Artificial Intelligence *CS361*

Prof. Abdel-Rahman Hedar

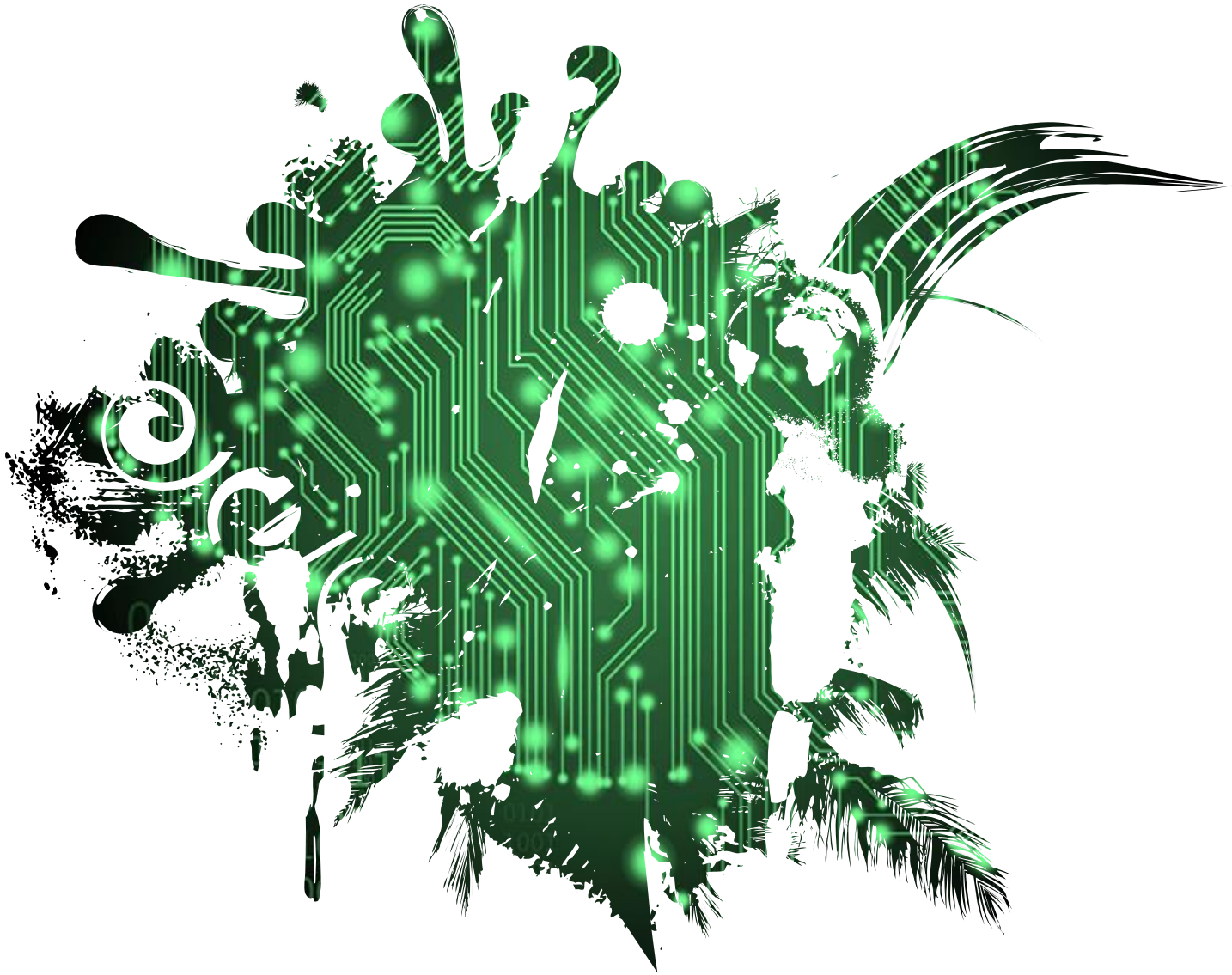
Solving Problems by Searching

Chapter 3 – Part I

Contents

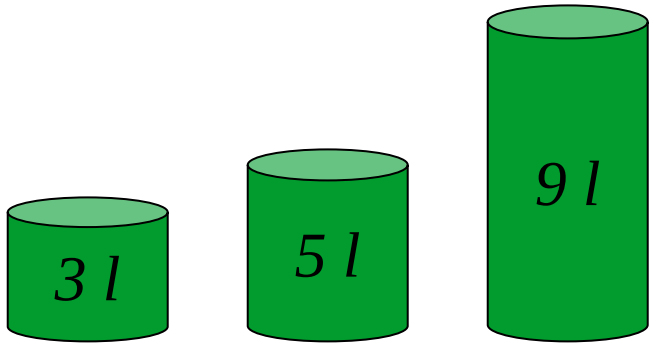
- Introduction to Problem Solving
- Complexity
- Uninformed search
  - Problem formulation
  - Search strategies: depth-first, breadth-first
- Informed search
  - Search strategies: best-first, A*
  - Heuristic functions

Introduction



Example: Measuring Problem!

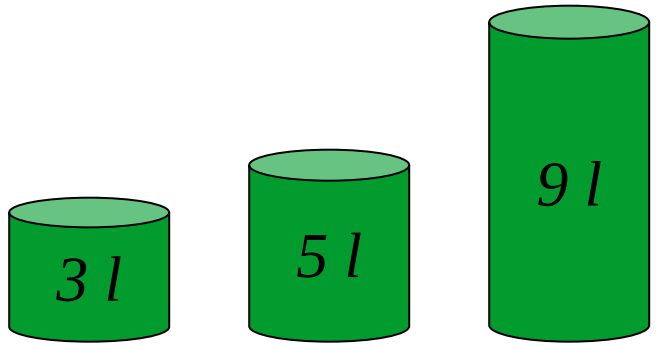
» Problem: Using these three buckets, measure 7 liters of water.



Example: Measuring Problem!

- **(one possible) Solution:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	**7**	**goal**



Which solution do we prefer?

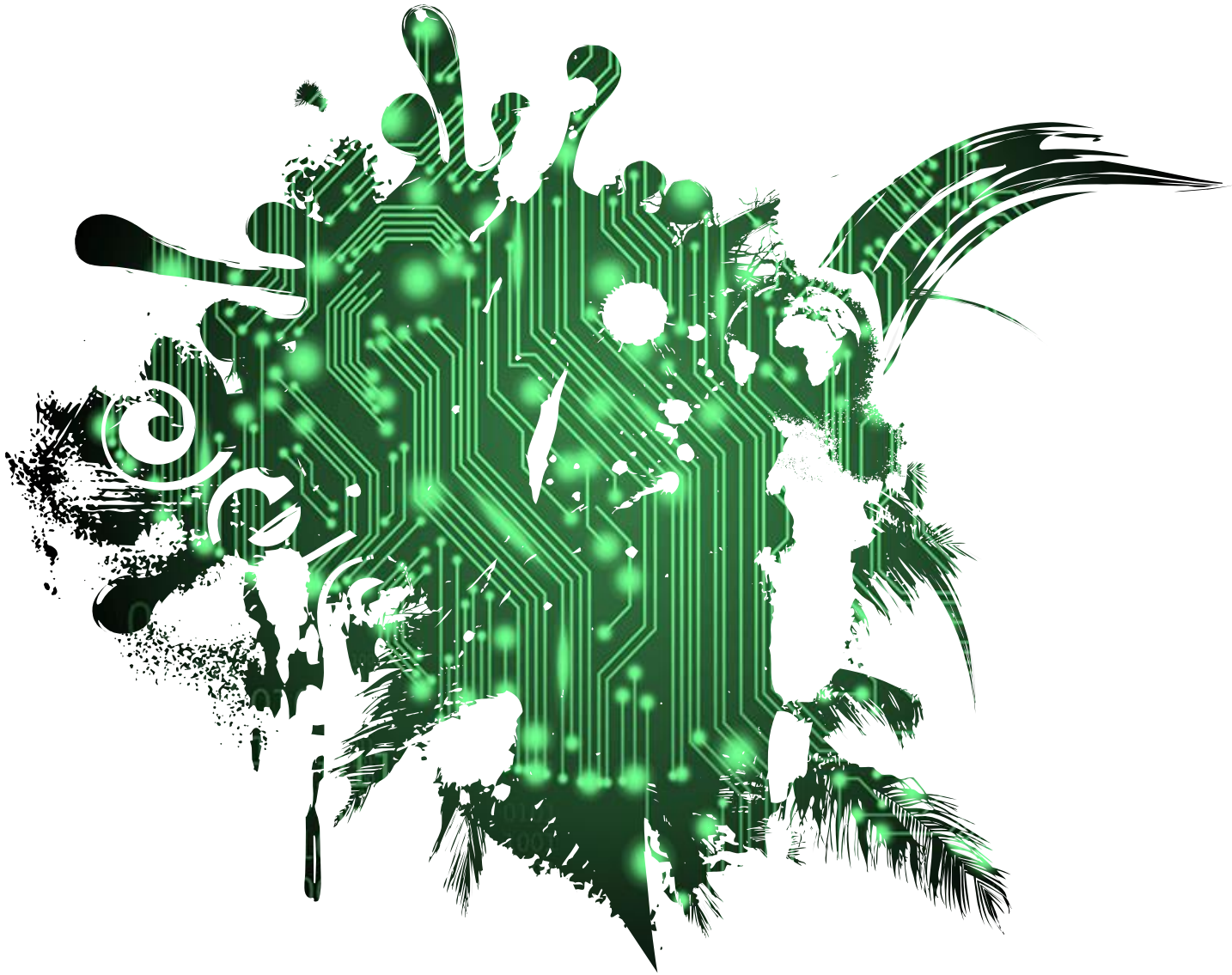
- **Solution 1:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	**7**	**goal**

- **Solution 2:**

a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3	0	**7**	**goal**

Problem-Solving Agents



Problem-Solving Agents

- Problem solving
  - Goal formulation
  - Problem formulation (states, operators)
  - Search for solution
- Problem formulation
  - Initial state
  - Operators
  - Goal test
  - Path cost

Problem-Solving

- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
- **Search for solution** is the process of looking for a sequence of actions that reaches the goal.
  - A search algorithm takes a problem as input and returns a solution in the form of an action sequence.

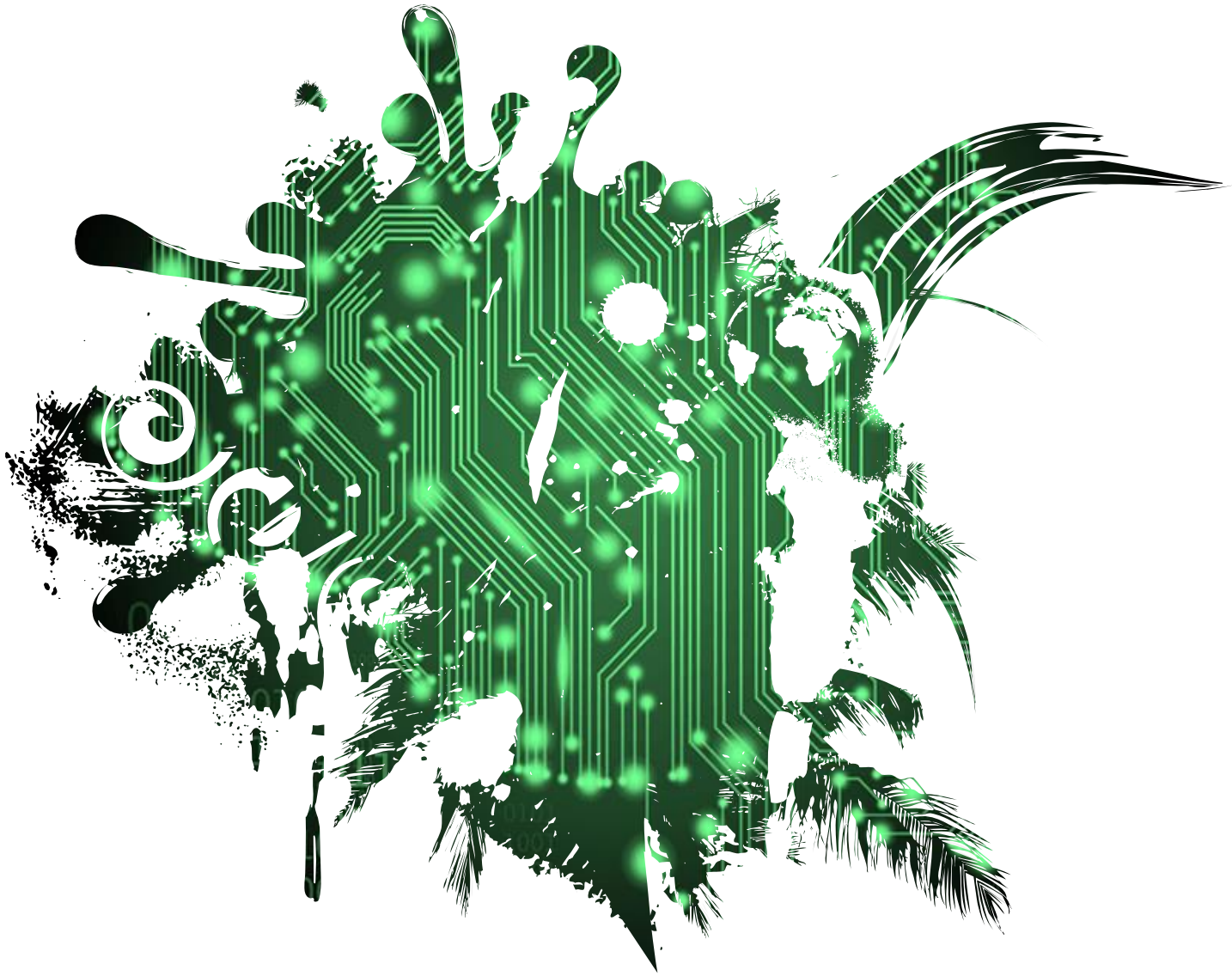
Problem Formulation

- **The initial state** that the agent starts in.
- A description of the possible **actions** available to the agent.
  - Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is applicable in s.
- **Transition model** is a description of what each action does specified by a function RESULT(s, a) that returns the state that results from doing action a in state s.
  - We also use the term successor to refer to any state reachable from a given state by a single action.

Problem Formulation

- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test**, which determines whether a given state is a goal s
- A **path cost** is a function that assigns a numeric cost to each path.
- The **step cost** of taking action a in state s to reach state s′ is denoted by c(s, a, s′).

Examples of Search Problems



Example 1: Buckets

- Measure 7 liters of water using a 3-liter, a 5-liter, and a 9-liter buckets.
- Formulate goal: Have 7 liters of water in 9-liter bucket
- Formulate problem:
  - States: amount of water in the buckets
  - Operators: Fill bucket from source, empty bucket
- Find solution: sequence of operators that bring you from current state to the goal state

Example 2: Vacuum World

- Simplified world: 2 locations, each may or not contain dirt, each may or not contain vacuuming agent.
- Goal of agent: clean up the dirt.

States: Integer Dirt, Robot Location
Actions: L, R, S, Off
Goal Test: No dirt
Path Cost: 1 for (L, R, S), 0 for (Off)

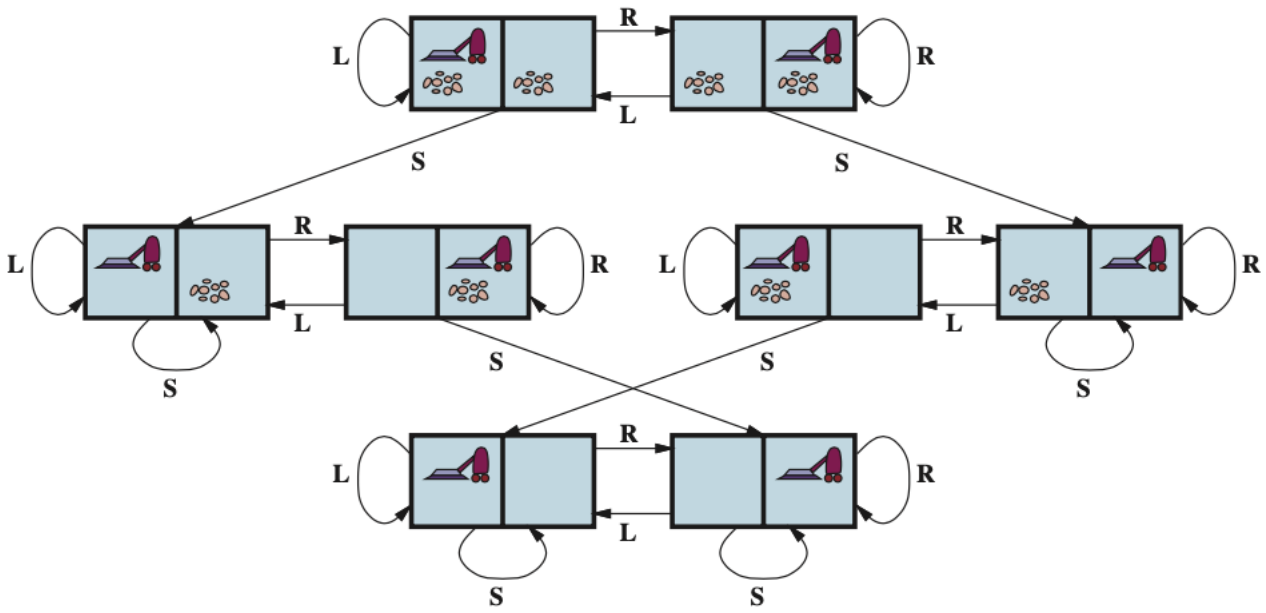
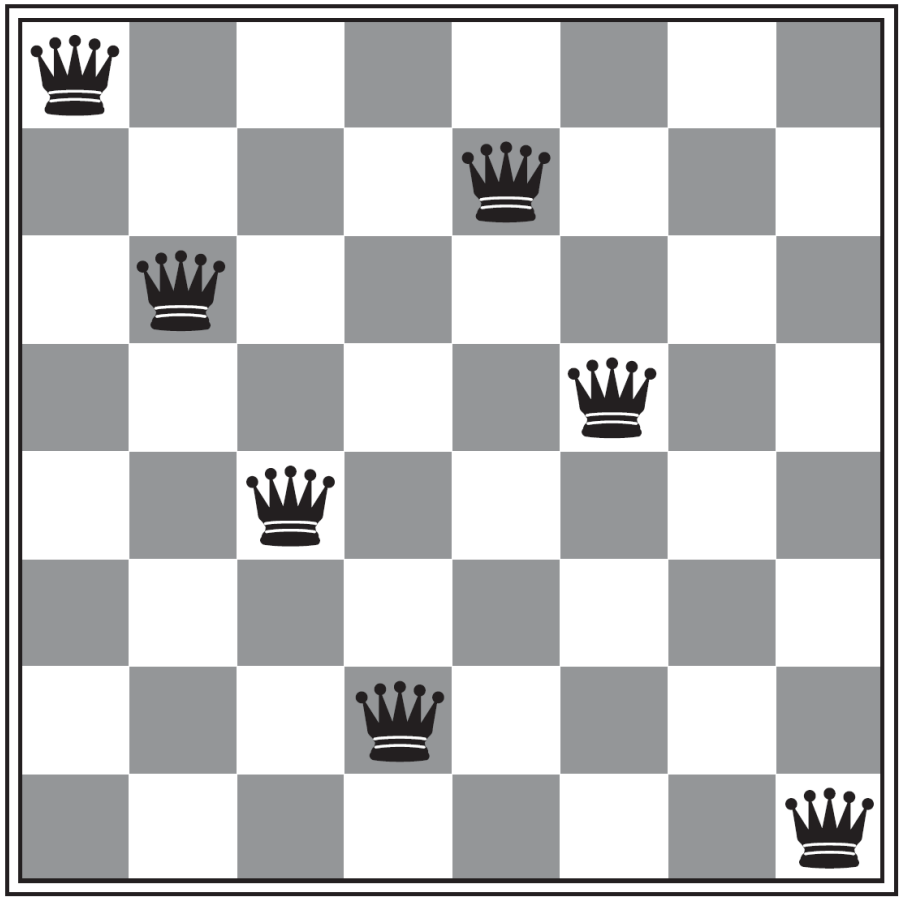


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Example 3: 8-Queens Problem

- The goal of the **8-queens 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.
- An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem,
  - this means that each action adds a queen to the state.
- A **complete-state formulation** starts with all 8 queens on the board and moves them around.

Example 3: 8-Queens Problem



Example 3: 8-Queens Problem

- **States:** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model:** Returns the board with a queen added to the specified square.
- **Goal test:** 8 queens are on the board, none attacked.

In this formulation, we have $64 \times 63 \times \ldots \times 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate.

Example 4: Romania

- In Romania, on vacation. Currently in Arad.
- Flight leaves tomorrow from Bucharest.
- Formulate goal be in Bucharest
- Formulate problem:
  - states: various cities
  - operators: drive between cities
- Find solution:
  - sequence of cities, such that total driving distance is minimized.

Example 4: Romania

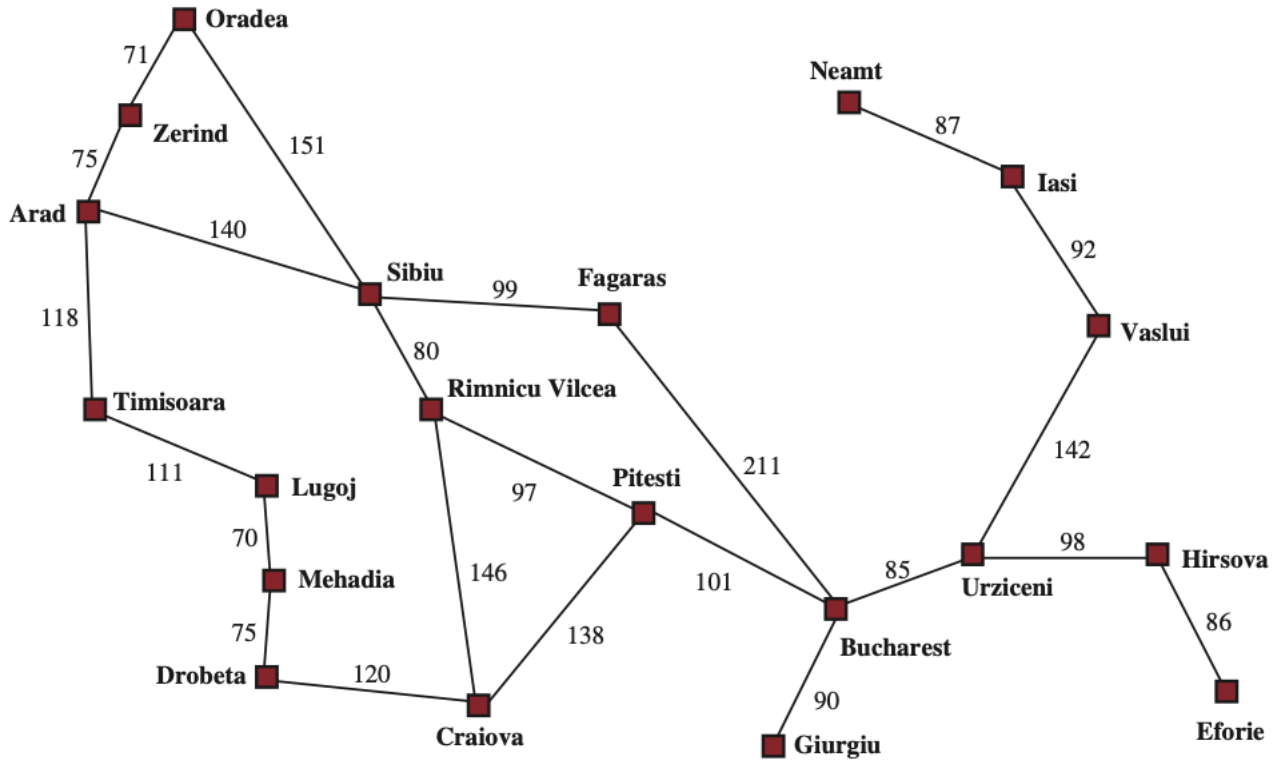


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Example 5: 8-Puzzle

- **State:** integer location of tiles (ignore intermediate locations)
- **Operators:** moving blank left, right, up, down (ignore jamming)
- **Goal test:** does state match goal state?
- **Path cost:** 1 per move

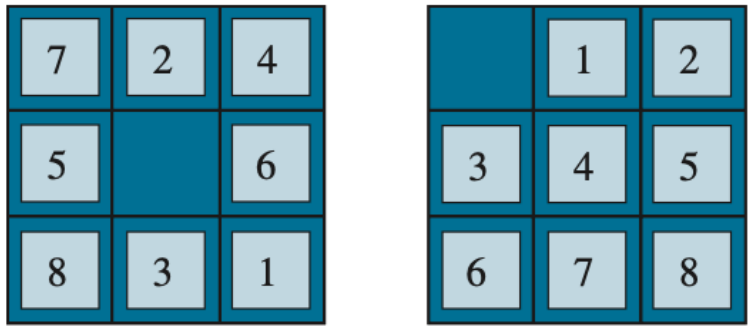


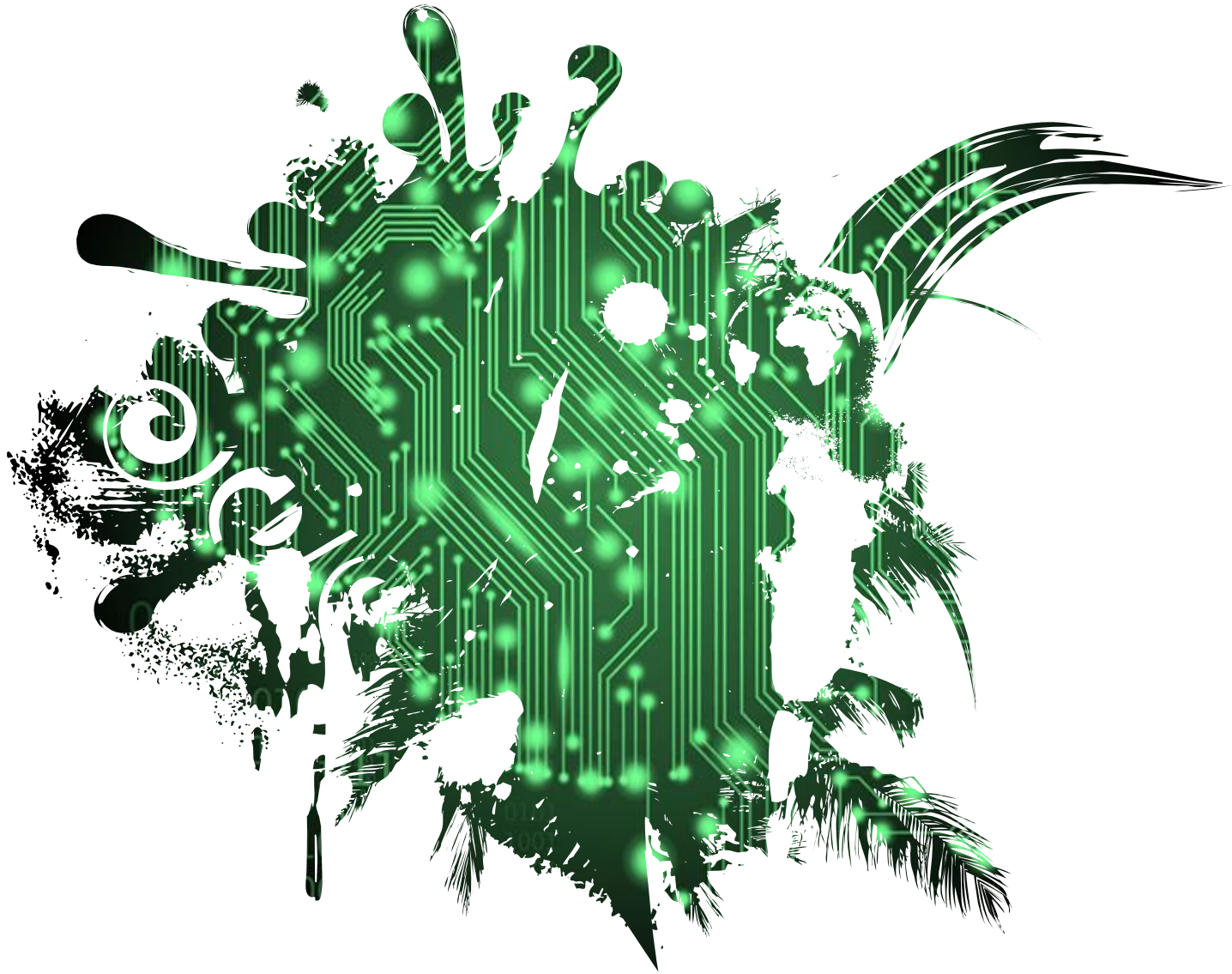
Figure 3.3 A typical instance of the 8-puzzle.

Example 5: 8-Puzzle

» Why search algorithms?
- 8-puzzle has 9!/2 = 181, 440 states
- 15-puzzle has 16!/2 ≈ 10^12 states
- 24-puzzle has 25!/2 ≈ 10^25 states
- Note: we have two possible solutions, the blank block in the first or in the last.

» So, we need a principled way to look for a solution in these huge search spaces.

Search Algorithms



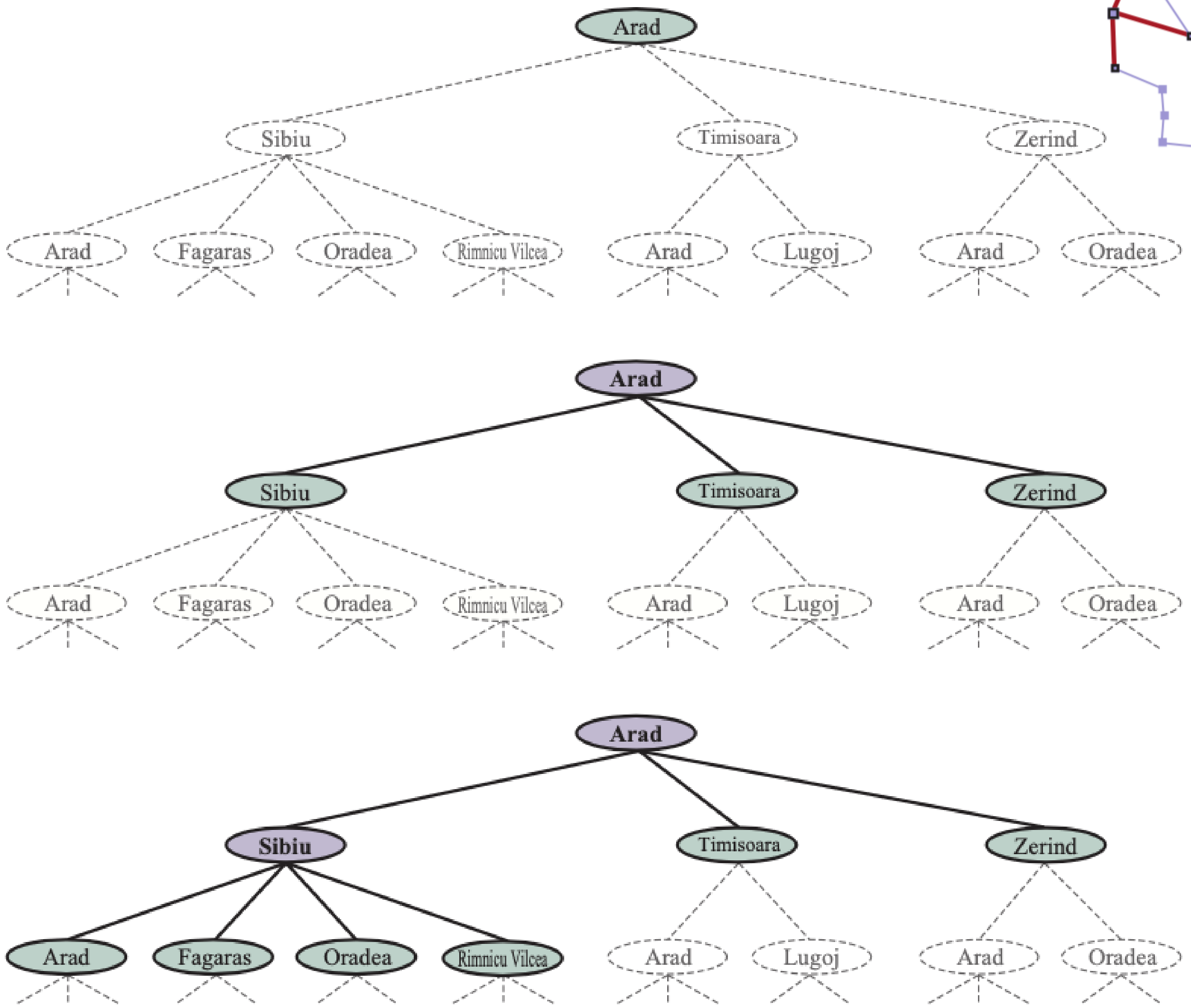
Search Algorithms

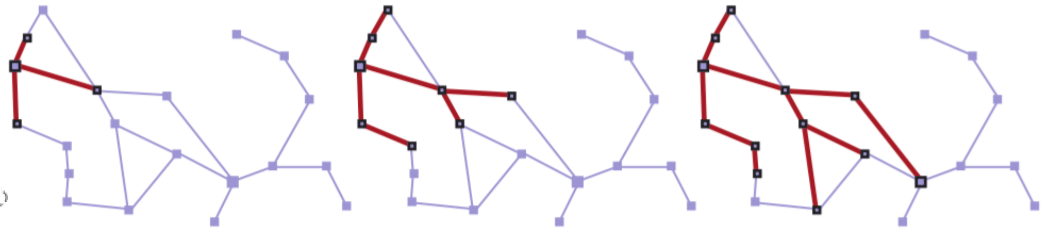
» Basic idea:

- offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

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

Search Algorithms: Examples





Search Algorithms: Grids

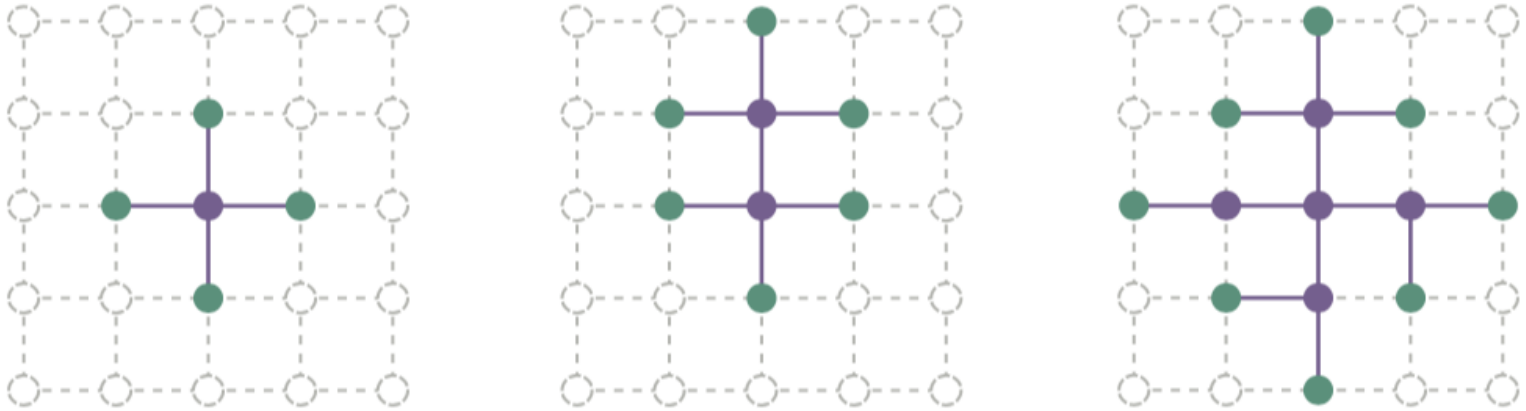


Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

Infrastructure for Search Algorithms

- For each node *n* of the tree, we have a structure that contains four components: *n*.STATE, *n*.PARENT, *n*.ACTION, and *n*.PATH-COST
- The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action),
        PARENT = parent, ACTION = action,
        PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Infrastructure for Search Algorithms

» The appropriate data structure for this is a queue. The operations on a queue are as follows:
  - EMPTY?(queue) returns true only if there are no more elements in the queue.
  - POP(queue) removes the first element of the queue and returns it.
  - INSERT(element, queue) inserts an element and returns the resulting queue.

Infrastructure for Search Algorithms

» Three common variants of queues are:

- the first-in, first-out or FIFO queue, which pops the *oldest* element of the queue;
- the last-in, first-out or LIFO queue (also known as a stack), which pops the *newest* element of the queue; and
- the priority queue, which pops the element of the queue with the highest priority according to some ordering function.

Search Strategies

- A strategy is defined by picking the order of node expansion.
- Strategies are evaluated along the following dimensions:
  - Completeness: does it always find a solution if one exists?
  - Time complexity: number of nodes generated/expanded
  - Space complexity: maximum number of nodes in memory
  - Optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be ∞)

Tree Sesrch

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

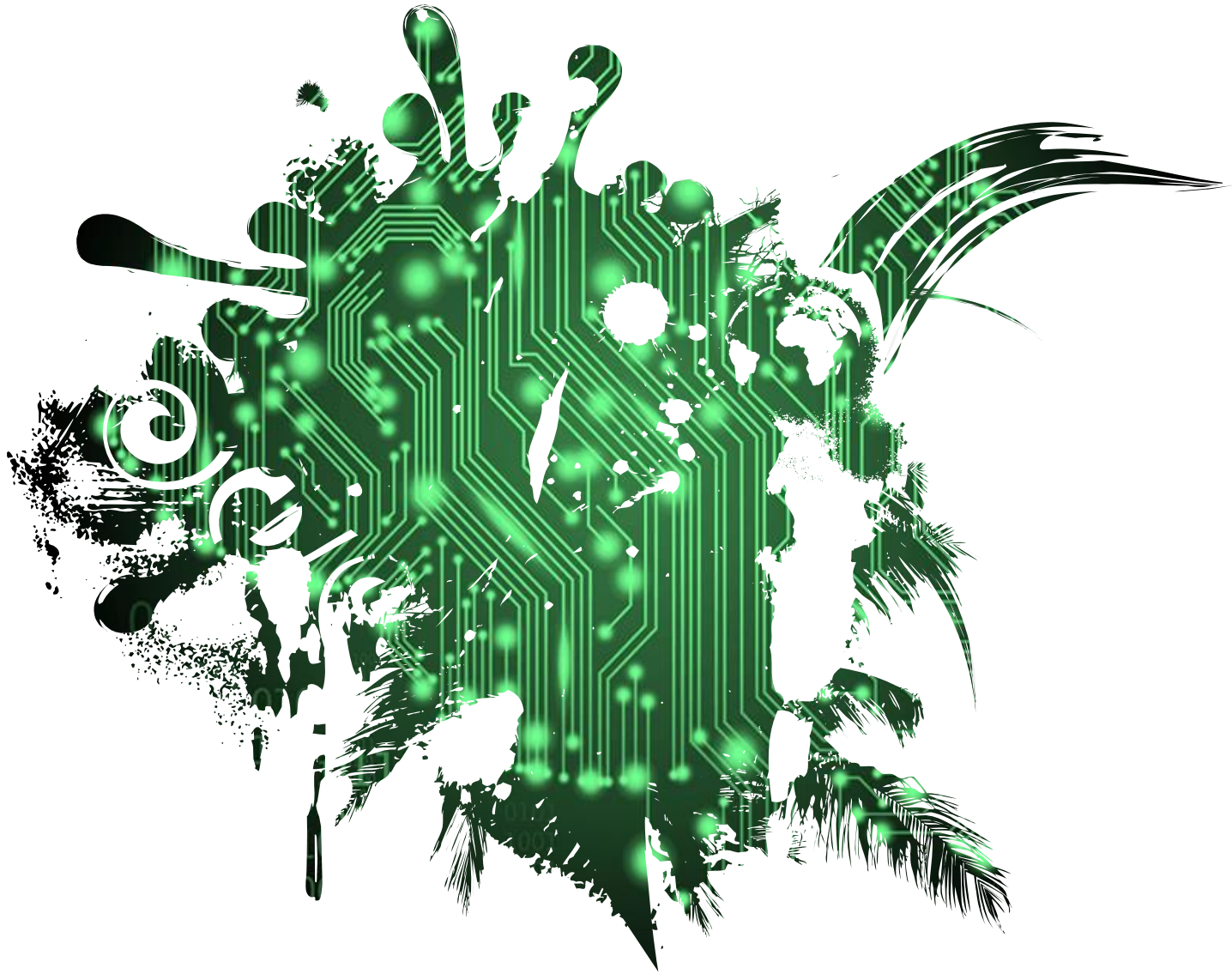
Uninformed Search Strategies

- Uninformed strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search

- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-First Search



Breadth-First Search

» Expand shallowest unexpanded node

» Implementation:

- *frontier* is a FIFO queue

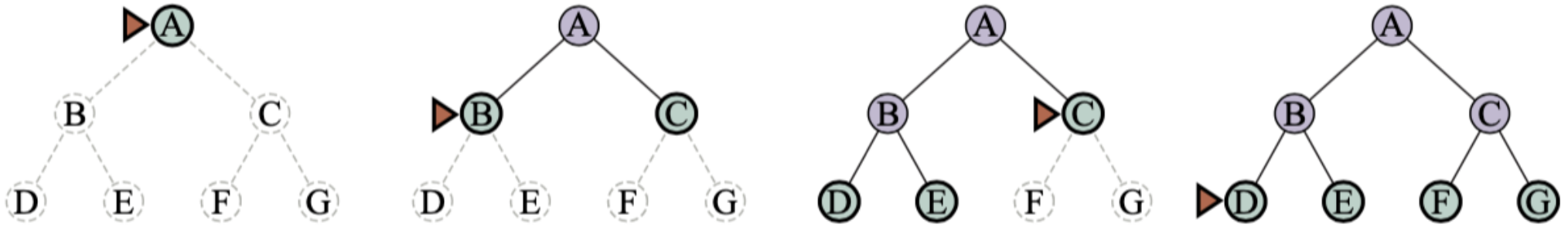


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Breadth-First Search

» **Complete:** Yes (if b is finite).

» **Time:** $1 + b + b^2 + \cdots + b^d = O(b^d)$.

» **Space:** $O(b^{d+1}) = O(b^d)$.

» **Optimal:** Yes, (if step cost = 1). In general, not optimal.

Best-First Search Algorithm

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node ← NODE(STATE=problem.INITIAL)
    frontier ← a priority queue ordered by f, with node as an element
    reached ← a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s] ← child
                add child to frontier
    return failure

function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Uniform-Cost Search

» Expand least-cost unexpanded node

» Implementation:

- *frontier* = queue ordered by path cost, lowest first

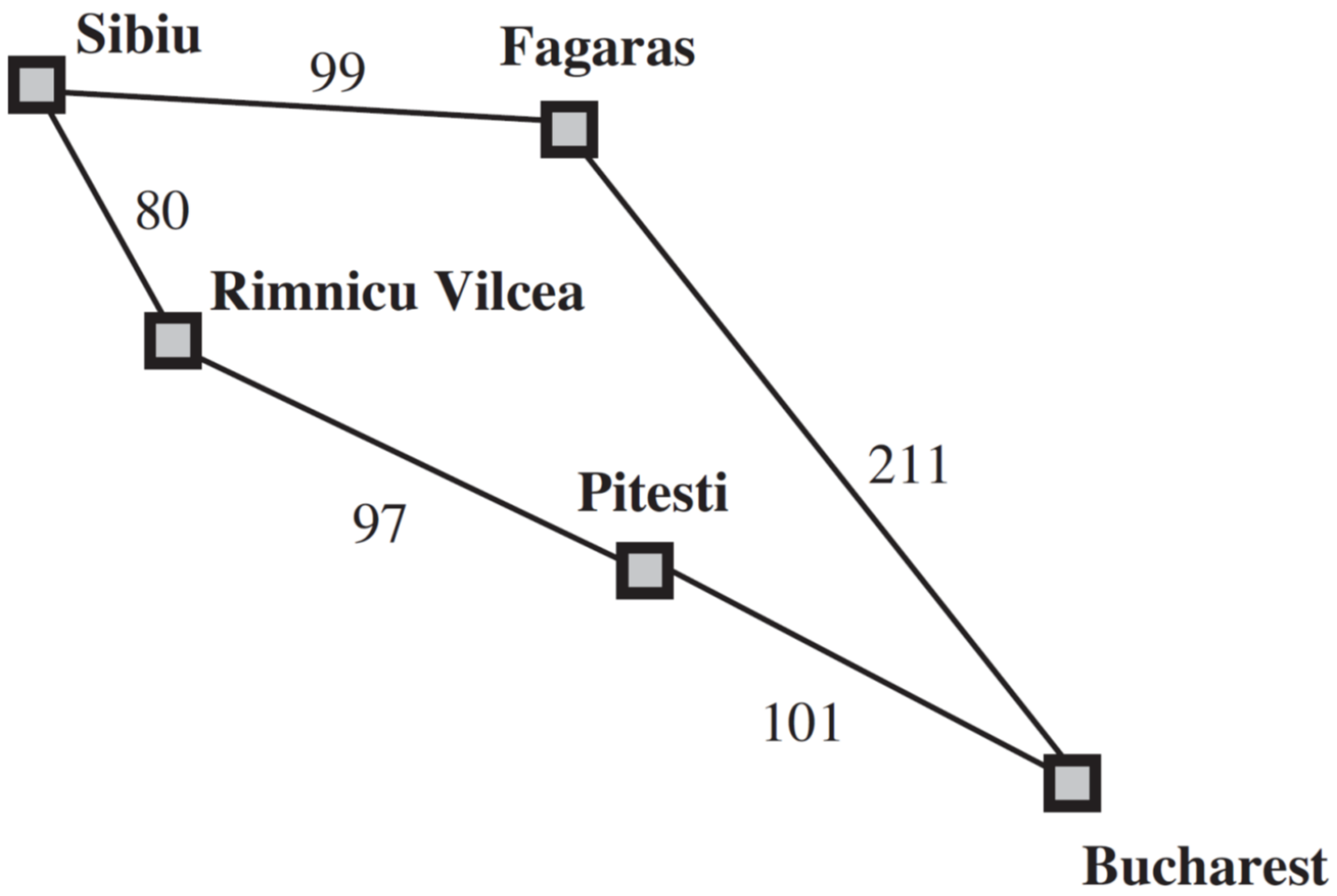
» Equivalent to breadth-first if step costs all equal

BFS & UCS Algorithms

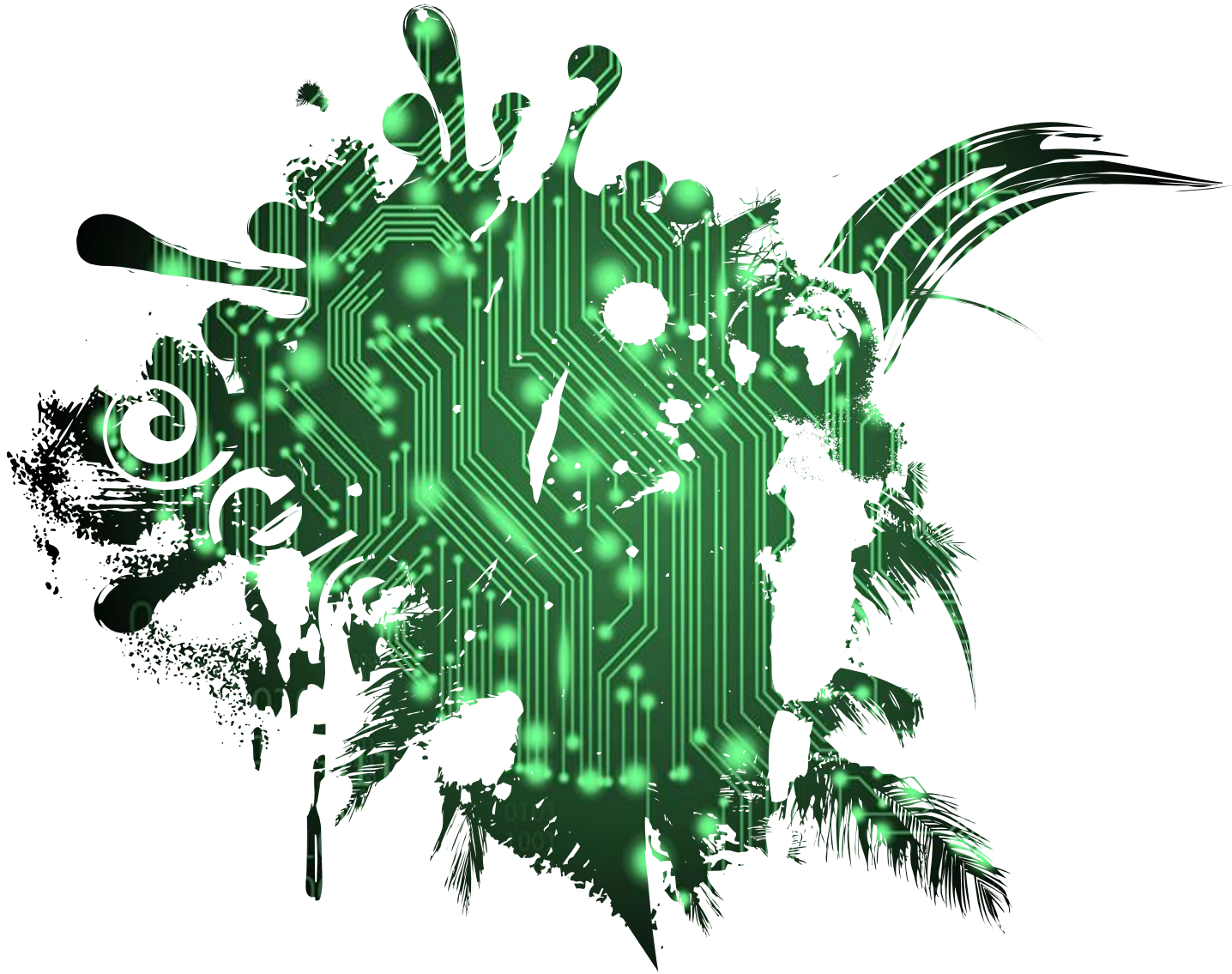
```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

UCS Example

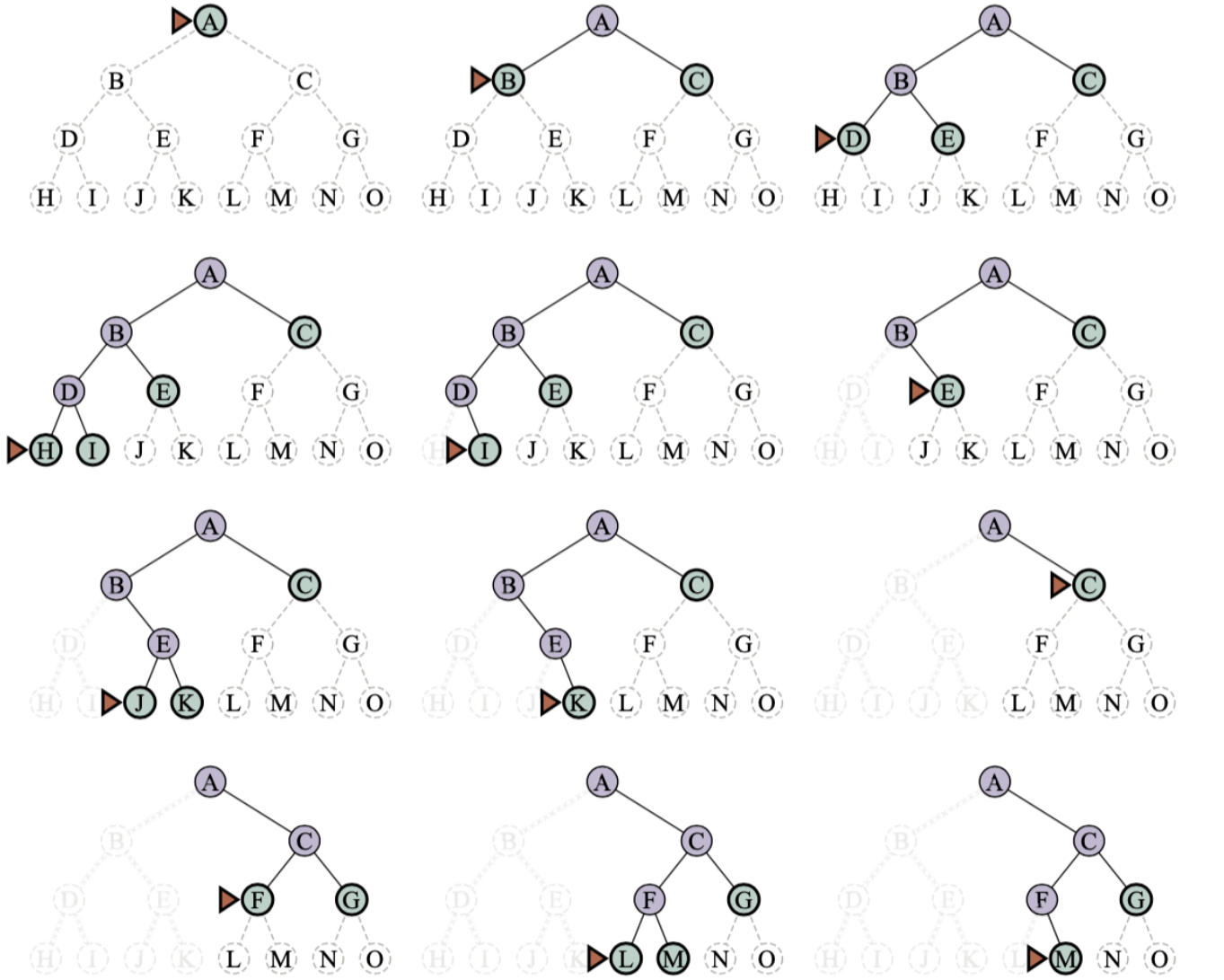

Sibiu
99
Fagaras
80
Rimnicu Vilcea
211
Pitesti
97
101
Bucharest

Depth-First Search



Depth-First Search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* is a LIFO queue



Depth-First Search

» **Complete:** No (Faills in infinite depth sapce or).

» **Time:** $O(b^m)$, terrible if m is much greater than d.

» **Space:** $O(bm)$, linear space

» **Optimal:** No.

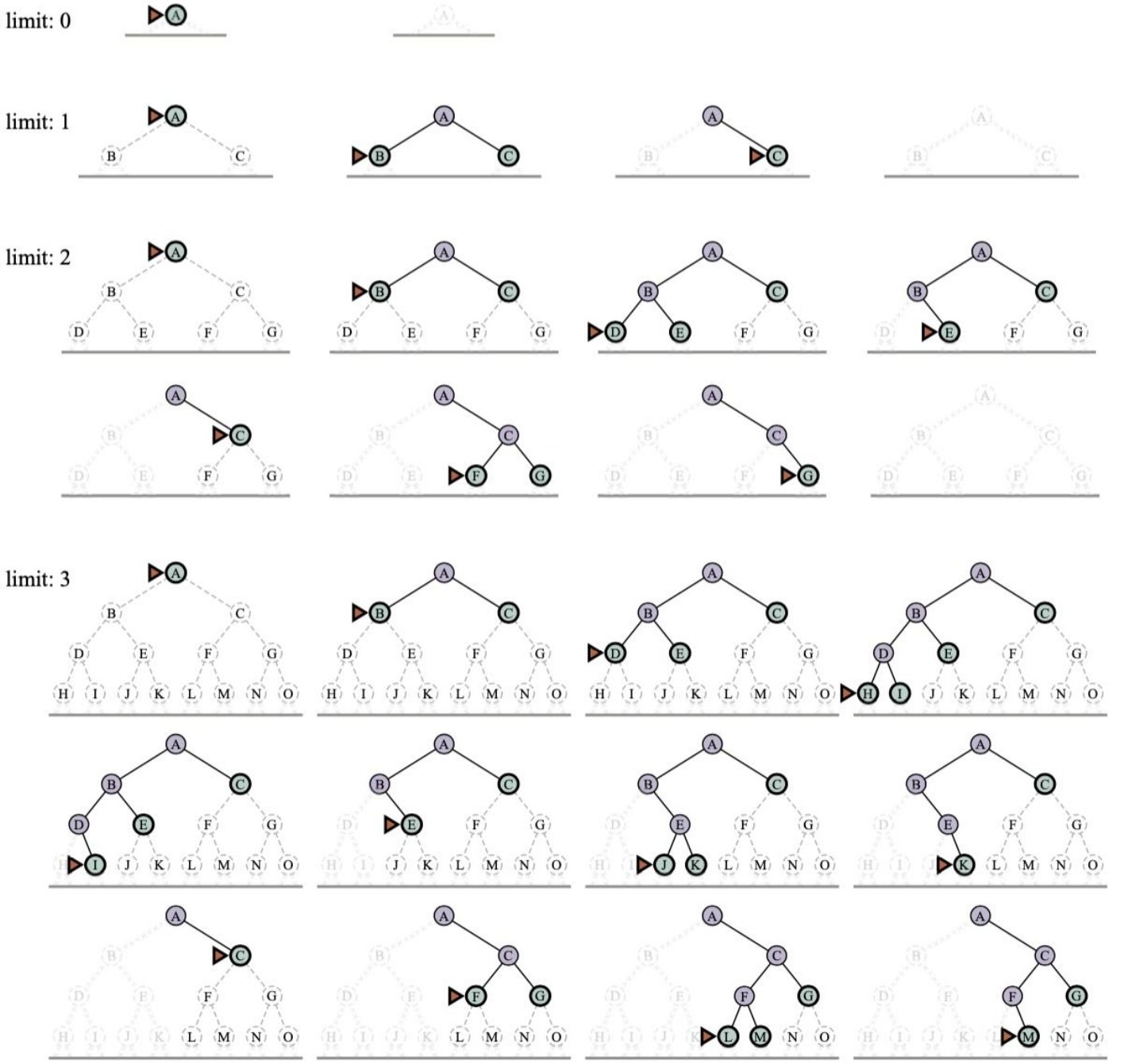
Depth-Limited Search & Iterative Deepening Search

- Depth-Limited Search = depth-first search with depth limit l,
- i.e., nodes at depth l have no successors.
- Iterative Deepening Search is successive implementation of Depth-Limited Search with incremental l.

DLS & IDS Algorithms

```
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

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

IDS Example



Conclusion

- Problem formulation usually requires abstracting away real-world details.
- Once problem is formulated in abstract form, complexity analysis helps us picking out best algorithm.
- Variety of uninformed search strategies; difference lies in method used to pick node that will be further expanded.
- Iterative deepening search only uses linear space and not much more time than other uniformed search strategies.

Questions & Comments

Abdel-Rahman Hedar

hedar@au.edu.eg

https://www.aun.edu.eg/fci/