Introduction to Artificial Intelligence and Problem Formulations

Dr. Tapas Kumar Mishra

Assistant Professor
Department of Computer Science and Engineering
SRM University
Amaravati-522 240, Andhra Pradesh, India
Email: tapaskumar.m [at] srmap [dot] edu [dot] in

The Nature of Environment

Properties of Environments

1. **Accessible vs. inaccessible:** If an agent's sensory apparatus gives it access to the **complete state of the environment**, then we say that the environment is accessible to that agent.
2. **Deterministic vs. nondeterministic:** If the next state of the environment is completely determined by the **current state** and the actions selected by the agents, then we say the environment is deterministic.
3. **Episodic vs. nonepisodic:** The agent's experience is **divided** into **episodes**.
4. **Static vs. dynamic:** If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static.
5. **Discrete vs. continuous: Chess** is **discrete**. **Taxi driving** is **continuous**.

Hardest Case

Inaccessible, Nondeterministic, Nonepisodic, Dynamic and Continuous

The Nature of Environment

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

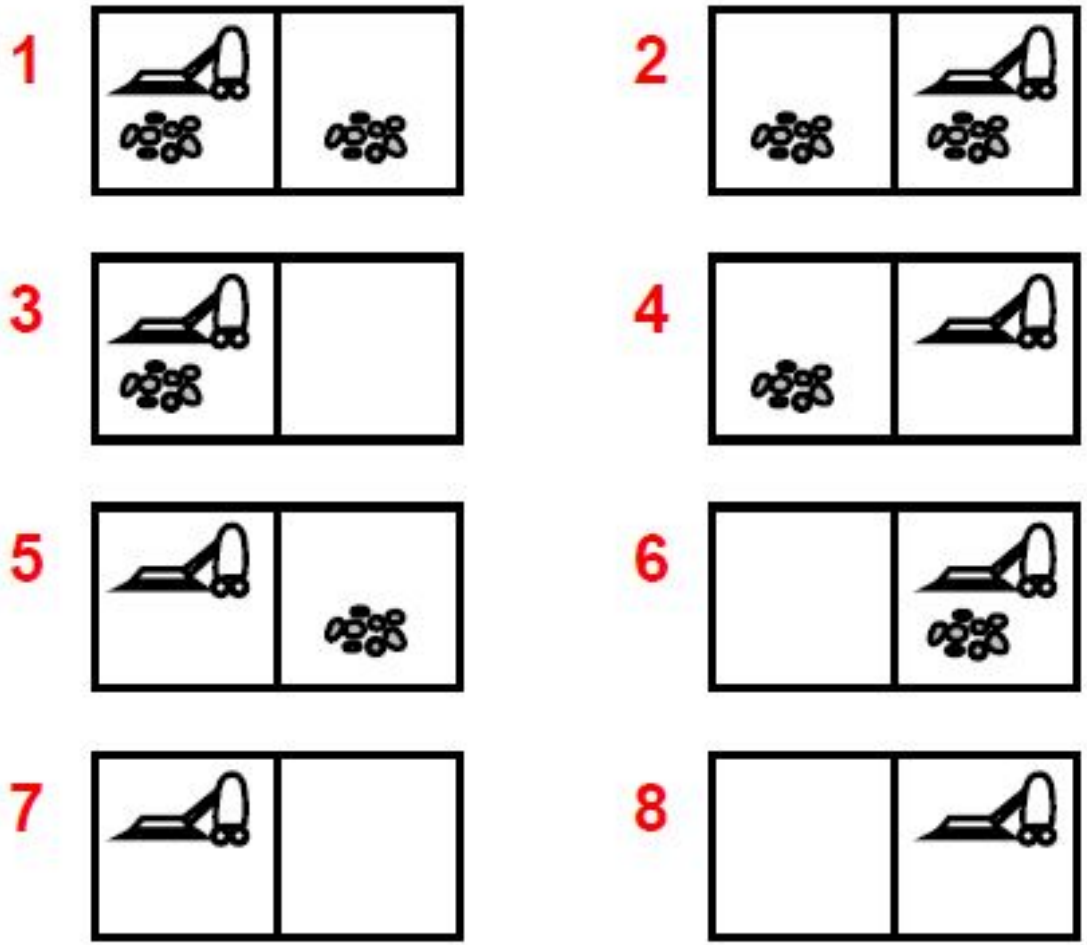
Problem Solving

- A **goal** and **a set of means for achieving the goal** are called a **problem**, and the process of exploring what the means can do is called **search**.
- **Problem-solving agents** decide what to do by **finding sequences of actions** that lead to desirable states.
- **Goal formulation**, based on the **current situation**, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider and follows goal formulation.
- We have a "**formulate**, **search**, **execute phase**" design for the **agent**.
- After formulating a **goal** and a **problem** to solve, the agent calls a **search** procedure to solve it.

Formulating Problems

- There are **four** essentially different types of problems.
  1. single state problems (deterministic, fully observable)
  2. multiple-state problems (non-observable)
  3. contingency problems (nondeterministic and/or partially observable)
  4. exploration problems (unknown state space - "online")
- Let the world contain just **two locations**.
- Each location may or may not contain **dirt**, and the agent may be in one location or the other.
- There are **8 possible world states**.
- The agent has **three possible actions** in this version of the vacuum world: **Left**, **Right** and **Suck**.

Formulating Problems



- The goal is equivalent to the state set $\{7, 8\}$.

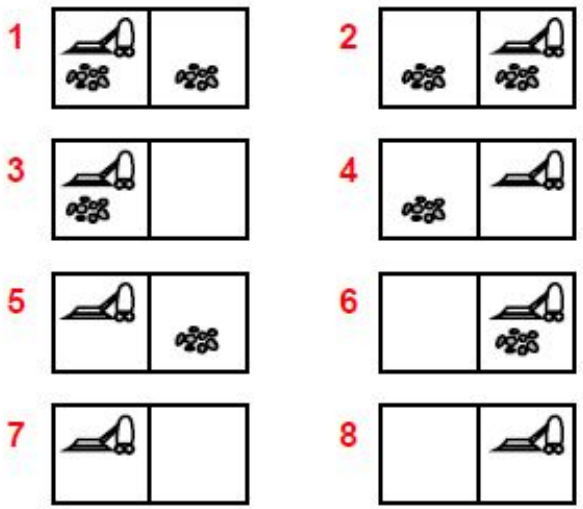
Formulating Problems

Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??
$[Right, Suck, Left, Suck]$

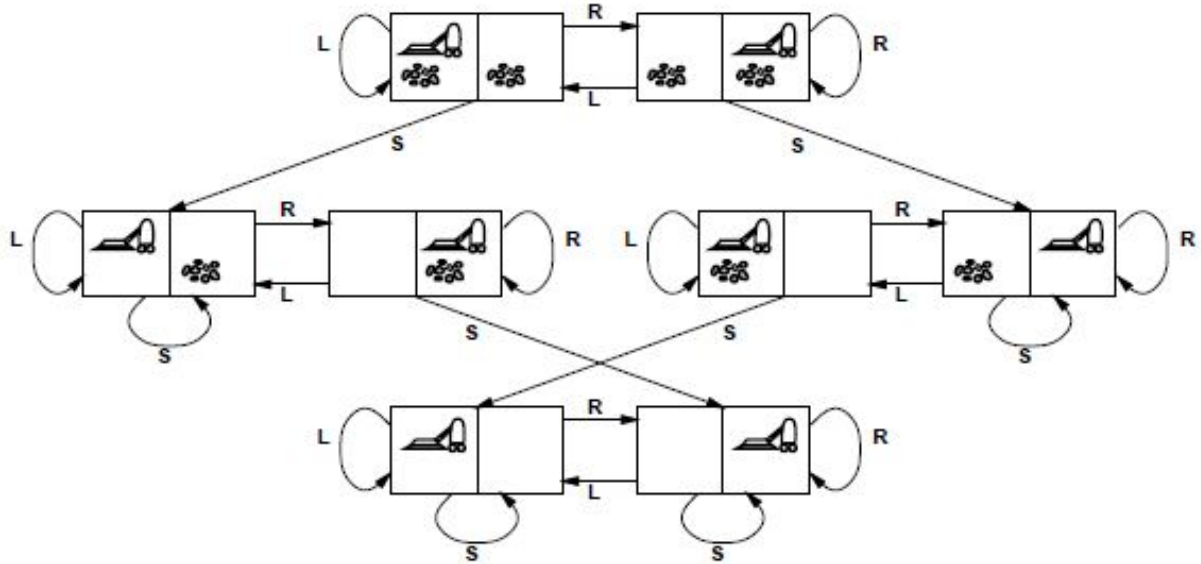
Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
$[Right, \textbf{if } dirt \textbf{ then } Suck]$



- Guaranteed to reach a goal state: [Right, Suck, Left, Suck]
- The agent learns a "map" of the environment, which it can then use to solve subsequent problems. We call this an **exploration problem**.

Vacuum World State Space Graph

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt
path cost??: 1 per action (0 for $NoOp$)

Well-defined Problems and Solutions - State Space Search

- The **initial state** that the agent knows itself to be in.
- **Successor function:** Given a particular state x, $S(x)$ returns the set of states reachable from x by any single action.
- **State space:** The set of all states reachable from the initial state by any sequence of actions.
- A **Path** in the state space is simply any sequence of actions leading from one state to another.
- A **Path cost** function is a function that assigns a cost to a path.
- Together, the **initial state**, **operator set**, **goal test** and **path cost** function define a **problem**.
- The output of a search algorithm is a **solution**, that is, a path from the initial state to a state that satisfies the **goal test**.
- The **total cost** of the search is the **sum** of the **path cost** and the **search cost**.

A simplified road map of part of Romania

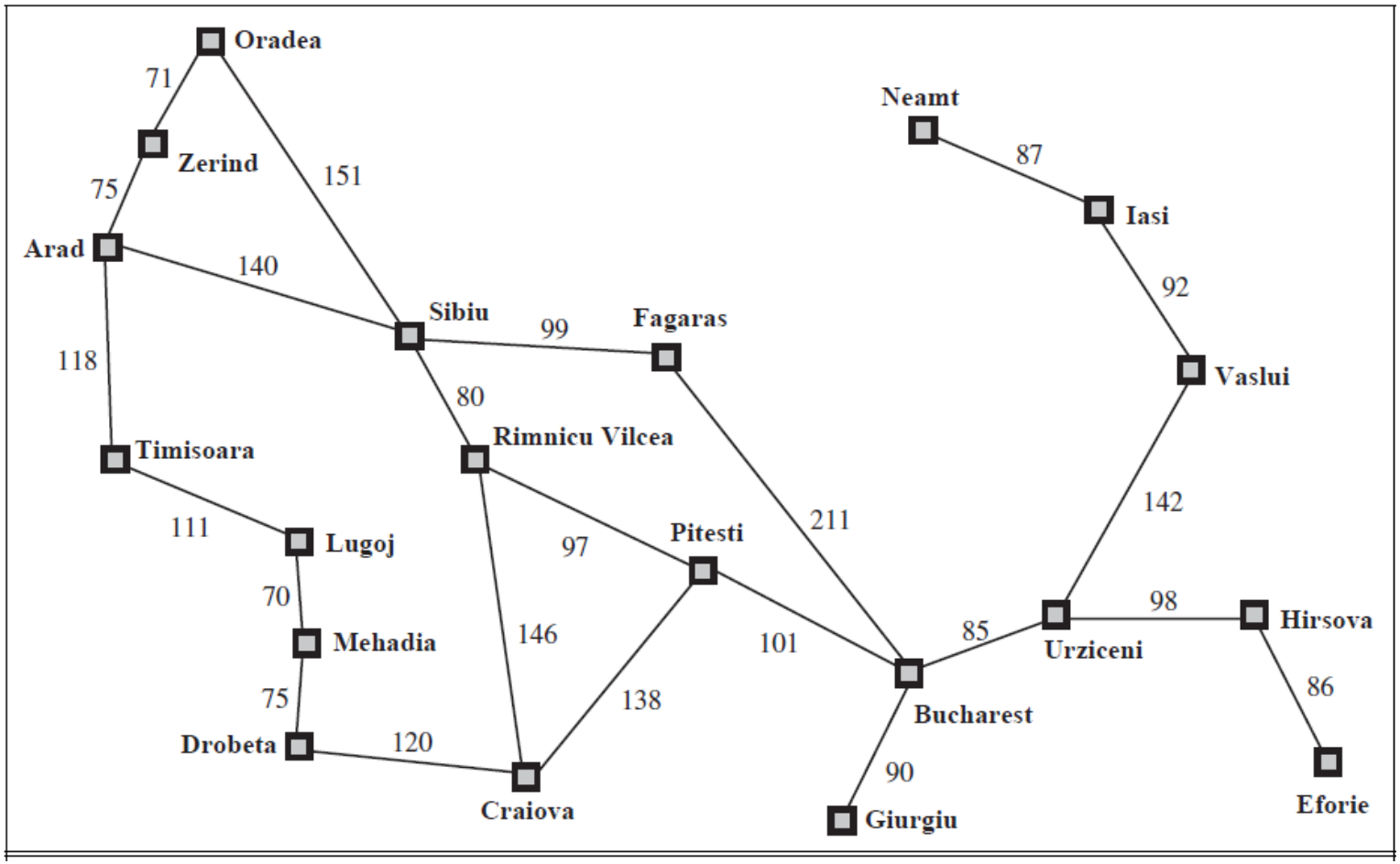


Figure: Page 68, Russel and Norvig, 3rd Edition, 2016 Reprint.

Single-State Problem Formulation

Single-state problem formulation

A problem is defined by four items:

initial state e.g., "at Arad"

successor function $S(x)$ = set of action–state pairs
e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

goal test, can be
explicit, e.g., $x =$ "at Bucharest"
implicit, e.g., $NoDirt(x)$

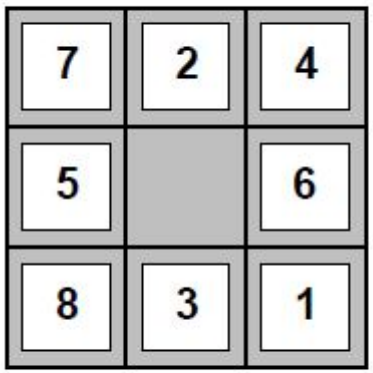
path cost (additive)
e.g., sum of distances, number of actions executed, etc.
$c(x, a, y)$ is the step cost, assumed to be ≥ 0

A solution is a sequence of actions
leading from the initial state to a goal state

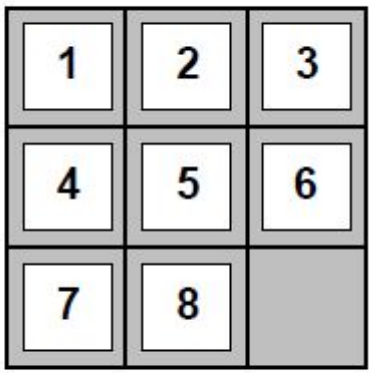
Example Problems

Toy Problems

Example: The 8-puzzle



Start State



Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move

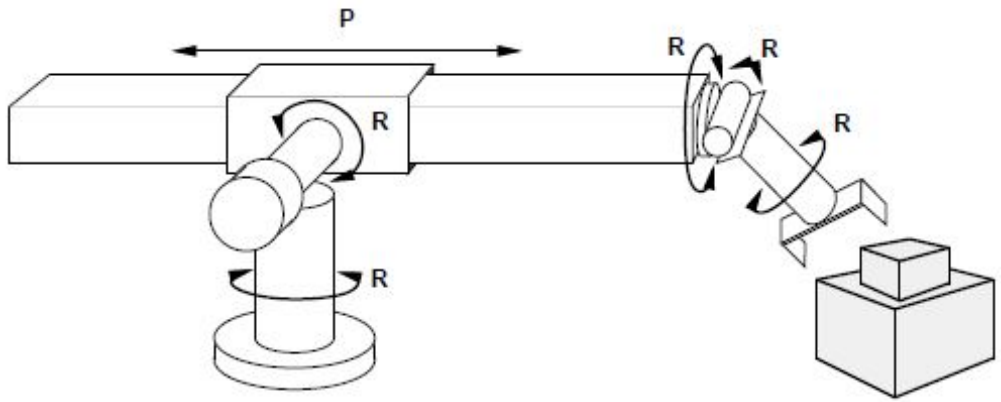
[Note: optimal solution of n-Puzzle family is NP-hard]

- The 8-puzzle belongs to the family of **sliding-block puzzles**.

Example Problems

Robotic Assembly

Example: robotic assembly



states??: real-valued coordinates of robot joint angles
parts of the object to be assembled

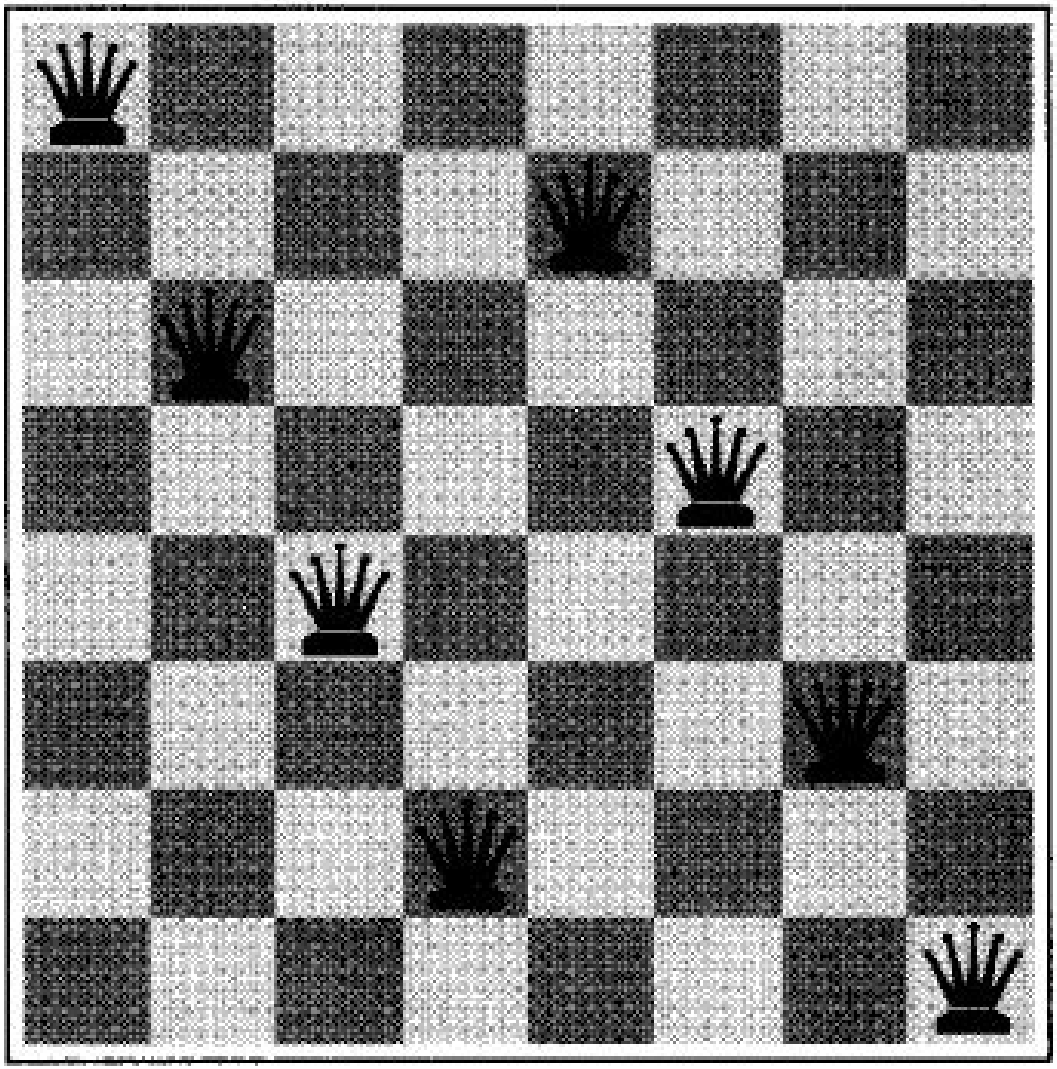
actions??: continuous motions of robot joints

goal test??: complete assembly **with no robot included!**

path cost??: time to execute

Example Problems

8-Queen Problem



Example Problems

8-Queen Problem

The goal of the **8-Queen problem** is to place **eight queens** on a **chessboard** such that **no queen attacks any other**. (A queen attacks any piece in the **same row**, **column** or **diagonal**)

The queen in the **rightmost column** is attacked by the queen at **top left**.

Goal test: 8 queens on board, none attacked.
Path cost: zero
States: any arrangement of 0 to 8 queens on board.
Operators: add a queen to any square.
We have 64^8 possible sequences to investigate.
States: arrangements of 0 to 8 queens **with none attacked**.
Operators: Place a queen in the left-most empty column such that it is not attacked by any other queen.

A quick calculation shows that there are only 2057 possible sequences to investigate.

Example Problems

Crypt arithmetic

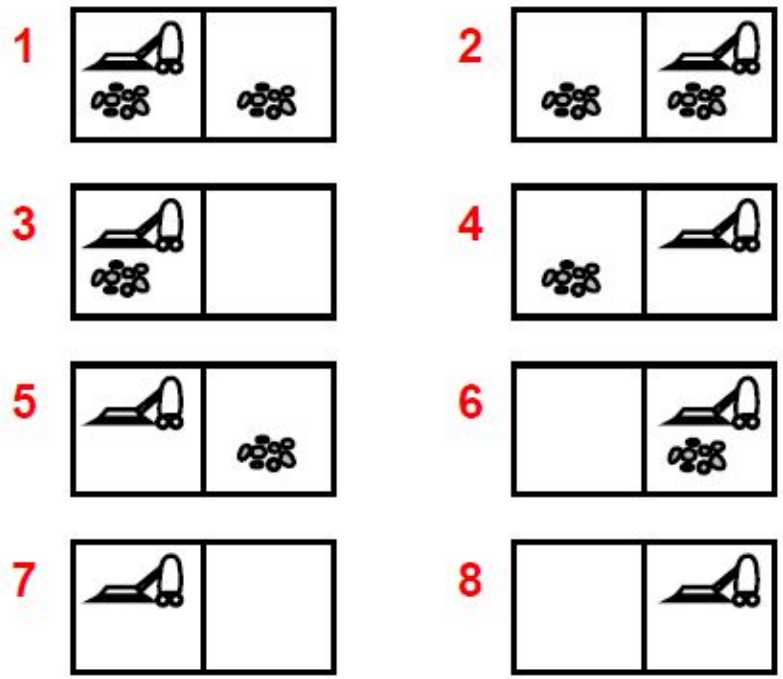
States: a cryptarithmetic puzzle with some letters replaced by digits.
Operators: Replace all occurrences of a letter with a digit not already appearing in the puzzle.
Goal test: Puzzle contains only digits, and represents a correct sum.
Path cost: Zero. All solutions equally valid.

```
 FORTY      Solution:    29786      F=2, 0=9, R=7, etc.
+  TEN                     850
+  TEN                     850
 -----                   -----
 SIXTY                   31486
```

Example Problems

The Vacuum World

States: one of the eight states
Operators: move left, move right, suck.
Goal test: no dirt left in any square.
Path cost: each action costs 1.



Example Problems

Missionaries and Cannibals Problem

Cannibals - A person who eats other human beings
State space - [Missionaries, Cannibals, Boat]
Initial State - [3, 3, 1]
Goal State - [0, 0, 0]
Operators - adding or subtracting the vectors [1, 0, 1], [2, 0, 1], [0, 1, 1], [0, 2, 1] or [1, 1, 1]
Path - moves from [3, 3, 1] to [0, 0, 0]
Path Cost - river trips / number of crossings

Example Problems

Missionaries and Cannibals Problem

Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

Solution

1. Move 2 cannibals to the right
2. Move 1 cannibal back to the left
3. Move 2 cannibals to the right
4. Move 1 cannibal back to the left
5. Move 2 missionaries to the right
6. Move 1 missionary and 1 cannibal back to the left
7. Move 2 missionaries to the right
8. Move 1 cannibal back to the left
9. Move 2 cannibals to the right
10. Move 1 cannibal back to the left
11. Move 2 cannibals to the right

Water Jug Problem

Problem

You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug and the ground on which water may be poured.
Neither jug has any measuring markings on it.
How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State

We will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug.
Note $0 \leq x \leq 4$ and $0 \leq y \leq 3$.
Our initial state: $(0, 0)$

Goal Predicate

State $= (2, y)$ where $0 \leq y \leq 3$.

Water Jug Problem

Operators

We must define a set of operators that will take us from one state to another.

1. Fill 4-gal jug	(x, y) $x < 4$	$\rightarrow$	$(4, y)$
2. Fill 3-gal jug	(x, y) $y < 3$	$\rightarrow$	$(x, 3)$
3. Empty 4-gal jug on ground	(x, y) $x > 0$	$\rightarrow$	$(0, y)$
4. Empty 3-gal jug on ground	(x, y) $y > 0$	$\rightarrow$	$(x, 0)$
5. Pour water from 3-gal jug to fill 4-gal jug	(x, y) $0 < x + y \geq 4$ and $y > 0$	$\rightarrow$	$(4, y - (4 - x))$
6. Pour water from 4-gal jug to fill 3-gal jug	(x, y) $0 < x + y \geq 3$ and $x > 0$	$\rightarrow$	$(x - (3 - y), 3)$
7. Pour all of water from 3-gal jug into 4-gal jug	(x, y) $0 < x + y \leq 4$ and $y \geq 0$	$\rightarrow$	$(x + y, 0)$
8. Pour all of water from 4-gal jug into 3-gal jug	(x, y) $0 < x + y \leq 3$ and $x \geq 0$	$\rightarrow$	$(0, x + y)$

Water Jug Problem

Solution

Gals in 4-gal jug	Gals in 3-gal jug	Rule Applied
0	0	-
4	0	1. Fill 4
1	3	6. Pour 4 into 3 to fill
1	0	4. Empty 3
0	1	8. Pour all of 4 into 3
4	1	1. Fill 4
2	3	6. Pour into 3

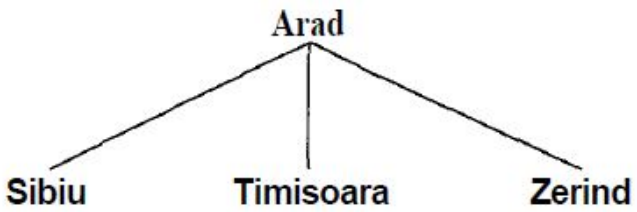
Real-World Problems

- Route Finding
- Touring and travelling salesperson problems
- VLSI layout: cell layout and channel routing
- Robot navigation
- Assembly sequencing

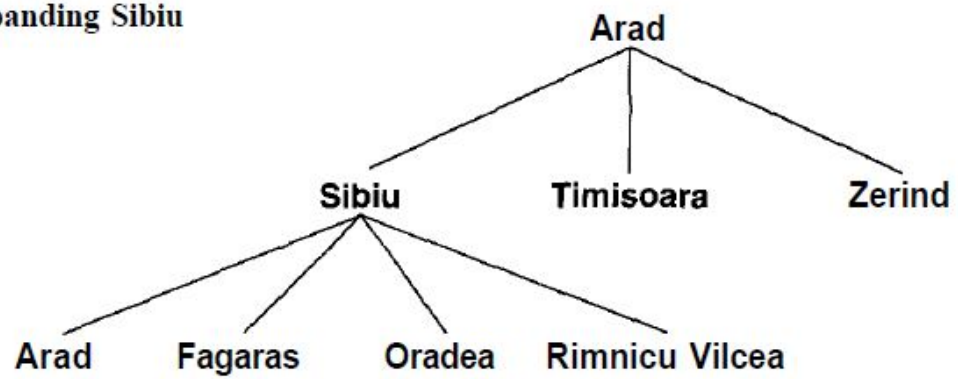
Searching for Solutions

(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu



Searching for Solutions

- There are many ways to represent nodes. We will assume a **node** is a **data structure** with **five components**.
  1. the **state** in the **state space** to which the node corresponds
  2. the **node** in the **search tree** that generated this node (this is called the **parent node**)
  3. the **operator** that was applied to generate the node;
  4. the **number of nodes** on the path from the **root to this node** (the **depth** of the node);
  5. the **path cost** of the path from the **initial state to the node**
- The collection of nodes that are **waiting to be expanded** is called the **fringe** or **frontier**.

Search Strategies

- Evaluate strategies in terms of **four** criteria.
  1. **Completeness:** is the strategy guaranteed to find a solution when there is one?
  2. **Time complexity:** how long does it take to find a solution?
  3. **Space complexity:** how much memory does it need to perform the search?
  4. **Optimality:** does the strategy find the highest-quality solution when there are several different solutions?

Uninformed/Blind and Informed Search

Uninformed/Blind Search

No preference is given to the order of successor node generation and selection.

1. Breadth-First Search
2. Uniform-Cost Search
3. Depth-First Search
4. Depth-Limited Search
5. Iterative Deepening Search

Informed Search

Some information about the problem space is used to compute a preference among the children for exploration and expansion.

1. Heuristic Function
2. Best First Search
3. A$*$ Search
4. AO$*$ Search
5. Hill Climbing Search

Uninformed/Blind Search - Breadth-First Search

Algorithm

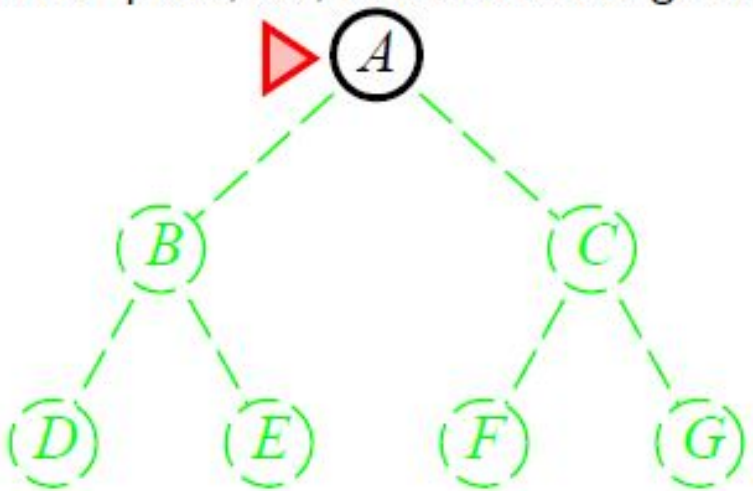
1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element from the queue and place all the children at the end of the queue in **any order**.
5. Return to step 2.

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



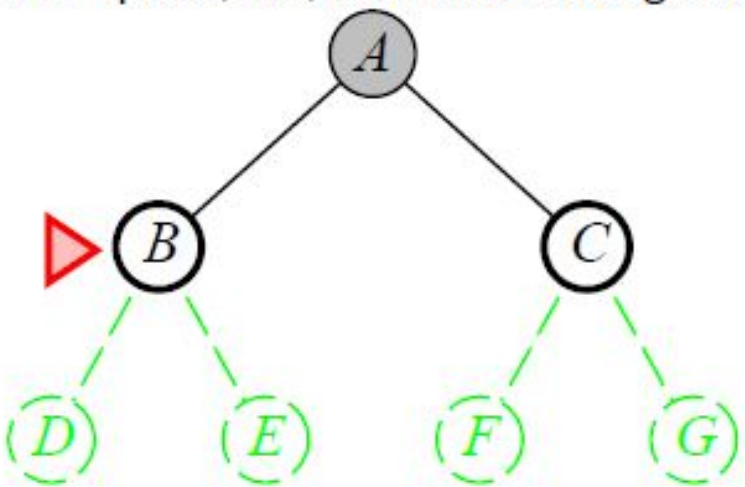
- The root node is expanded first, then all the nodes generated by the root node are expanded next.
- All the nodes at depth d in the search tree are expanded before the nodes at depth $d + 1$

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

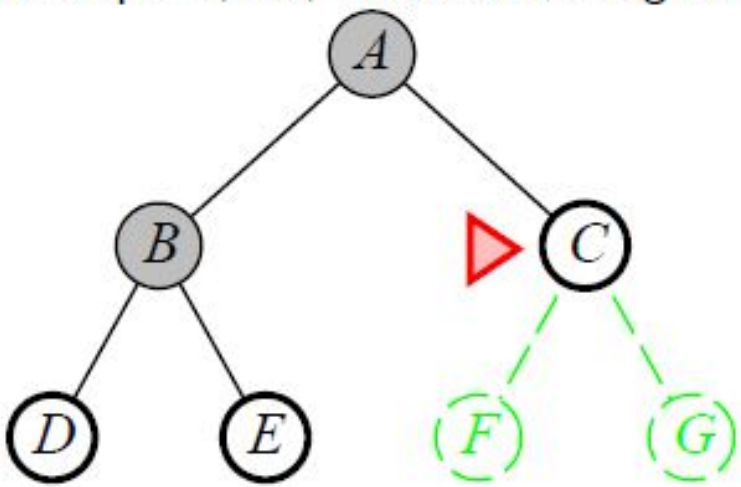


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

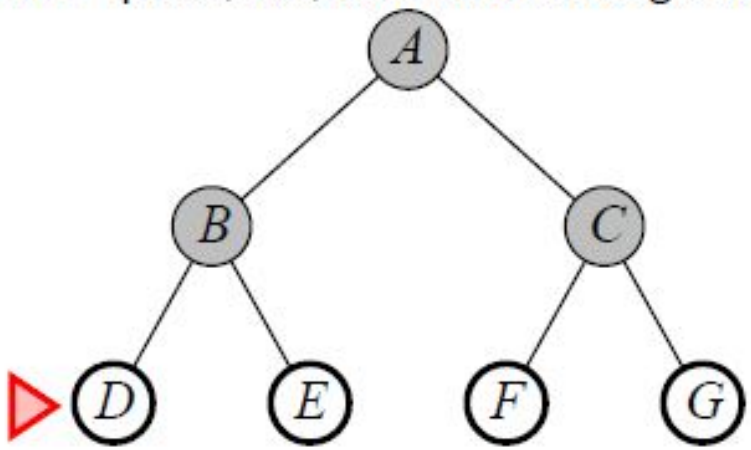


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Uninformed/Blind Search - Breadth-First Search

- b - maximum **branching factor** of the search tree
- d - **depth** of the least-cost solution
- m - maximum depth of the state space (may be ∞)

Properties

1. **Complete:** Yes (if b is finite)
2. **Time:** $1 + b + b^2 + \cdots + b^d = O(b^d)$
3. $O(b^d)$ (keeps every node in memory)
4. **Optimal:** Yes (if cost $= 1$ per step); not optimal in general
5. It is **optimal** provided the **path cost** is a **nondecreasing function** (i.e., 7, 8, 8, 10, 11, 11, 12) of the depth of the node. (This condition is usually satisfied only when **all operators** have the **same cost**.)

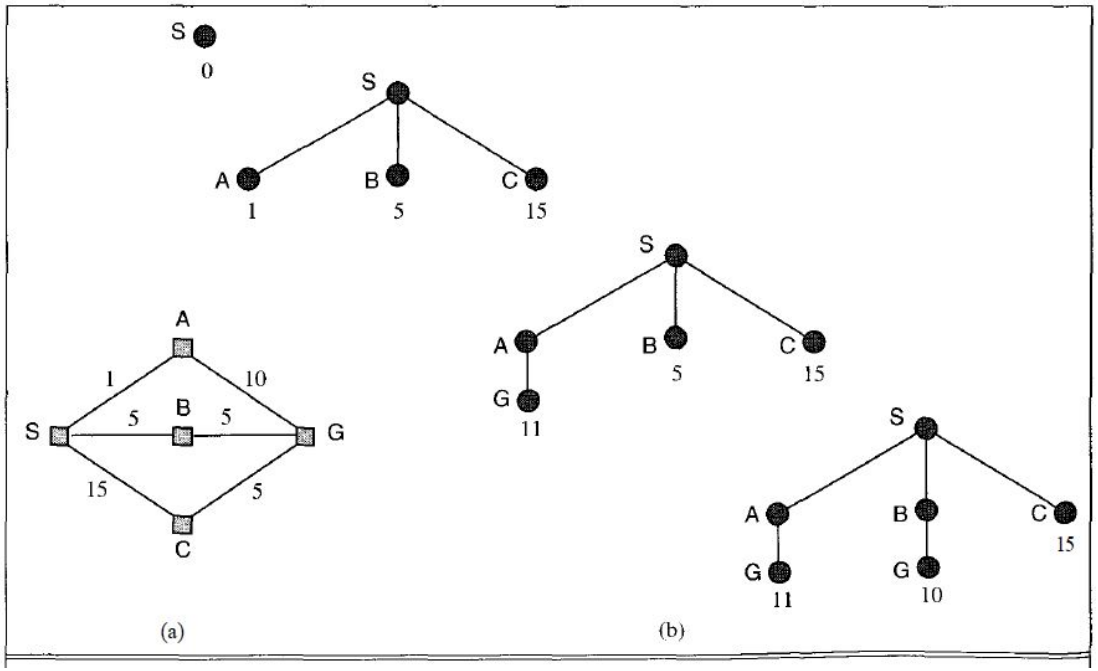
Uninformed/Blind Search - Breadth-First Search

- The memory requirements are a bigger problem for breadth-first search than the execution time.
- $b = 10$, $d = 2$, Nodes $= 1 + 10 + 10^2 = 111$

Depth (d)	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	**111**	**1 seconds**	**11 kilobytes**
4	11111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11111 terabytes

Uninformed/Blind Search - Uniform Cost Search

- Uniform cost search **modifies** the breadth-first strategy by always expanding the **lowest-cost node** on the **fringe** (as measured by the **path cost** $g(n)$), rather than the **lowest-depth node**.
- The problem is to get from S to G and the cost of each operator is marked.
- The strategy first expands the initial state, yielding paths to A, B and C. Because the path to A is cheapest, it is expanded next, generating the **path** SAG, which is **in fact a solution**, though **not the optimal one**.



Uninformed/Blind Search - Uniform Cost Search

- The next step is to expand *SB*, generating *SBG*, which is now the **cheapest path** remaining in the queue, so it is **goal-checked** and returned as the **solution**.
- If every operator has a **nonnegative cost**, then the **cost of a path** can **never decrease** as we go along the path.
- But if some operator had a **negative cost**, then nothing but an **exhaustive search** of all nodes would find the **optimal solution**.

Properties

Same as BFS.

Uninformed/Blind Search - Depth-First Search

Algorithm

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element, and place the children at the front of the queue in **any order**.
5. Return to step 2.

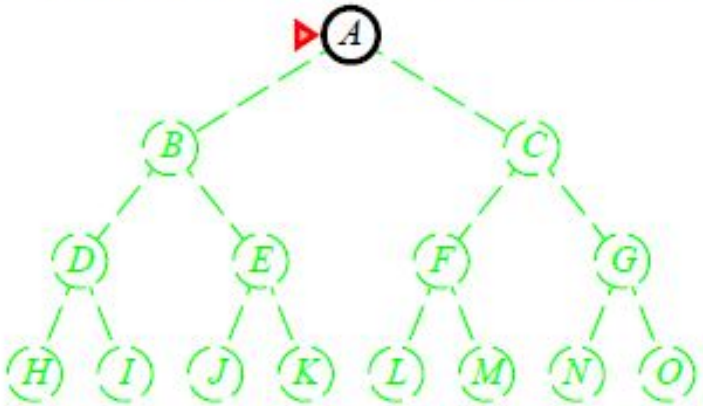
- Depth-first may actually be faster than breadth-first.
- **Implementation:** Stack

Depth-first search

Expand deepest unexpanded node

Implementation:

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

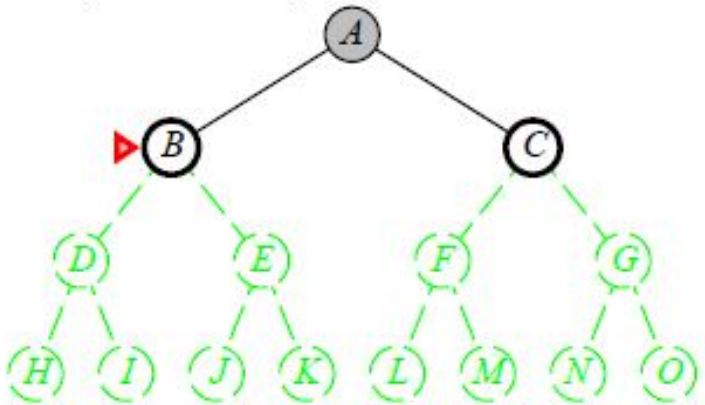


Depth-first search

Expand deepest unexpanded node

Implementation:

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

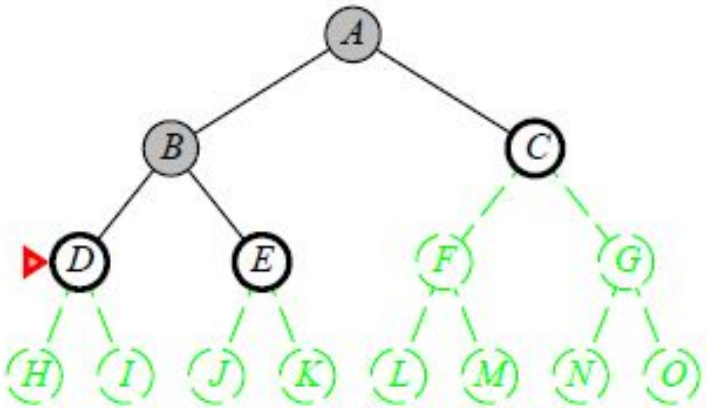


Depth-first search

Expand deepest unexpanded node

Implementation:

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

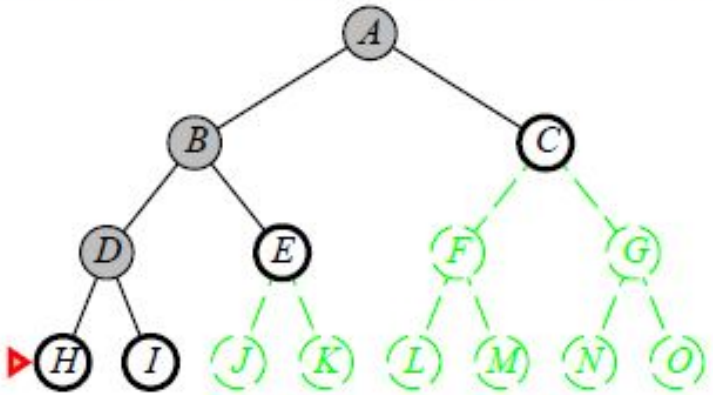


Depth-first search

Expand deepest unexpanded node

Implementation:

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

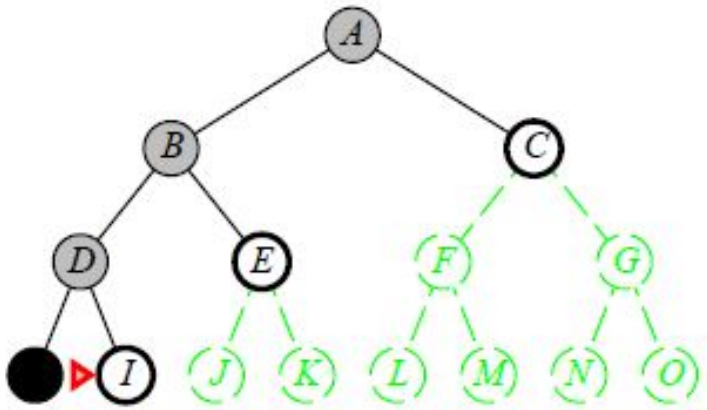


Depth-first search

Expand deepest unexpanded node

Implementation:

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

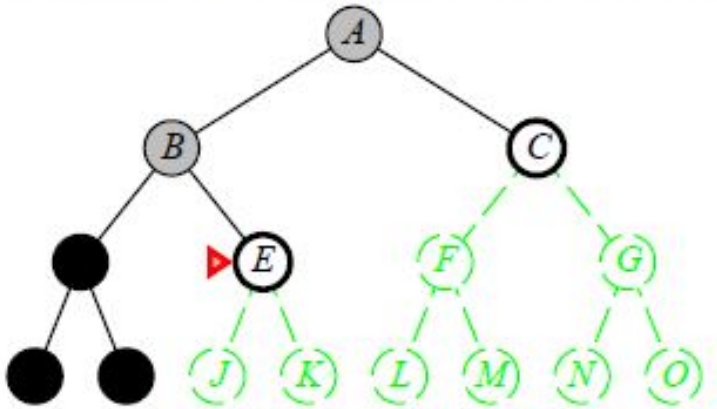


Depth-first search

Expand deepest unexpanded node

Implementation:

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

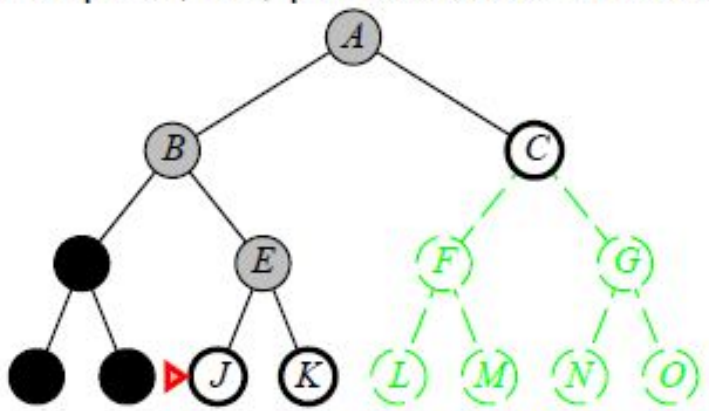


Depth-first search

Expand deepest unexpanded node

Implementation:

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

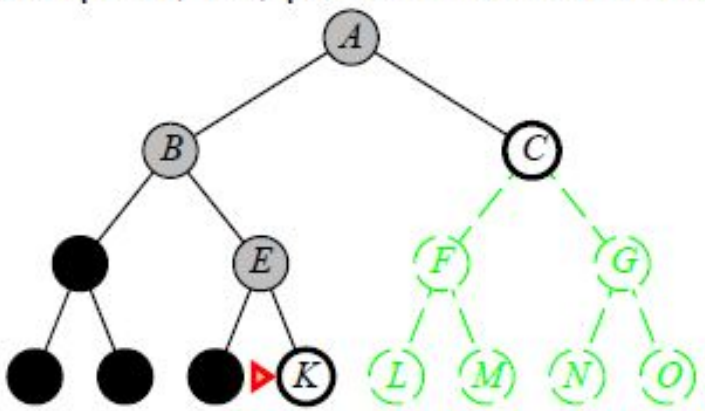


Depth-first search

Expand deepest unexpanded node

Implementation:

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

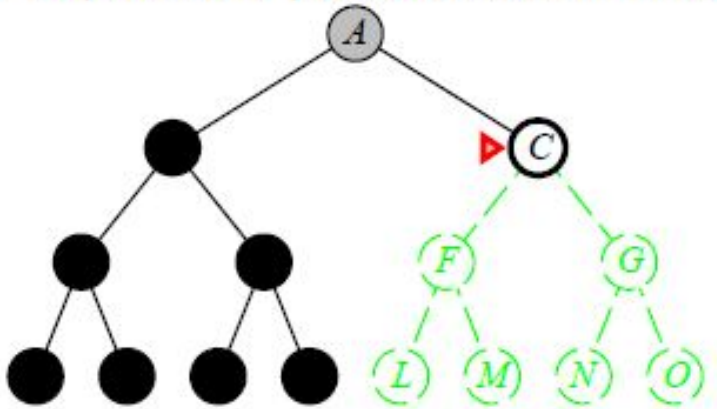


Depth-first search

Expand deepest unexpanded node

Implementation:

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

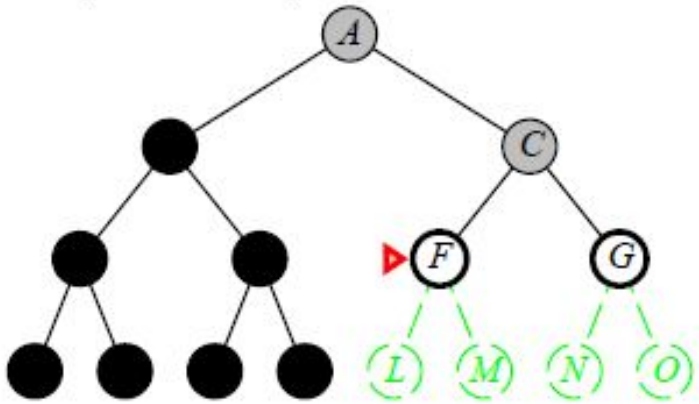


Depth-first search

Expand deepest unexpanded node

Implementation:

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

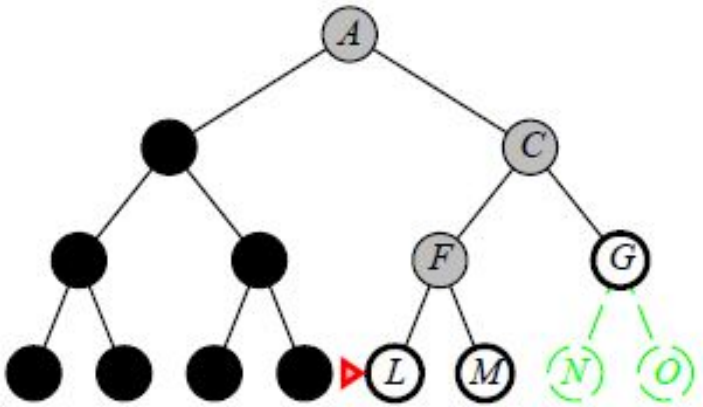


Depth-first search

Expand deepest unexpanded node

Implementation:

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

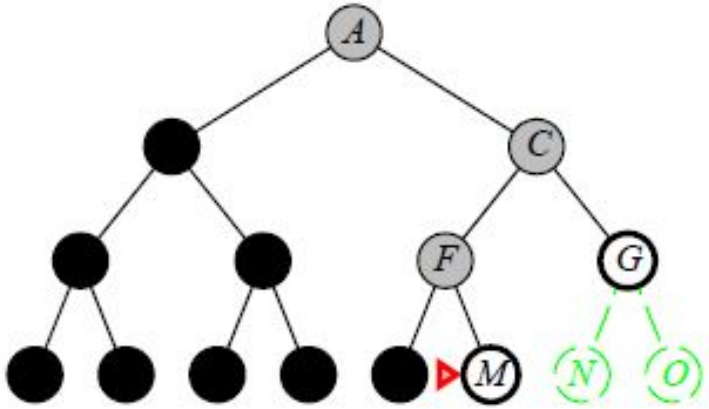


Depth-first search

Expand deepest unexpanded node

Implementation:

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



Properties of depth-first search

Complete?? 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

- It can get stuck going down the wrong path.
- Depth-first search should be avoided for search trees with large or infinite maximum depths.


Thank You!


SRM UNIVERSITY
AP