Problem solving and searching

Problem Solving Agents

A simple problem-solving agent:

- It first formulates a goal and a problem,
- searches for a sequence of actions that would solve the problem
- executes the actions one at a time.
- When this is complete, it formulates another goal and starts over.

Problem Solving

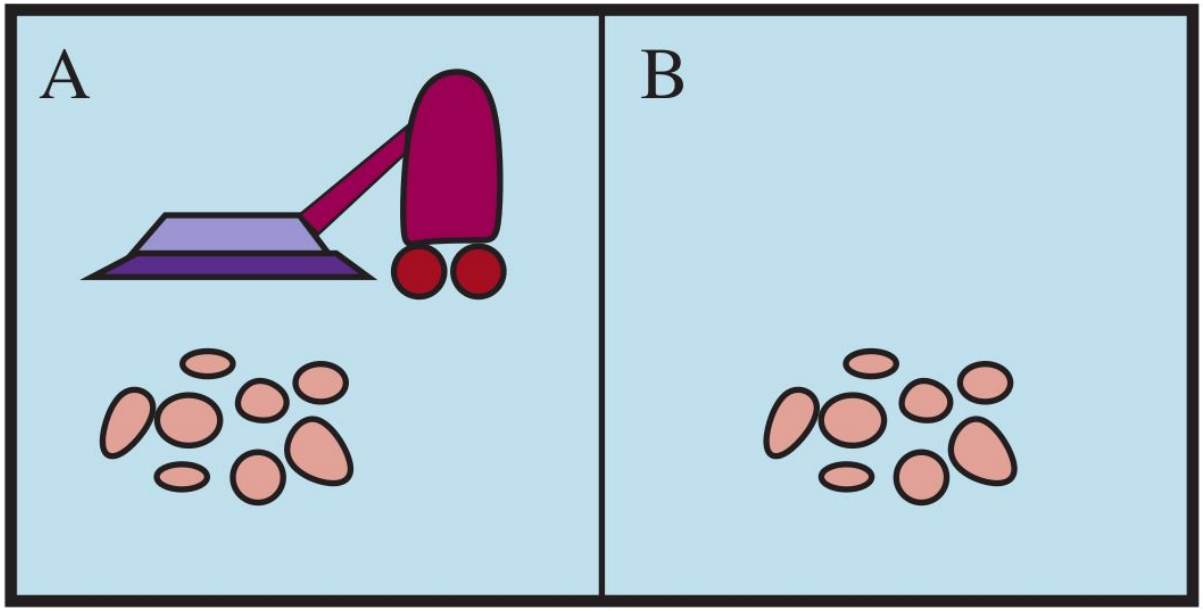
- Goal Formulation
- Problem Formulation
  - Initial State
  - Actions (operators)
  - Transition Model
  - Goal Test
  - Path Cost

Problem Solving

- **Goal Formulation**: Based on current situation and performance measure.
- **Problem Formulation**: given a goal, what actions and states to consider
  - **Initial State**: The initial state that the agent starts in.
  - **Actions**: description of the possible actions available to the agent.
  - **Transition Model**: describes what each action does, i.e., maps a state and action to a resulting state
  - **Goal Test**: determines whether a given state is a goal state.
  - **Path Cost**: A path cost function that assigns a numeric cost to each path.

Example: vacuum world

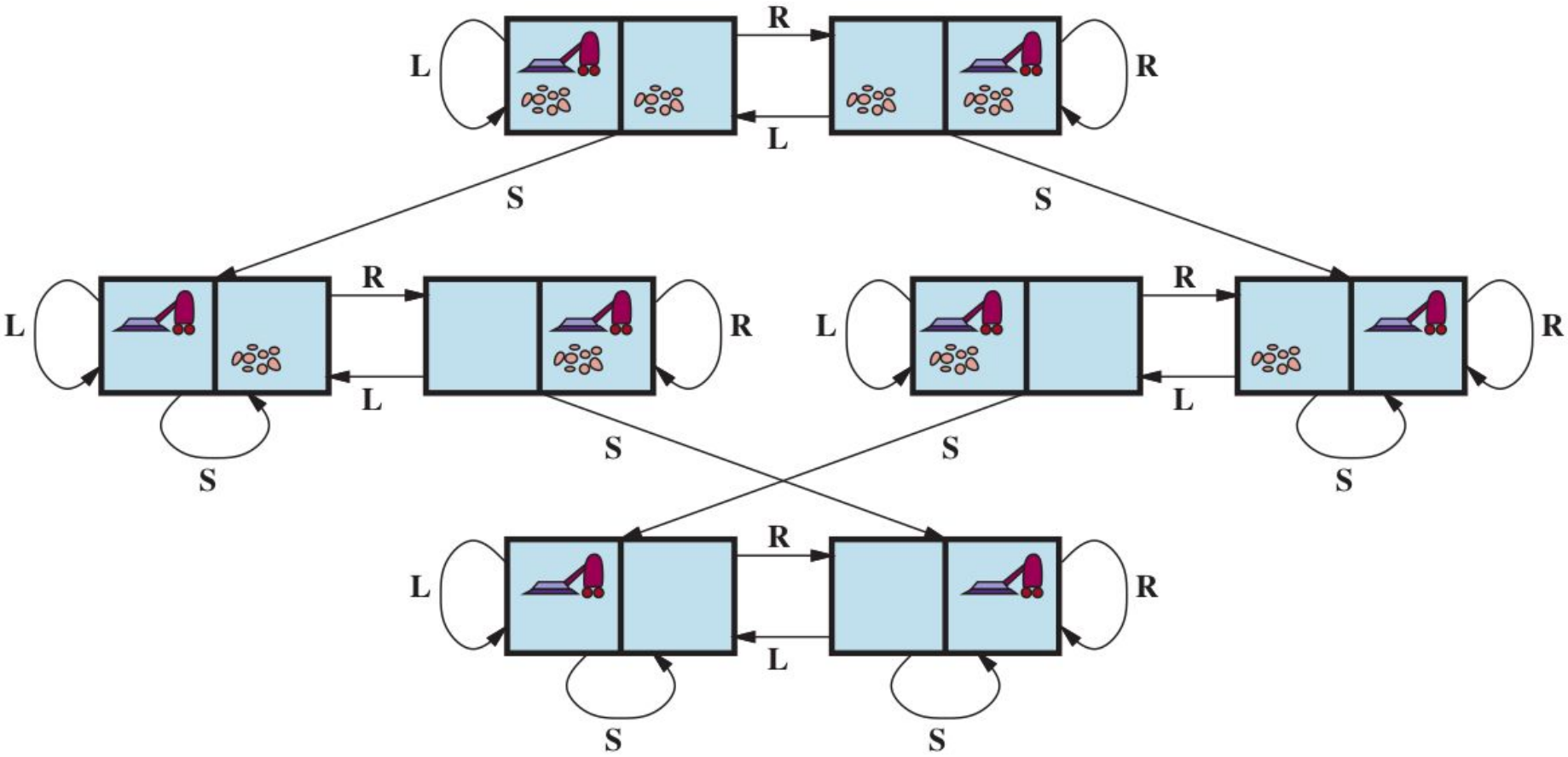
- Goal Formulation
- Problem Formulation
  - Initial State
  - Actions (operators)
  - Transition Model
  - Goal Test
  - Path Cost



Example: vacuum world

- **States**: A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. Total 8 states.
- **Initial state**:
- **Actions**:
- **Transition mode**l:
- **Goal states**:
- **Path cost**:

Example: vacuum world



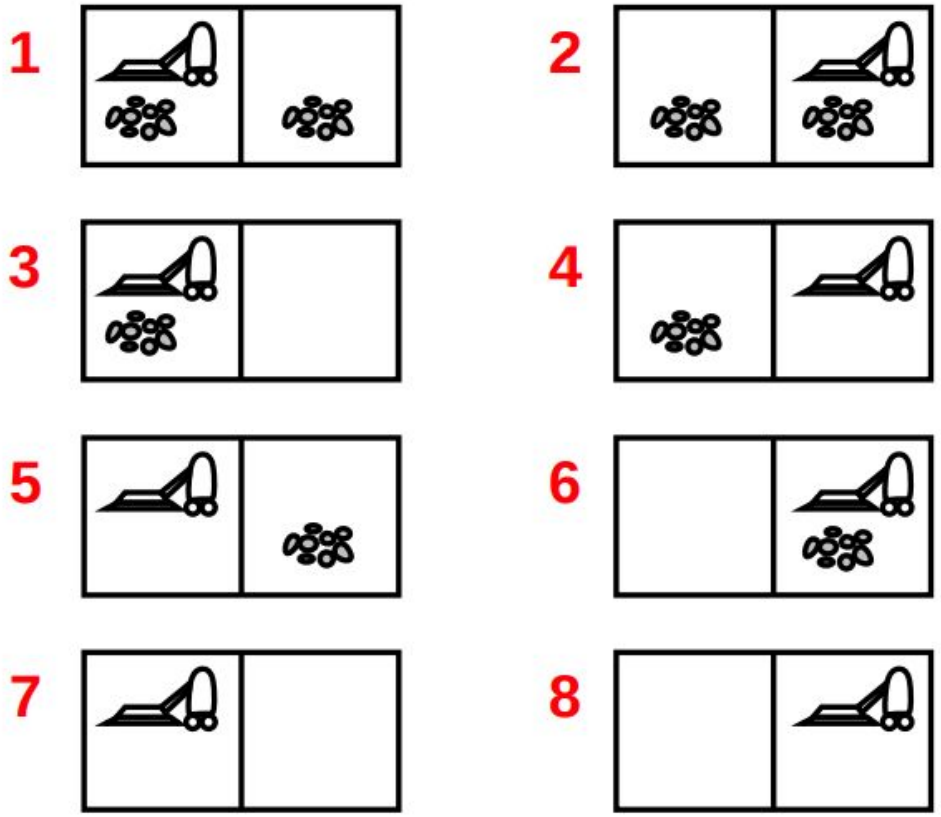
Example: vacuum world

- **States**: Total 8 states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: Suck, move Left, and move Right.
- **Transition mode**l: Maps a state and action to a resulting state.
- **Goal states**: The states in which every cell is clean.
- **Path cost**: Each action costs 1.

Example: vacuum world

Single-state, start in #5.

Solution??

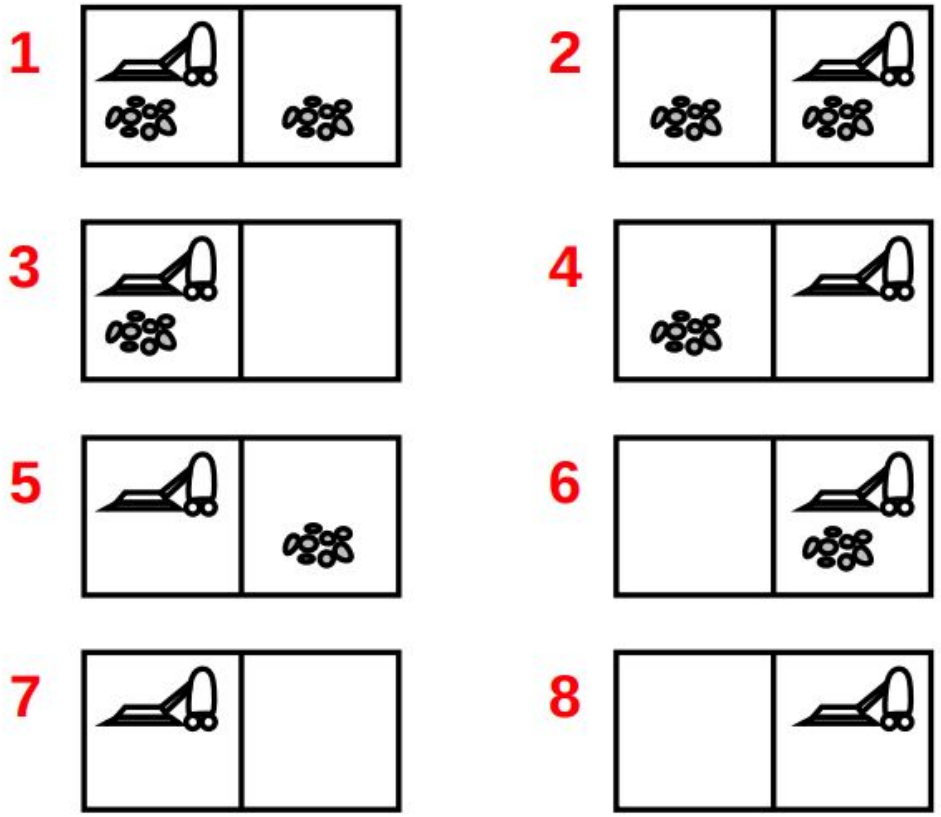


Example: vacuum world

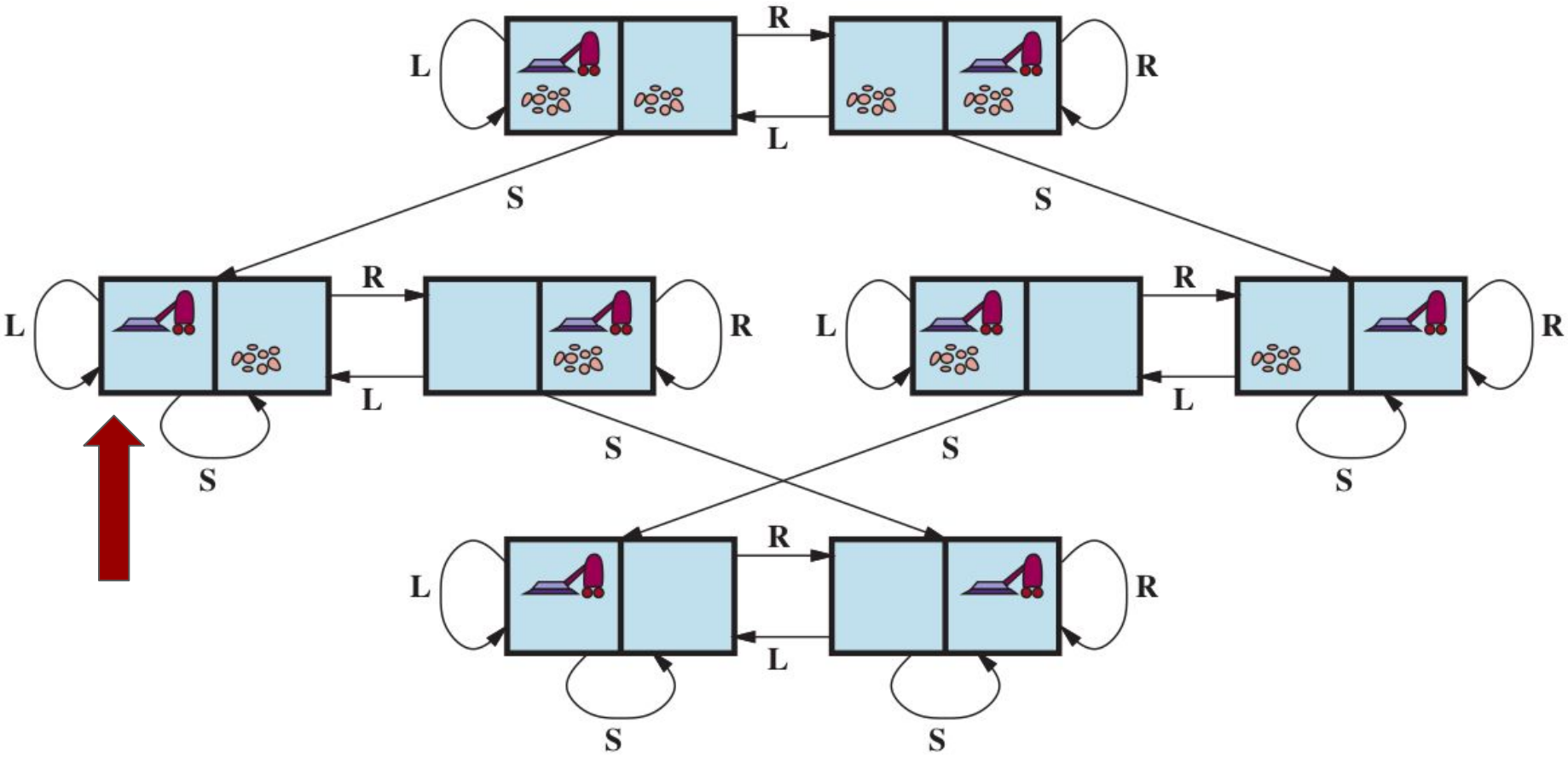
Single-state, start in #5.

Solution??

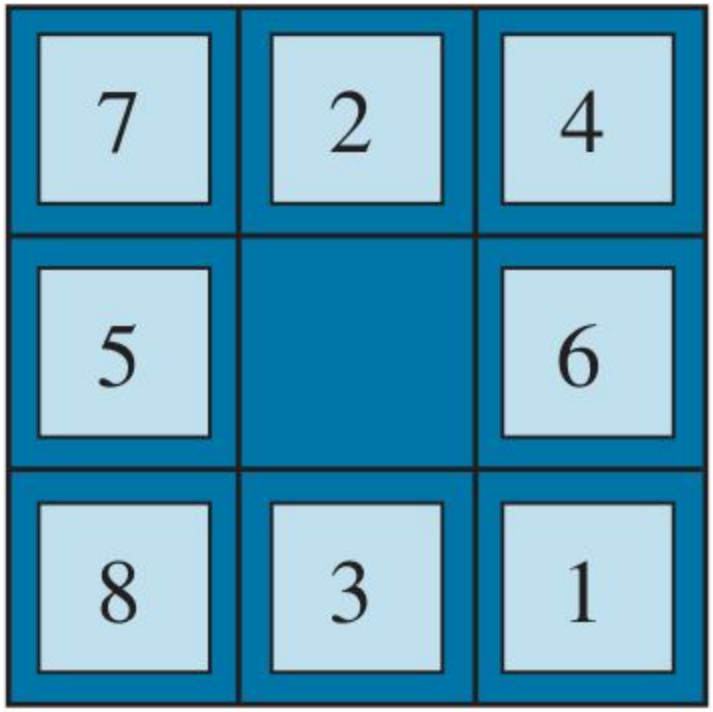
[Right, Suck]



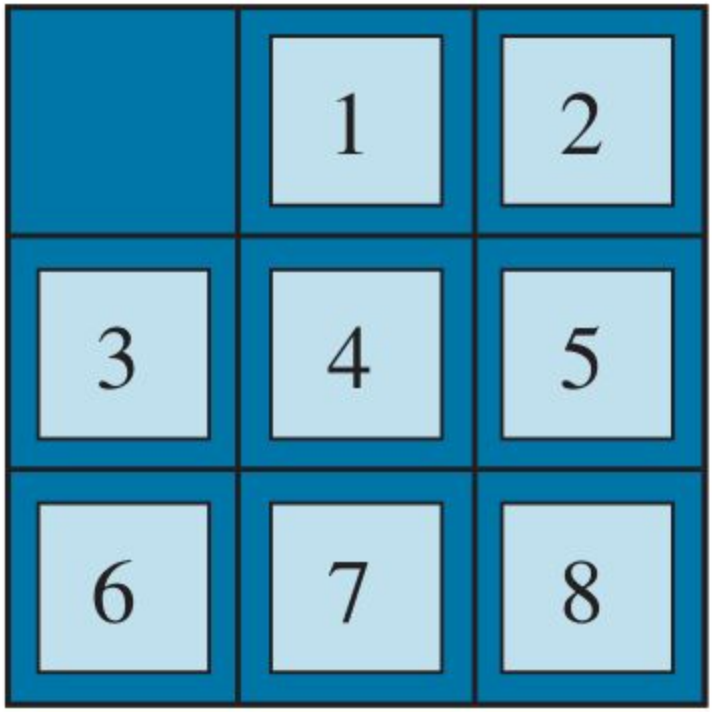
Example: vacuum world



Example: The 8-puzzle



Start State



Goal State

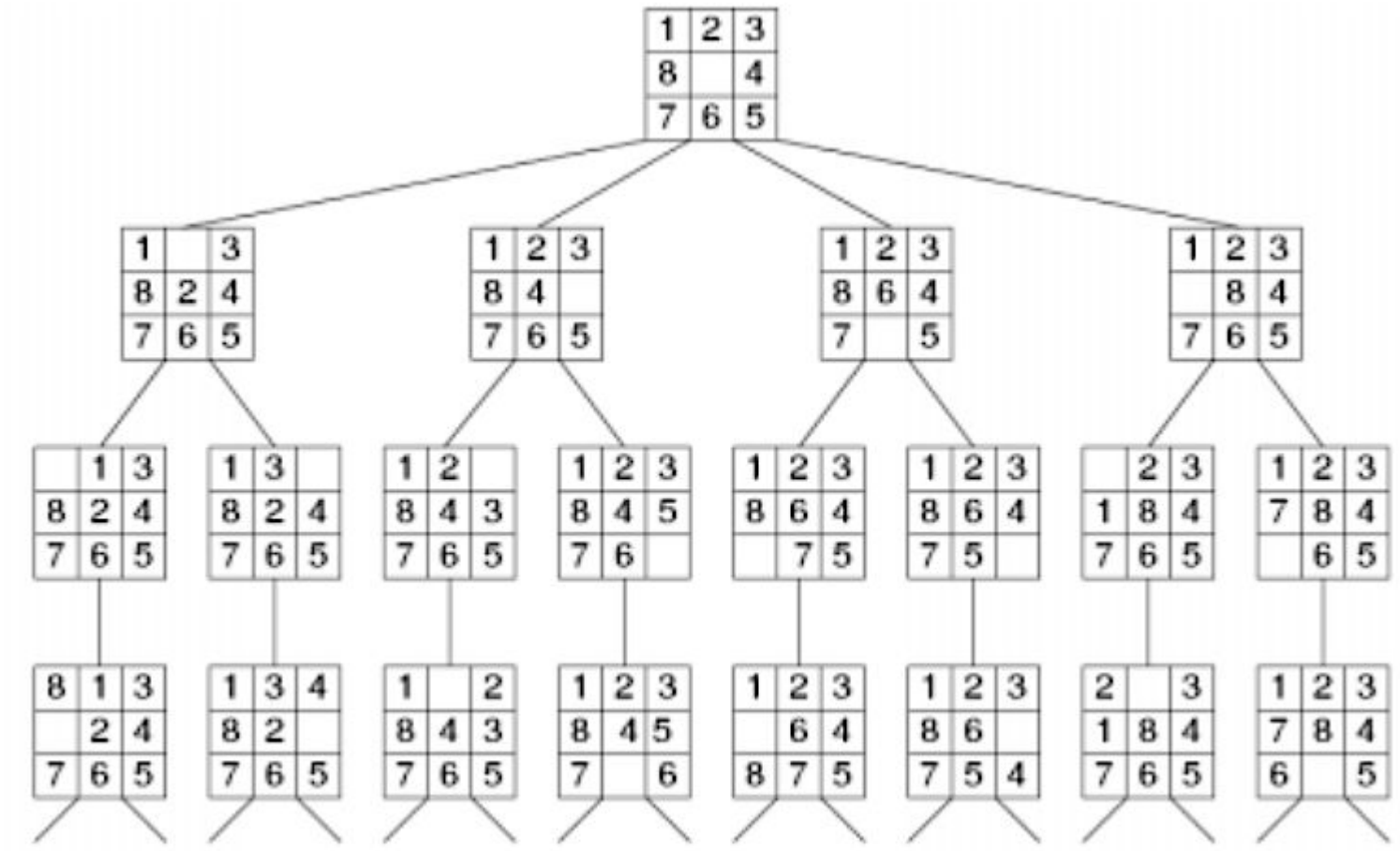
Example: The 8-puzzle

- **States**:
- **Initial state**:
- **Actions**:
- **Transition model**:
- **Goal state**:
- **Action cost**:

Example: The 8-puzzle

- **States**: A state description specifies the location of each tiles.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: blank space moving Left, Right, Up, or Down.
- **Transition model**: Maps a state and action to a resulting state;
- **Goal state**: a state with the numbers in order.
- **Action cost**: Each action costs 1.

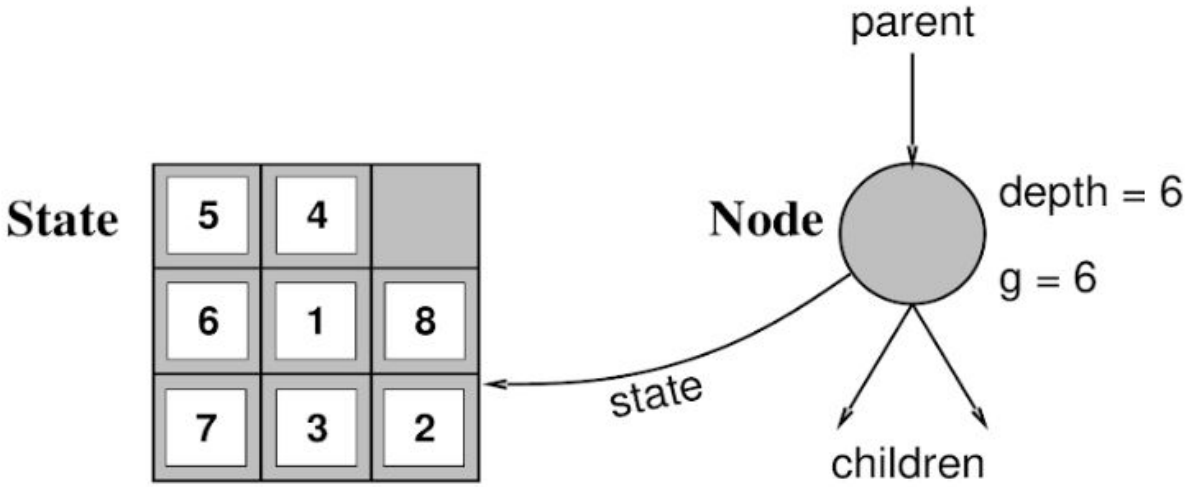
Example: The 8-puzzle



Example: The 8-puzzle

State v/s Nodes

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

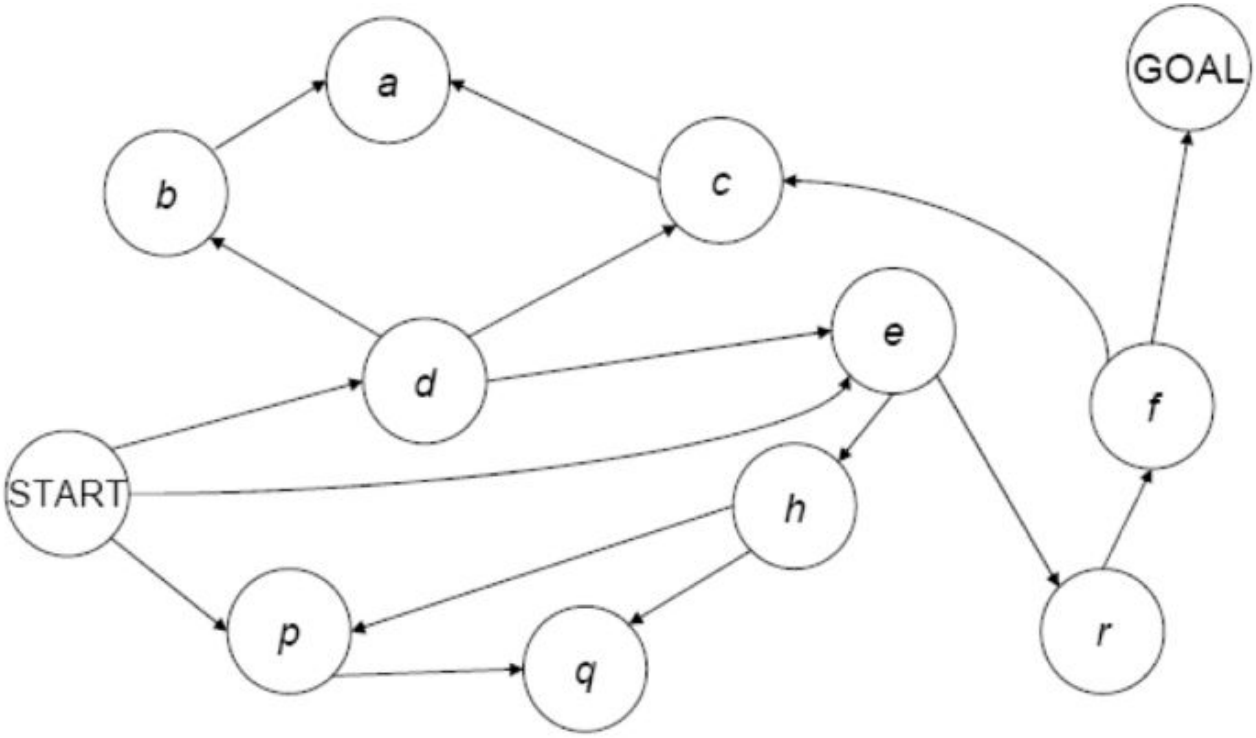


Search data structures (node)

A node in the search tree is represented with four components:

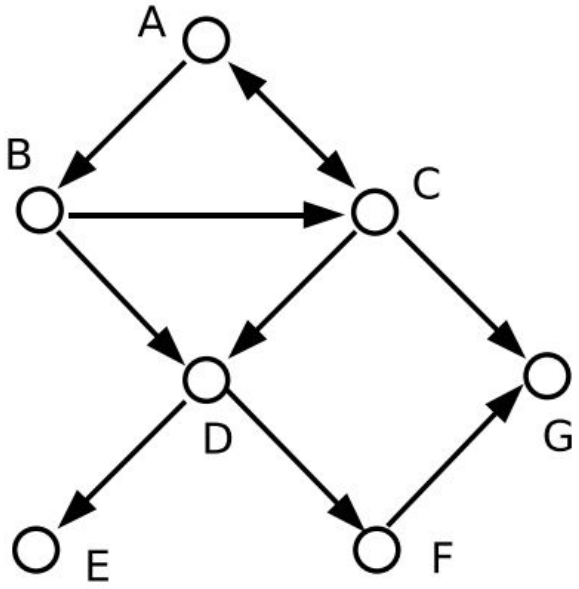
- *node.STATE* : the state to which the node corresponds;
- *node.PARENT*: the node in the tree that generated this node;
- *node.ACTION* : the action that was applied to the parent's state to generate this node;
- *node.PATH-COST* : the total cost of the path from the initial state to this node.

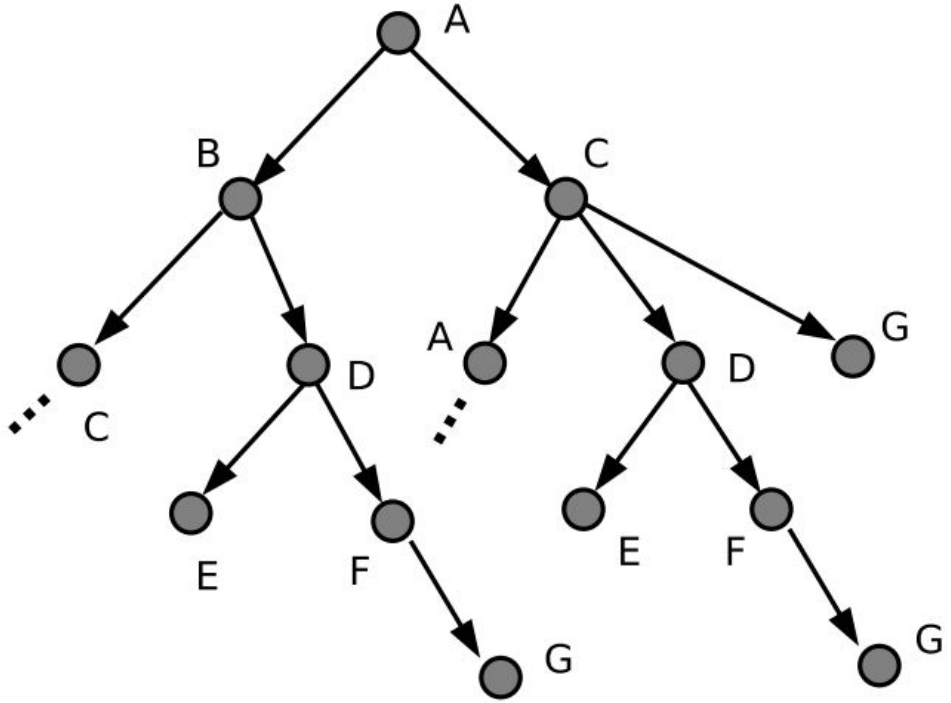
Drawing Search: Graphical Representation



Vertices represent states in the search space and edges represents transitions resulting from actions (assumes a finite search space).

State space vs. search tree





- Search tree is created by searching through the state space.
- Search tree can be infinite even if the state space is finite

State and State Space

- A ***state*** is a situation that an agent can find itself in
  - *world states:* The actual situations of real world
  - *representational states:* abstract descriptions of the real world used by the agent.
- **State space** is a *graph* whose nodes are the set of all states, and whose links are actions that transform one state into another
- ***Successor function****:* given a *state*, it returns a set of (action, state) pairs, where each state is the state reachable by taking action.

Search tree and nodes

- A **search tree** is a tree (a graph with no loops) in which the root node is the start state and the set of children for each node consists of the states reachable by taking any action
- A ***search node*** is a node in the search tree. It is a data structure constituting part of a search tree that includes state, parent node, action, path cost.
- The ***branching factor*** in a search tree is the number of actions available to the agent

State space search

There are two types of strategies:

- Blind (uninformed) search
- Heuristic (directed, informed) search

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

Search data structures (Queues)

Three kinds of queues are used in search algorithms:

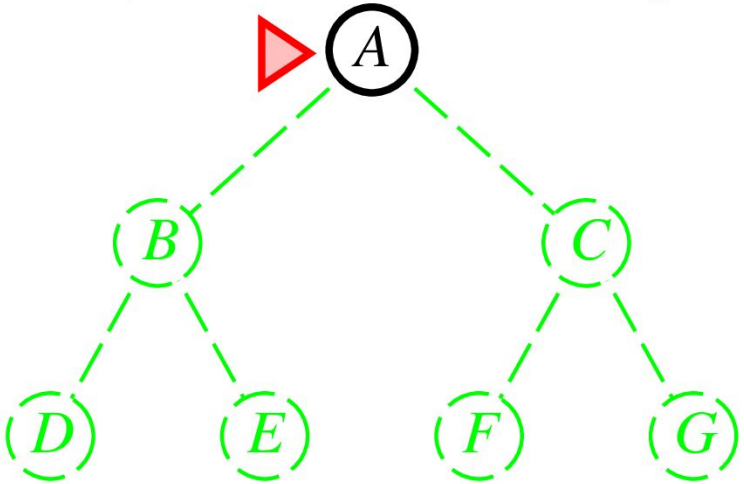
- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in *breadth-first search*.
- A **LIFO queue** or last-in-first-out queue (also known as a **stack**) pops first the most recently added node; we shall see it is used in *depth-first search*.
- A **priority queue** first pops the node with the minimum cost according to some evaluation function, f . It is used in *best-first search*.

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

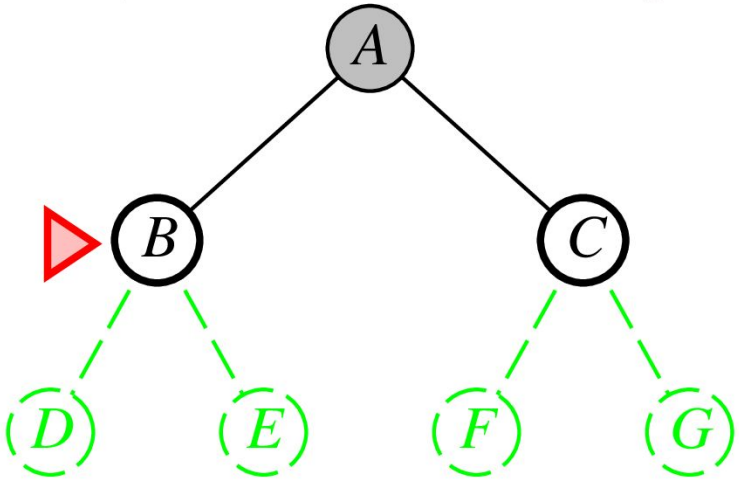


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

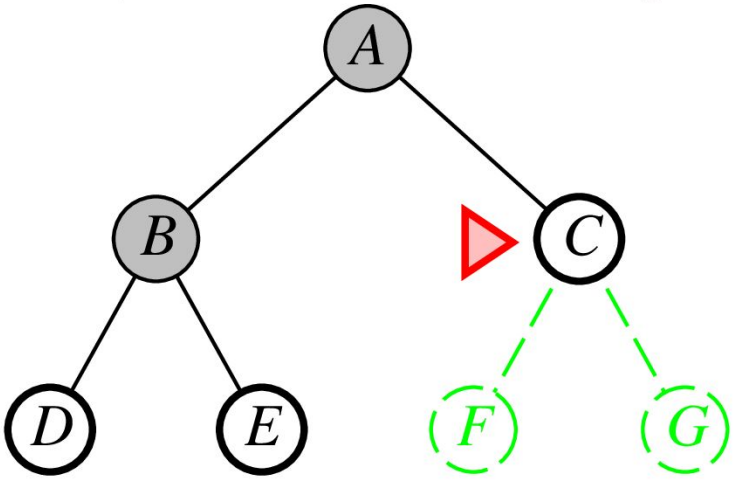


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

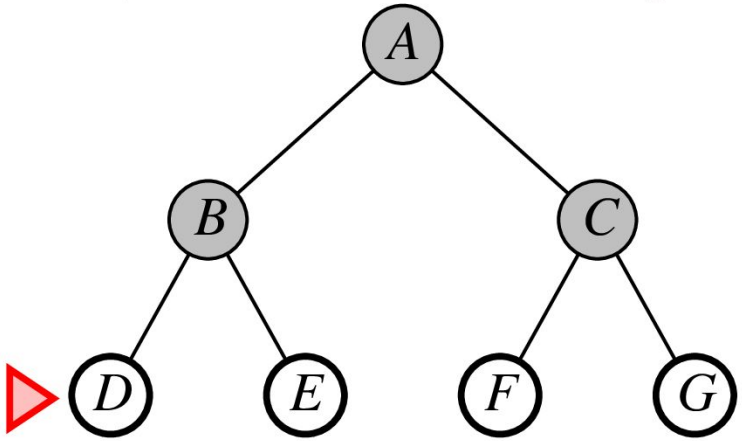


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



An example journey

How to reach Buzet from Pula?

$$problem = (s_0, \text{succ}, \text{goal})$$

$$s_0 = Pula$$

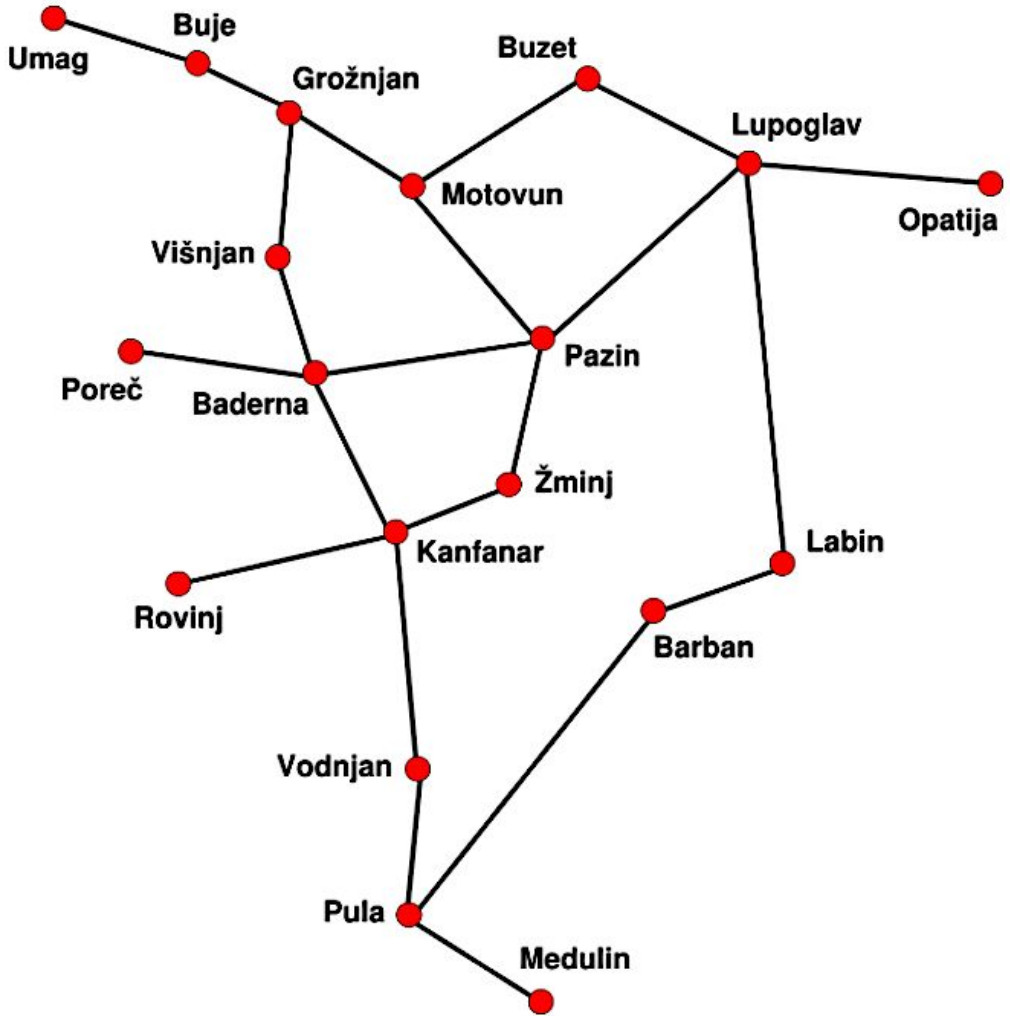
$$\text{succ}(Pula) = \{Barban, Medulin, Vodnjan\}$$

$$\text{succ}(Vodnjan) = \{Kanfanar, Pula\}$$

$$\vdots$$

$$\text{goal}(Buzet) = \top$$

An example journey



How to reach Buzet from Pula?

$$problem = (s_0, \text{succ}, \text{goal})$$

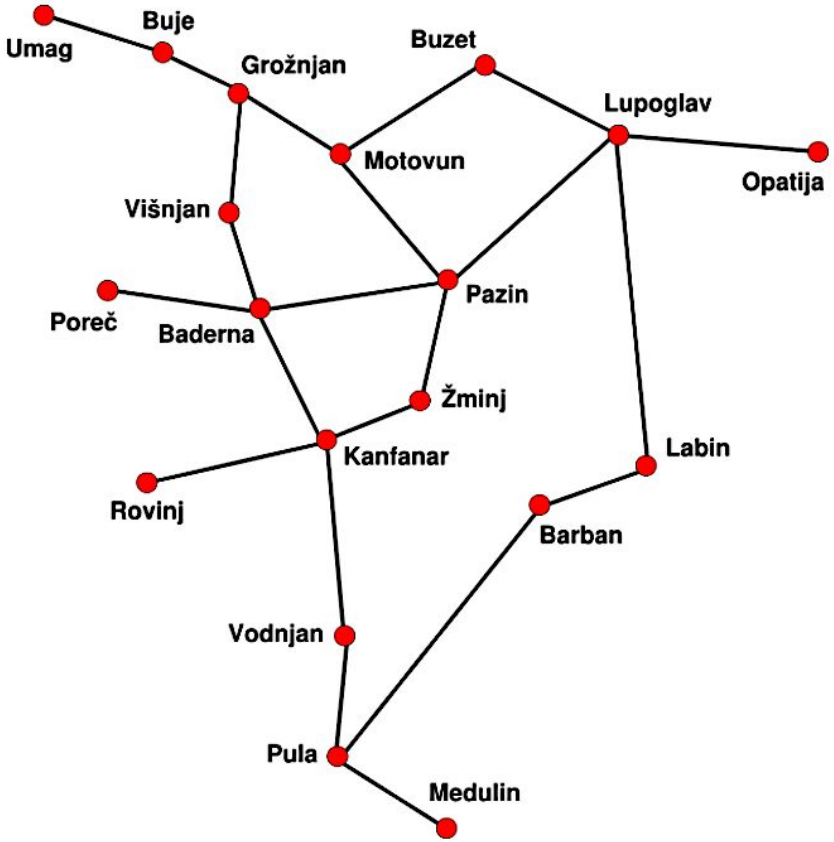
$$s_0 = Pula$$

$$\text{succ}(Pula) = \{Barban, Medulin, Vodnjan\}$$

$$\text{succ}(Vodnjan) = \{Kanfanar, Pula\}$$

$$\vdots$$

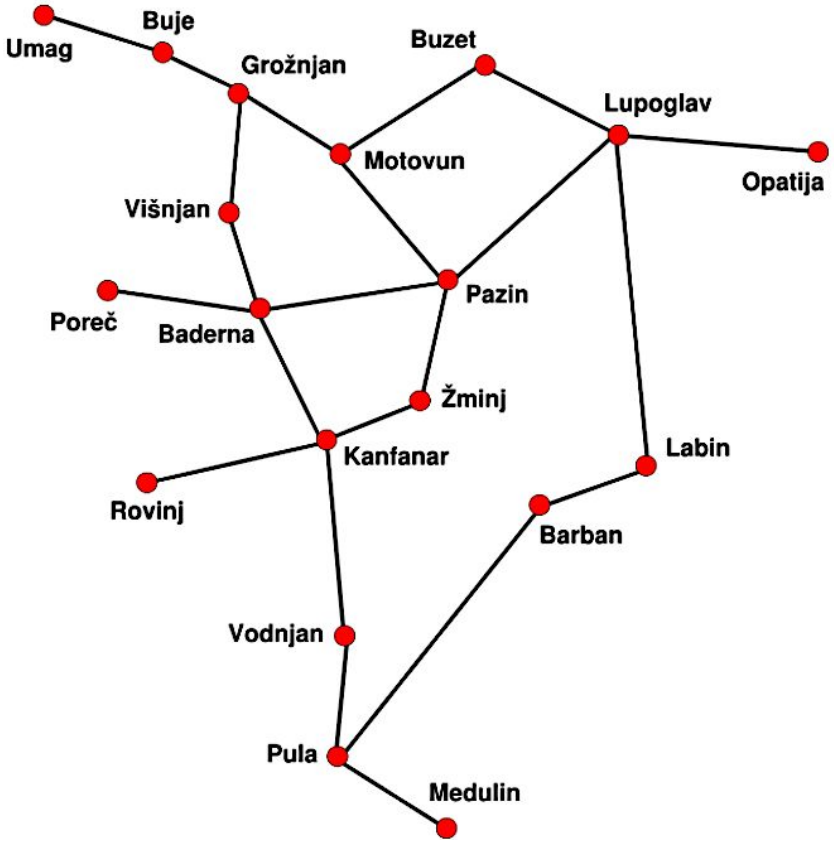
$$\text{goal}(Buzet) = \top$$



$$open = [\,(Pula, 0)\,]$$

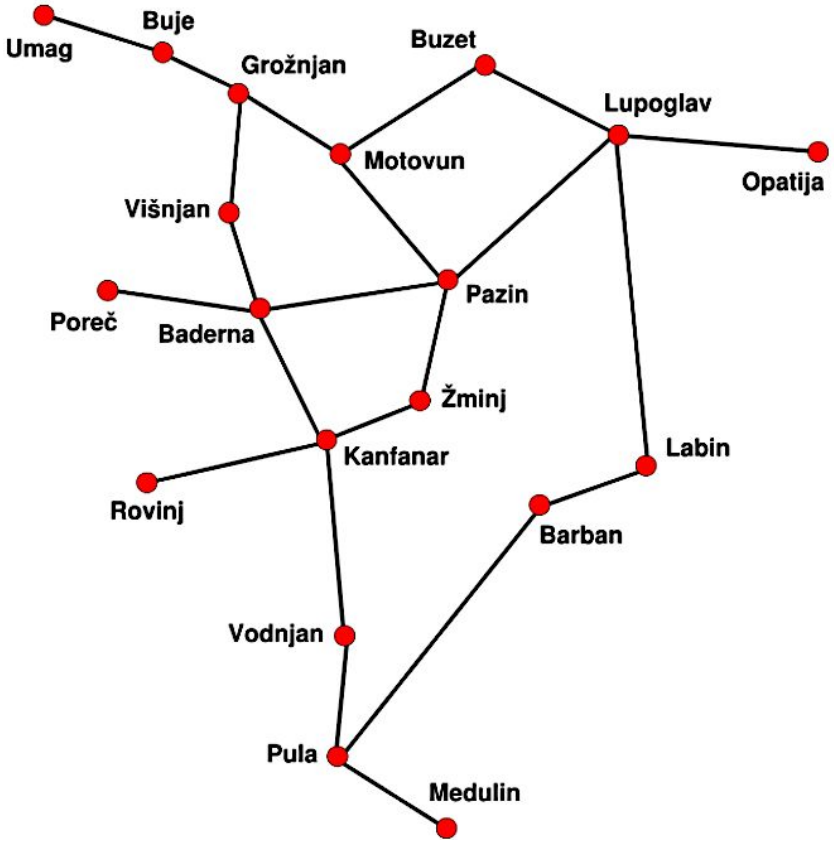
$$\text{expand}(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\}$$
$$open = [\,(Vodnjan, 1), (Barban, 1), (Medulin, 1)\,]$$

$$\text{expand}(Vodnjan, 1) = \{(Kanfanar, 2), (Pula, 2)\}$$
$$open = [\,(Barban, 1), (Medulin, 1), (Kanfanar, 2), (Pula, 2)\,]$$



$$\text{expand}(Barban, 1) = \{(Labin, 2), (Pula, 2)\}$$
$$open = [\,(Medulin, 1), (Kanfanar, 2), (Pula, 2), (Labin, 2), (Pula, 2)\,]$$

$$\text{expand}(Medulin, 1) = \{(Pula, 2)\}$$
$$open = [\,(Kanfanar, 2), (Pula, 2), (Labin, 2), (Pula, 2), (Pula, 2)\,]$$



$$\text{expand}(Kanfanar, 2) =$$
$$\{(Baderna, 3), (Rovinj, 3), (Vodnjan, 3)(Zminj, 3)\}$$
$$open = [\,(Pula, 2), (Labin, 2), (Pula, 2), (Pula, 2), (Baderna, 3), \ldots$$

Properties of breadth-first search

Complete??

Time??

Space??

Optimal??

- b is the branching factor;
- d is the depth of the shallowest solution
- m is the maximum depth of the search tree;

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step);

$Space$ is the big problem;

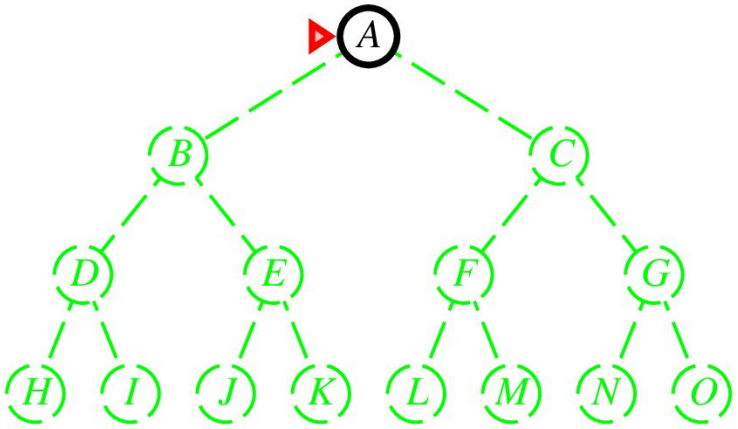
- b is the branching factor;
- d is the depth of the shallowest solution
- m is the maximum depth of the search tree;

Depth-first search

Expand deepest unexpanded node

Implementation:

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

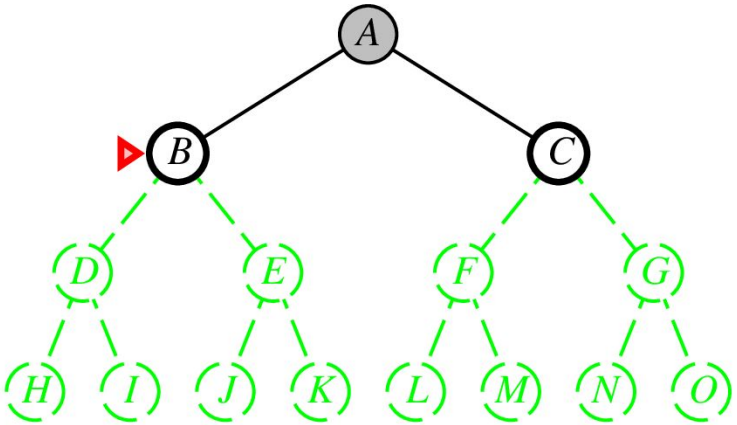


Depth-first search

Expand deepest unexpanded node

Implementation:

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

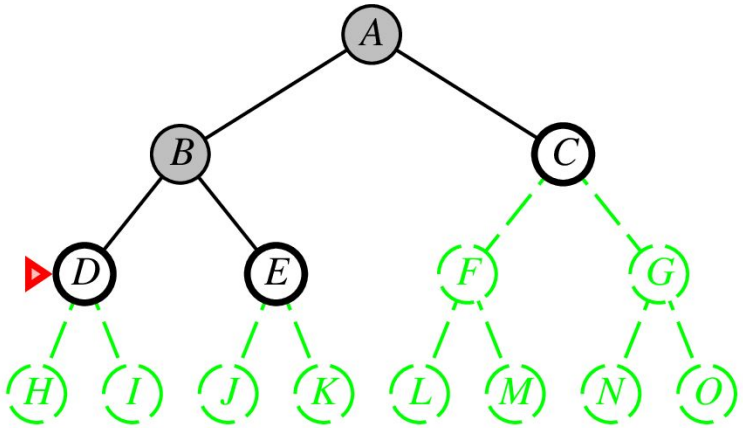


Depth-first search

Expand deepest unexpanded node

Implementation:

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

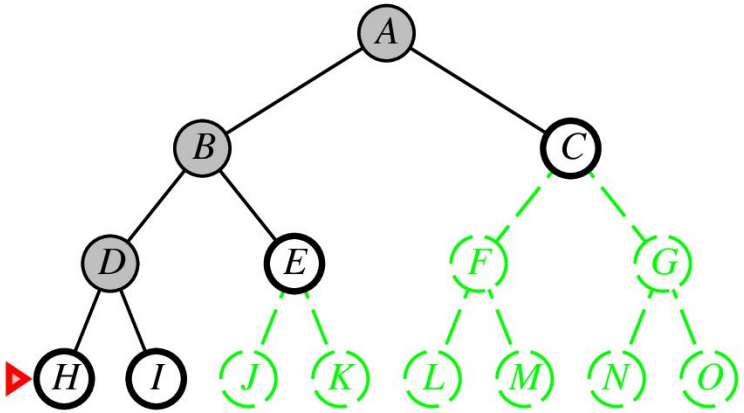


Depth-first search

Expand deepest unexpanded node

Implementation:

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

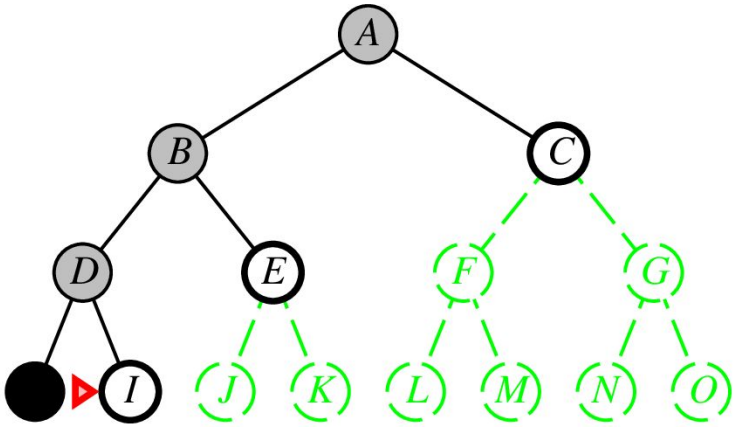


Depth-first search

Expand deepest unexpanded node

Implementation:

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

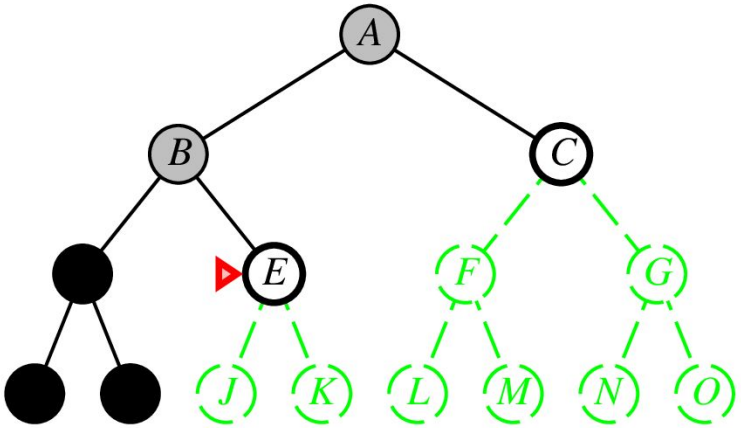


Depth-first search

Expand deepest unexpanded node

Implementation:

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

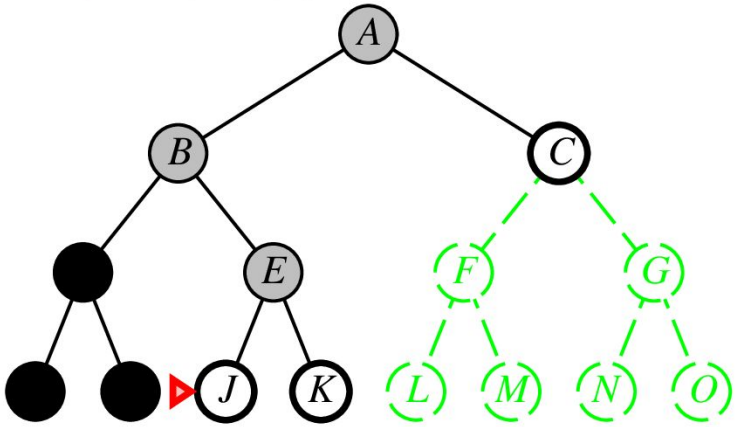


Depth-first search

Expand deepest unexpanded node

Implementation:

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

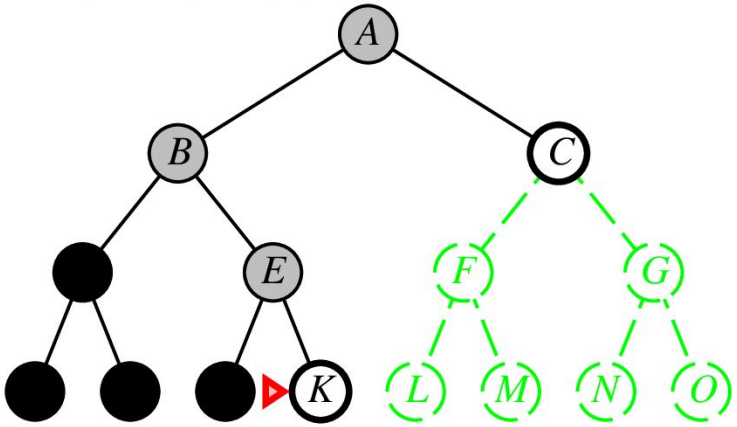


Depth-first search

Expand deepest unexpanded node

Implementation:

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

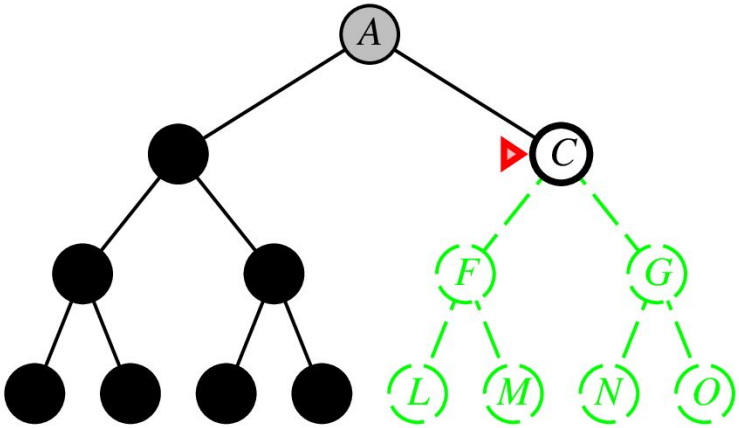


Depth-first search

Expand deepest unexpanded node

Implementation:

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

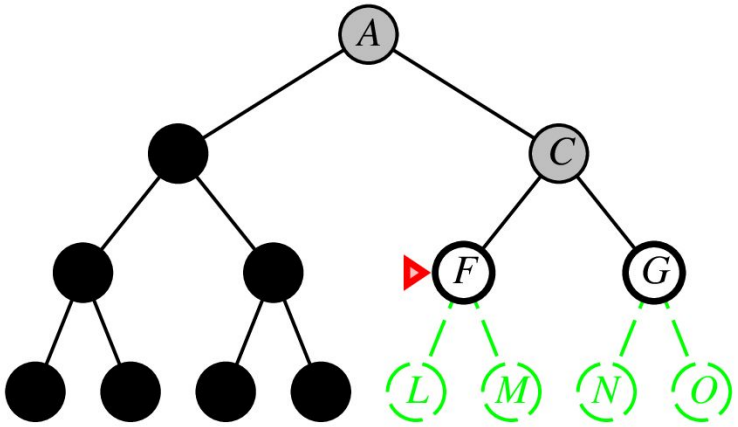


Depth-first search

Expand deepest unexpanded node

Implementation:

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

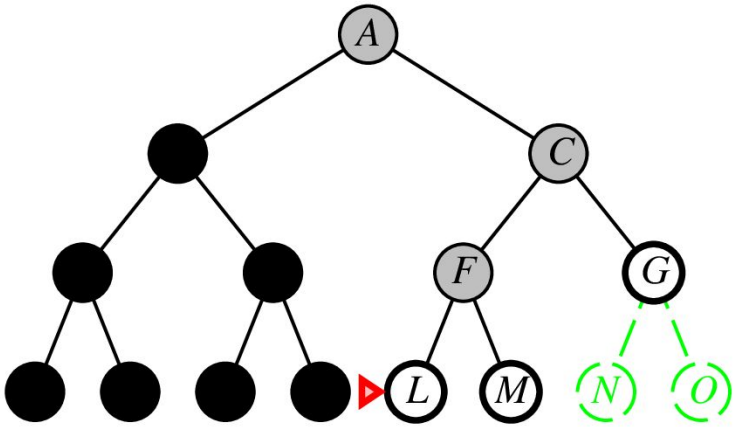


Depth-first search

Expand deepest unexpanded node

Implementation:

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

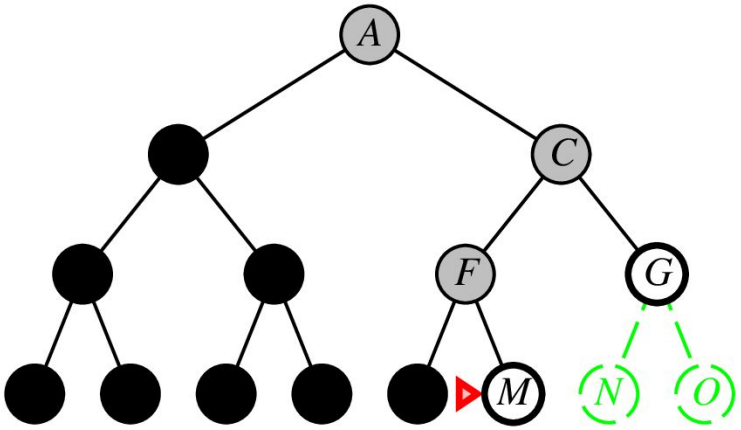


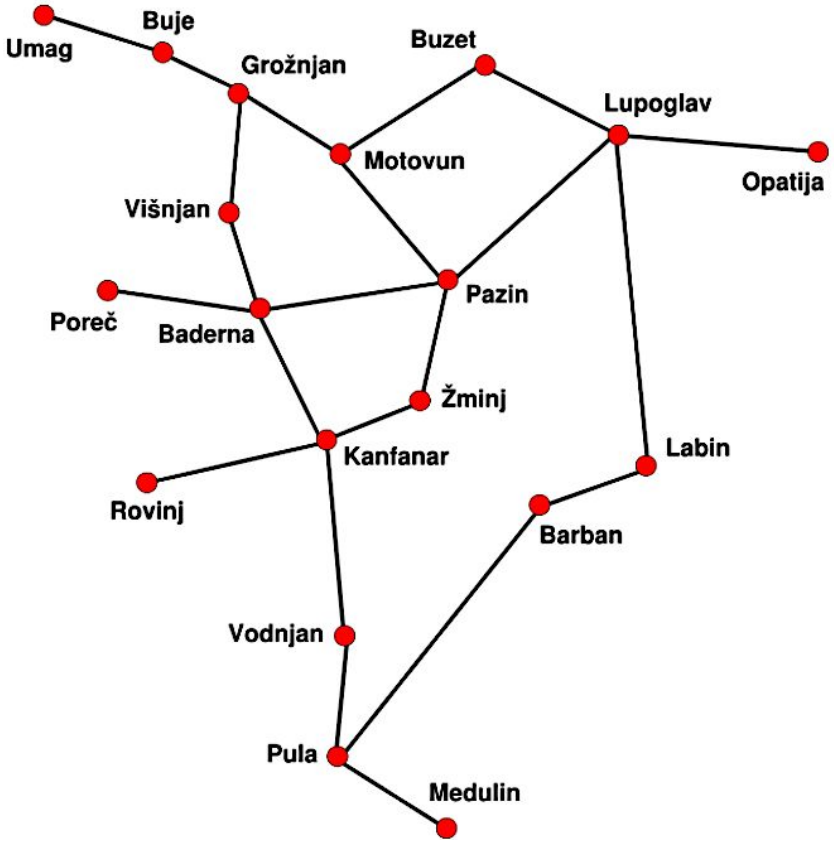
Depth-first search

Expand deepest unexpanded node

Implementation:

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

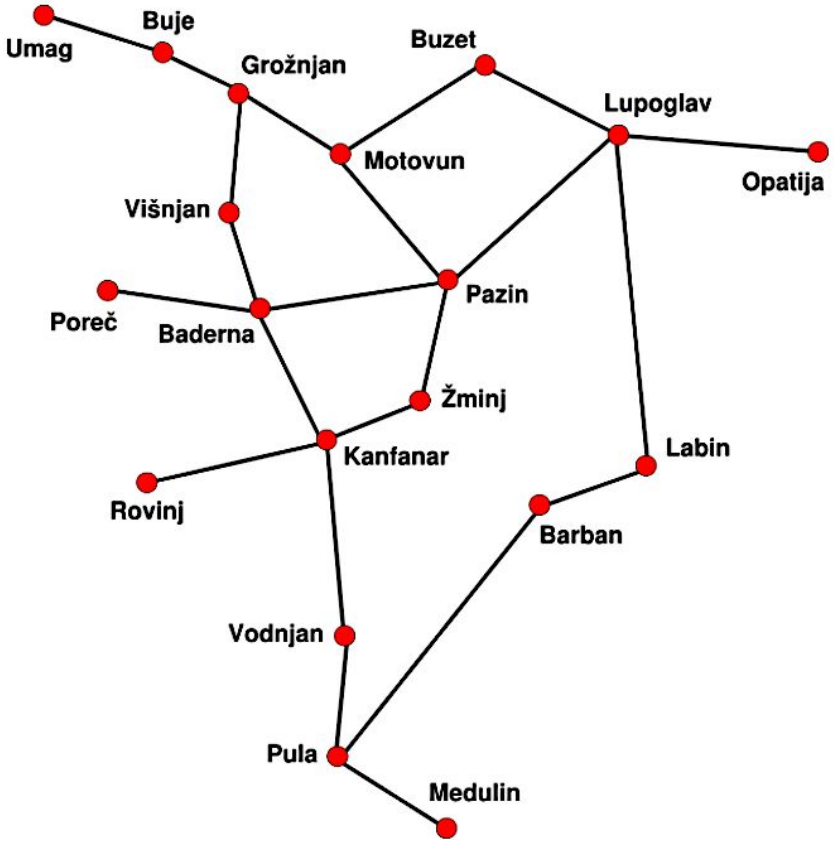




$$open = [\,(Pula, 0)\,]$$

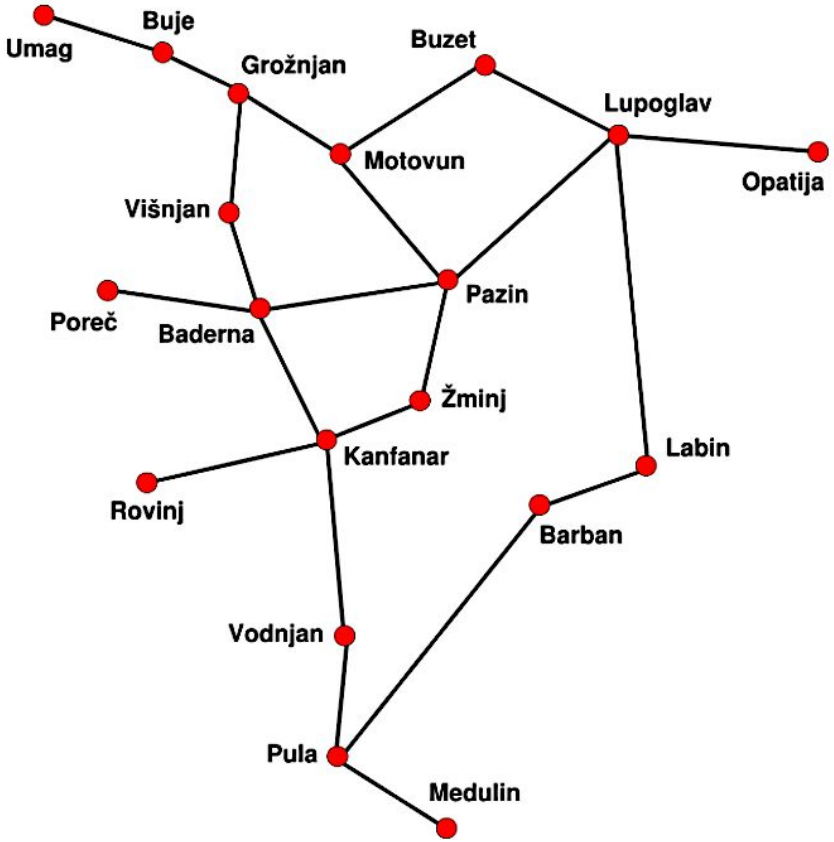
$$\text{expand}(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\}$$

$$open = [\,(Vodnjan, 1), (Barban, 1), (Medulin, 1)\,]$$



$$\text{expand}(\,Vodnjan, 1) = \{(Kanfanar, 2), (Pula, 2)\}$$
$$open = [\,(Kanfanar, 2), (Pula, 2), (Barban, 1), (Medulin, 1)\,]$$

$$\text{expand}(Kanfanar, 2) =$$
$$\{(Baderna, 3), (Rovinj, 3), (\,Vodnjan, 3), (Zminj, 3)\}$$
$$open = [\,(Baderna, 3), (Rovinj, 3), (\,Vodnjan, 3), (Zminj, 3), (Pula, 2), \ldots$$



$$\text{expand}(Baderna, 3) = \{(Porec, 4), (Visnjan, 4), (Pazin, 4), (Kanfanar, 4)\}$$

$$open =$$

$$[(Porec, 4), (Visnjan, 4), (Pazin, 4), (Kanfanar, 4), (Baderna, 3), \ldots$$

Properties of depth-first search

Complete??

Time??

Space??

Optimal??

- b is the branching factor;
- m is the maximum depth of the search tree;
- d is the depth of the shallowest solution

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

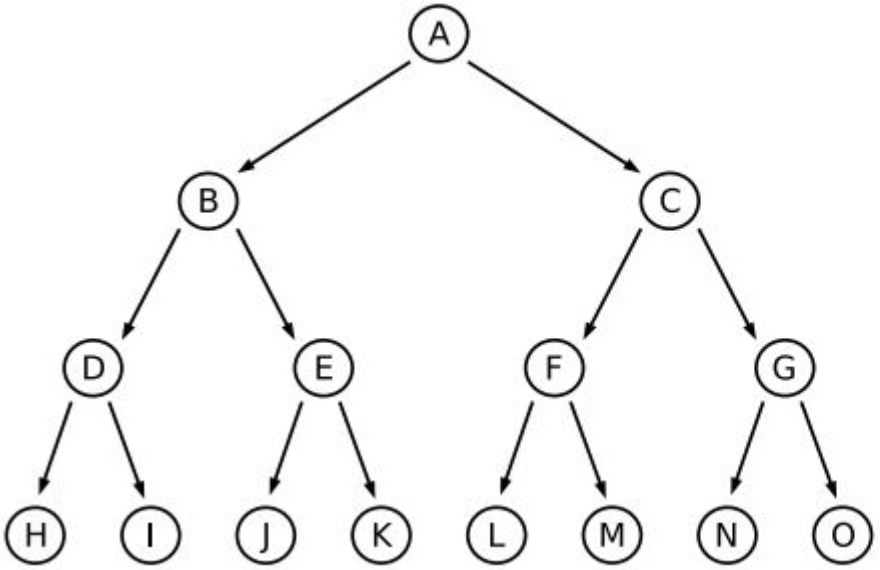
- b is the branching factor;
- m is the maximum depth of the search tree;
- d is the depth of the shallowest solution

Depth-limited search

= depth-first search with depth limit l

Implementation:

Nodes at depth l have no successors



$l = 0$: A

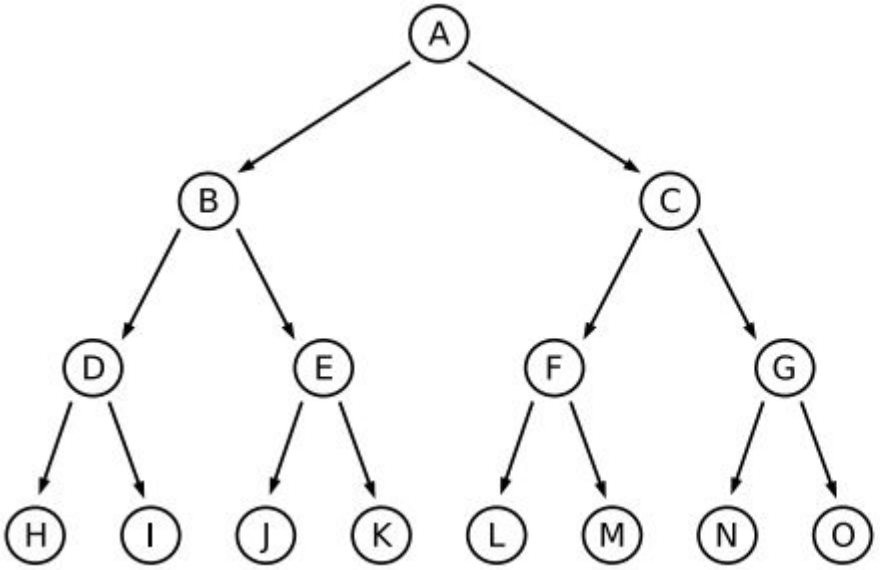
$l = 1$: A, B, C

Depth-limited search

= depth-first search with depth limit l

Implementation:

Nodes at depth l have no successors



$l = 0$: A
$l = 1$: A, B, C
$l = 2$: A, B, D, E, C, F, G

Properties of Depth-limited search

Complete??

Time??

Space??

Optimal??

- b is the branching factor;
- m is the maximum depth of the search tree;
- d is the depth of the shallowest solution
- l is the depth limit

Properties of Depth-limited search

Complete: No

Time: $O(b^l)$

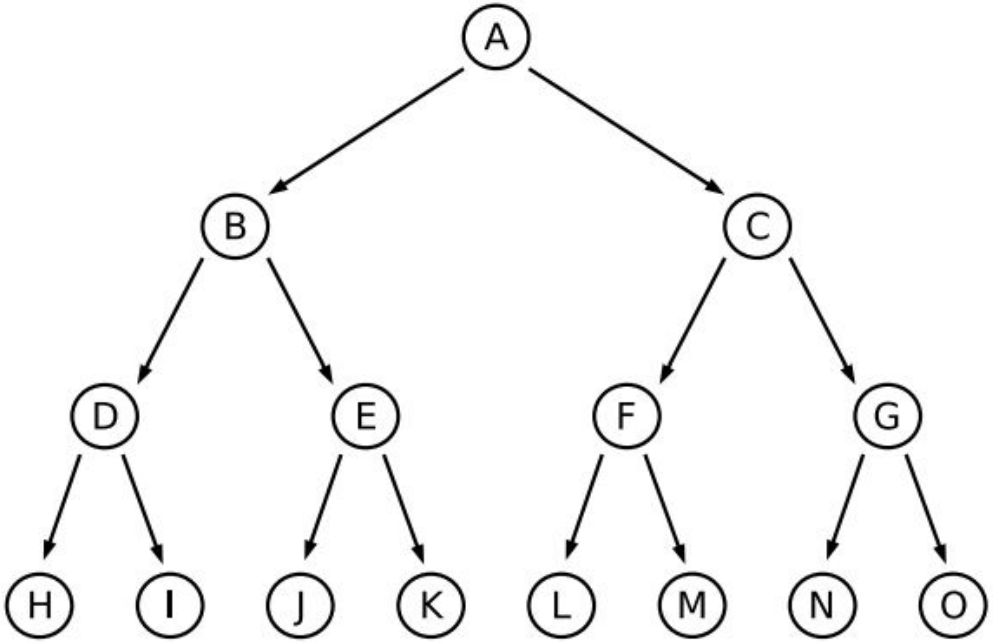
Space: $O(bl)$

Optimal: No

- b is the branching factor;
- m is the maximum depth of the search tree;
- d is the depth of the shallowest solution
- l is the depth limit

Iterative deepening search

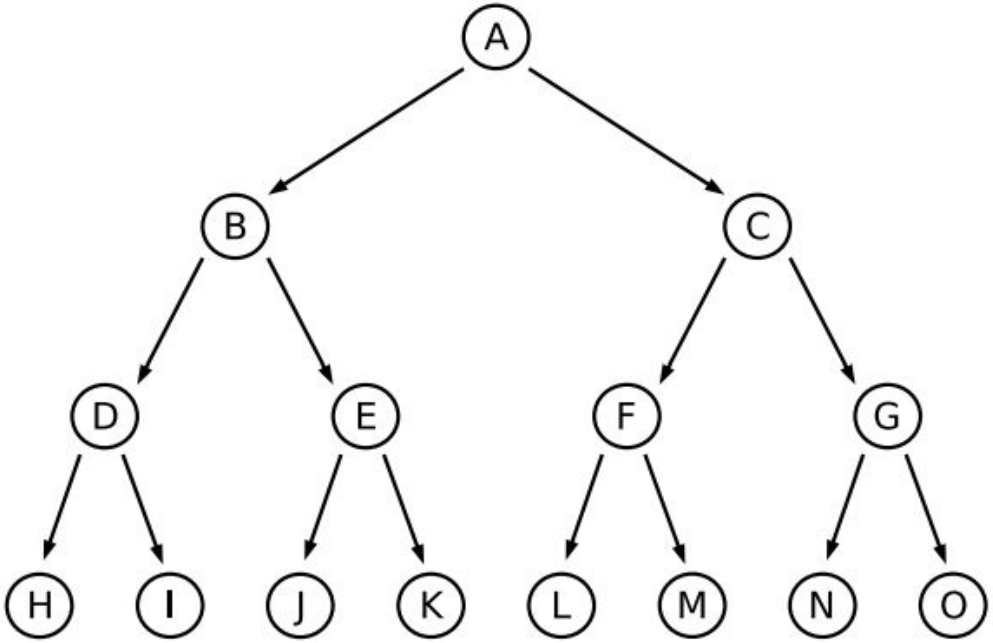
- Iterative deepening repeatedly applies depth-limited search with increasing limits.



A, A,

Iterative deepening search

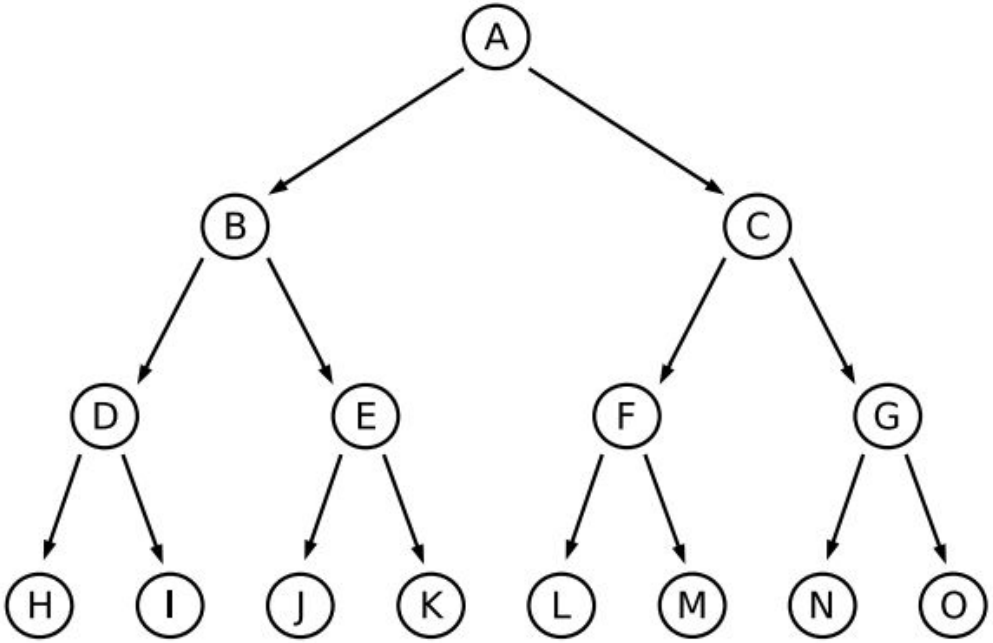
- Iterative deepening repeatedly applies depth-limited search with increasing limits.



A, A, B, C,

Iterative deepening search

- Iterative deepening repeatedly applies depth-limited search with increasing limits.



A, A, B, C, A, B, D, E, C, F,
G A, B, D, H, I, E, J, K, . . .

Iterative deepening search

- Iterative deepening search solves the problem of picking a good value for ℓ by trying all values: first 0, then 1, then 2, and so on.
- The search is performed until either a solution is found, or the depth-limited search returns the failure value rather than the cutoff value.
- Effectively combines the advantages of DFS and BFS

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Properties of iterative deepening search

Complete??

Time??

Space??

Optimal??

- b is the branching factor;
- m is the maximum depth of the search tree;
- d is the depth of the shallowest solution
- l is the depth limit

Properties of iterative deepening search

Complete?? Yes

Time?? $O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

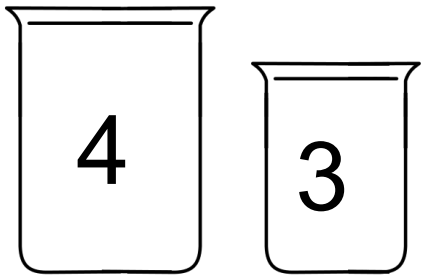
- *b* is the branching factor;
- *m* is the maximum depth of the search tree;
- *d* is the depth of the shallowest solution
- *l* is the depth limit

Comparing search algorithms

Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	No	No	Yes
Optimal cost?	Yes	No	No	Yes
Time	$O(b^d)$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(bm)$	$O(b\ell)$	$O(bd)$

- *b* is the branching factor;
- *m* is the maximum depth of the search tree;
- *d* is the depth of the shallowest solution
- l is the depth limit

water jug problems



two jugs of capacity 4 and 3 liters

Start State: (0,0)

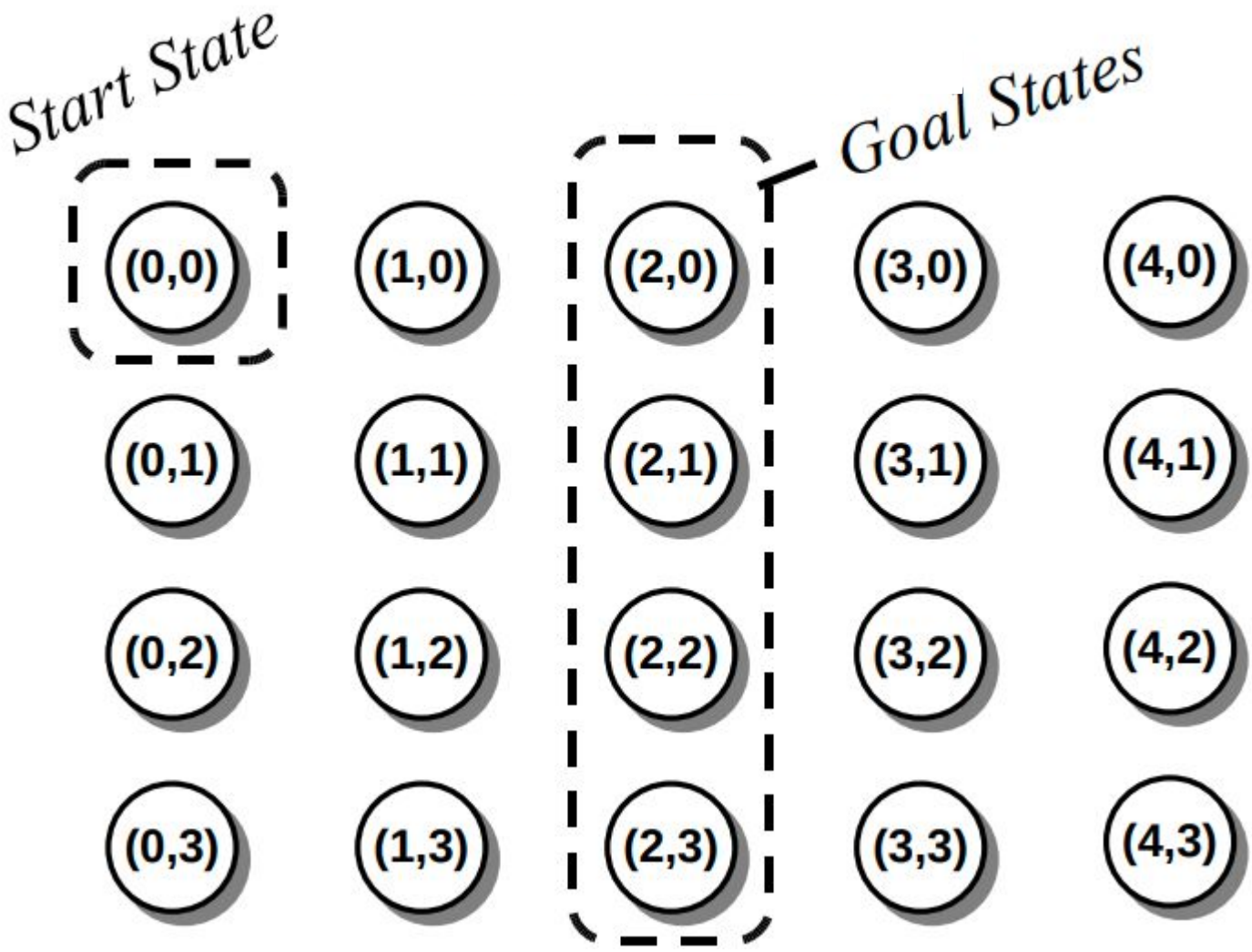
Goal State: (2, y)

$y \in \{0, 1, 2, 3\}$

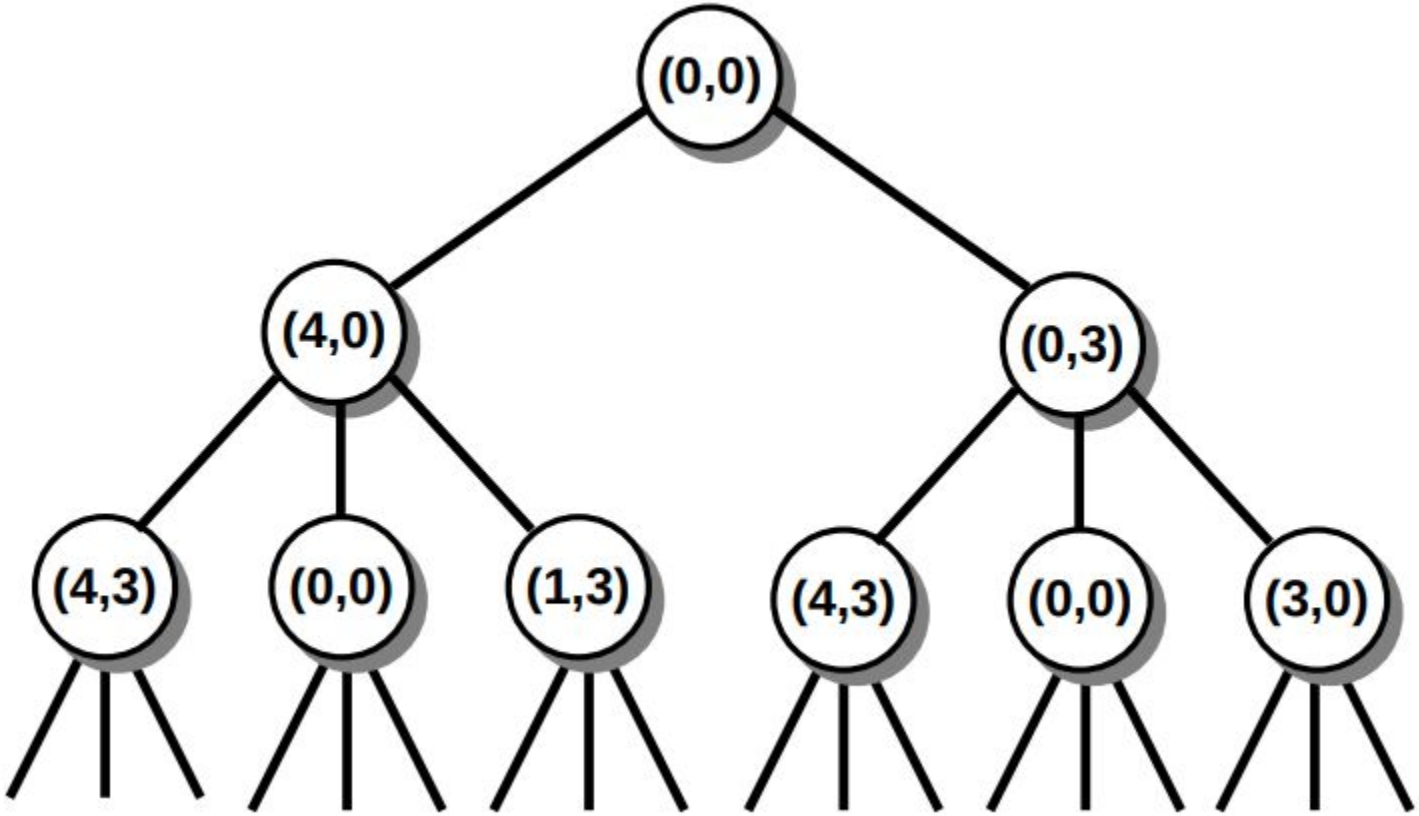
We basically perform three operations to achieve the goal.

1. Fill water jug.
2. Empty water jug
3. Transfer water from one jug to the other

water jug problems

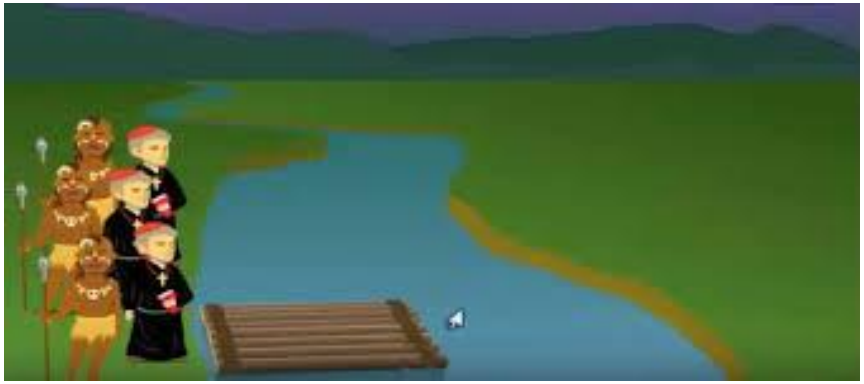


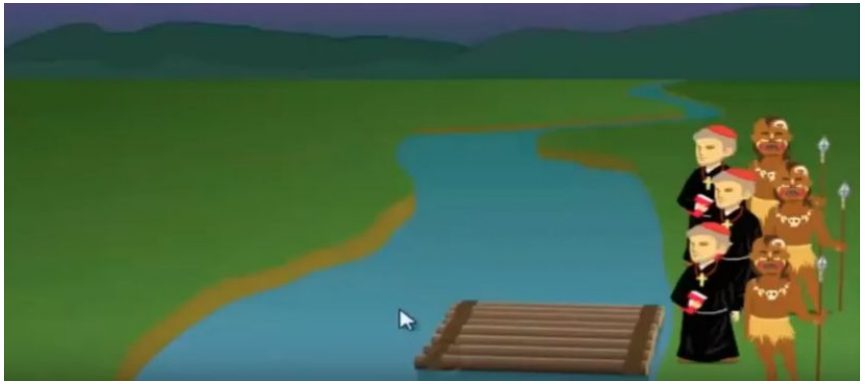
water jug problems



Missionaries and Cannibals problem

- Three missionaries and three cannibals must be brought over by boat
- The boat can carry at most two people.
- Constraint: missionaries present on the bank cannot be outnumbered by cannibals.





Missionaries and Cannibals problem

- **States**?
- **Initial state**?
- **Actions**?
- **Transition model**?
- **Goal state**?
- **Action cost**?

Missionaries and Cannibals problem

• **States**: Combination of missionaries and cannibals and boat on each side of the river

- number of missionaries on the left side of the coast, {0, 1, 2, 3}
- number of cannibals on the left side of the coast, {0, 1, 2, 3}
- position of the boat, {L, R}

• **Initial state**: 3 Missionaries, 3 Cannibals, and boat on the left side

E.g., (3, 3, L)

Missionaries and Cannibals problem

- **Actions**: raid the boat with one or two persons across the river in either direction to the other side, e.g., = {(1, 1), (2, 0), (0, 2), (1, 0), (0, 1)}
- **Transition model**: Maps a state and action to a resulting state. Note the constraint that missionaries can't be outnumbered by cannibals.
- **Goal state**: 3 Missionaries, 3 Cannibals, and boat on the right side. E.g., (0, 0, R)
- **Action cost**: require minimum numbers of movement.

More problems