n\}$ be a state path
- Path cost from start to the last state:
  $J(\sigma) = \sum_{i=0}^{n-1} d(s_i, s_{i+1})$
- Let's call it g(s) for succinctness, where s is s_n

Cost



Path cost: $g(current)$

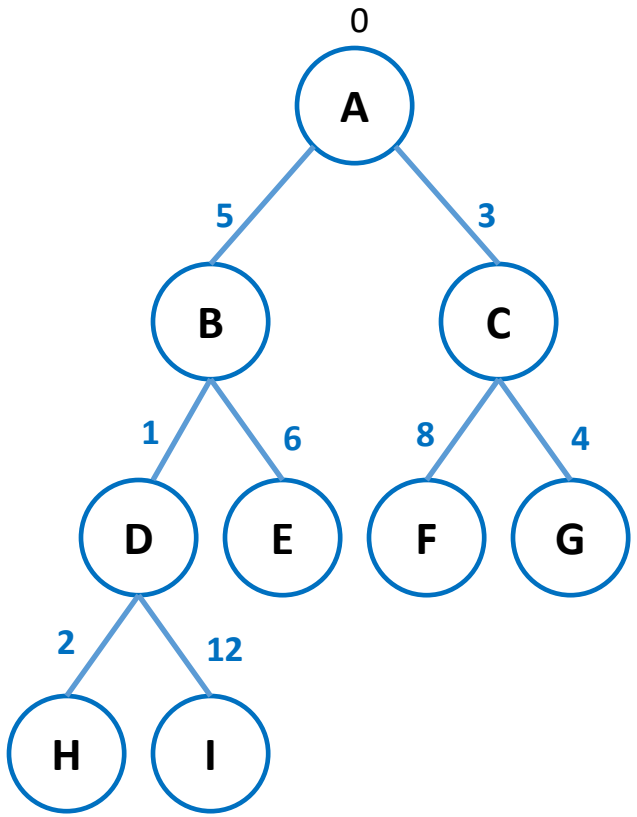
Uniform Cost Search

Idea: expand the lowest cost node first

Implementation: Frontier is a priority que based on cumulative cost

Uninformed b/c costs are part of the problem!

Equivalent to BFS if all actions have the same cost



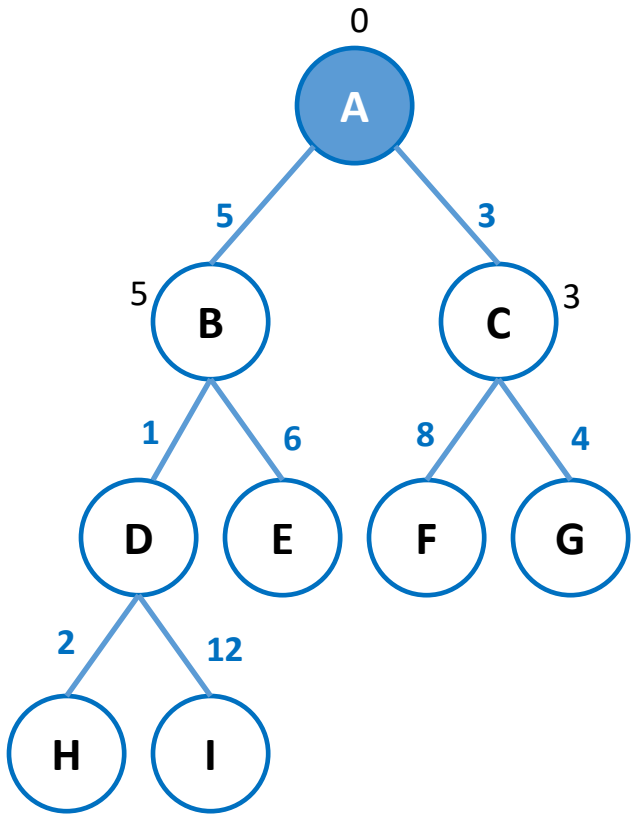
Uniform Cost Search

Idea: expand the lowest cost node first

Implementation:
Frontier is a priority que based on cumulative cost

Uninformed b/c costs are part of the problem!

Equivalent to BFS if all actions have the same cost



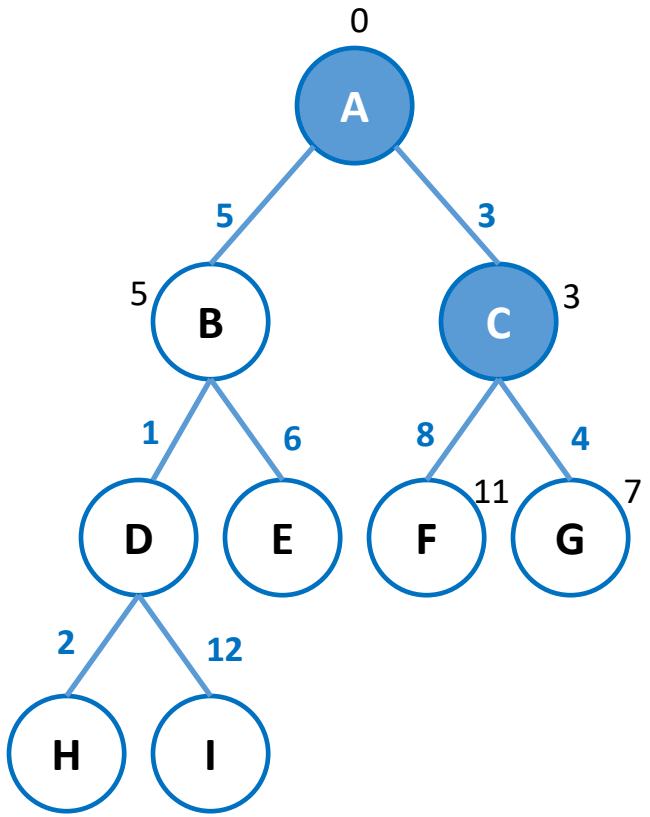
Uniform Cost Search

Idea: expand the lowest cost node first

Implementation:
Frontier is a priority que based on cumulative cost

Uninformed b/c costs are part of the problem!

Equivalent to BFS if all actions have the same cost



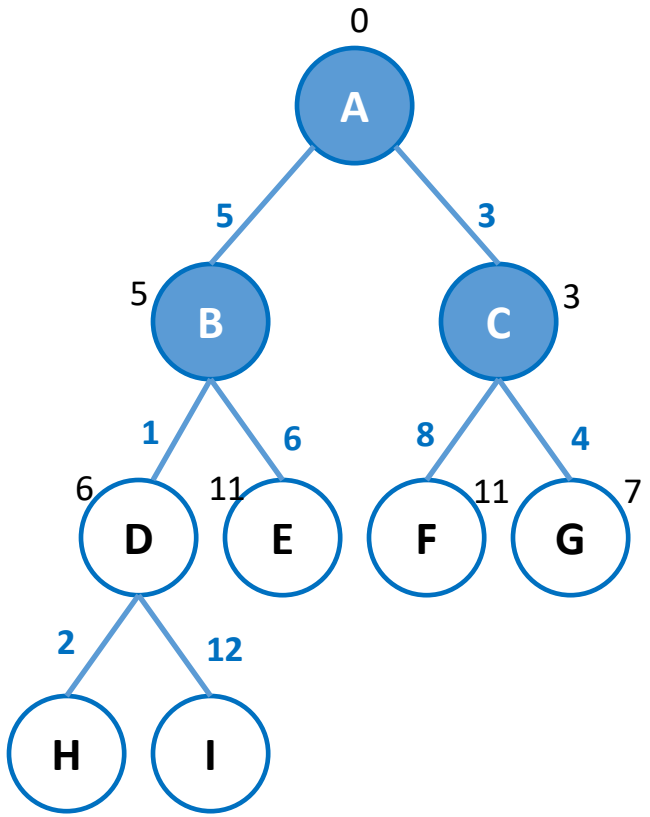
Uniform Cost Search

Idea: expand the lowest cost node first

Implementation:
Frontier is a priority que based on cumulative cost

Uninformed b/c costs are part of the problem!

Equivalent to BFS if all actions have the same cost



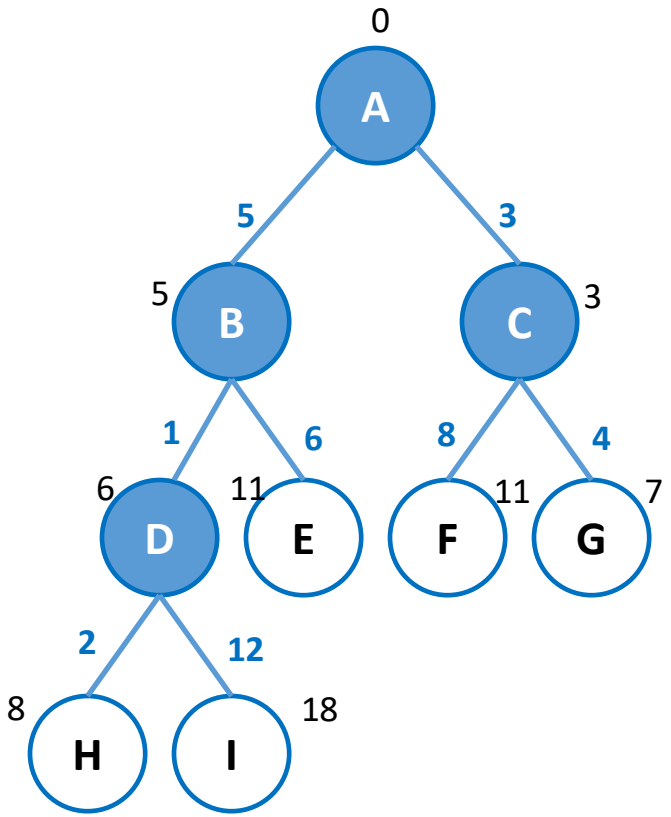
Uniform Cost Search

Idea: expand the lowest cost node first

Implementation:
Frontier is a priority que based on cumulative cost

Uninformed b/c costs are part of the problem!

Equivalent to BFS if all actions have the same cost



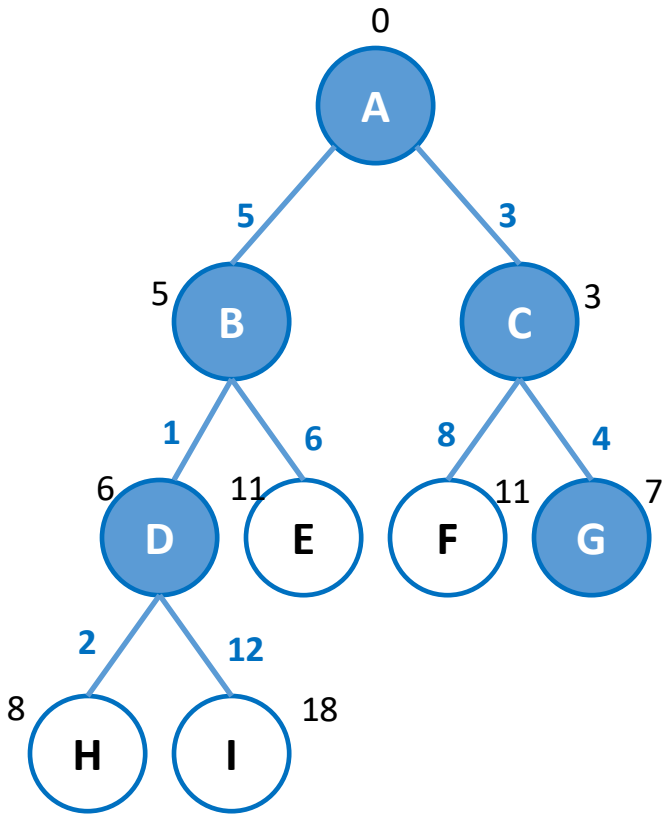
Uniform Cost Search

Idea: expand the lowest cost node first

Implementation:
Frontier is a priority que based on cumulative cost

Uninformed b/c costs are part of the problem!

Equivalent to BFS if all actions have the same cost



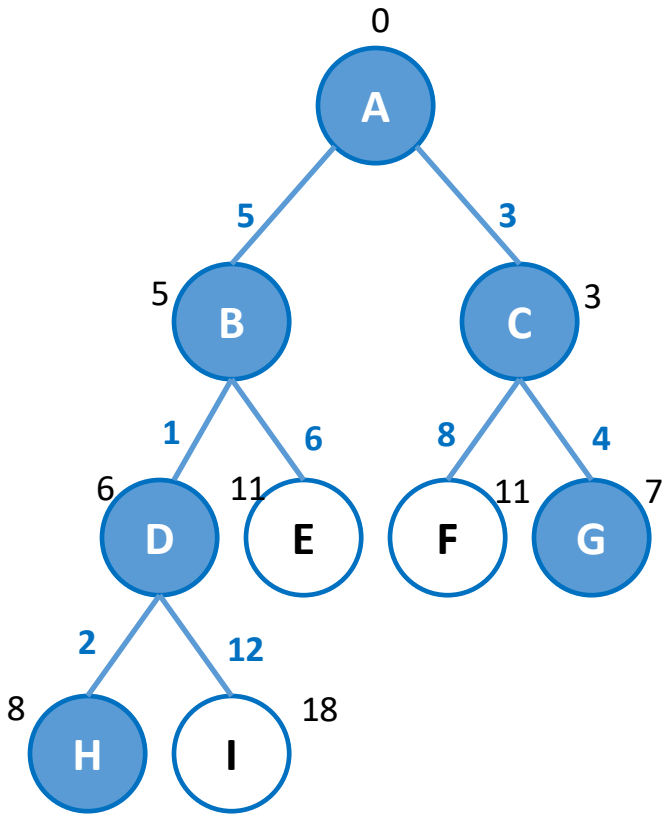
Uniform Cost Search

Idea: expand the lowest cost node first

Implementation:
Frontier is a priority que based on cumulative cost

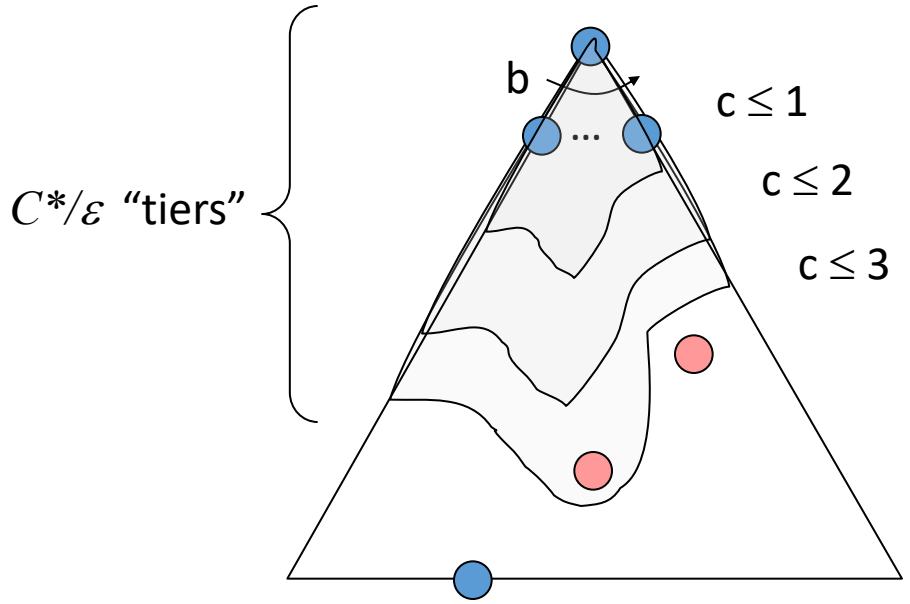
Uninformed b/c costs are part of the problem!

Equivalent to BFS if all actions have the same cost

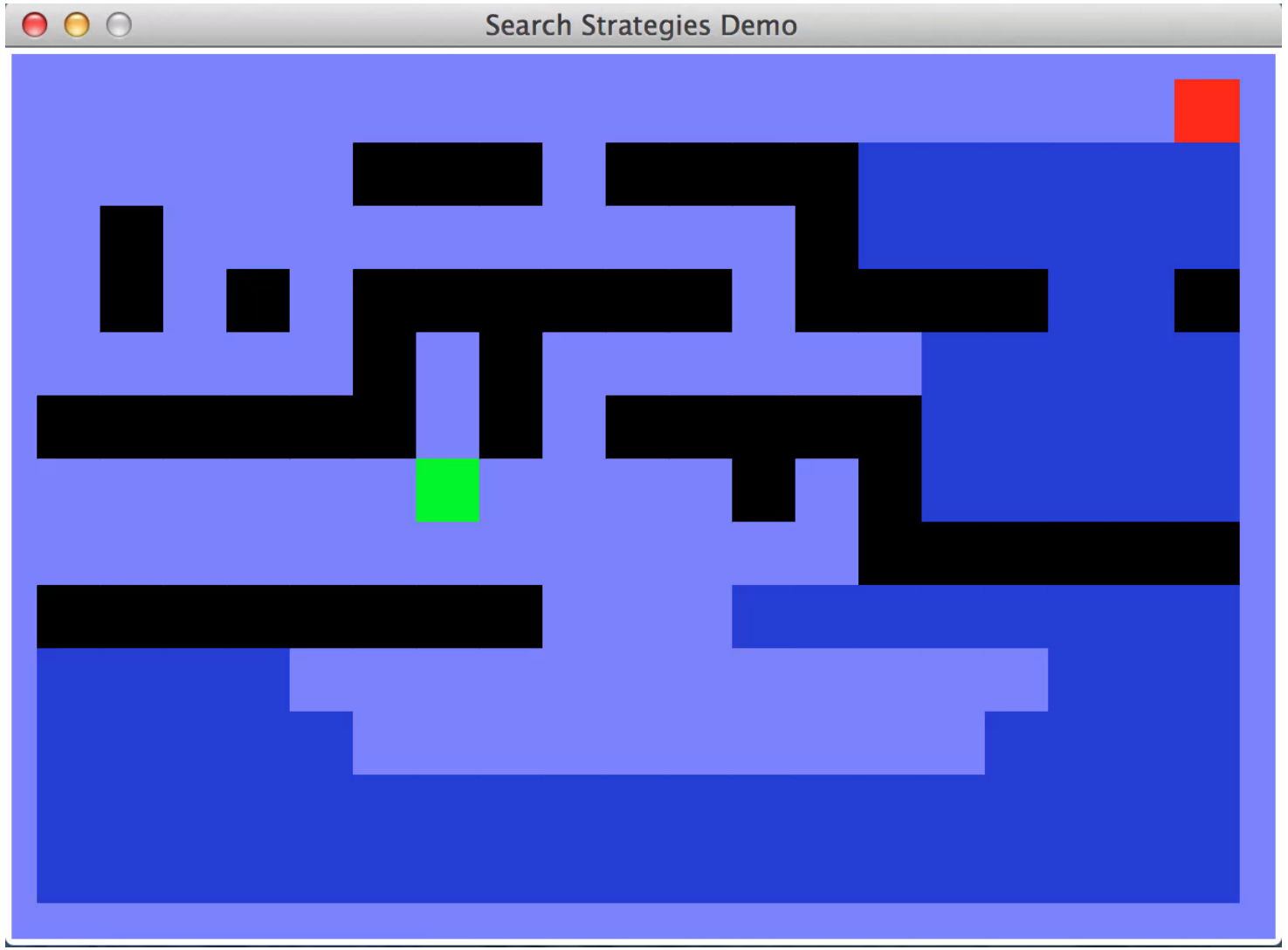


UCS Properties

- **Completeness** ?
  - Yes
- **Optimality** ?
  - Yes (positive costs)
- **Time Complexity** ?
  - O($b^{C^*/\varepsilon}$)
- **Space Complexity** ?
  - O($b^{C^*/\varepsilon}$)

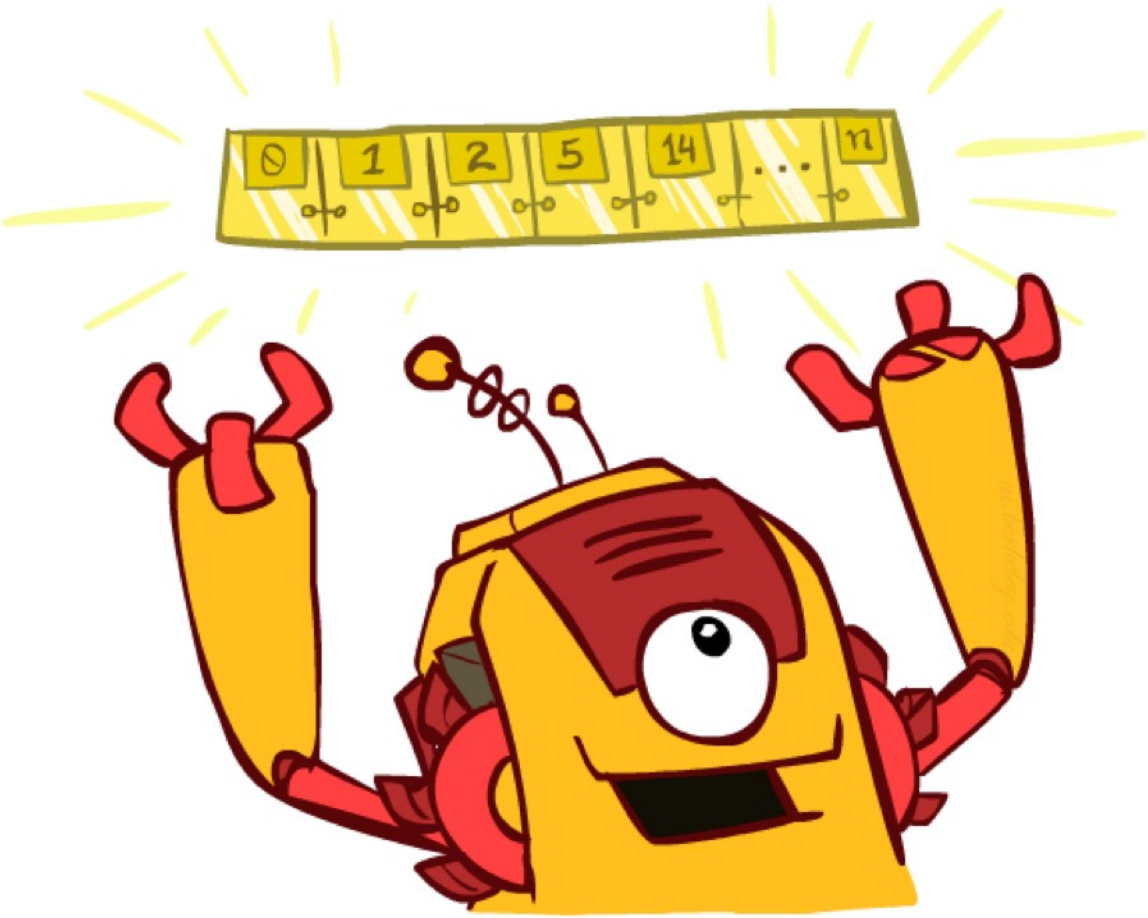


- If the solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε


Search Strategies Demo

The One Queue

- All these search algorithms are the same except for the frontier strategies
  - Conceptually, all frontiers are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable frontier object



Uninformed Search 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*

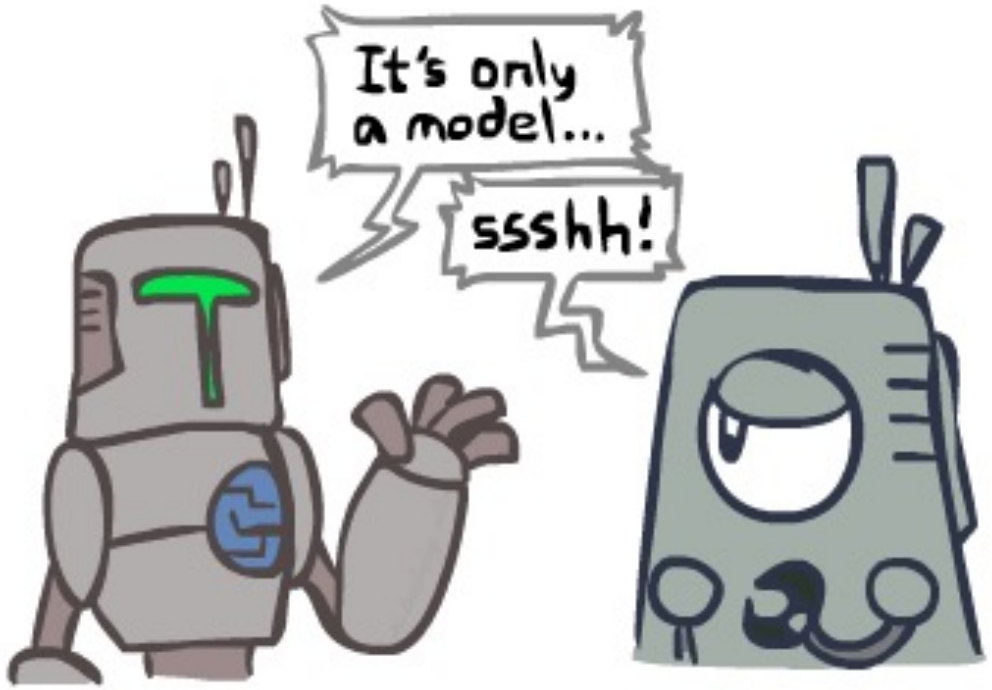
b: branching factor
d: solution depth
m: max depth
l: given limit
C^*: optimal cost
ε: minimum arc cost

- BFS: Optimality only holds for uniform costs
- UCS: Optimality holds for non-negative costs
- Iterative Deepening: Optimality only holds for uniform costs and tree search version
- Space complexities ignore the visited/explored set (i.e. the tree search versions)
- Branching factors are assumed to be finite
- For completeness, graphs are assumed to be either finite or that there is a goal in finite depth

Why uninformed?

Search and Models

- Search operates over models of the world
  - The agent doesn't try all the plans out in the real world!
  - Planning is all "in simulation"
  - Your search is only as good as your models…



Search Gone Wrong?