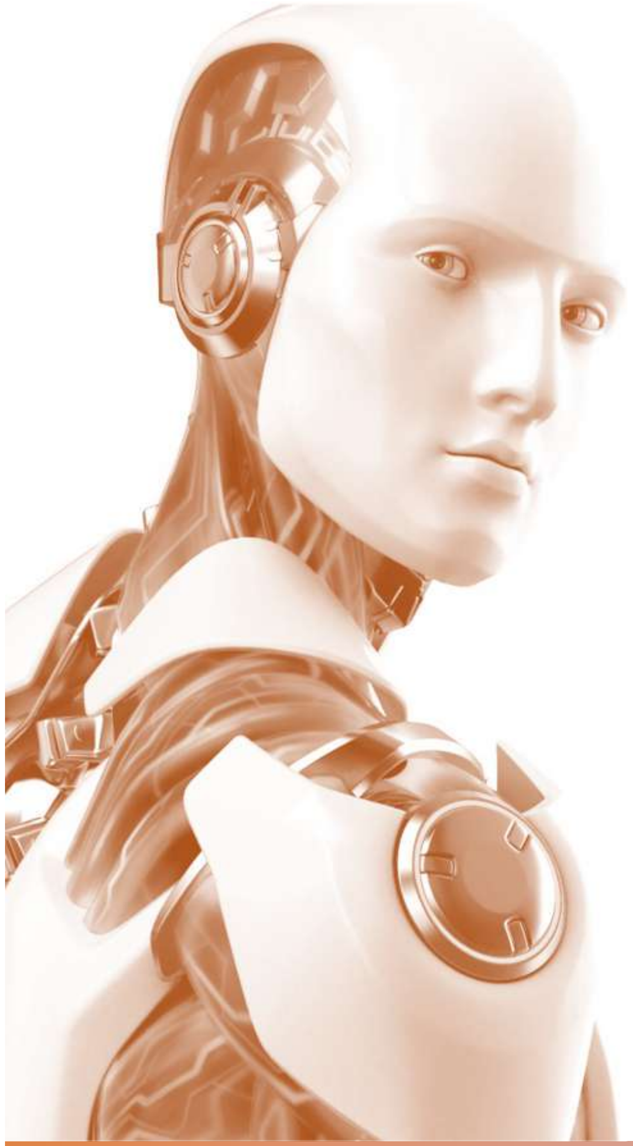
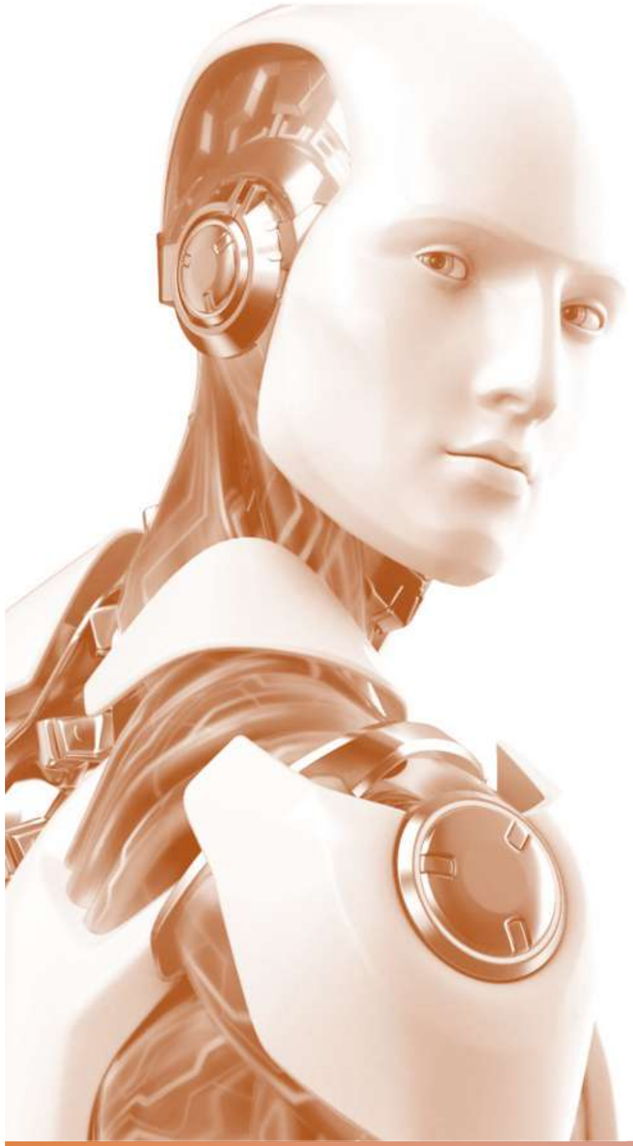


MC906/MO416 Introduction to AI

Lecture 4

UNICAMP



Introduction to AI

Lecture 4 - Problem-based agents and search without information

Profa. Dra. Esther Luna Colombini
esther@ic.unicamp.br

Prof. Dr. Alexandre Simões
alexandre.simoes@unesp.br

LaRoCS – Laboratory of Robotics and Cognitive Systems

Summary

- Model building and problems
  - Goal-based agent
- Search methods without information
  - Breadth-first search
  - Search with uniform cost
  - Depth-first search
  - Depth-limited Search
  - Iterative deepening search
  - Bidirectional search
- Exercise

Model building and problems

Water jugs problem

- There are two jugs: a 4-liter and a 3-liter jug, initially empty, and a fountain that spouts water in abundance.

- **Goal**: Get 2 liters in any of the jugs.
- Possible **actions:**
  - Fill the vases
  - Empty the vases
  - Complete one vase with another
  - Throwing the vase content at another

Review...

- Agent types:
  - **Simple reactive agent:** Select actions based on current perception, ignoring the rest of the perception history
  - **Model-based agent:** Controls the part of the world that it cannot see now while maintaining an internal state that depends on the history of perceptions
  - **Goal-based agent:** Combines your goal with information on the results of possible actions in order to choose actions that achieve its goals
  - **Utility-based agent:** Use a performance measure (utility function) that allows a comparison between different states of the world, allowing to select the sequence of actions

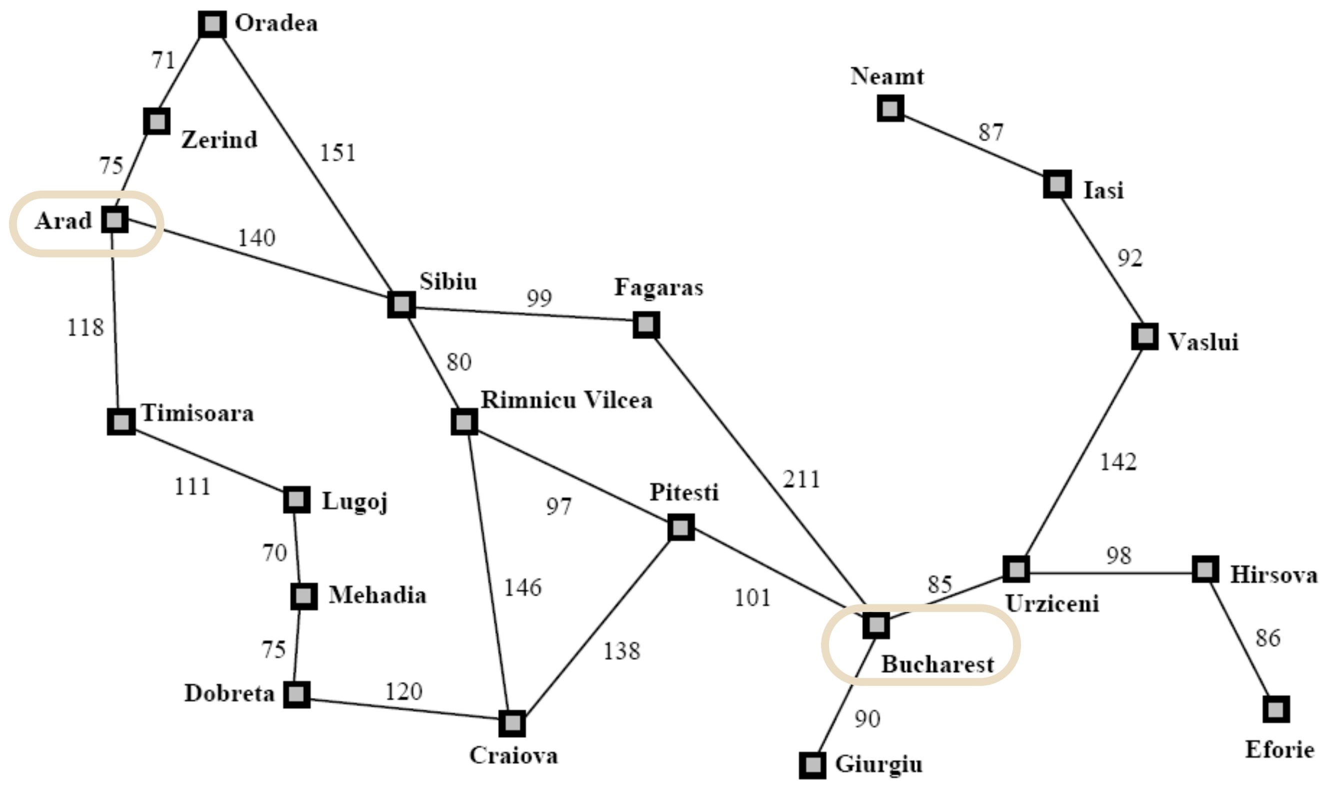
Goal-oriented agent

- How can an agent find a sequence of actions that achieve its goals when no single action is able to do so?

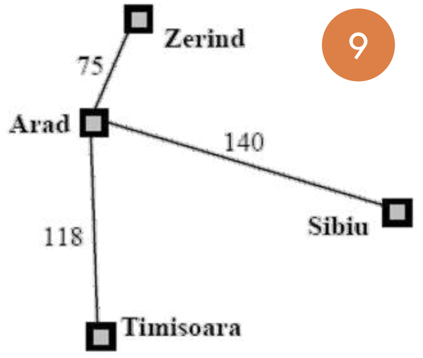
- **Objective (Goal):** set of states in the world

Solving the problem with search methods

- **Objective:** to go from Arad to Bucharest
- **Question:** how to model the problem? Observable, discrete and deterministic environment!



Problem definition



- Components:
  1. **Initial state:** where the agent starts.
     - Ex: *In(Arad)*
  2. **Agent's actions:**

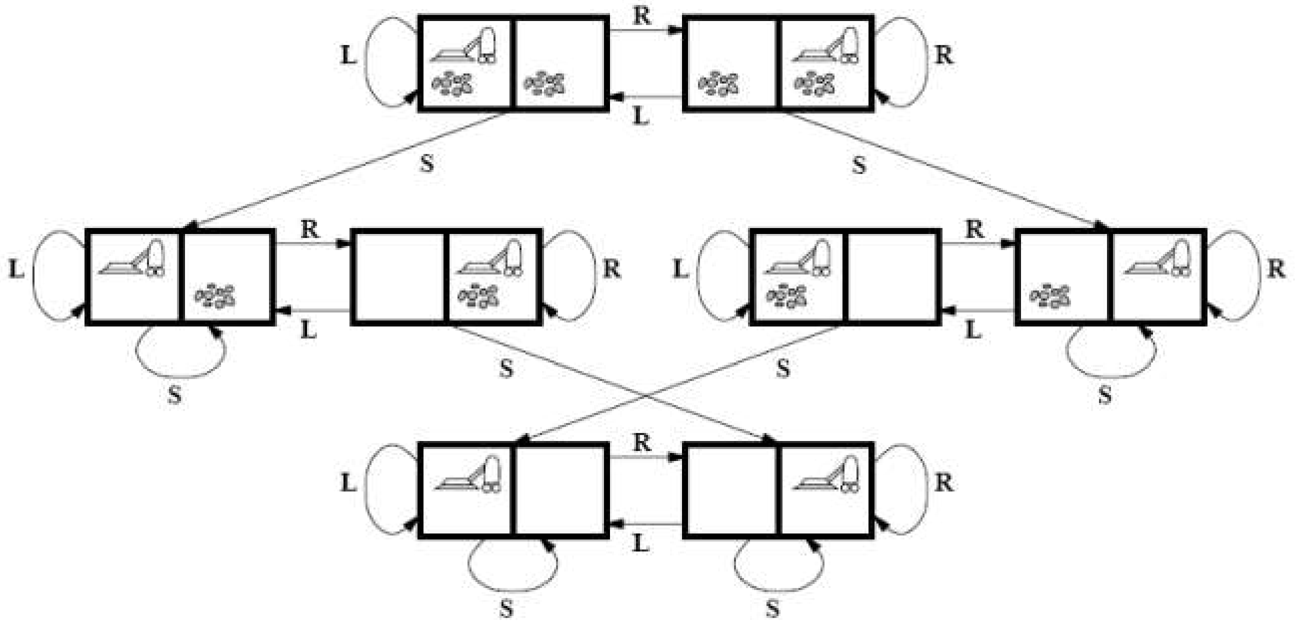
     SUCCESSOR(*x*) returns a set of ordered pairs <*action*, *successor*>, in that each action is one of the valid actions (**Applicable action**) in state *x*

     Example:

     InArad: {<*GoTo(Sibiu)*, *In(Sibiu)*> ,
     < *GoTo(Timisoara)*, *In(Timisoara)*>,
     < *GoTo(Zerind)*,*In(Zerind)*>}
  3. **Objective test:** determines whether a given state is the objective state (or goal)
  4. **Path cost:** function that assigns a numerical cost to each path

Vacuum Cleaner World

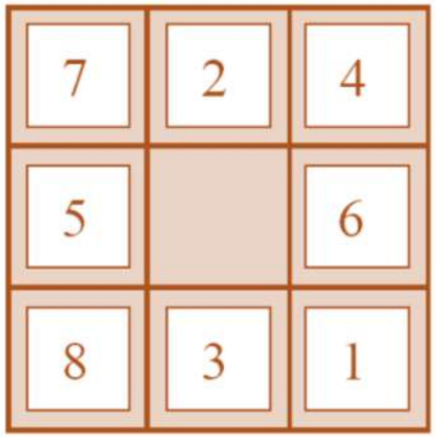
- **Initial state:** any state
- **Successor function:** generates valid states that result from trying to perform the three actions (L: Left, R: Right and S: Vacuum). The state space is:



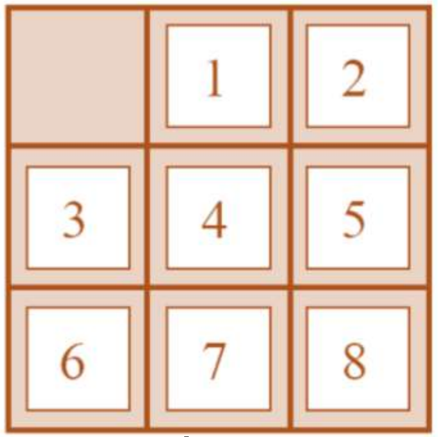
- **Objective test:** check if all squares are clean
- **Path cost:** each step costs 1

8-piece puzzle

- **Initial state:** any state
- **Successor function:** generates the valid states that result from the attempt to perform the four actions: empty space moves to Left, Right, Up, Down.
- **Objective test:** check if the state corresponds to the final configuration
- **Path cost:** each step costs 1



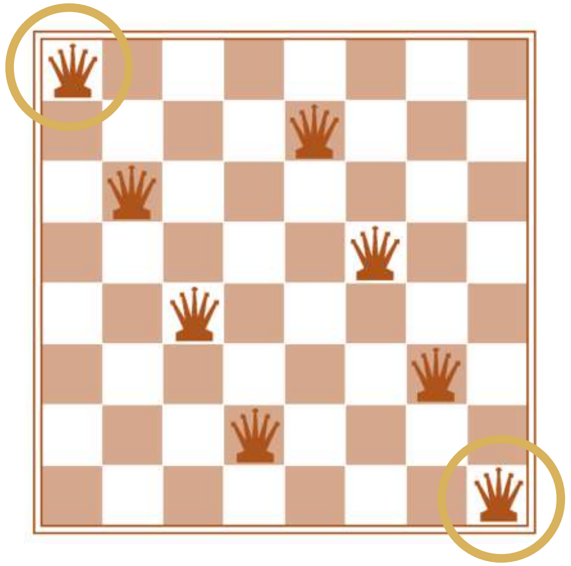
Initial state

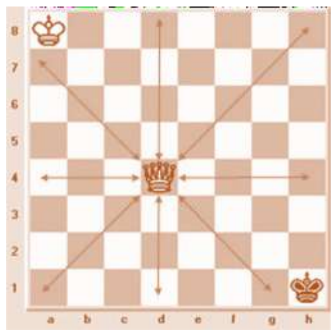


Final state

8 queens problem

- **Initial state:** empty board
- **Successor function:** place a queen in any empty square
- **Objective test:** 8 queens are on the board and none are attacked
- **Cost of the solution:** no interest (only the final state is important)





There are a total of 92 possible solutions and 12 distinct ones (excluding the symmetric solutions of rotation and reflection).

Real world problems ...

- Problems in determining trajectories (computer networks, military action planning, air travel, ...)
- VLSI layout problems
- Robot navigation issues
- Automatic assembly sequence problems
- Protein design
- Internet searches
- ...

A simple problem-solving agent

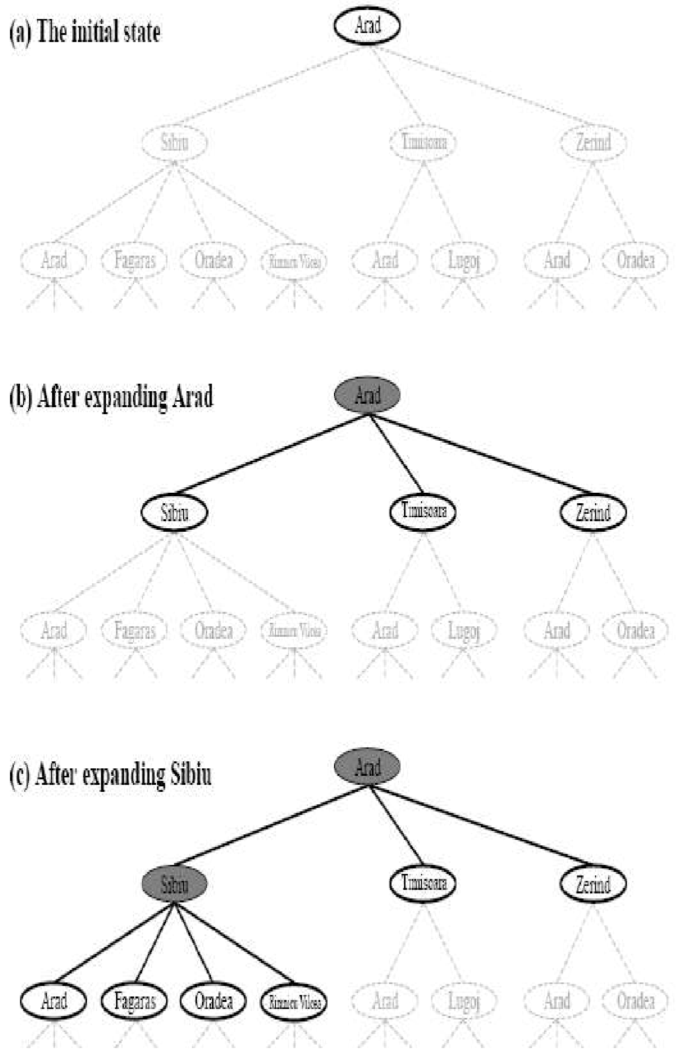
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
        persistent: **seq**, an action sequence, initially empty
                    **state**, some description of the current world state
                    **goal**, a goal, initially null
                    **problem**, a problem formulation

        state ← UPDATE-STATE(state, percept)

        if seq is empty then
                goal ← FORMULATE-GOAL(state)
                problem ← FORMULATE-PROBLEM(state, goal)
                seq ← SEARCH(problem)
                if seq = failure then return a null action
        action ← FIRST(seq)
        seq ← REST(seq)
        return action
```

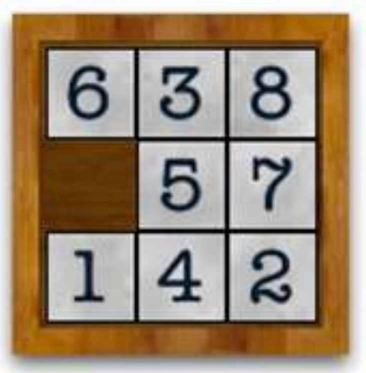
In search of solutions ...

- State Space Search
  1. Current state
  2. Expansion
  3. Generation of new states
- Tree search techniques:
  - What is the best strategy?



States

- How to represent the state of the game below?

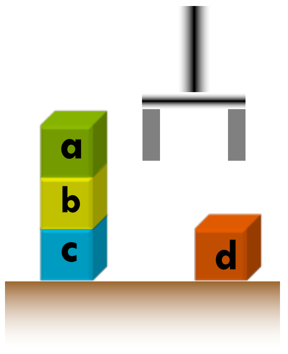


Vector : [6,3,8,*,6,7,1,4,2]

Logic :

on_place(1,6); on_place(2,3); on_place(3,8); on_place(4,0)...

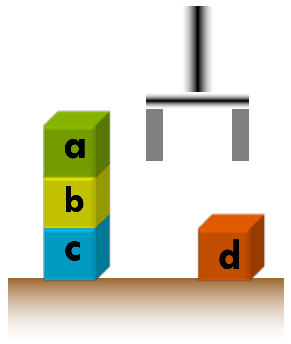
- Another Example → Blocks World



on(a,b)
on(b,c)
ontable(d)
ontable(c)
clear(a)
clear(d)
handempty

Characteristics of States

- Characteristics of States
  - **Negation as Failure:** The state represents only what is true (TRUE) in the world. What is not represented is considered false.
  - **Data structure**
    - A state can be a vector of strings (name of predicates - attribute) Bad !!!
    - A state can be a vector of integers where each integer represents a predicate (Mapping each predicate to an integer).
      - Ex: world of Blocks



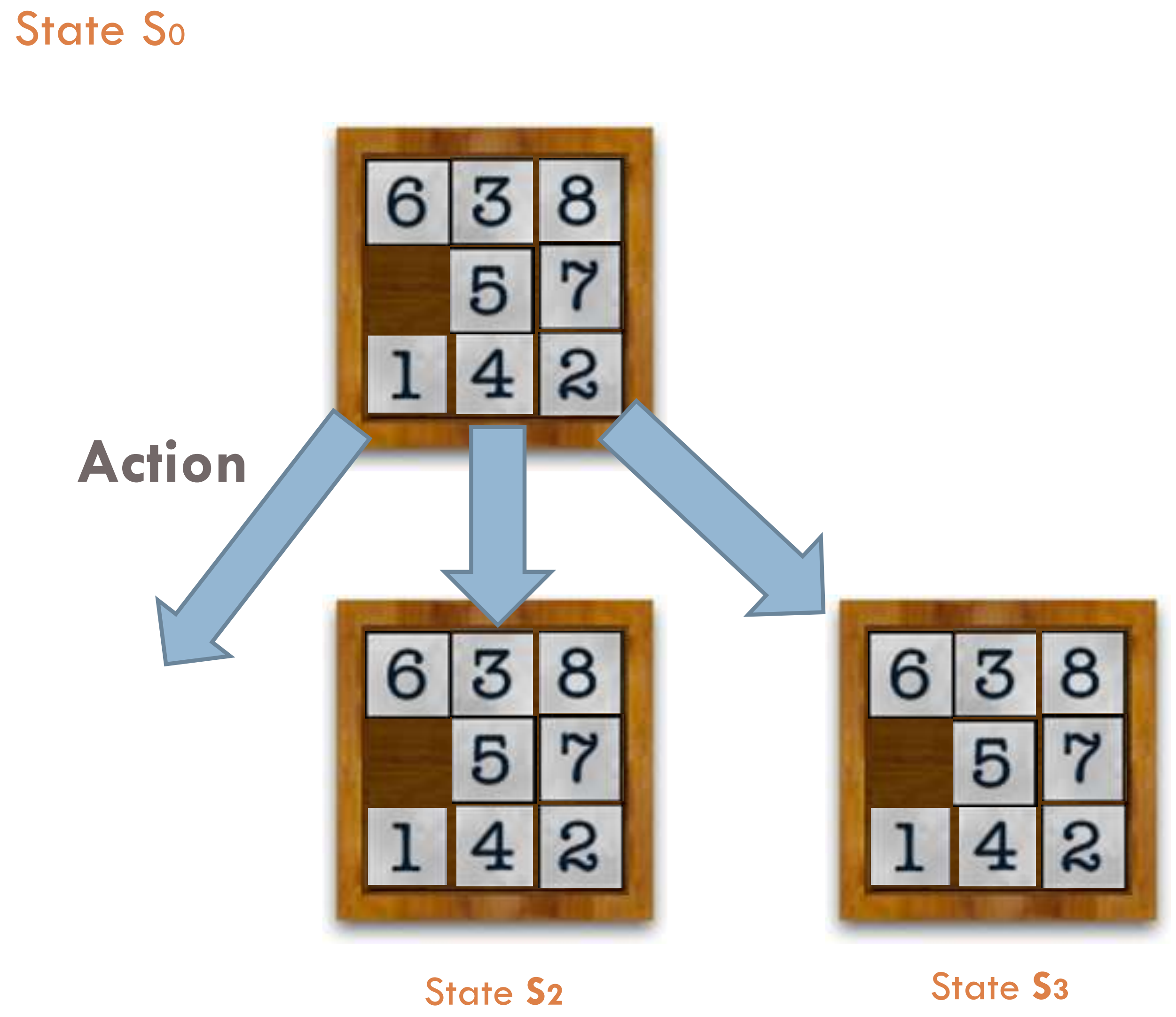
predicate	Number	predicate	Number
handempty	1	on(a,a)	6
holding (a)	2	on(a,b)	7
holding (b)	3	on(a,c)	8
holding (c)	4	on(a,d)	9
holding (d)	5	on(b,a)	10 .....

States

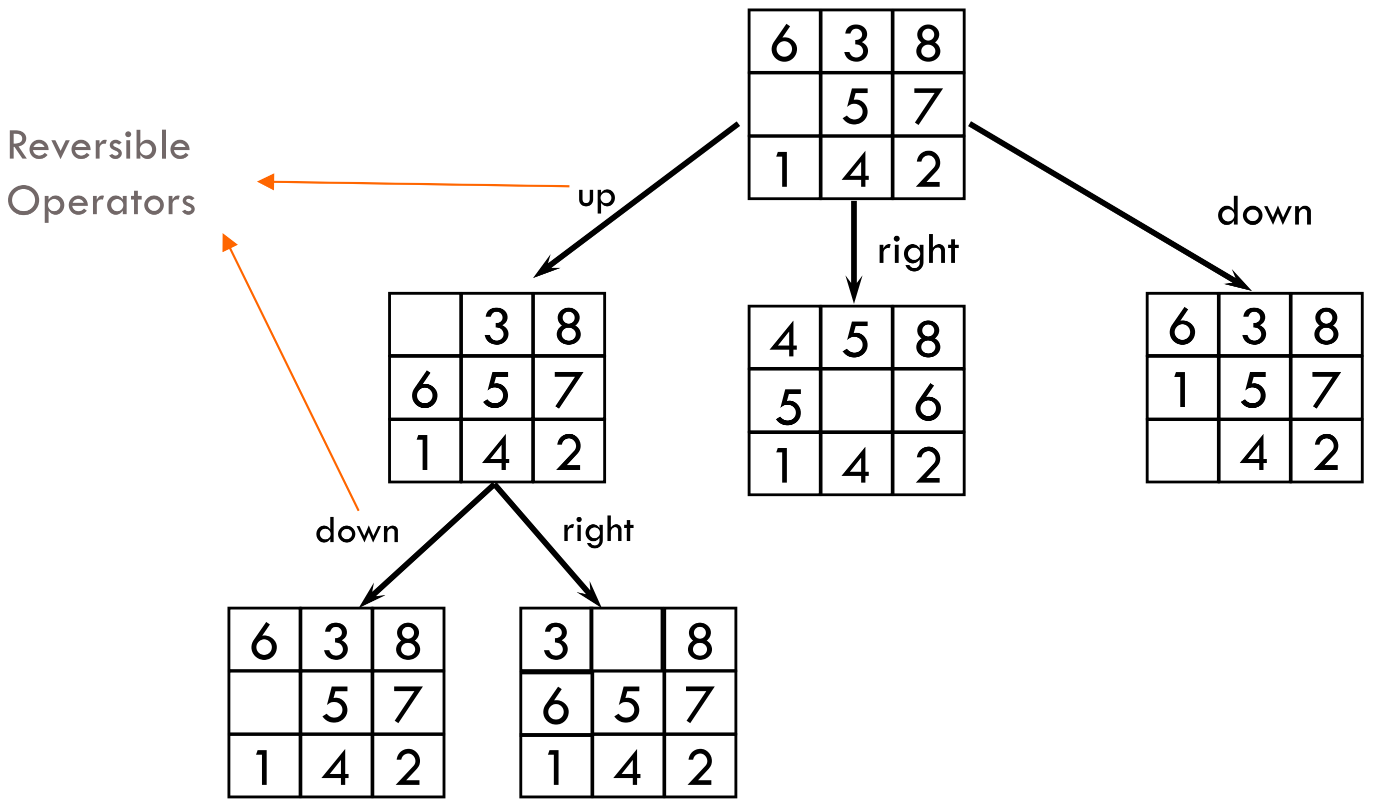
- Examples of Data Structures
  - Integers Vector
    - [1,4,6,7,3,2]
    - Querying a predicate is slow (traverses the entire vector)
    - Determining the set of predicates is quick (it is the vector itself)
  - Representative string
    - 111101100
    - Quick query of a predicate (just check the position)
    - Determining the set of predicates is slow (go through the entire string)

- All of this must be defined at the time of structuring knowledge representation
- Define which representation to use (vector, matrix, Logic mapped to integers, etc.)
- Define representation in a coherent Data Structure

State transition

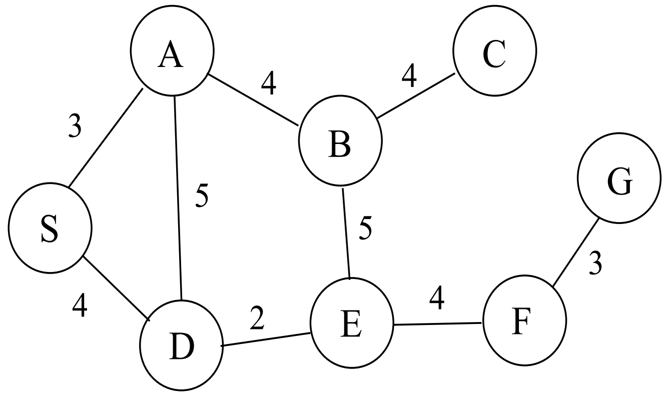


Actions: movement of empty space

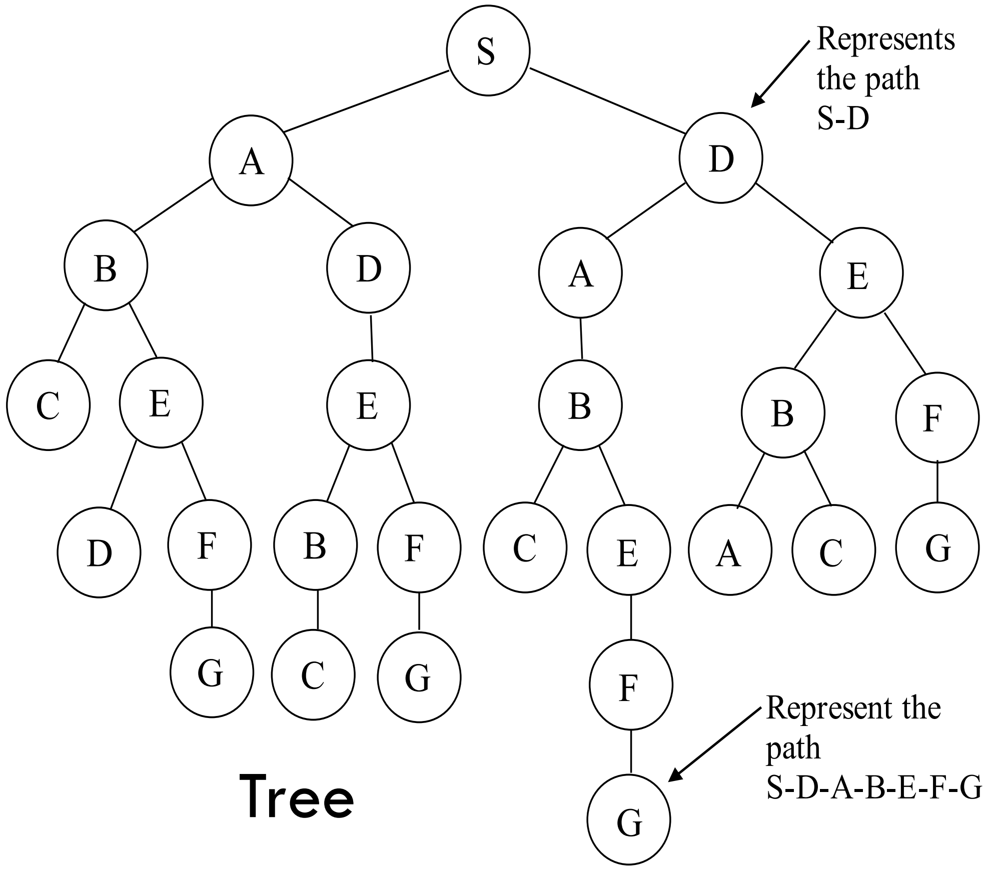


State Space in Graph and Tree

- A graph and its tree-like equivalent



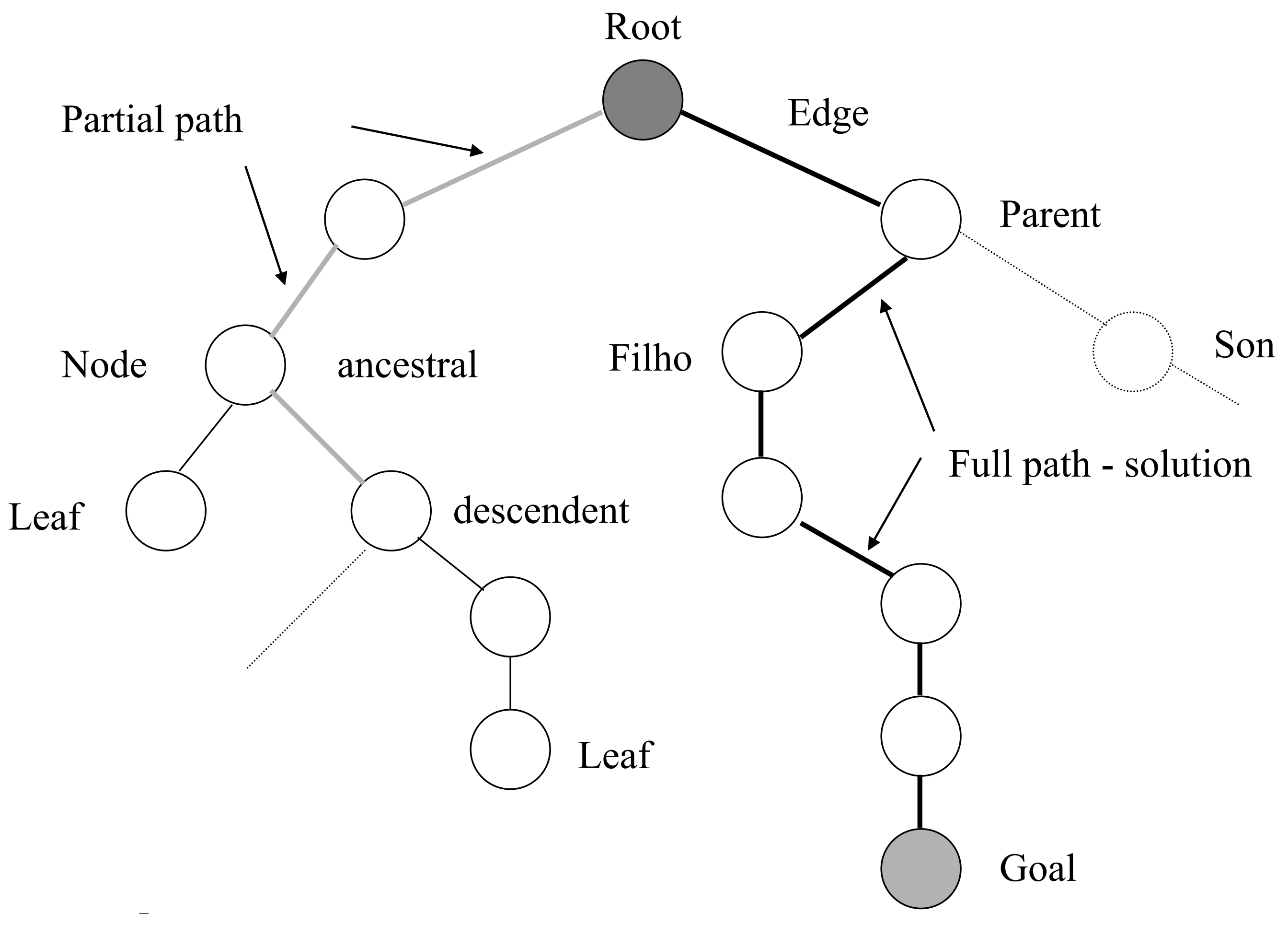
Graph



Tree

Search tree

- Detailing a tree

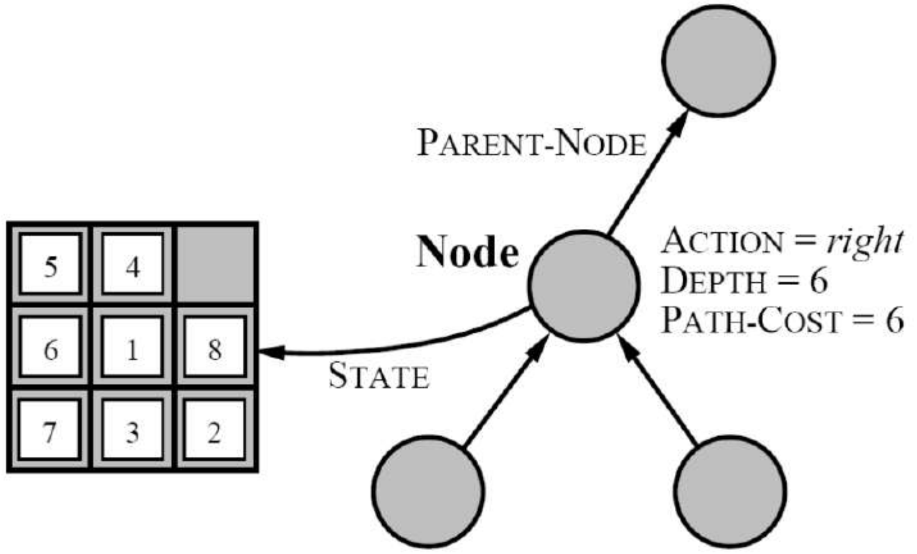


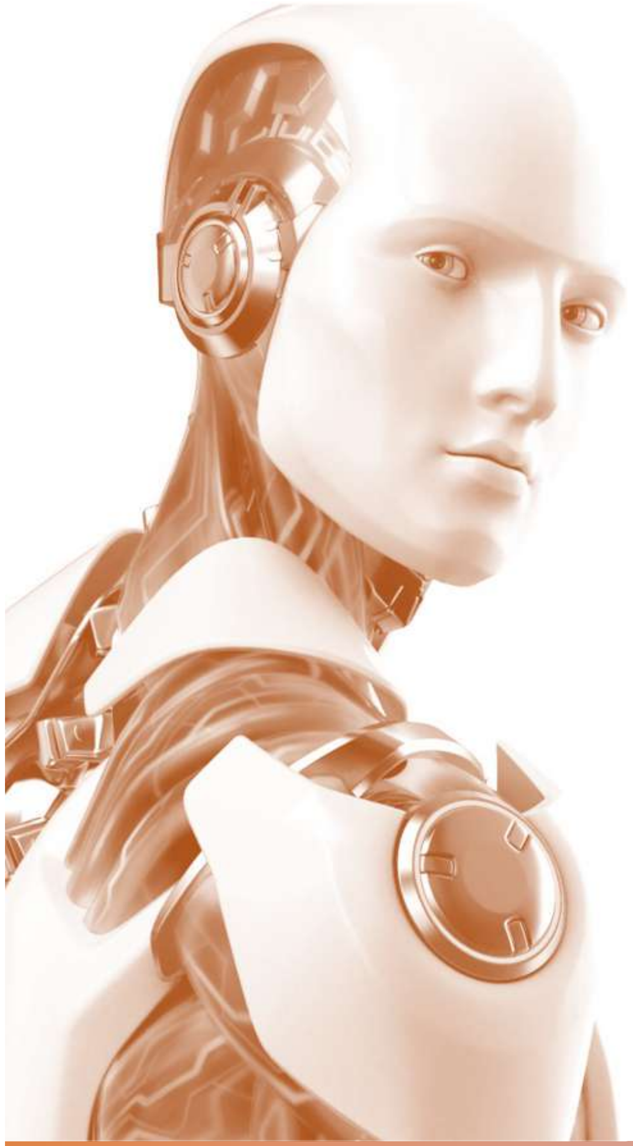
General tree search algorithm

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the list using the initial state of problem
    loop do
        if the list is empty then return failure
        choose a leaf node and remove it from the list
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the list
```

Components of a node

- **State:** the state in the state space that the node corresponds to
- **Parent node:** the node in the search tree that generated this node
- **Action:** the action that was applied to the parent to generate the node
- **Path cost:** the cost, traditionally denoted by g(x), of the path from the initial state to the node
- **Depth:** the number of steps along the path, from the initial state





MC906/MO416 Introduction to AI

Lecture 4 – PART II

Blind Search Methods

Representing a collection of nodes

- Node Collection: Queue
  - Possible operations on the queue:
  - CREATE-QUEUE (element, ...): creates a queue with the given element (s)
  - EMPTY?(queue): returns true only if there are no more elements in the queue
  - FIRST(queue): returns the first element of the queue
  - REMOVE-FIRST(queue): returns FIRST (queue) and removes it from the queue
  - INSERT (element, queue): inserts an element into the queue and returns the resulting queue
  - INSERT-ALL(elements, queue): inserts a set of elements into the queue and returns the resulting queue

General tree search algorithm

```
function TREE-SEARCH(problem, list) returns a solution, or
list
   list ← INSERT(CRREATE-NODE(INITIAL-STATE[problem]),
   list)
   repeat
        if EMPTY?(list) then return fail
        node ← REMOVE-FIRST(list)
        if TEST-GOAL[problem] applyied STATE[node]
                sucessful then return SOLUTION(node)
        list ← INSERT-ALL(EXPAND(node, problem), list)
```

Node expansion strategies

- **Blind search** strategies (without information about the problem):
  - Breadth-first search
  - Search with uniform cost
  - Depth-first search
  - Depth-limited Search
  - Iterative deepening search
  - Bidirectional search

Performance measures

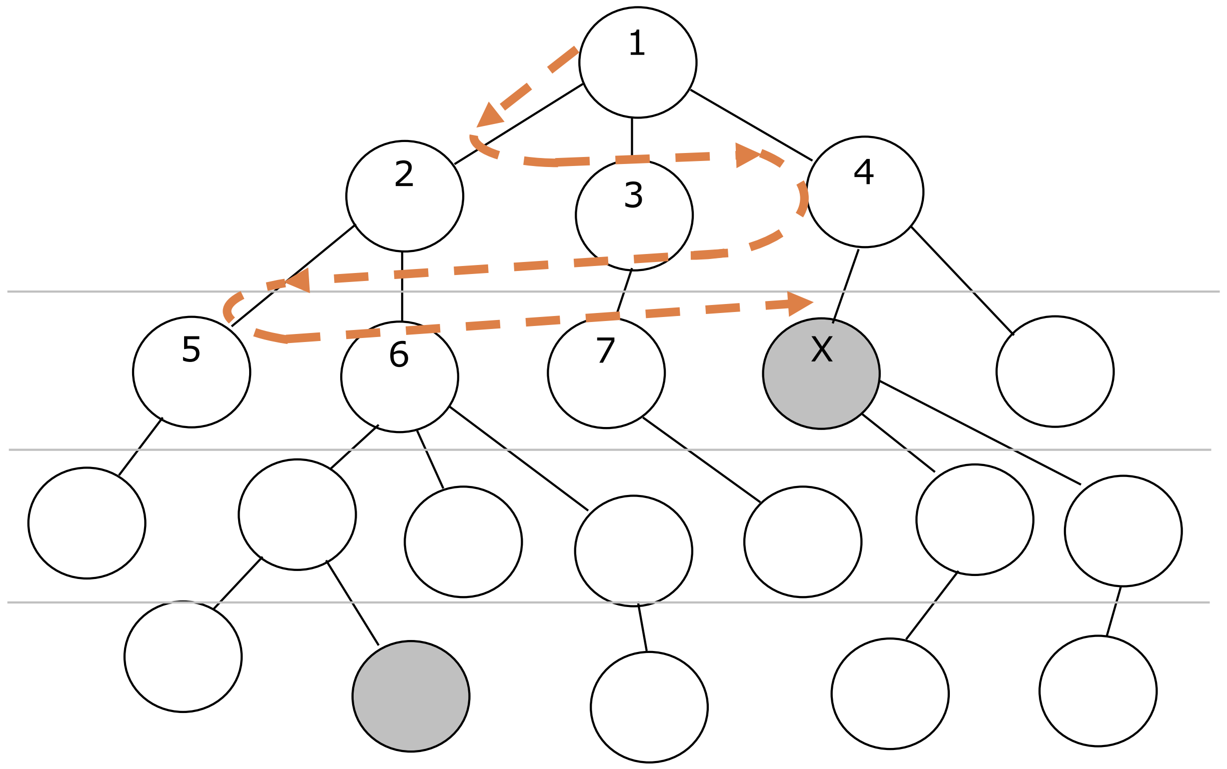
- **Completeness:** Does the algorithm guarantee to find a solution when it exists?
- **Optimization:** Does the strategy find the optimal solution?
- **Time complexity:** How long does it take to find a solution?
- **Memory complexity:** How much memory is needed to perform the search?

Breadth-first Search

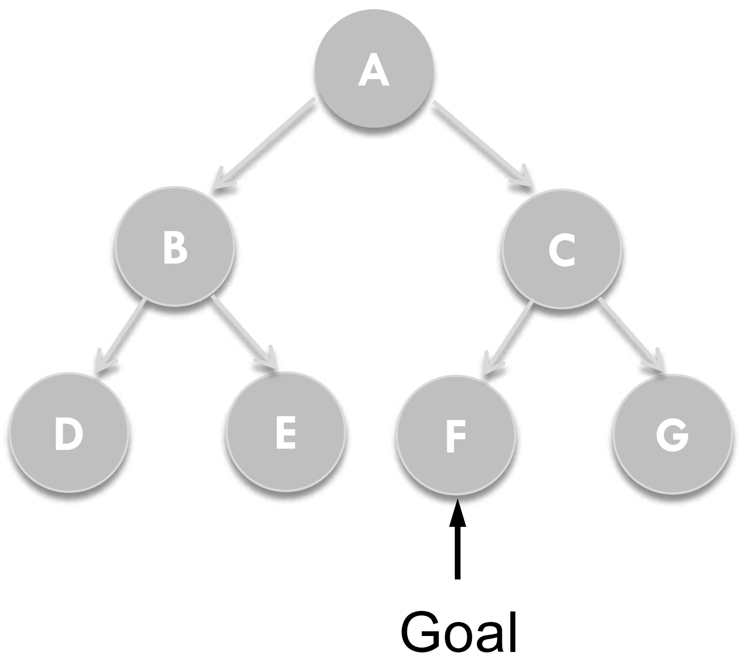
- The root node is expanded first, then all successors in the root only, then the successors of those nodes, and so on
- It can be implemented by doing:
  - SEARCH-IN-TREE (problem, QUEUE-FIFO())
- QUEUE "First In First Out":
- Place all newly arrived successors at the end of the queue
- Choose as successor the first node in the queue (shallow nodes expanded first)

Breadth-first Search

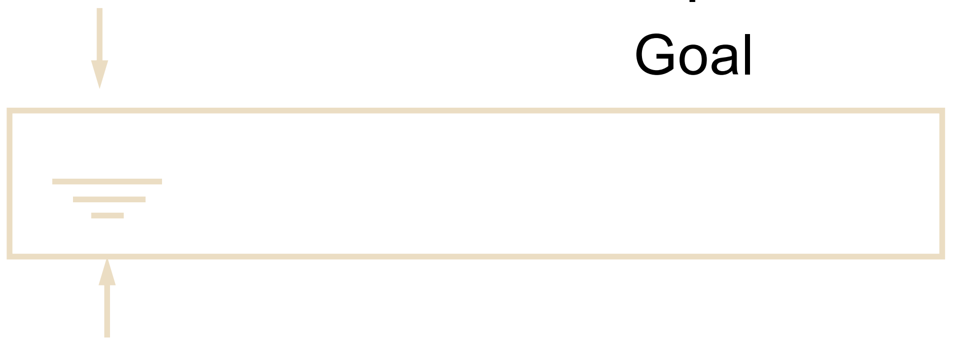
- Path strategy in search of the solution:



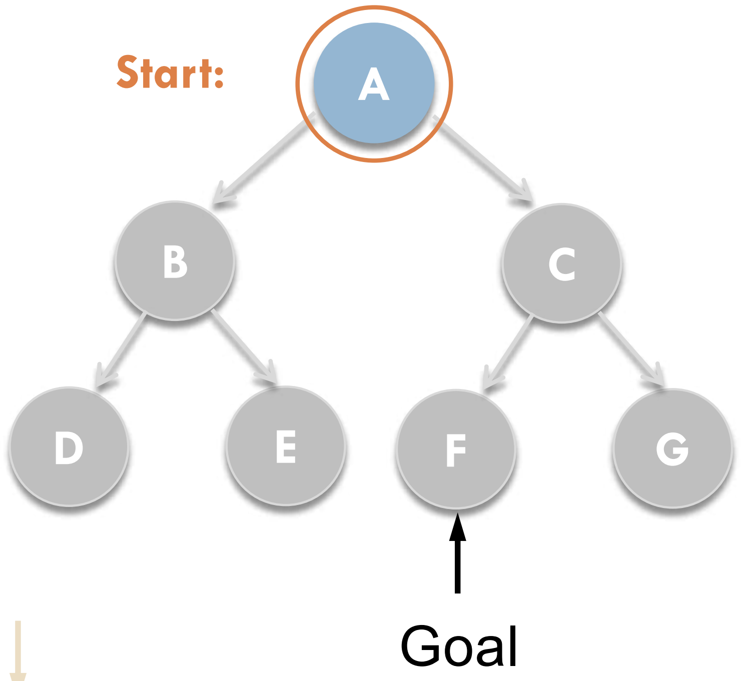
Breadth-first Search: example



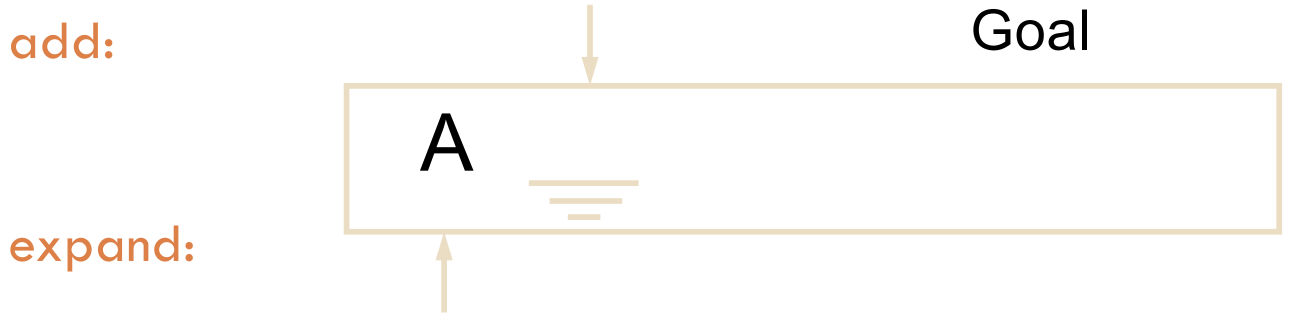
Queue:



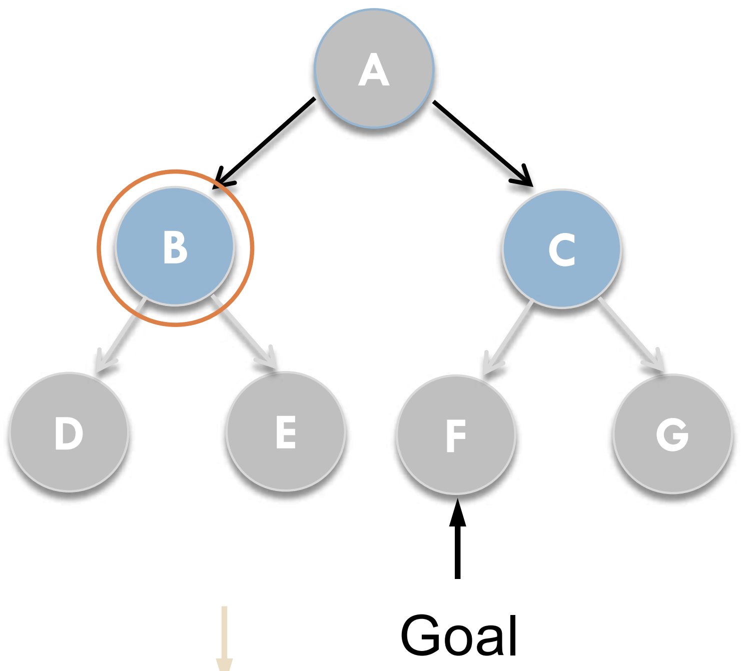
Breadth-first Search: example



Queue:

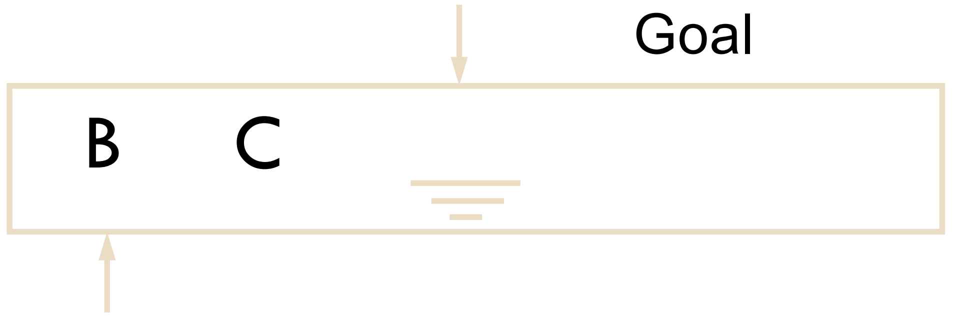


Breadth-first Search: example



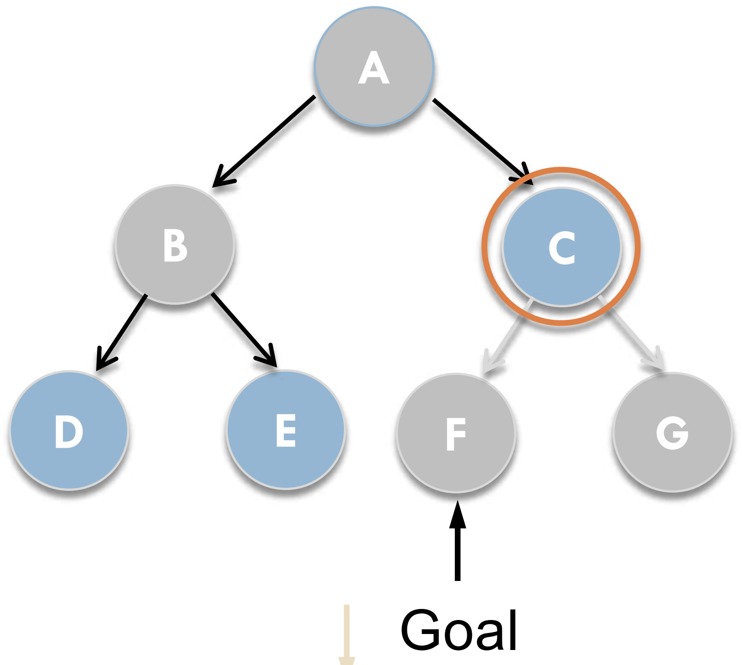
Queue:

add:



expand:

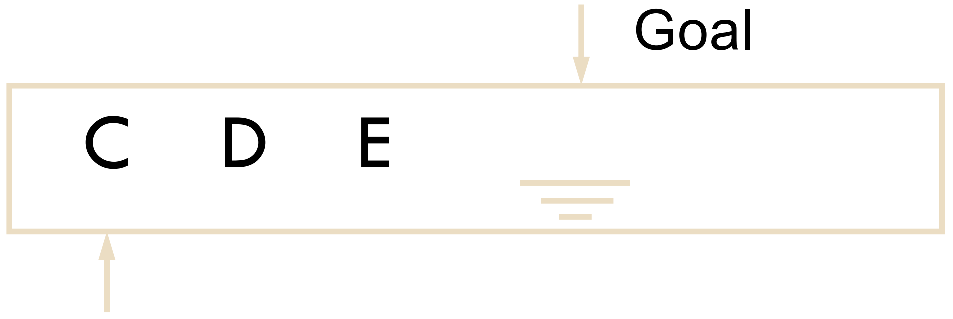
Breadth-first Search: example



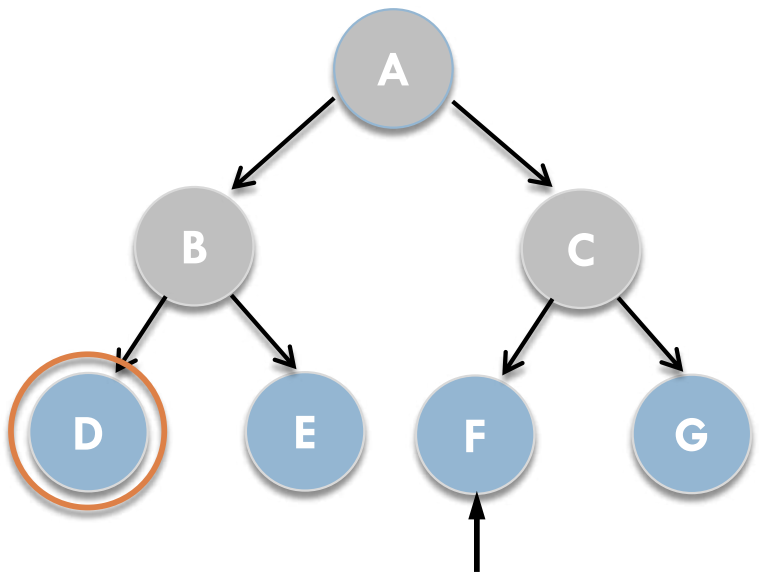
Queue:

add:

expand:



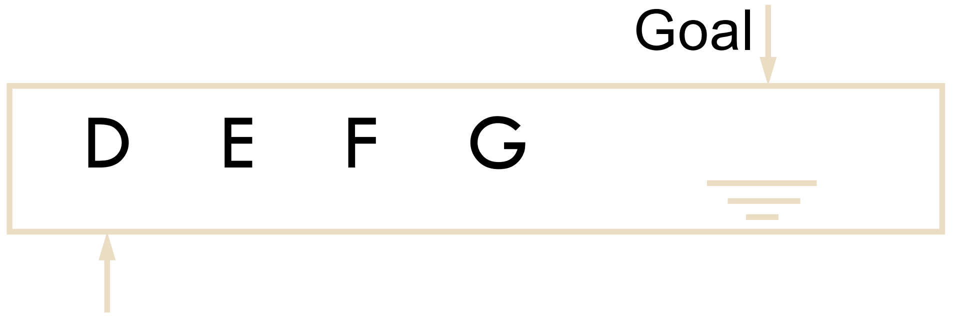
Breadth-first Search: example



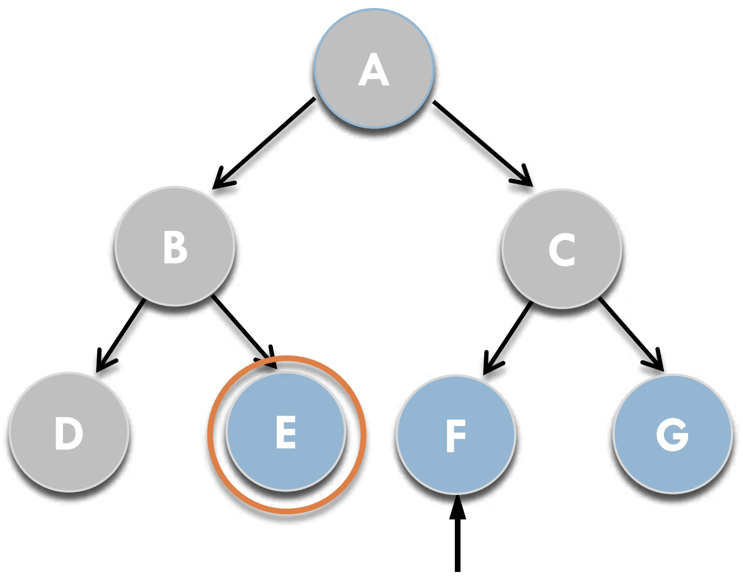
Queue:

add:

expand:



Breadth-first Search: example



Goal

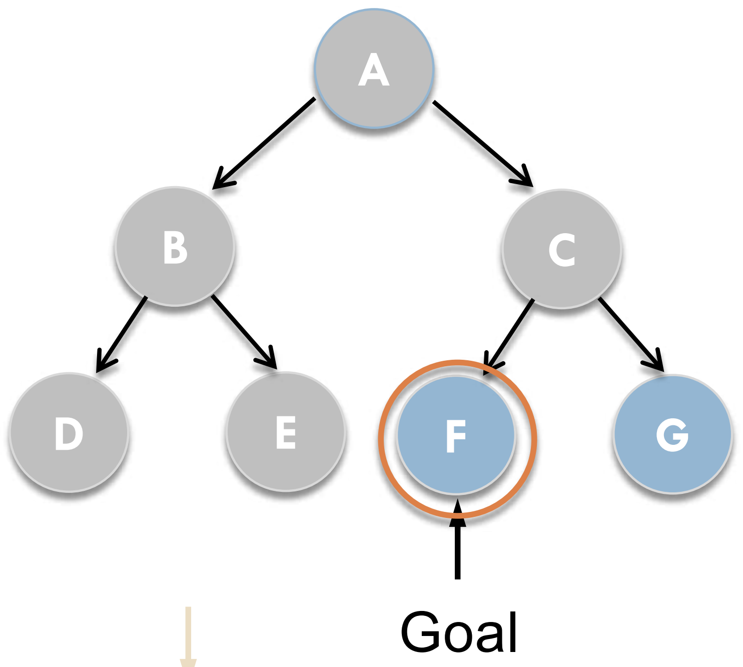
Queue:

add:

E F G

expand:

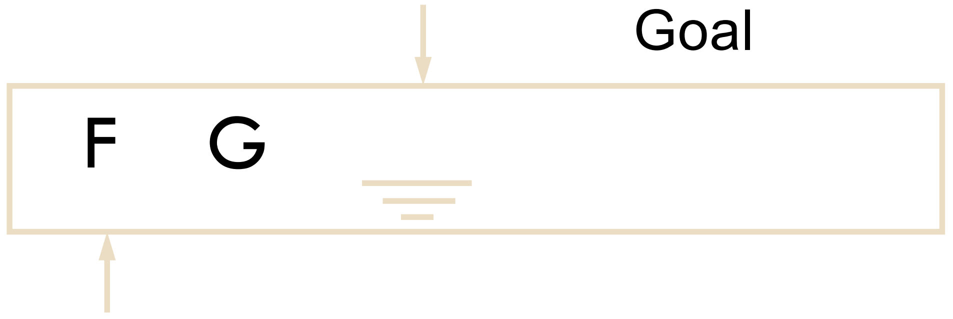
Breadth-first Search: example



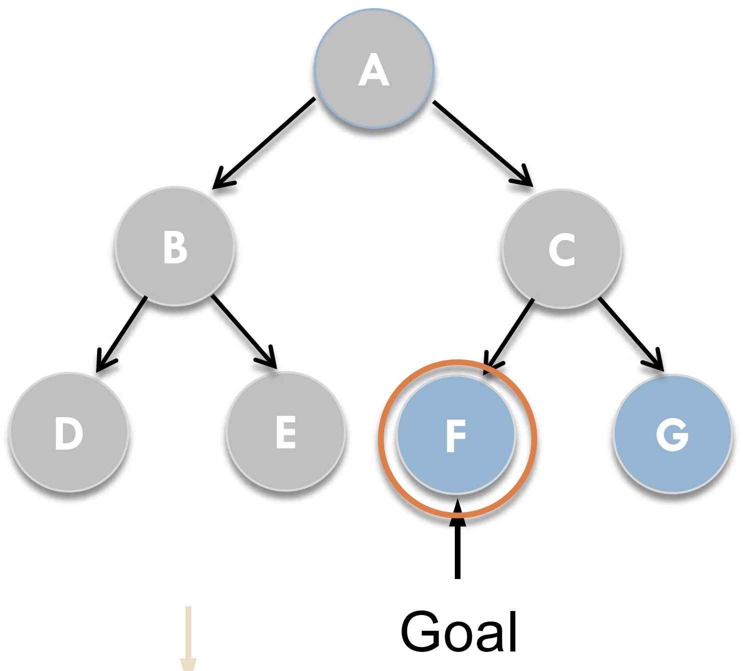
Queue:

add:

expand:



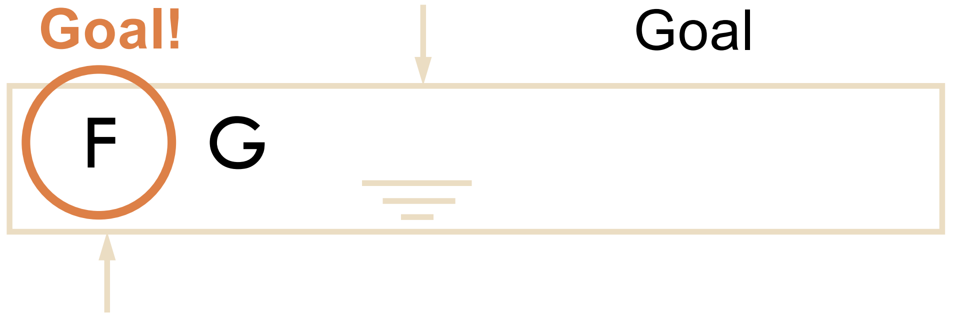
Breadth-first Search: example



Queue:

add:

Goal!

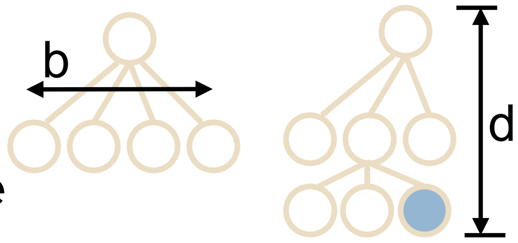


expand:

Breadth-first Search: performance

- Consider:
  - *b*: average number of successors for each node
  - d: shallowest depth of solution
- **Completeness:** complete, provided that d is finite
- **Optimization:** optimal, if all paths have the same cost
- **Time Complexity:**
  - Nodes per level: $1 \rightarrow b^1, 2 \rightarrow b^2, 3 \rightarrow b^3, \ldots, n \rightarrow b^n$
  - All nodes are expanded to level d, except the last (solution). Like: $b+b^2+b^3+\ldots+b^d+(b^{d+1}-b) = O(b^{d+1})$
- **Memory Complexity:**

  Nodes that have already been visited need to be stored in another vector in memory, as it may be necessary to consult them after finding the goal to determine the path from the initial state to the goal. Therefore, every node remains in memory (memory complexity is equal to time complexity)



Breadth-first Search: summary

- Always expands the shallowest node
- Memory and time requirements are a big problem

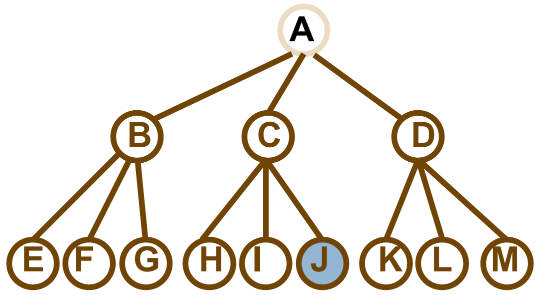
Depth	Nodes	Time	Memory
2	1,100	0,11 seg	1 megabyte
4	111,100	11 seg	106 megabyte
6	10^7	19 min	10 gigabyte
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabyte
12	10^{13}	35 years	10 pentabyte
14	10^{15}	3.523 years	1 exabyte

- Suppose b=10, analyse of 10,000 nodes/sec and 1,000 bytes/node

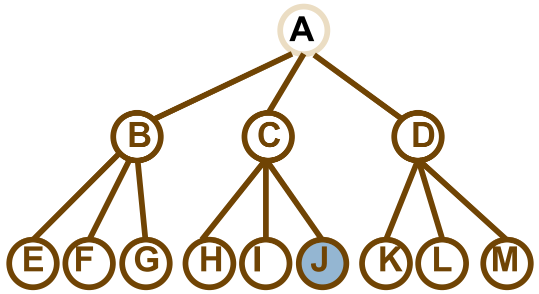
Uniform Cost Search

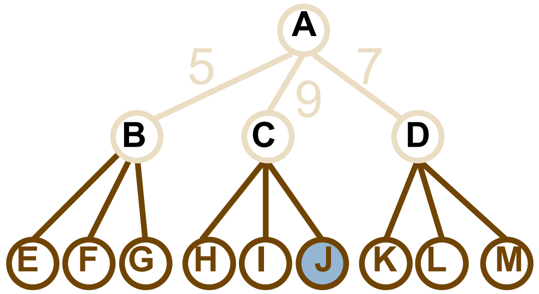
- **Breadth search:** optimal when the cost of all steps are equal
- **Uniform cost search:** instead of expanding the shallowest node, expand the node with the lowest cost path. Expands nodes in order of increasing cost
- Uses g(n)
  - "It doesn't matter how many steps the path takes, but only its total cost"
  - If all step costs are the same, it is identical to the search in extension
  - Use a priority Queue!

Uniform Cost Search: example

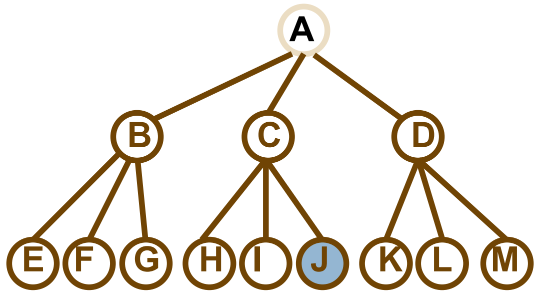


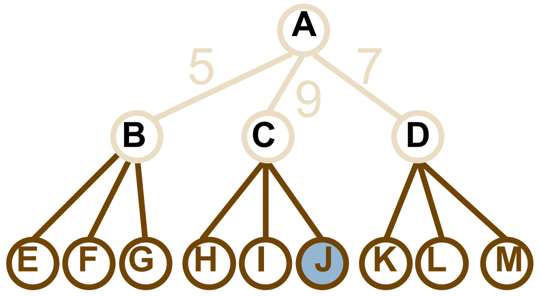
Uniform Cost Search: example

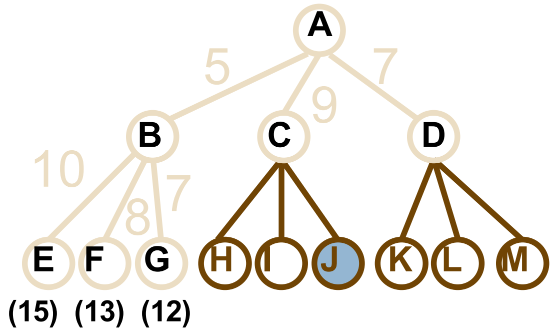




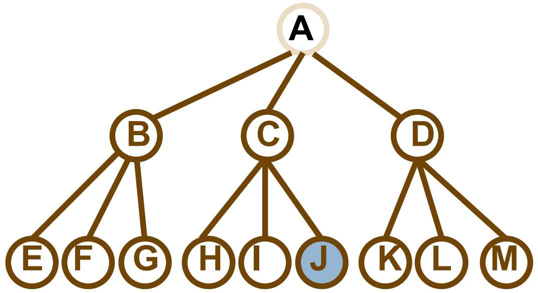
Uniform Cost Search: example

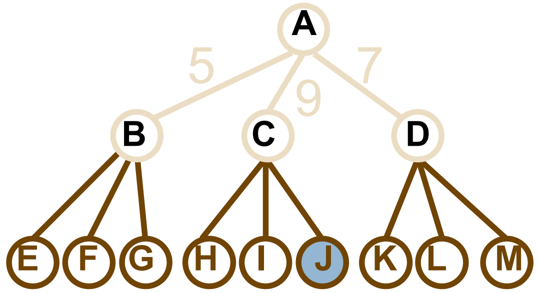


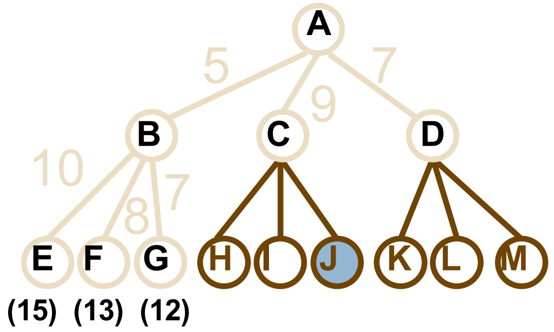


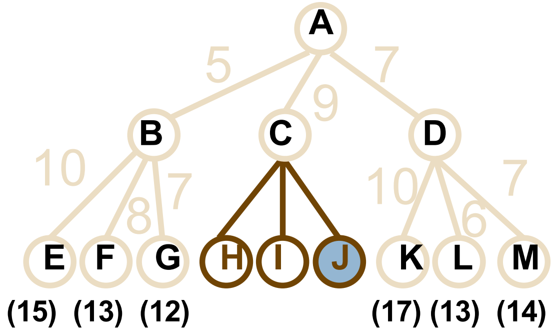


Uniform Cost Search: example

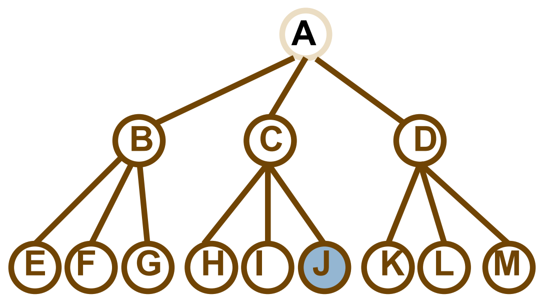


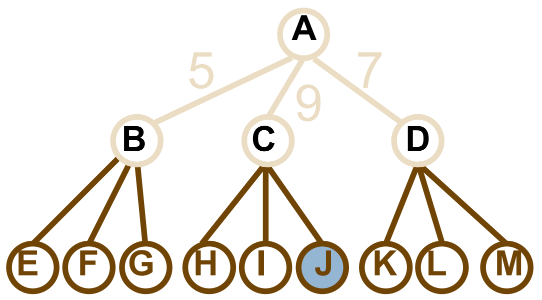


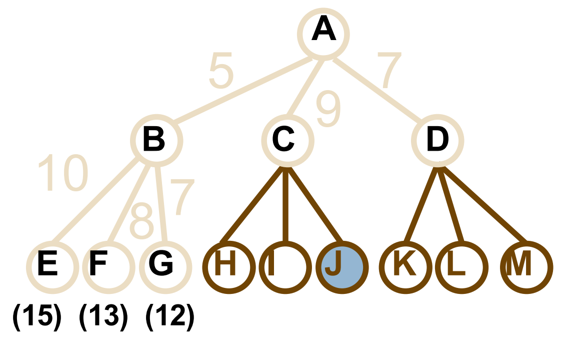


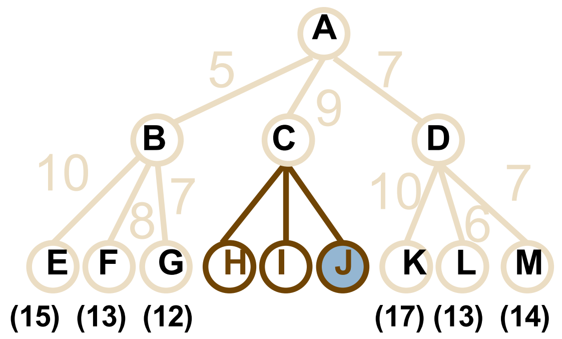


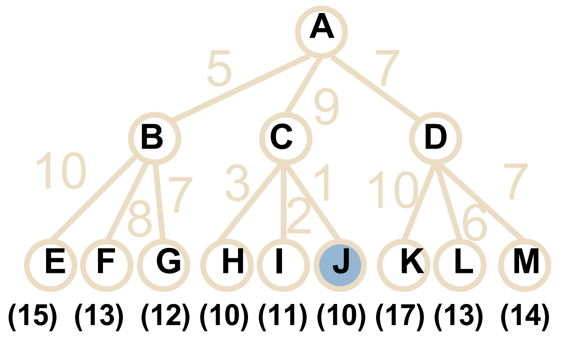
Uniform Cost Search: example



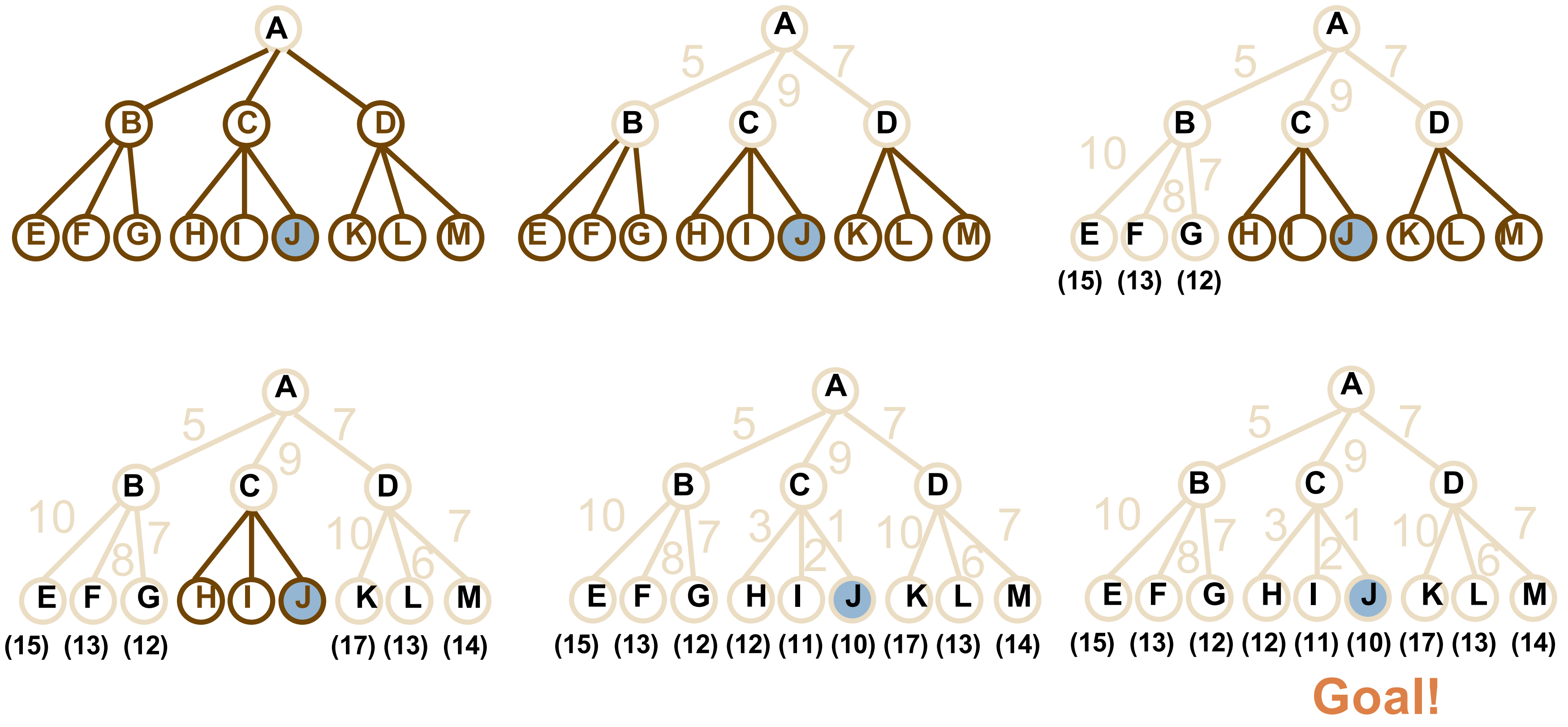








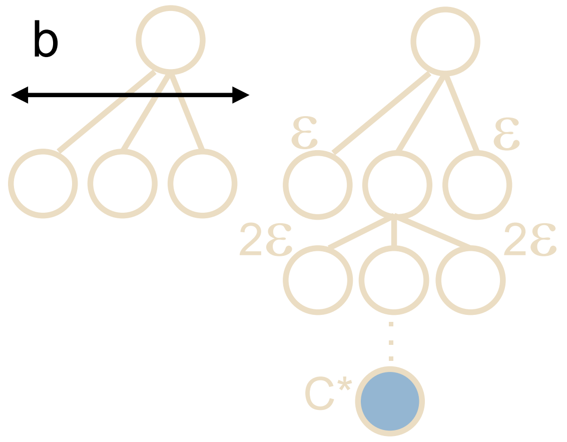
Uniform Cost Search: example



Explore large trees with small steps before exploring paths involving large steps.

Uniform Cost Search: example: performace

- Difficult to characterize in terms of b and d, since the search is path-oriented
- Since the algorithm can explore deeper solutions than d first, its complexity can be greater than $O(b^d)$
- Performance:
  - **Completeness:** complete if step cost $c >= \varepsilon$
  - **Optimization:** optimal
  - Memory and Time Complexity: $O(b^{[C^*/\varepsilon]})$
    - Optimal solution cost: C*
    - Cost of each step $>= \varepsilon$
    - The tree can be reinterpreted as dependent on cost. In the worst case, all nodes costing less than C* will be analyzed first



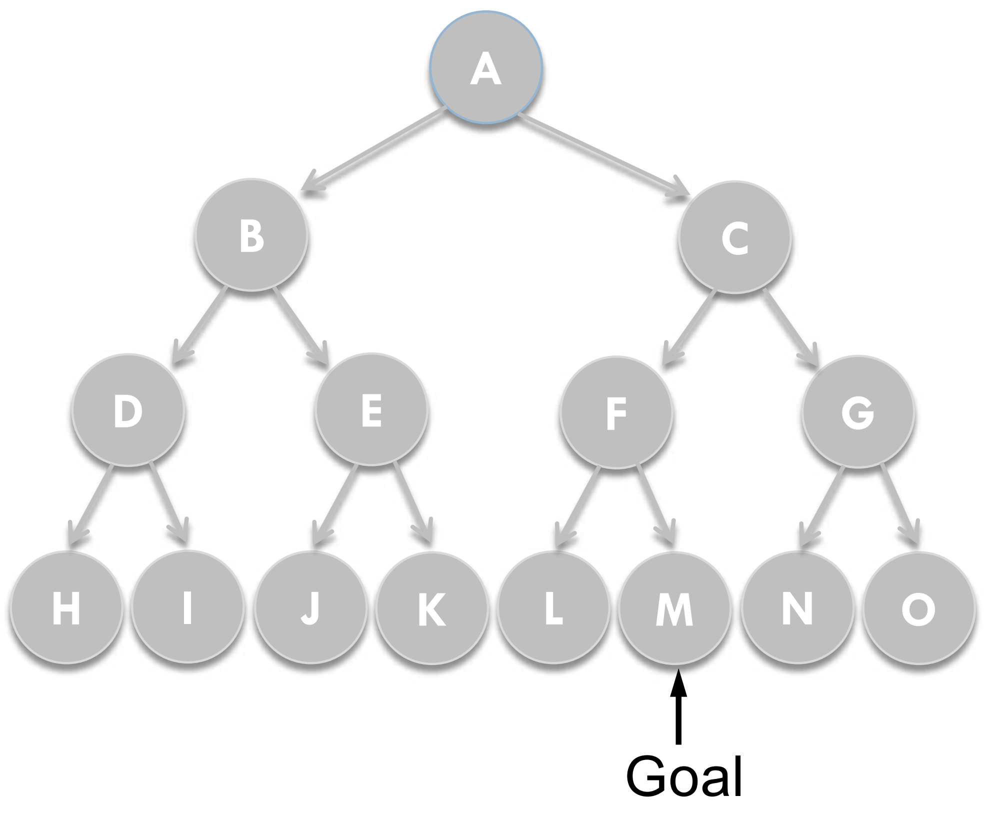
Depth-first

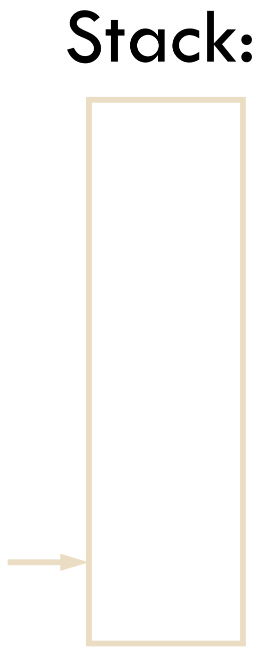
- Always expands the deepest node in the search tree.
- It can be implemented by doing:
  - TREE SEARCH(problem, STACK)

- Stack (Last in First Out):
  - Puts all newly arrived successors at the top of the stack
- Choose as successor the first node in the stack (last arrived nodes expanded first)

Depth-first Search

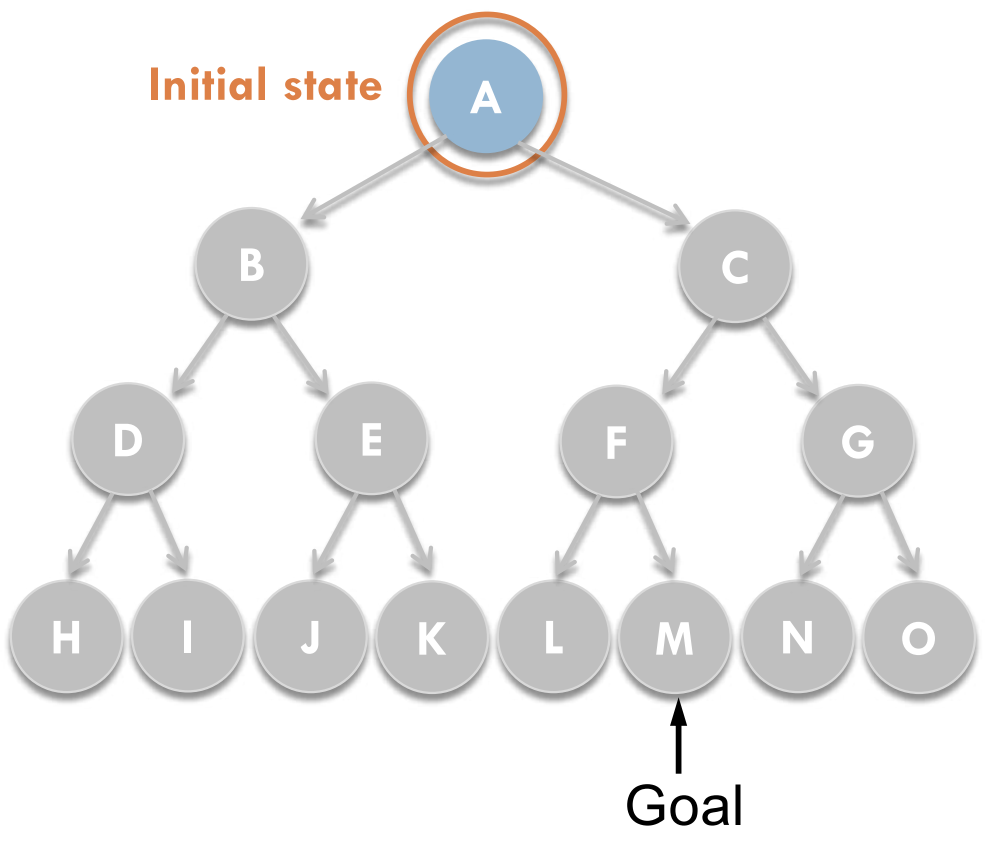
Put the right nodes in the stack first

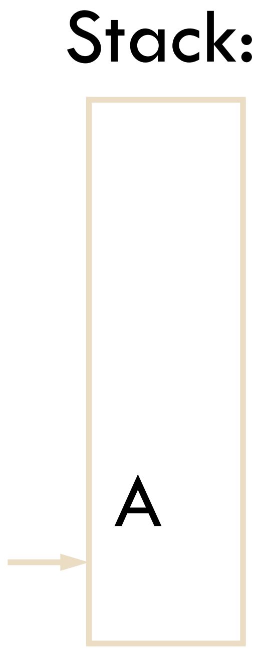




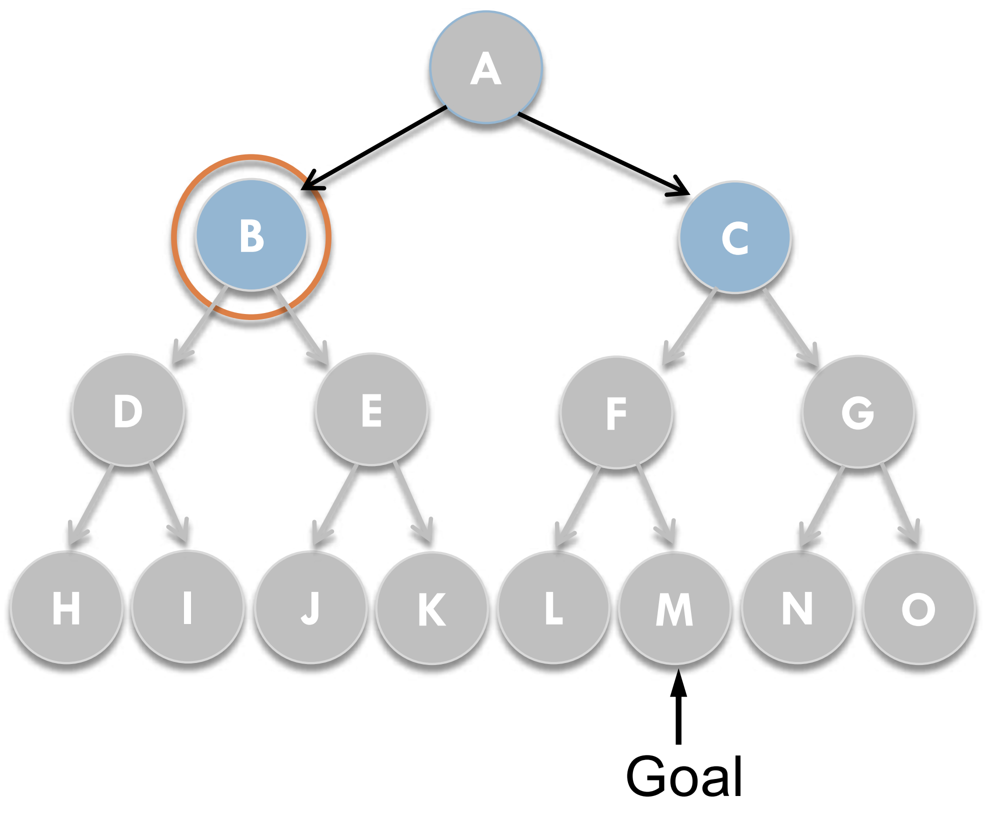
Depth-first Search

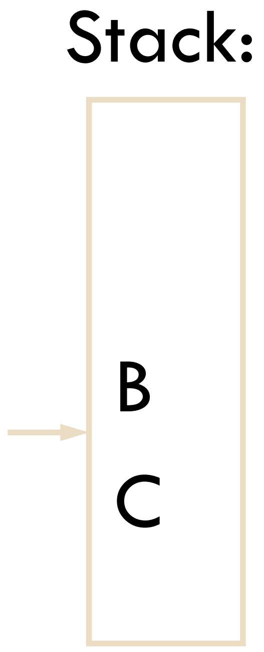
Put the right nodes in the stack first



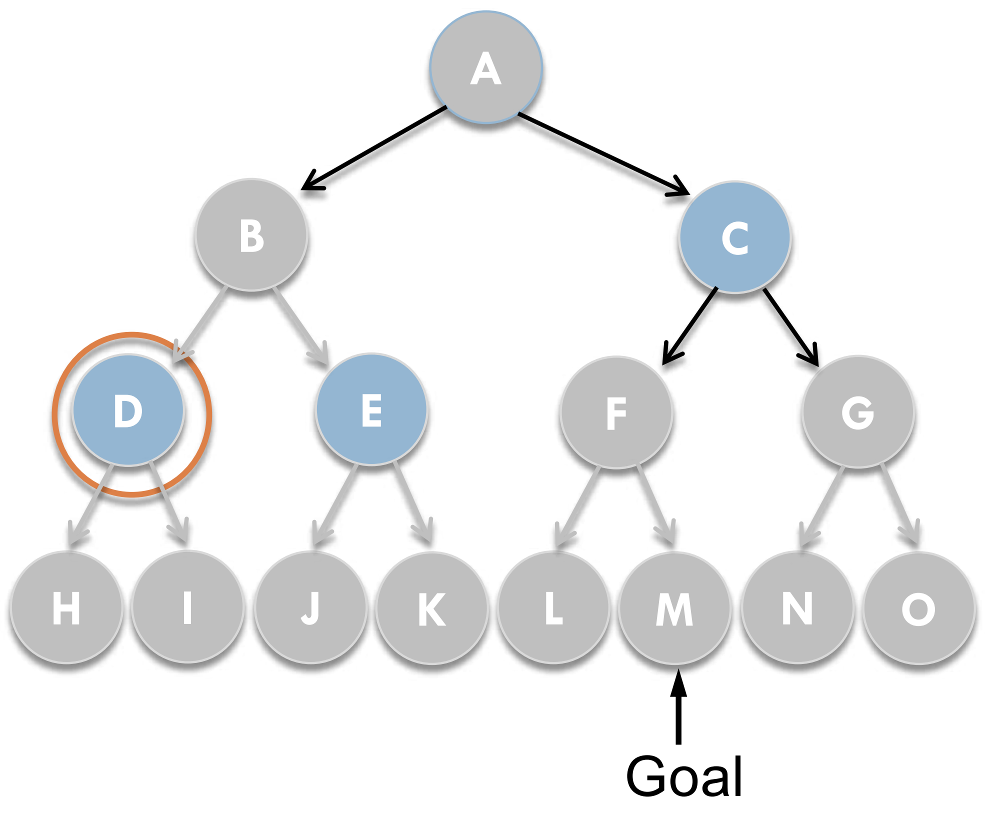


Depth-first Search

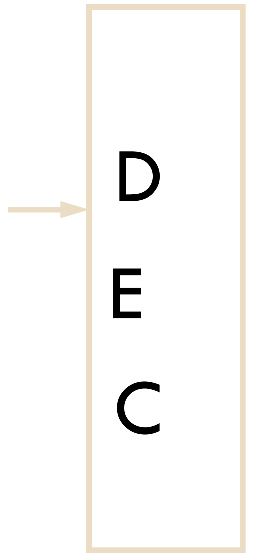




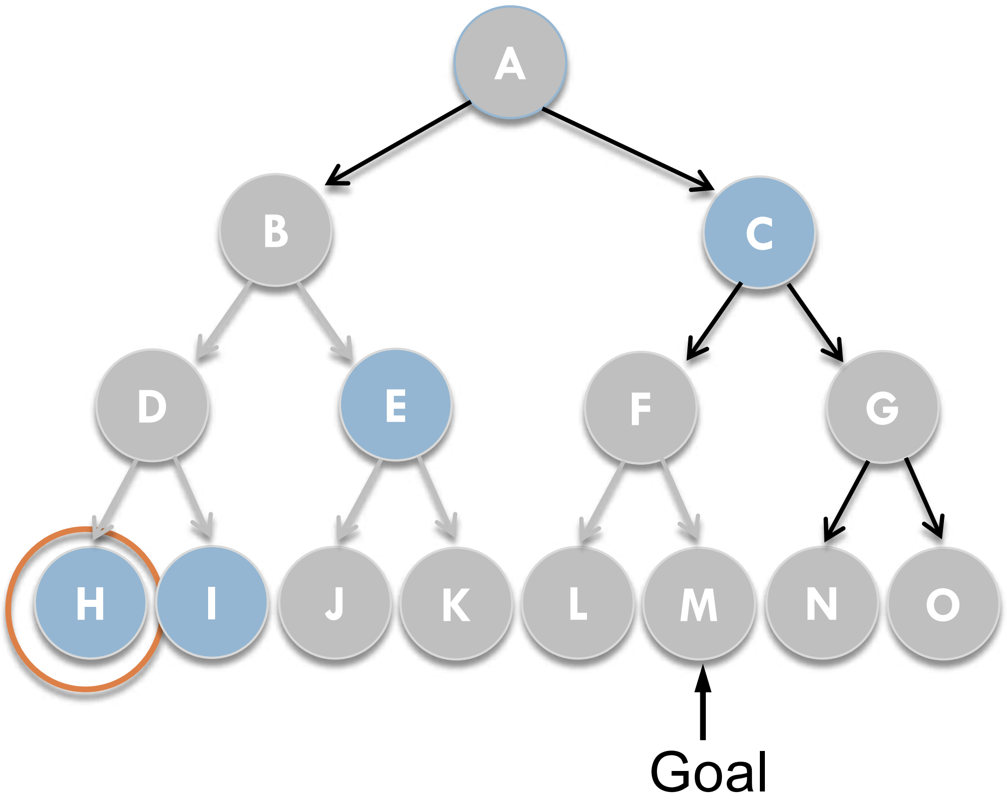
Depth-first Search



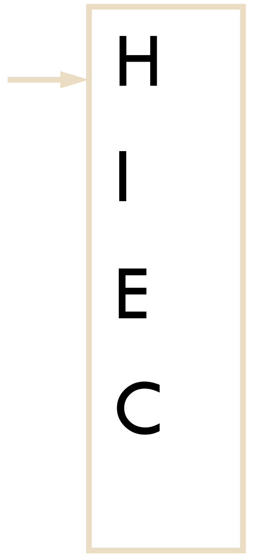
Stack:



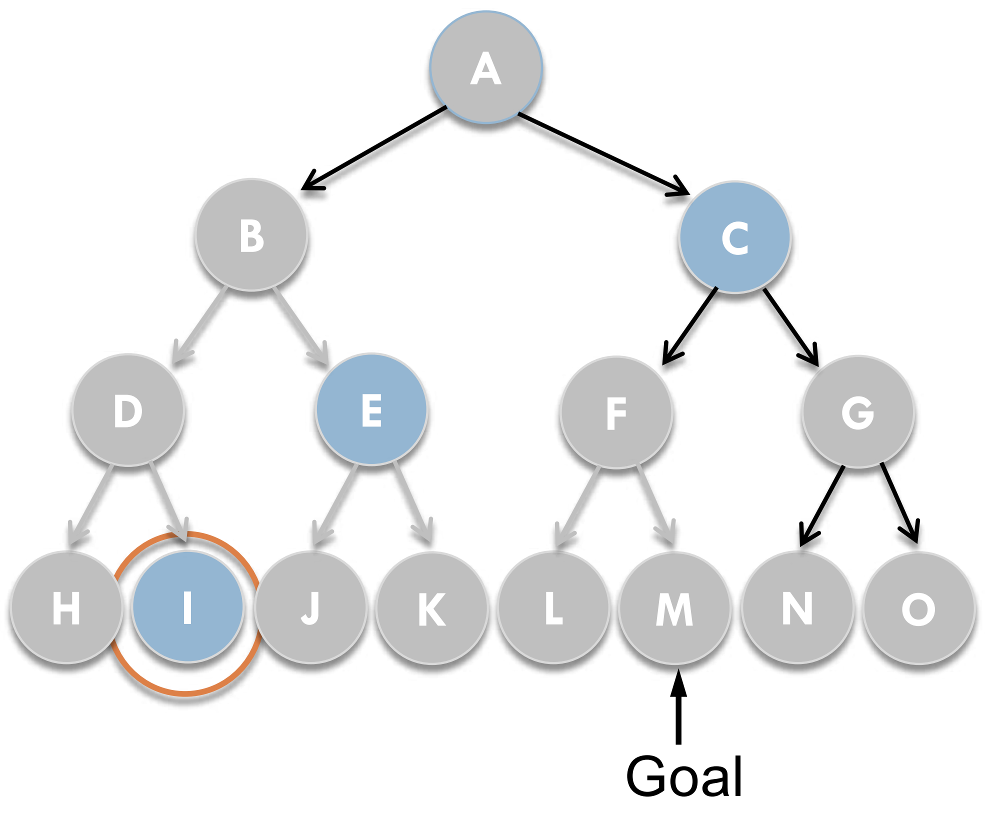
Depth-first Search

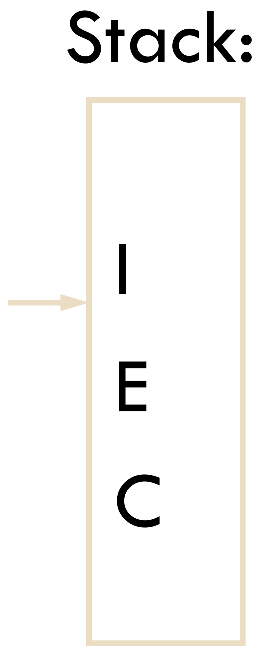


Stack:

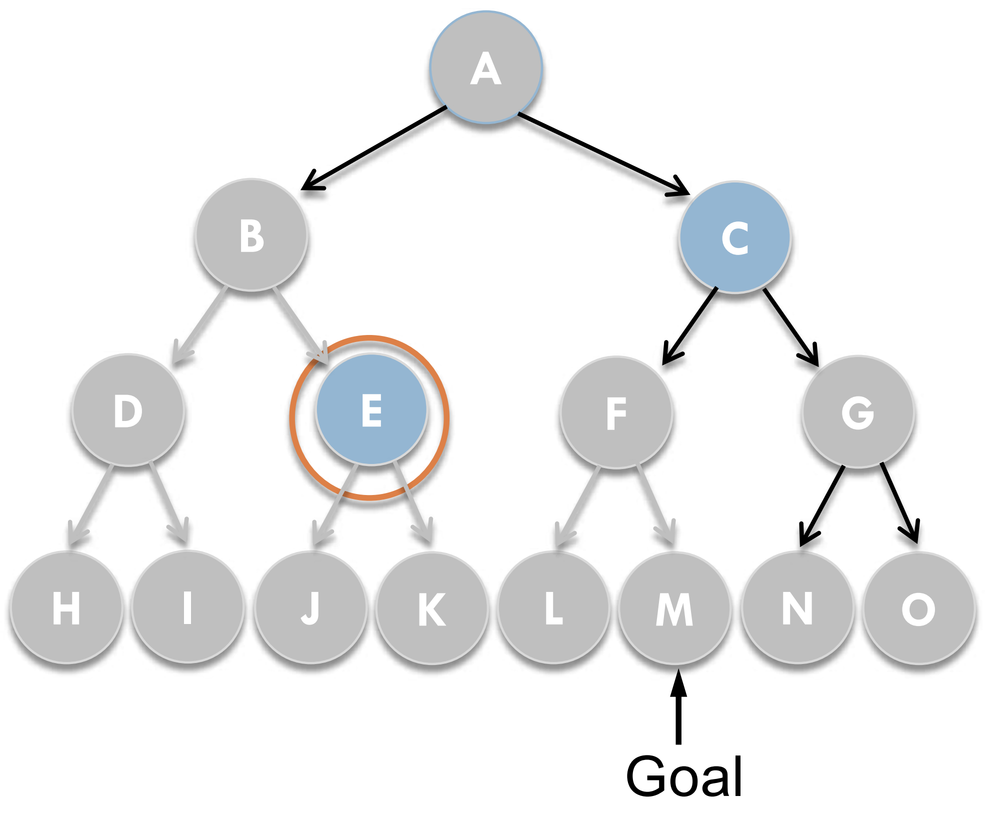


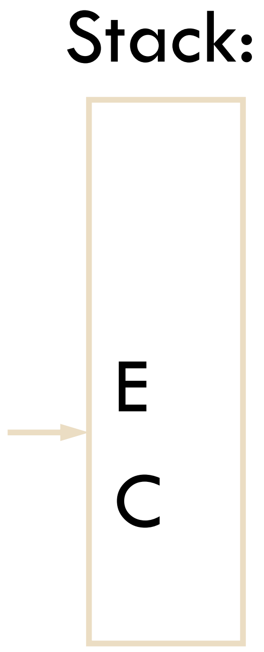
Depth-first Search



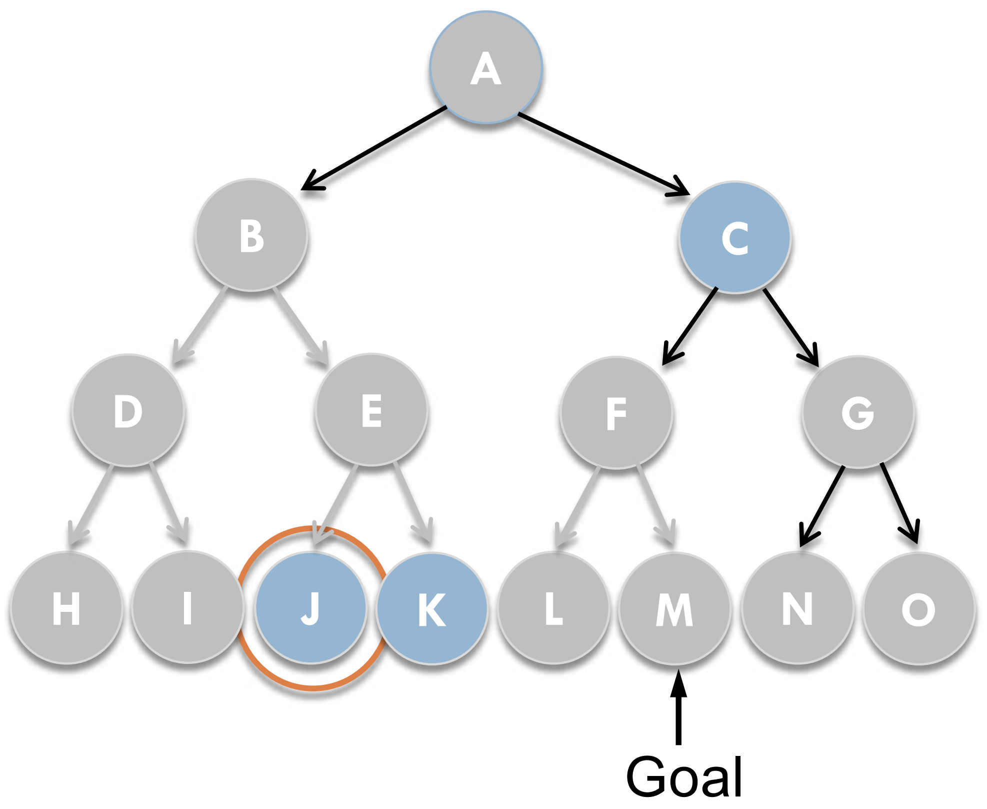


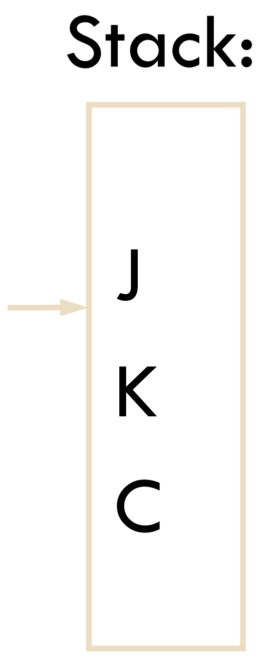
Depth-first Search



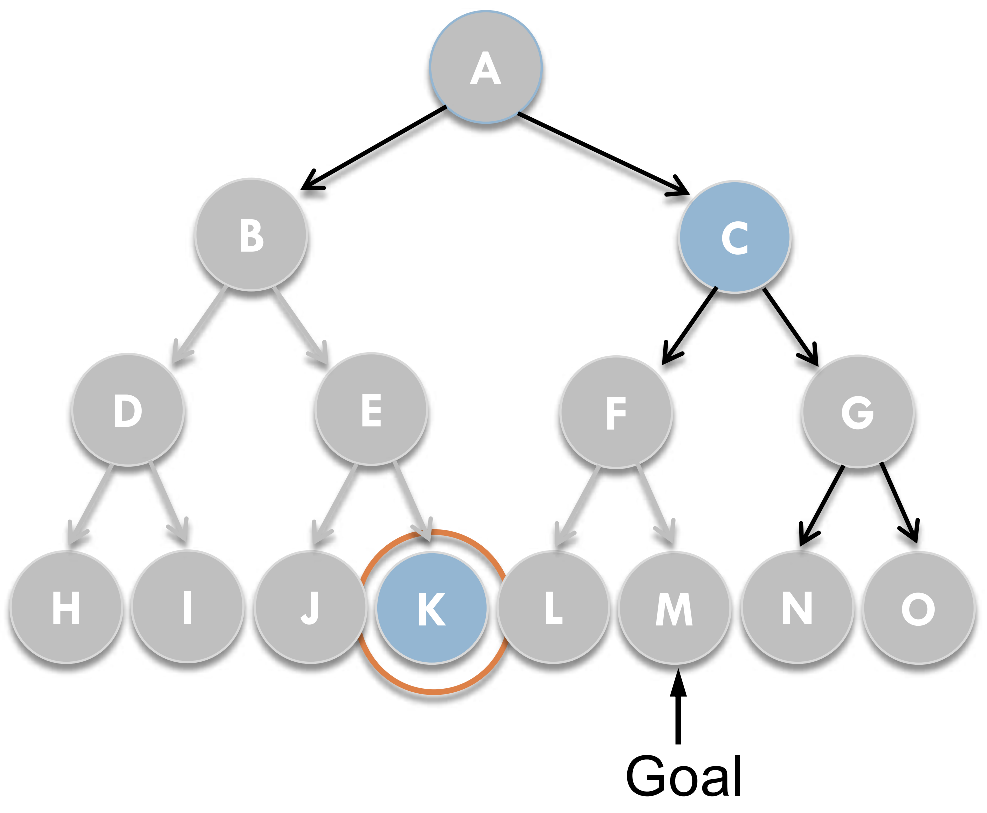


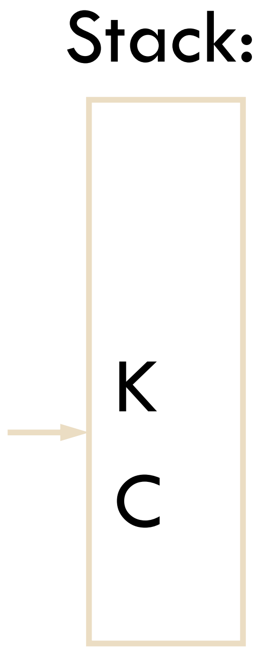
Depth-first Search



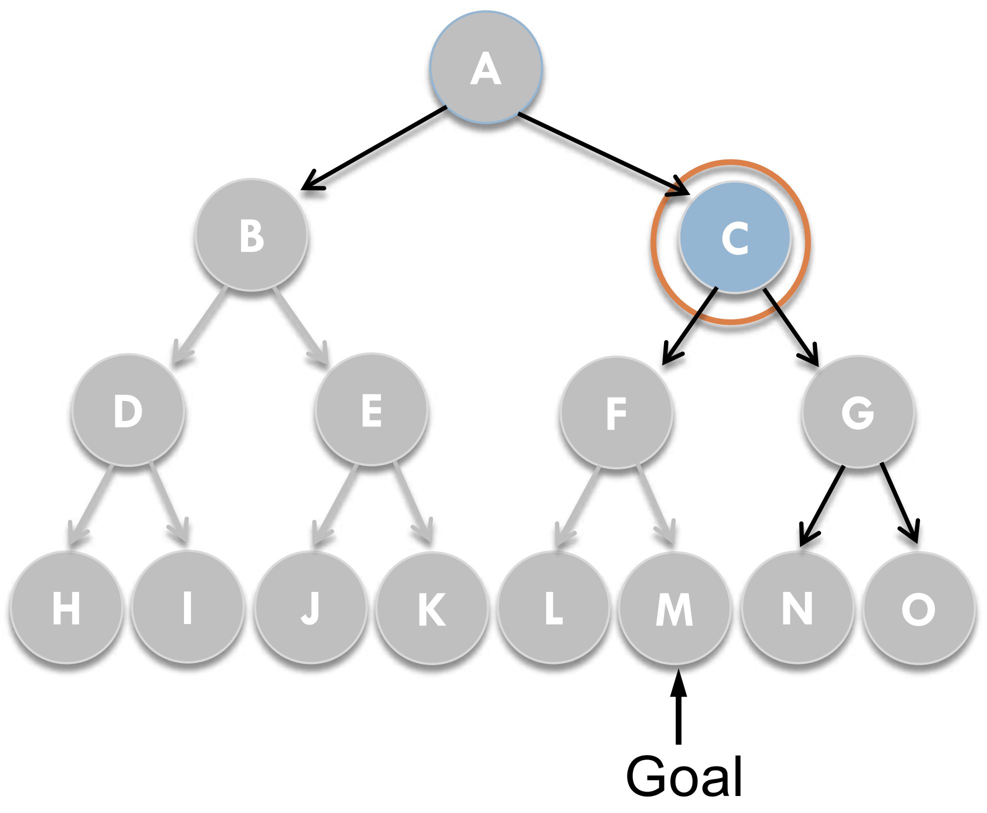


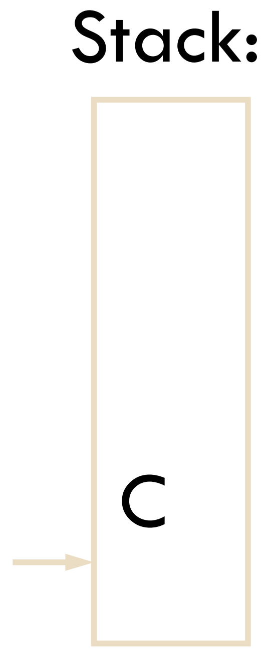
Depth-first Search



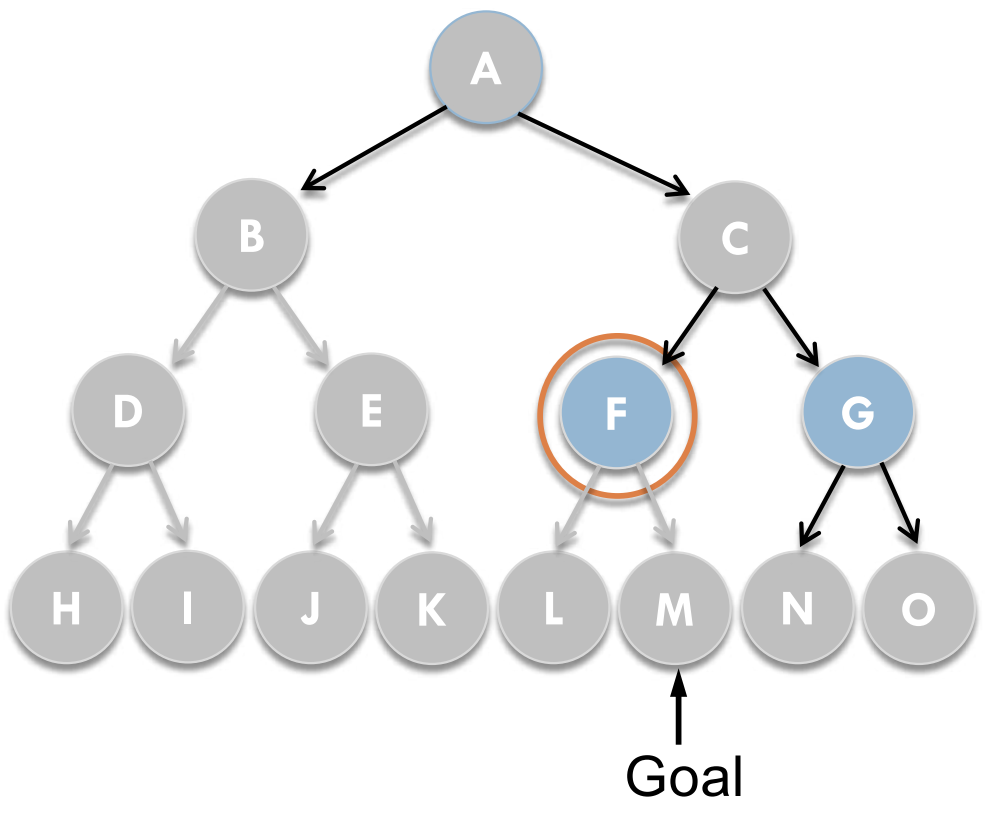


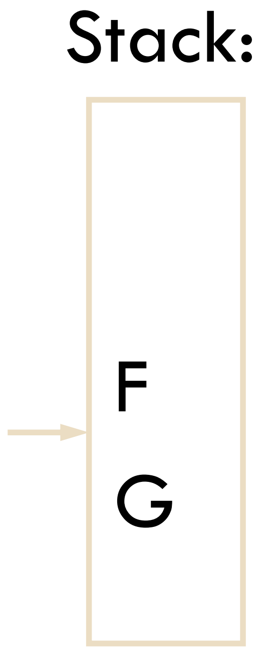
Depth-first Search



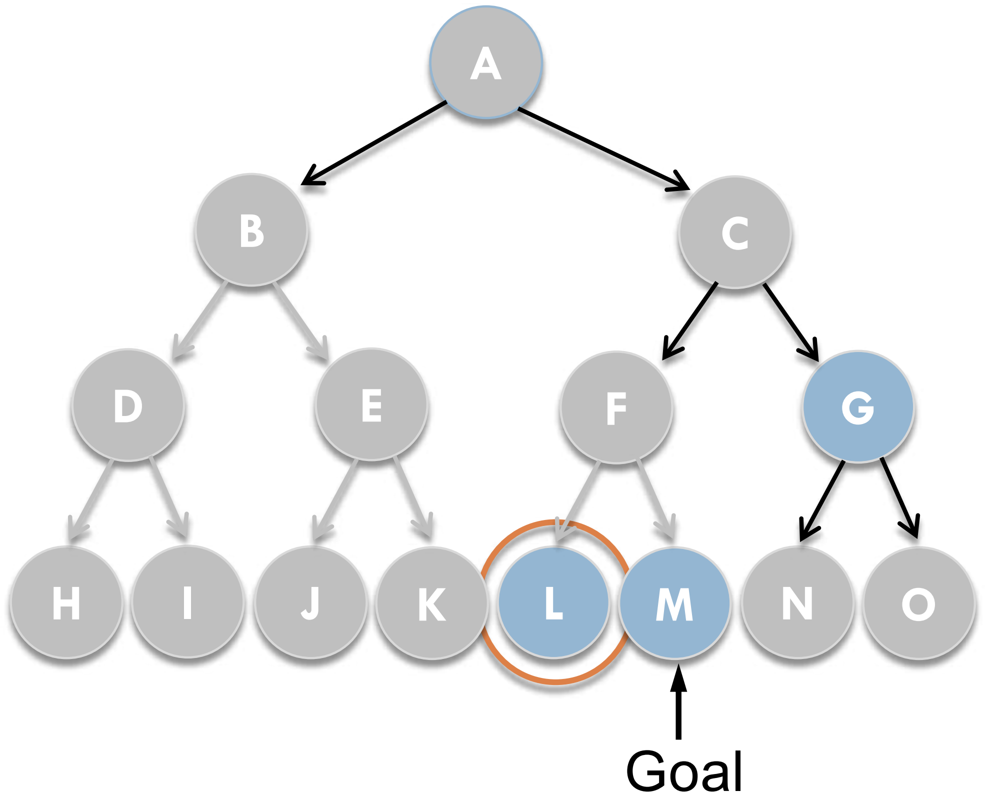


Depth-first Search

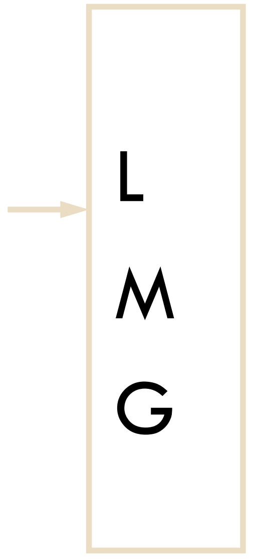




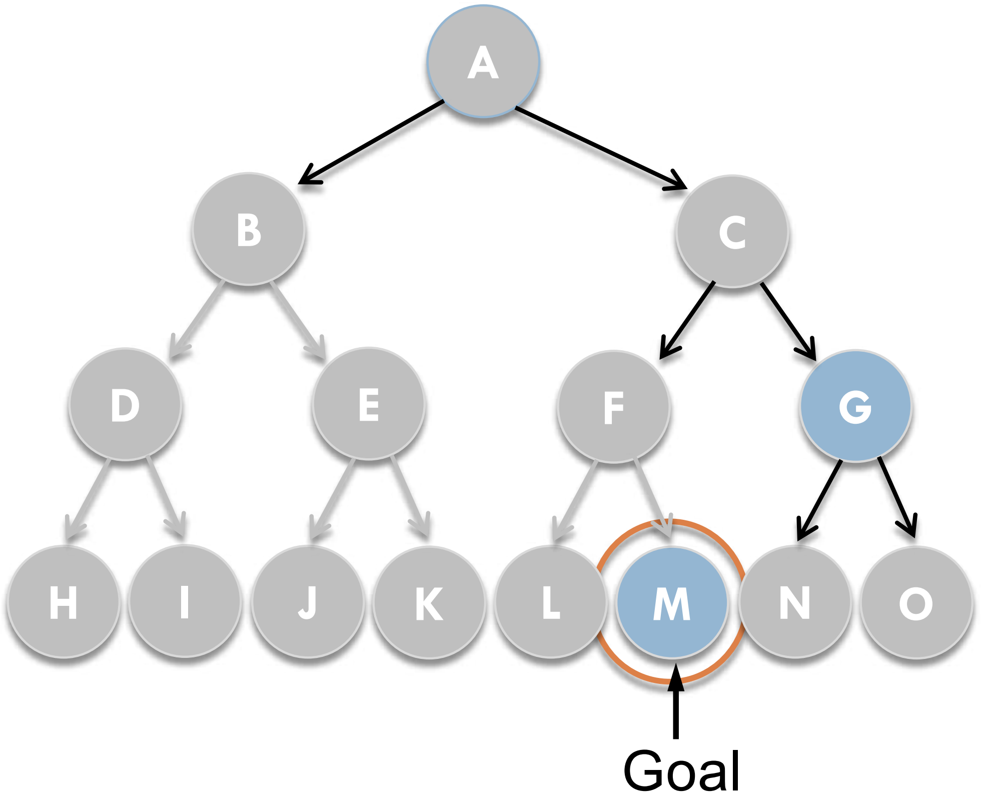
Depth-first Search

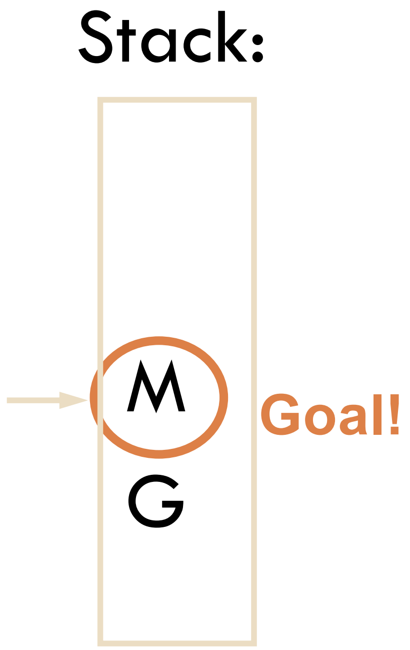


Stack:



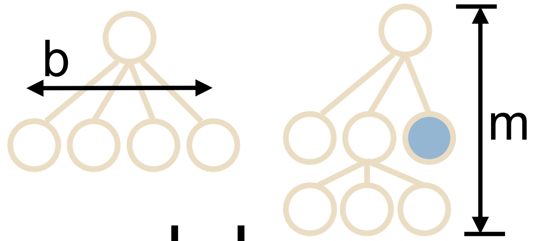
Depth-first Search





Depth-first Search: performance

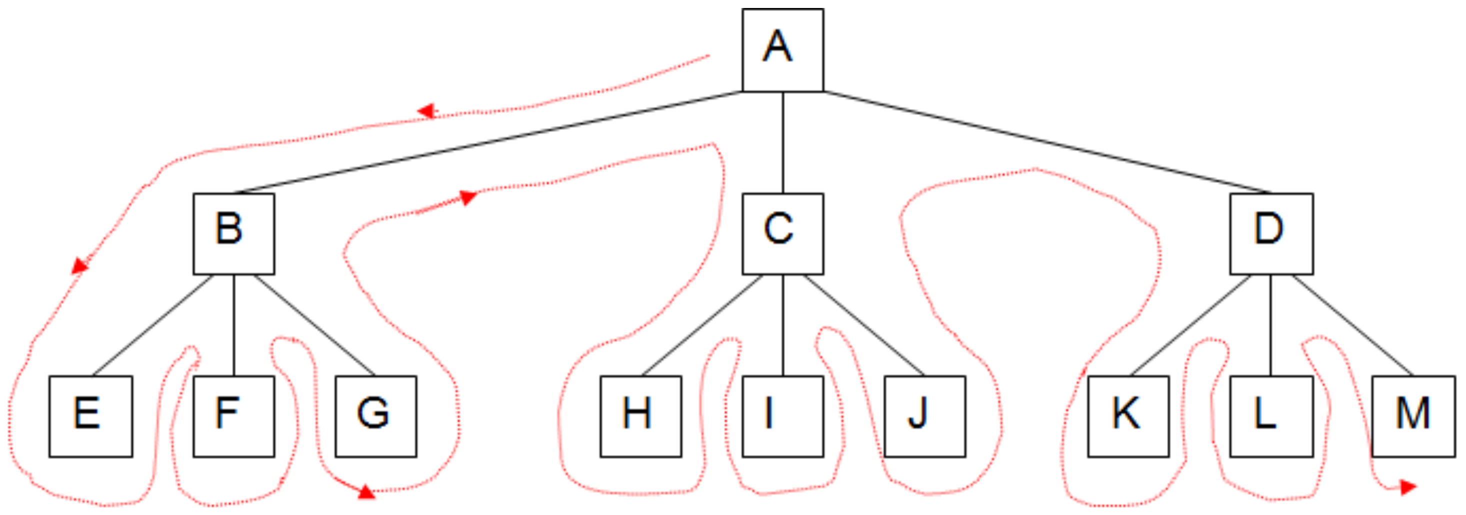
- Consider:
  - *b*: average number of successors for each node
  - d: maximum depth

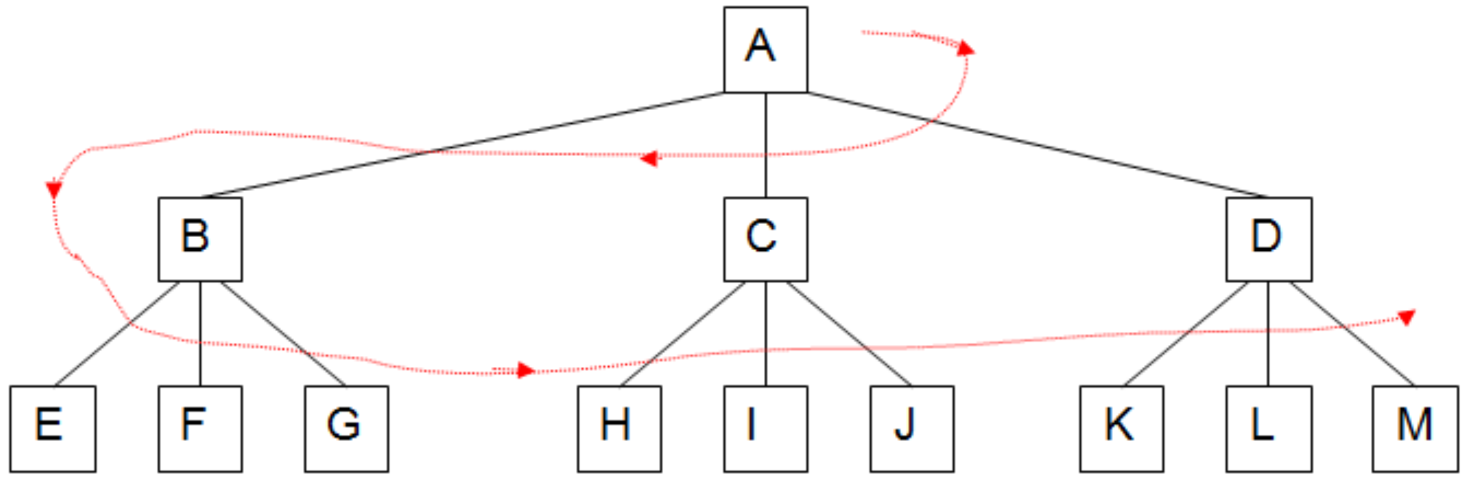


- **Completeness:** is not complete (if a branch has no end, the depth search never ends)
- **Optimization:** not optimal (does not always return the lowest cost node)
- **Time Complexity:** O(b^m)
- **Memory Complexity:** Once a branch of memory (a node and its descendants) has been fully explored, that part of the path can be removed from memory. It is essential that only a single path from the root to the leaf node being analyzed remain in memory, along with its non-expanded sibling nodes.

  Hence: it stores $bm+1$ nodes, i.e.: O(bm)

BFS x DFS





Depth-limited Search

- Depth search with limited depth limit in L
- Performance:
  - **Completeness:** incomplete if $L<d$
  - **Optimization:** Not optimal if $L>d$
  - **Time complexity:** $O(b^L)$
  - **Memory complexity:** $O(bL)$

Depth-limited Search: algorithm

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
        return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
        if GOAL-TEST(node) then return SOLUTION(node)
        else if limit = 0 then return cutoff
        else
                cutoff occurred? ← false
                for each action in ACTIONS(STATE) do
                        child ← CHILD-NODE(problem, node, action)
                        result ← RECURSIVE-DLS(child, problem, limit − 1)
                        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

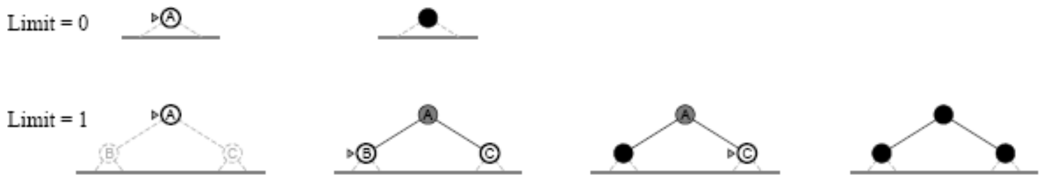
- Gradually increase the depth limit (0,1,2, ...) until you find the objective (L = d)
- Combines the advantages of searching in width and depth
- Algorithm:

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

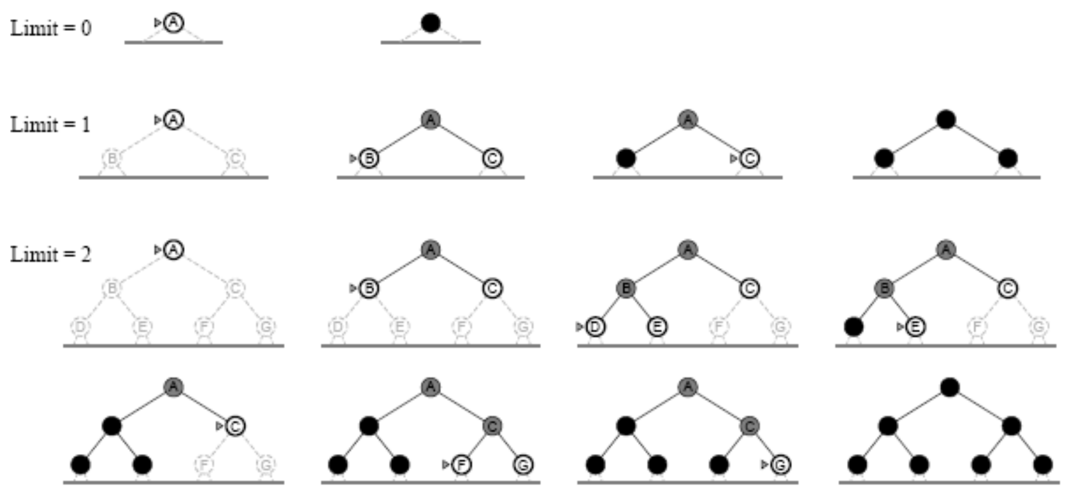
Iterative Deepening Search

Limit = 0

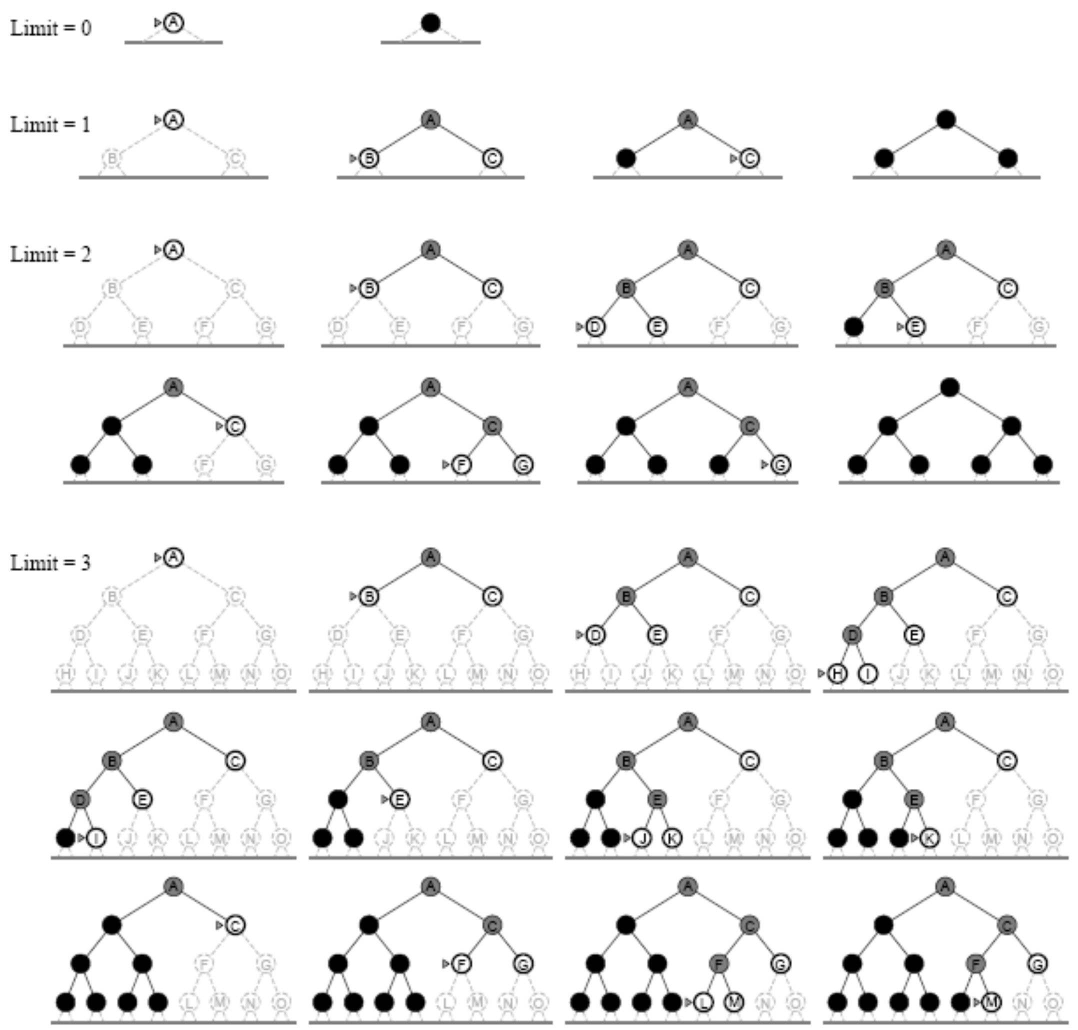
Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search



Search for iterative deepening: performance

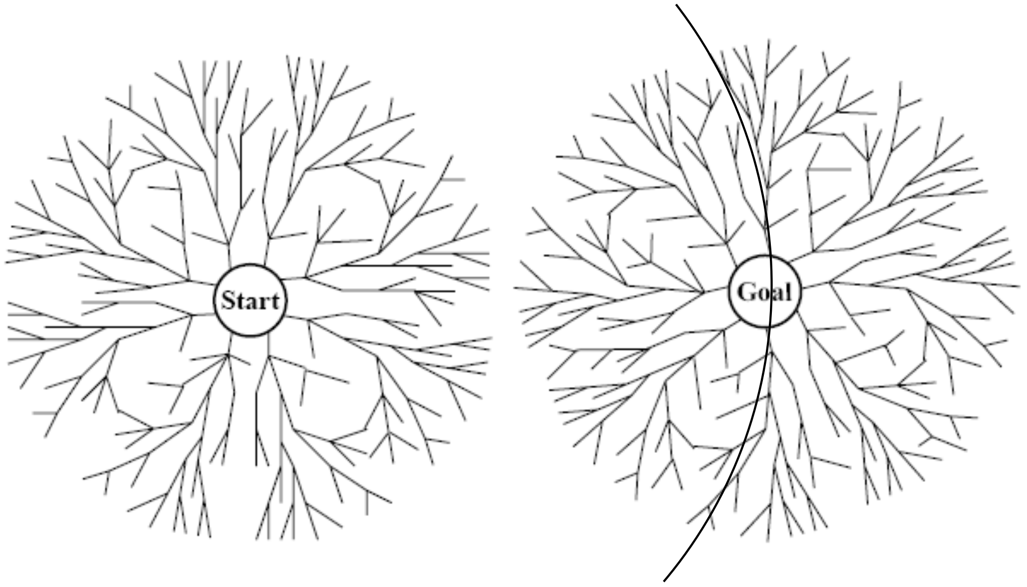
- **Completeness:** complete, as in the breadth-first search
- **Optimization:** optimal, as the breadth-first search
- **Time complexity:**

  It may seem like a waste, but ...

  - The first level nodes are generated **d** times, the second level nodes are generated (**d-1**) times, and so on ...
  - Hence: $(d)b + (d-1)b^2 + \ldots + (1)b^d \rightarrow O(b^d)$
- **Memory complexity:** it is a depth-depth search. Hence: $O(b.d)$

Bidirectional search

- Perform two simultaneous searches: one from the initial state to the objective, the other from the objective to the initial state, stopping when the searches are in the intermediate state
- Motivation: $b^{d/2}+b^{d/2}$ is much smaller than b^d



Bidirectional search

- Performance:
  - **Completeness:** complete
  - **Optimization:** optimal (for uniform cost steps)
  - **Time complexity:** $O(b^{d/2})$
  - **Memory complexity:** $O(b^{d/2})$
- Restrictions:
  - The goal needs to be known
  - The predecessors of a node must be computable

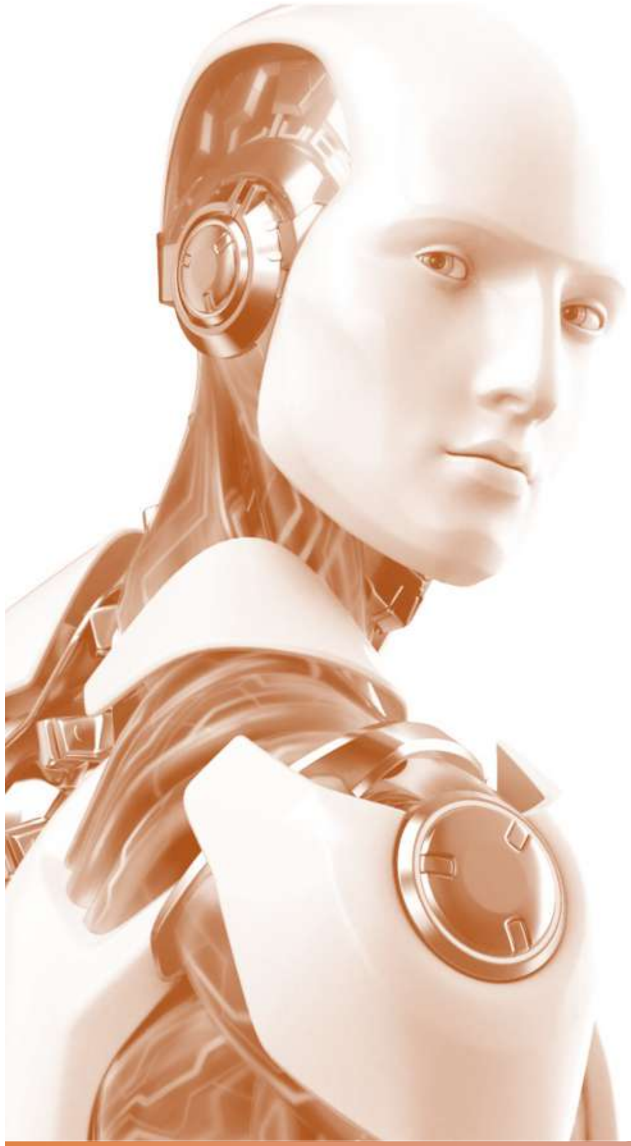
Search strategies without information: comparative table

Criteria	Breadth-first	Uniform cost	Depth-first	Depth Limited	Iterative Deepening	Bidirec. (if aplicable)
Complete	Yes	Yes	No	No	Yes	Yes
Time	$O(b^{d+1})$	$O(b^{[C^*/ \varepsilon]})$	$O(b^m)$	$O(b^L)$	$O(b^d)$	$O(b^{d/2})$
Memory	$O(b^{d+1})$	$O(b^{[C^*/ \varepsilon]})$	$O(bm)$	$O(b^L)$	$O(bd)$	$O(b^{d/2})$
Optimal	Yes	Yes	No	No	Yes	Yes

b branching factor
d depth of solution
m maximum depth of the search tree
l depth limit
C* cost of the optimal solution

Repeated states

- The search process can waste time expanding nodes already explored before
  - Repeated states can lead to infinite loops
  - Repeated states can turn a linear problem into an exponential problem
- Strategy
  - Compare nodes about to be expanded with nodes already visited.
  - If the node has already been visited, it will be discarded.
  - A list stores nodes already visited.



MC906/MO416 Introduction to AI

Lecture 4 – PART III

Water jug problem

There are two vessels: a 4-liter and a 3-liter vessel, initially empty, and a fountain that spouts water in abundance.

Objective: Get 2 liters in any of the jugs.

Possible actions:

- Fill the vases
- Empty the vessels
- Complete one vase with another
- Throwing a vase at another

Water jug problem: other solutions

- **State:** (x, y), where:
  - x is the amount of water in the 4 Liter container;
  - y is the amount of water in the 3 Liter vessel
- **Initial state:** (0,0)
- **Target States:** (X, 2) or (2, X)
- **Rules:**
  - R0: Fill_jug_4
  - R1: Fill_jug_3
  - R2: Empty_jug_4
  - R3: Empty_jug_3
  - R4: Complete_3_with_4
  - R5: Complete_4_with_3
  - R6: Empty_4_in_3
  - R7: Empty_3_in_4

Water jug problem: other solutions

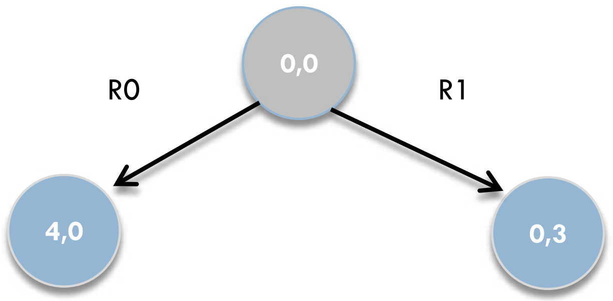
Breadth-first search:

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:

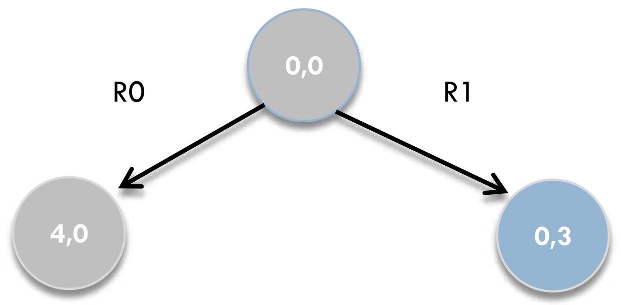


R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:



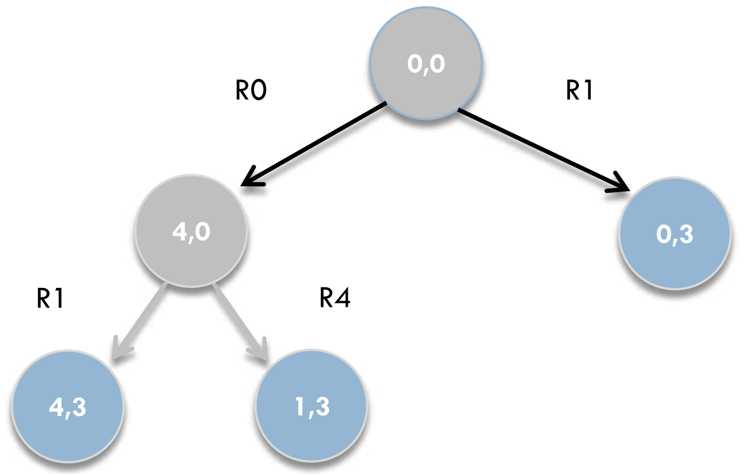
Is it a goal? No, expand

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:

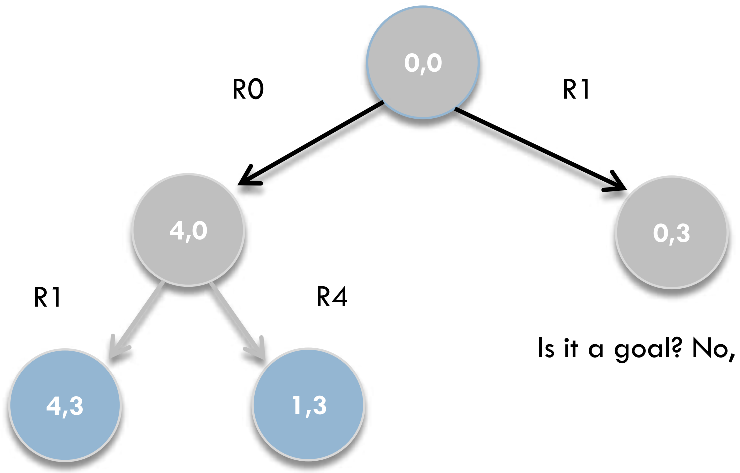


R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:



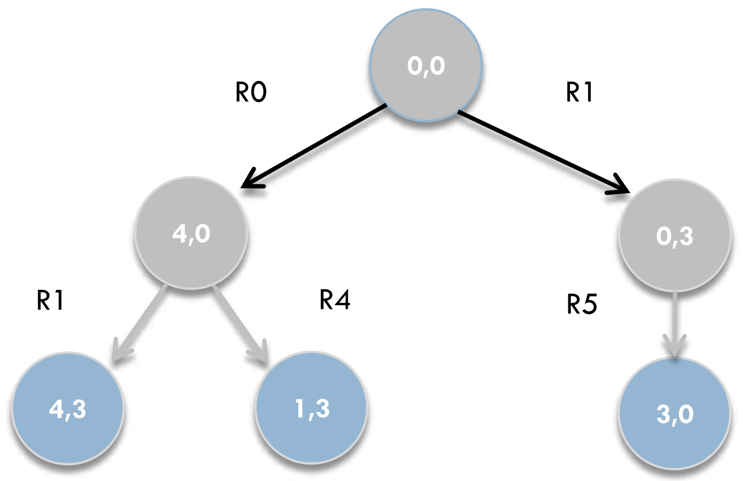
Is it a goal? No, expand

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:



R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

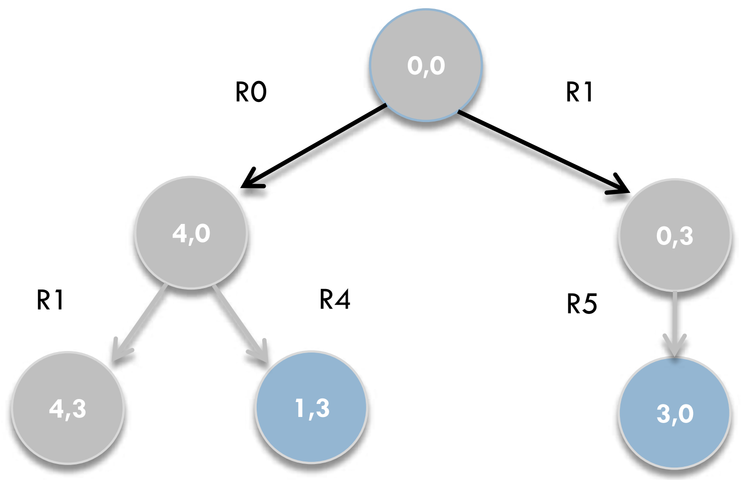
From R0->R7

Water jug problem: other solutions

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

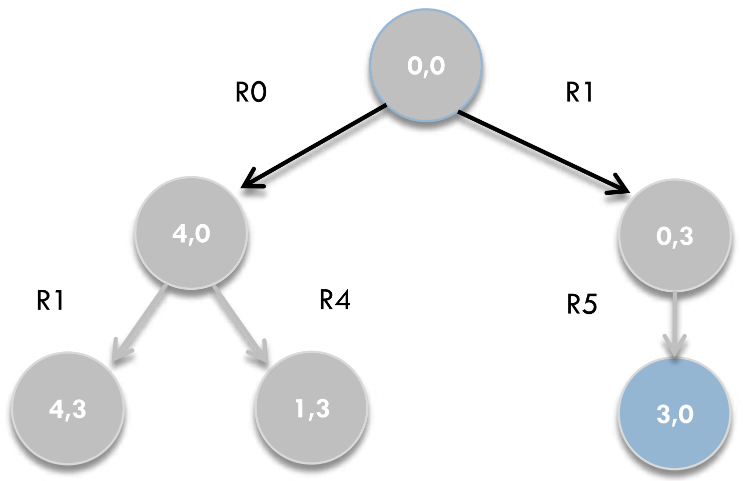
Breadth-first search:



Is it a goal? No, expand
Nothing to do as all possible actions lead to a state that has been visited (4,0) ou (0,3)

Water jug problem: other solutions

Breadth-first search:



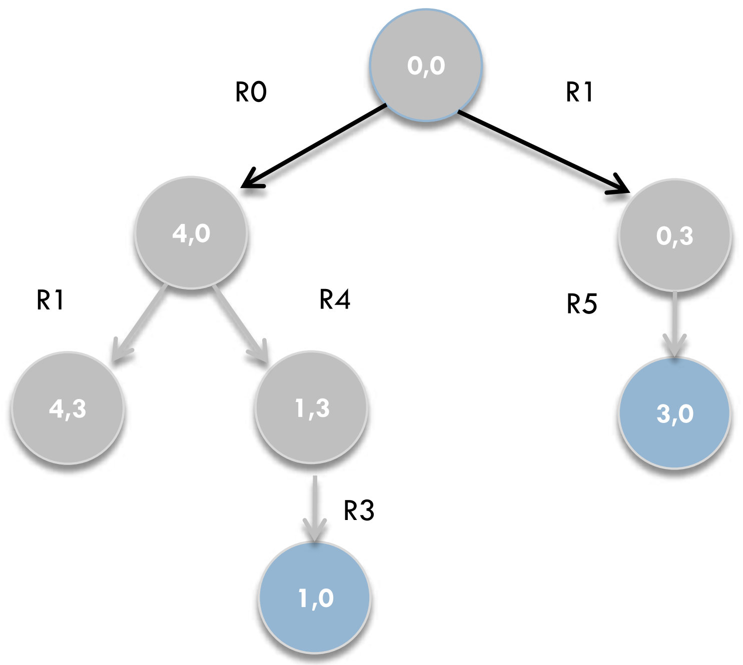
Is it a goal? No, expand

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:

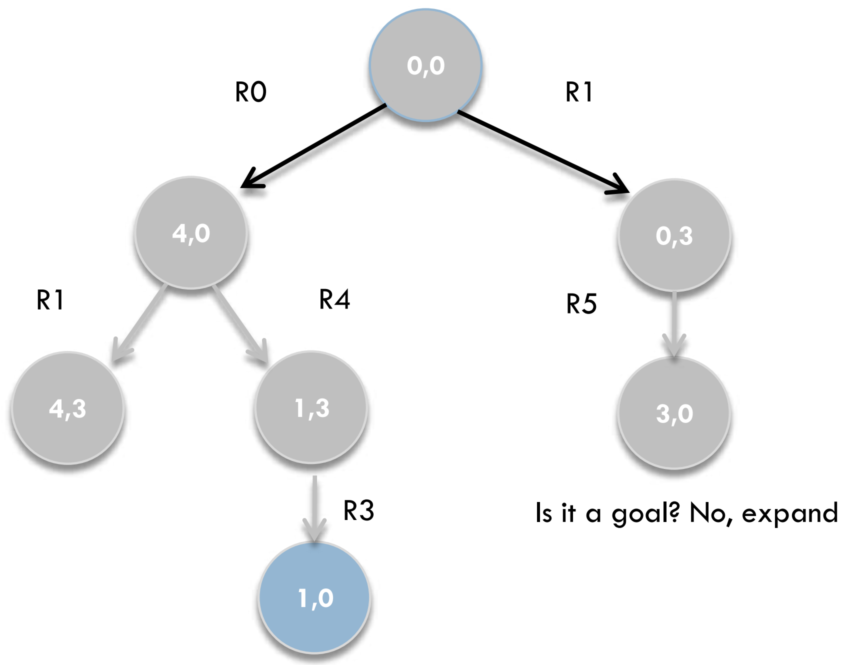


R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:



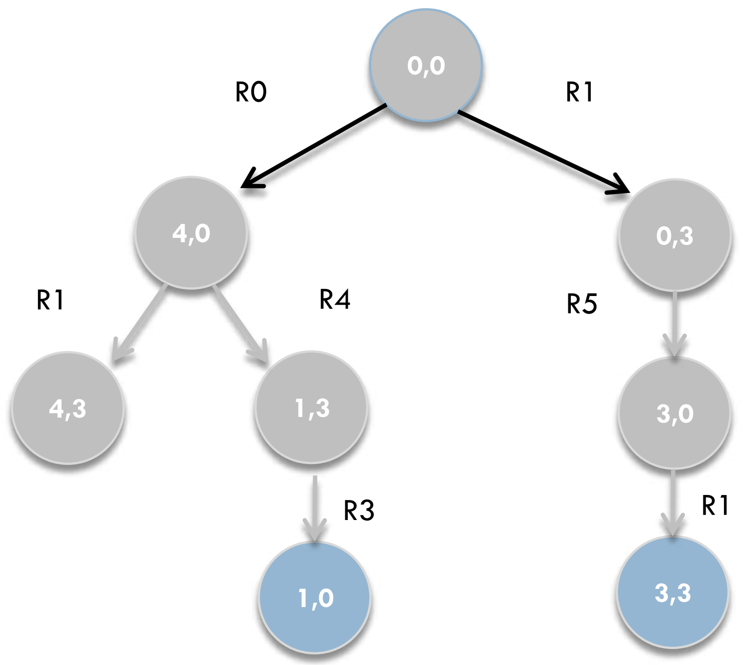
Is it a goal? No, expand

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:

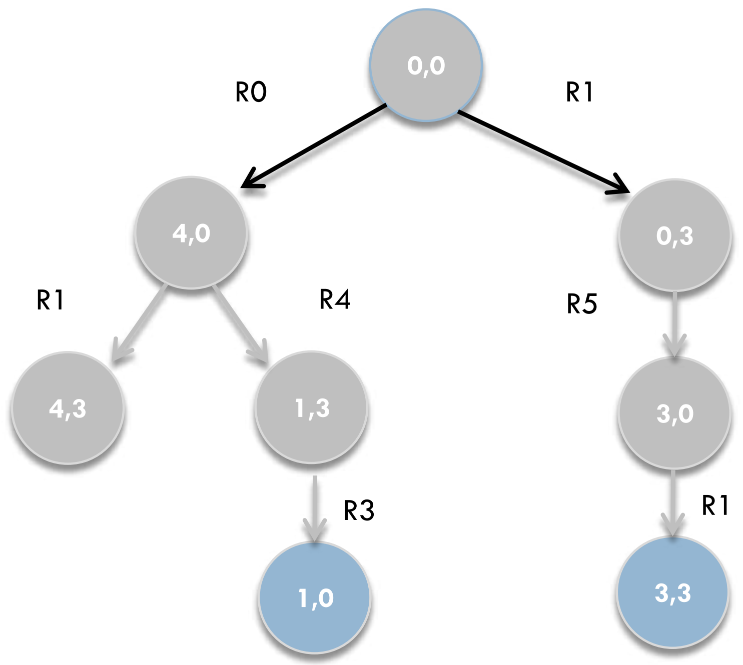


R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:

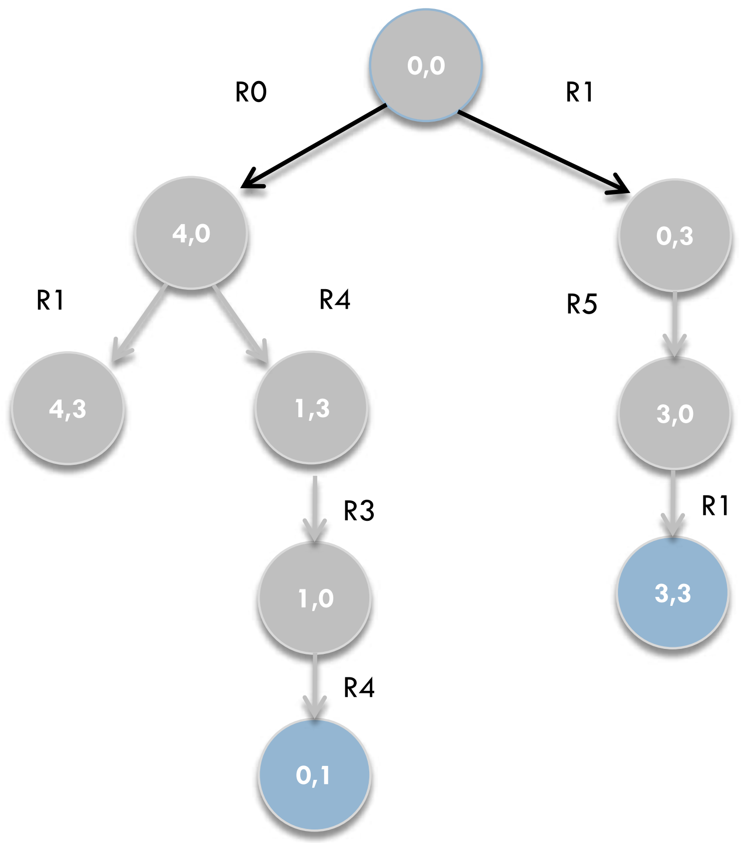


R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:

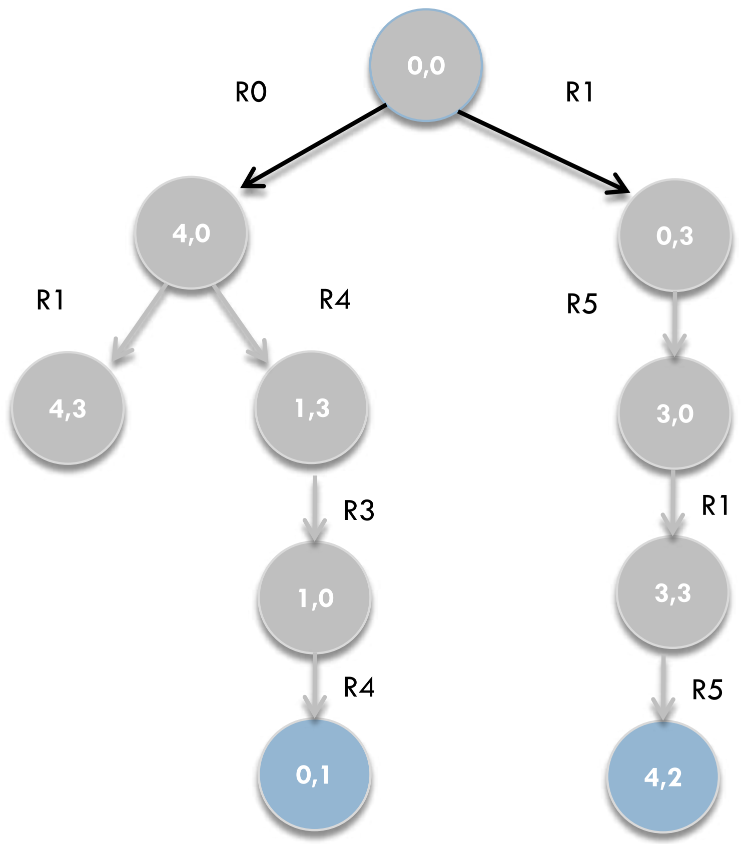


R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:



R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

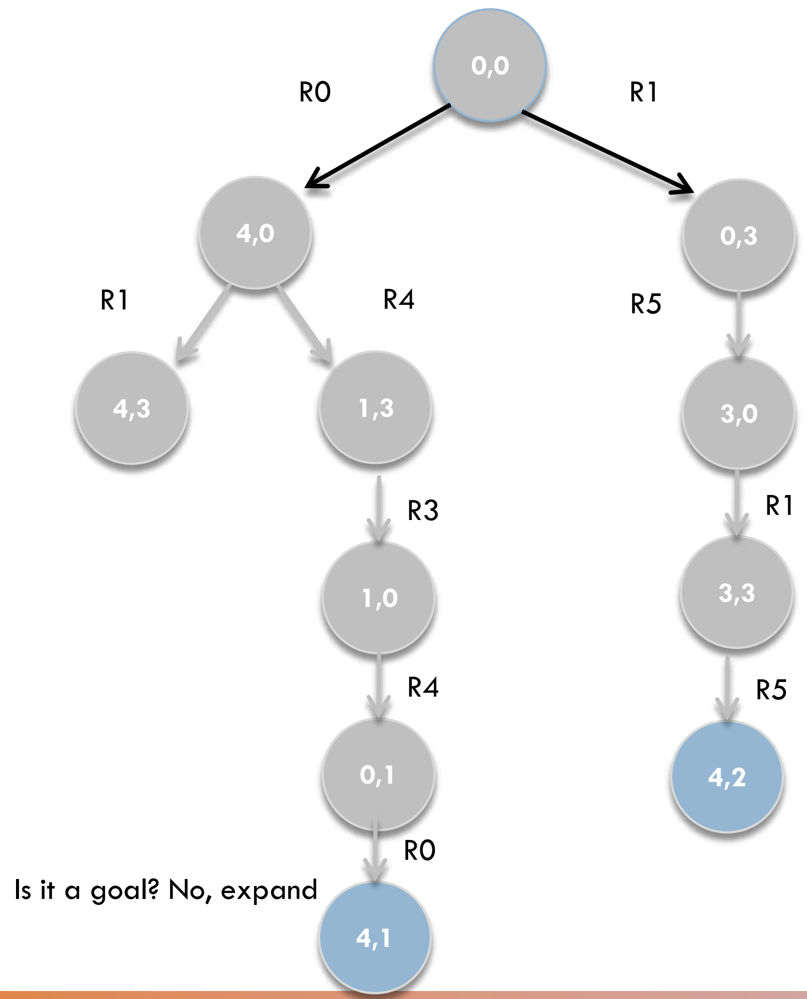
From R0->R7

Water jug problem: other solutions

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

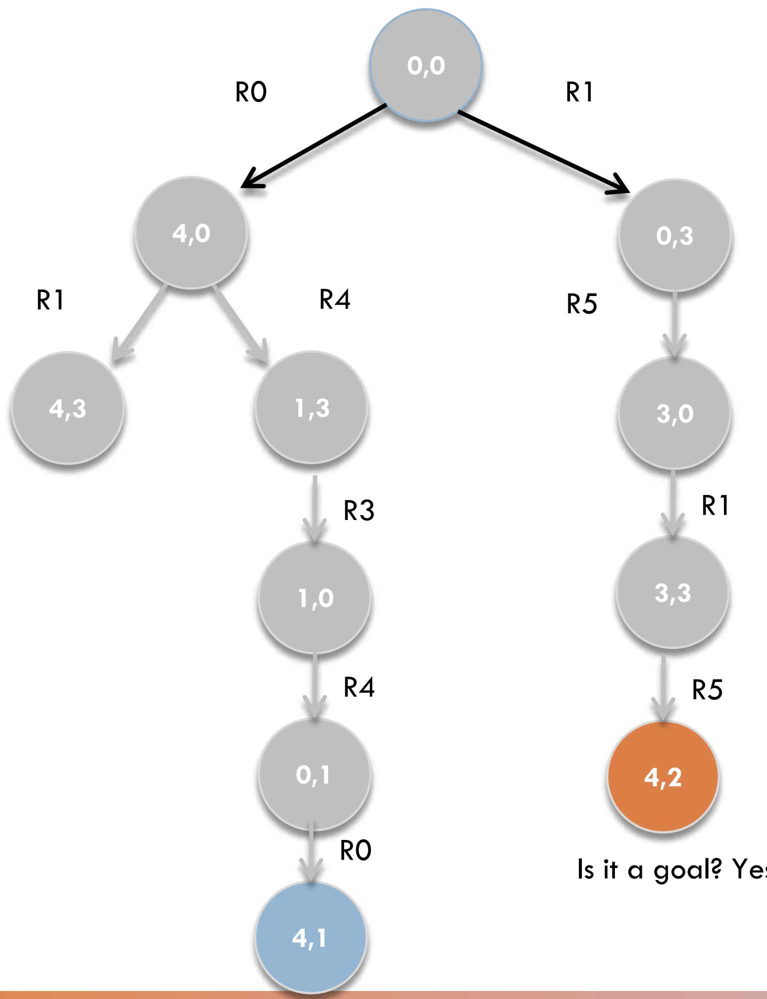
From R0->R7

Breadth-first search:



Water jug problem: other solutions

Breadth-first search:



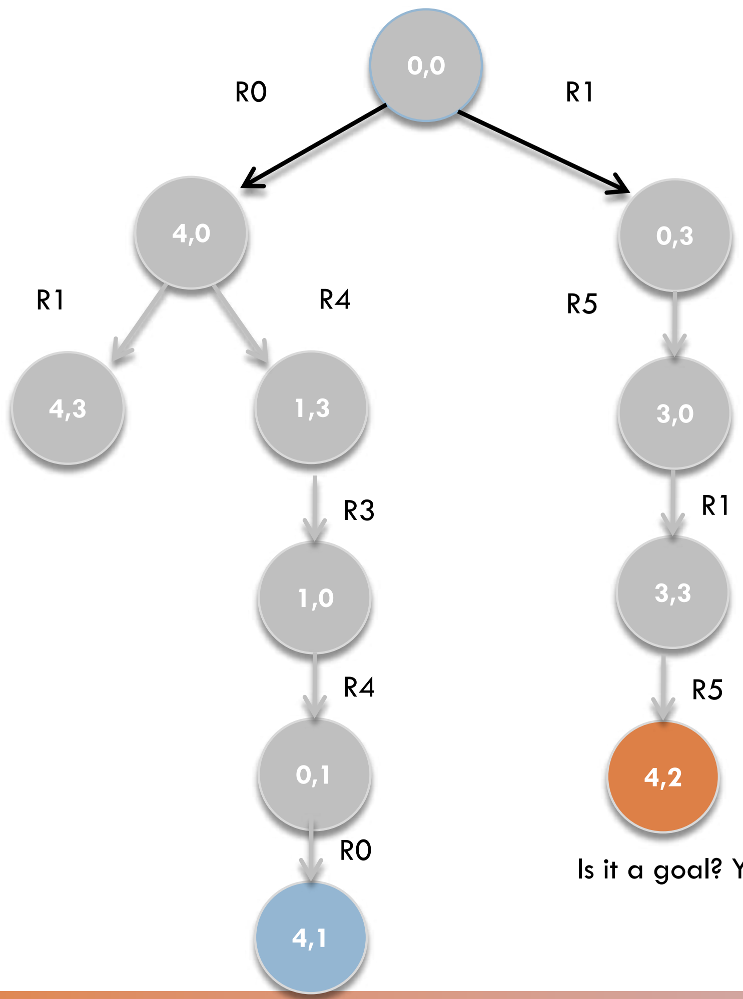
Is it a goal? Yes, return the path to the solution

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Water jug problem: other solutions

Breadth-first search:



R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Path to solution: R1, R5, R1, R5

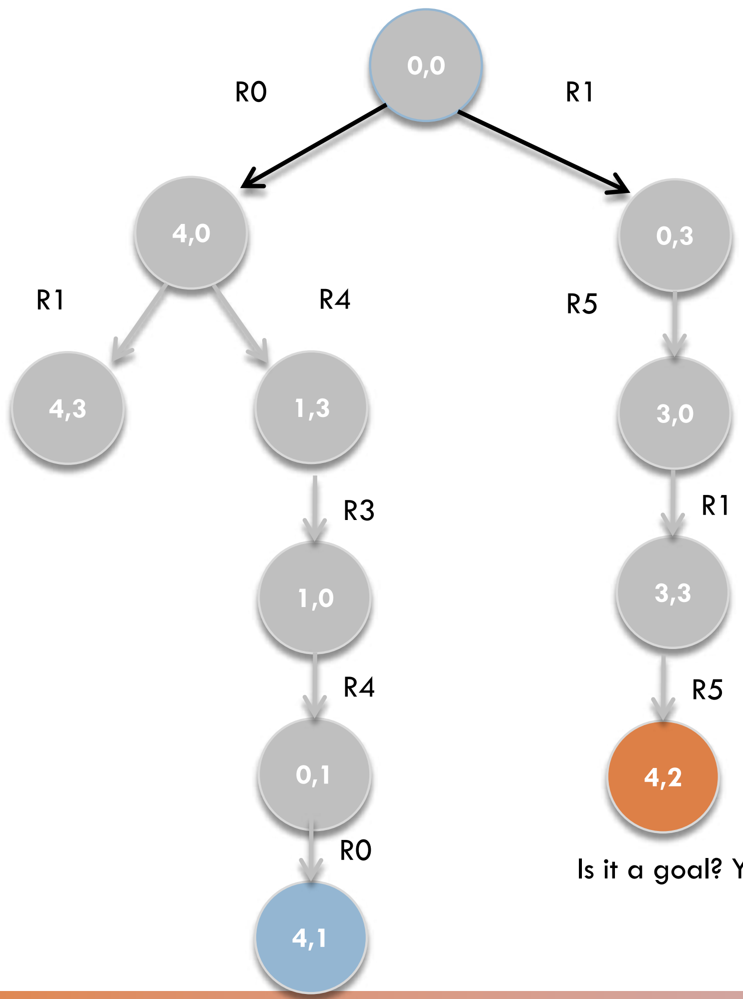
Is it a goal? Yes, return the path to the solution

Water jug problem: other solutions

R0: Fill_jug_4
R1: Fill_jug_3
R2: Empty_jug_4
R3: Empty_jug_3
R4: Complete_3_with_4
R5: Complete_4_with_3
R6: Empty_4_in_3
R7: Empty_3_in_4

From R0->R7

Breadth-first search:



What is the order of expansion of the nodes?

(0,0), (4,0), (0,3), (4,3), (1,3), (3,0), (1,0), (3,3), (0,1), (4,2)

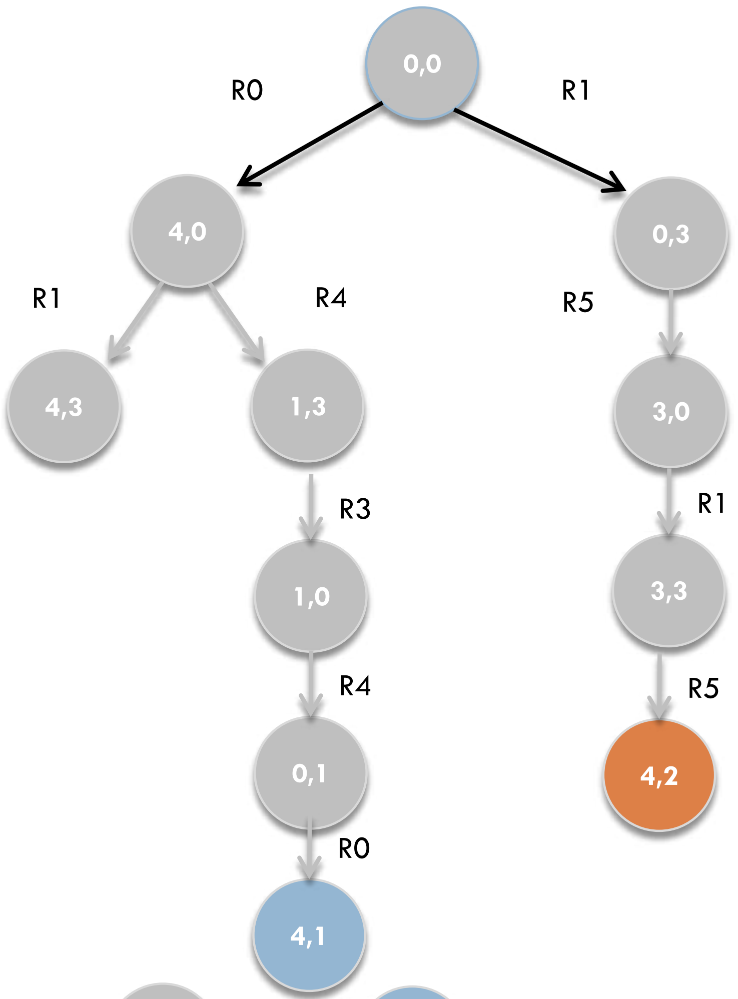
Which nodes were created?

(0,0), (4,0), (0,3), (4,3), (1,3), (3,0), (1,0), (3,3), (0,1), (4,1),(4,2)

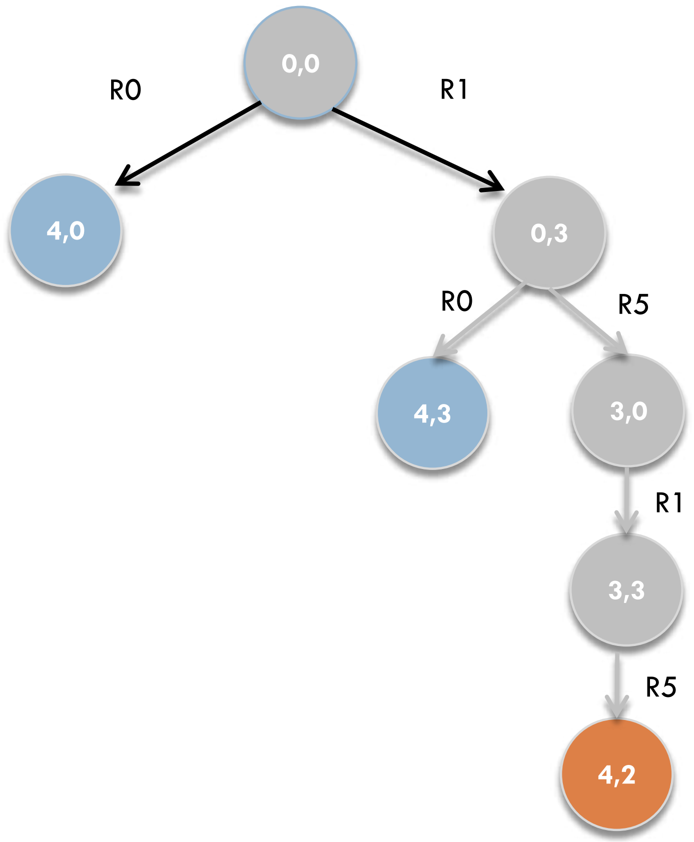
Is it a goal? Yes, return the path to the solution

Water jug problem: other solutions

Breadth-first search:



Depth-first search:

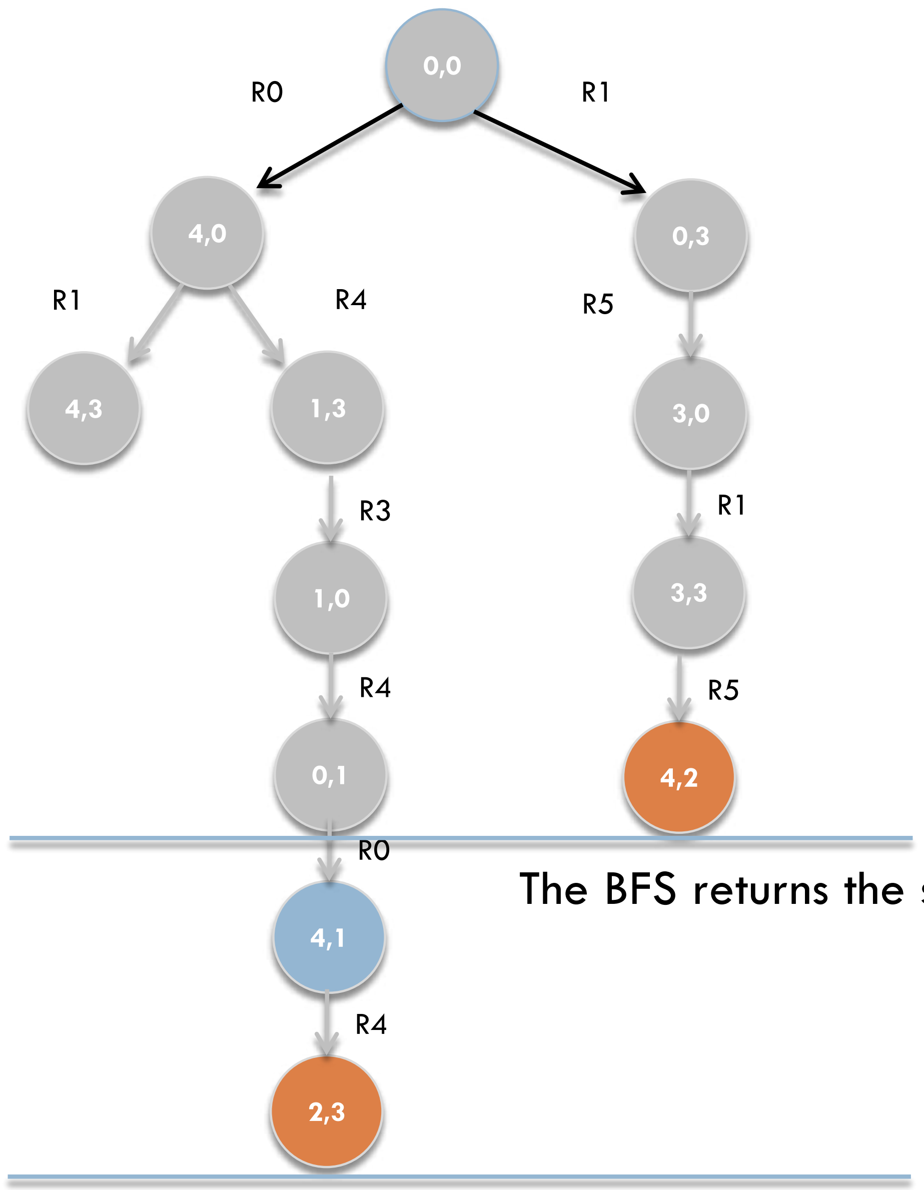


● Visited node ● Generated node, but not visited. Considering the rules executed from R0 to R7

Water jug problem: other solutions

If we could look deeper?

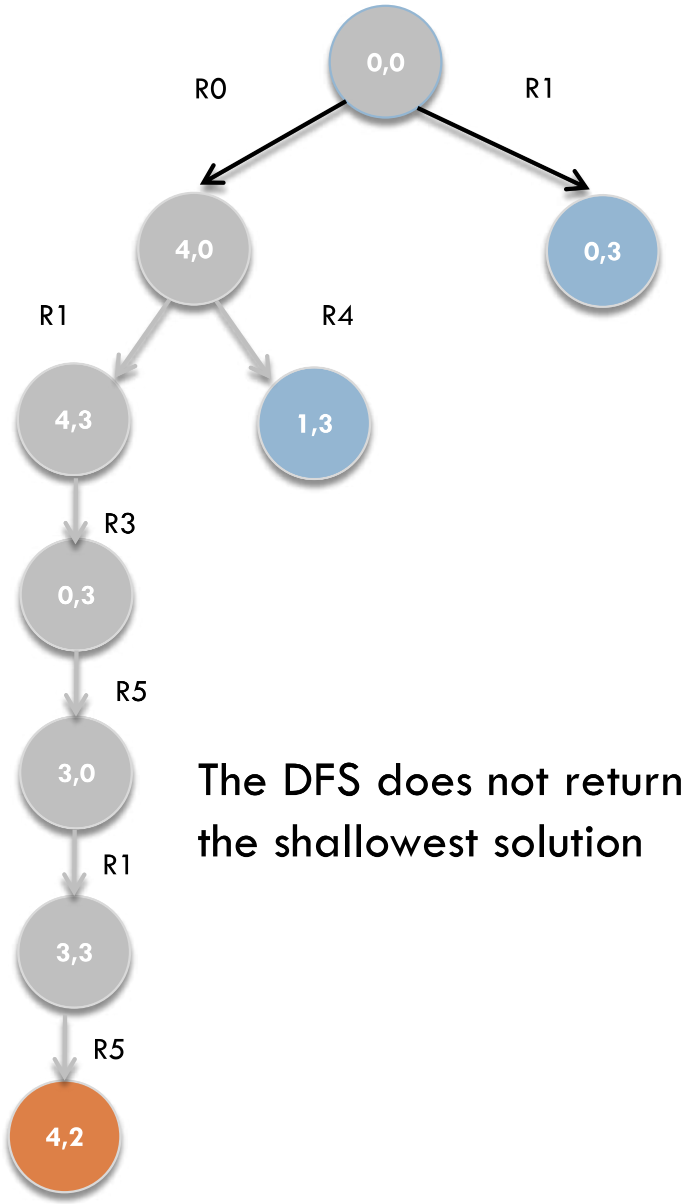
Breadth-first search:



The BFS returns the shallowest

Considering the we are opening the leftmost node

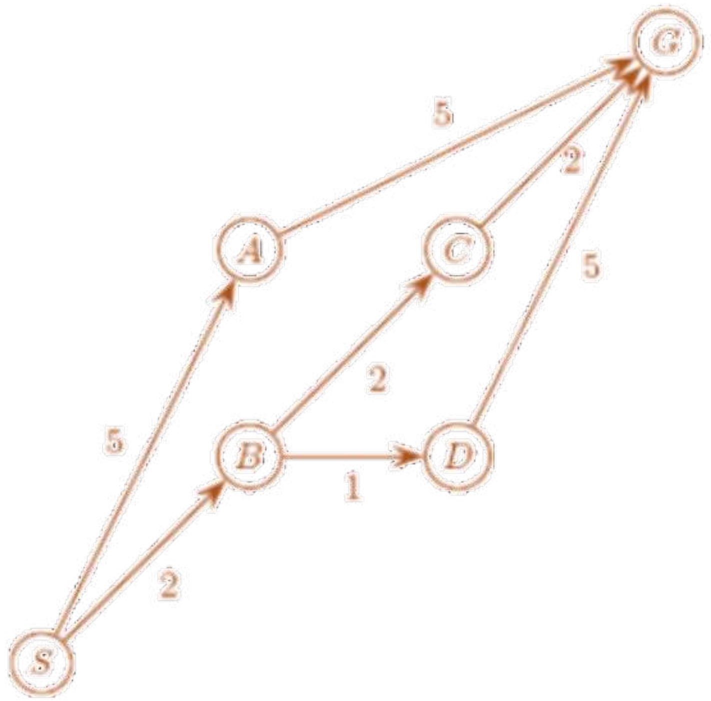
Depth-first search:



The DFS does not return the shallowest solution

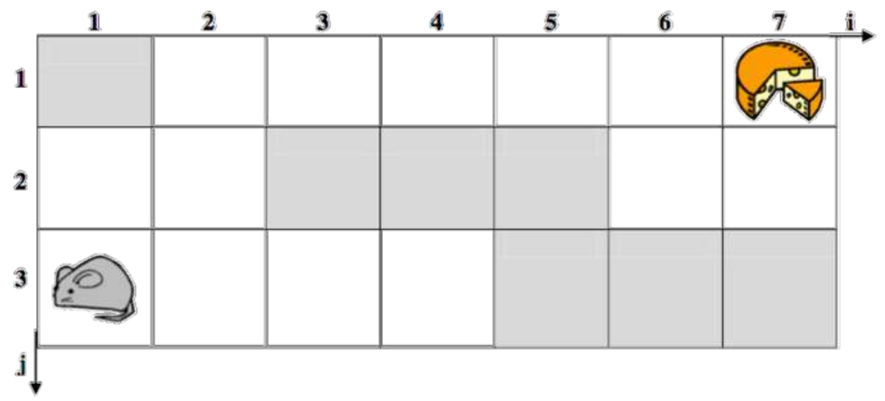
Path problem

- Consider node S as the initial state, G as the sole objective. The labels on the edges indicate the cost of traversing them. Consider the order of opening the nodes to be alphabetical. Compare the generated trees and paths found by one:
  - Breadth-first search
  - Depth-first search
  - Search with uniform cost



Maze problem

- The drawing below shows a mouse in a maze looking for a piece of cheese. In this maze, the mouse can perform the following actions:
  - R1: walk to the right
  - R2: walk to the left
  - R3: walk up
  - R4: walk down



- The rules are always applied in this order (R1, R2, R3, R4).
- White cells represent possible pathways and dark cells represent walls that the mouse cannot pass through. The cost of traveling from one square to another is given by g = 1, for each square covered in the maze. Admit that the mouse maintains a control to avoid repeated positions.
- Admit that the mouse performs the search offline, that is, it processes the entire search tree before going out in search of cheese. Admit that the mouse runs the entire algorithm to determine the best path for the cheese and only then begins its locomotion.
- Indicate the path the mouse will take in its search and display the search tree for the BFS and DFS algorithms.

Extra-class activities

Lecture 4

- Activities
  - Reading:
    - RUSSELL, S. NORVIG, P. Inteligência Artificial. 3ª edição. Capítulo 3.
  - Recommended exercises :
    - 3.1 to 3.15

References

Lecture 4

- RUSSELL, S. NORVIG, P. Inteligência Artificial. 3ª edição.
- Simões, A. S. Slides de aula: IA para Controle e Automação.