# Introduction to Artificial Intelligence

## Problem Solving and Search

**Bernhard Beckert**

**UNIVERSITÄT KOBLENZ-LANDAU**

**Summer Term 2003**

# Outline

- Problem solving
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

# Problem solving

## Offline problem solving

**Acting only with complete knowledge of problem and solution**

## Online problem solving

**Acting without complete knowledge**

## Here

**Here we are concerned with offline problem solving only**

# Example: Travelling in Romania

## Scenario

**On holiday in Romania; currently in Arad**
**Flight leaves tomorrow from Bucharest**

## Goal

**Be in Bucharest**

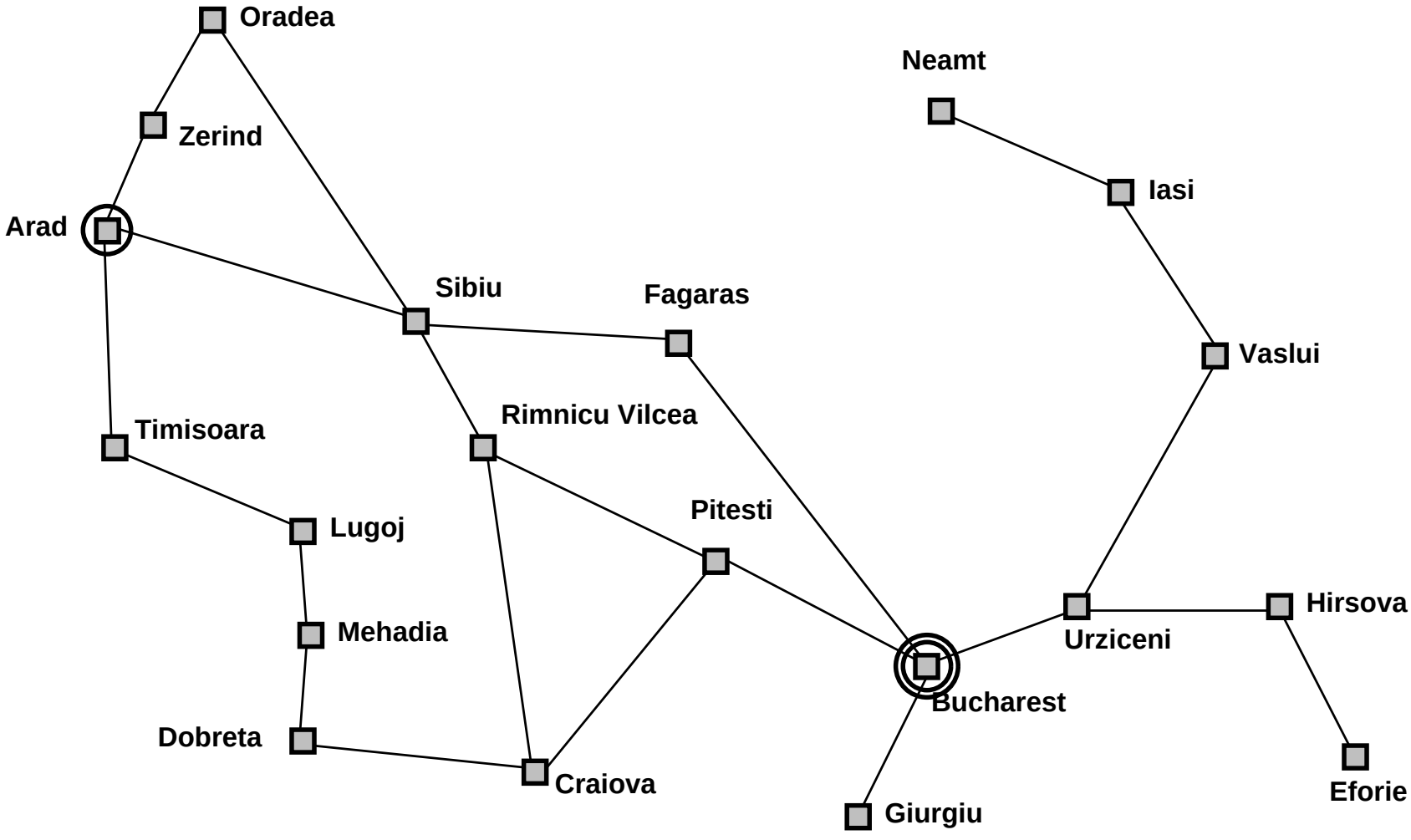
## Formulate problem

***States*: various cities**
***Actions*: drive between cities**

## Solution

**Appropriate sequence of cities**
**e.g.: Arad, Sibiu, Fagaras, Bucharest**

# Example: Travelling in Romania



# Problem types

## Single-state problem

- observable (at least the initial state)
- deterministic
- static
- discrete

## Multiple-state problem

- partially observable (initial state not observable)
- deterministic
- static
- discrete

## Contingency problem

- partially observable (initial state not observable)
- non-deterministic

# Example: vacuum-cleaner world

**Single-state**

**Start in:** $5$

**Solution:** [***right***, ***suck***]

**Multiple-state**

**Start in:** $\{1, 2, 3, 4, 5, 6, 7, 8\}$

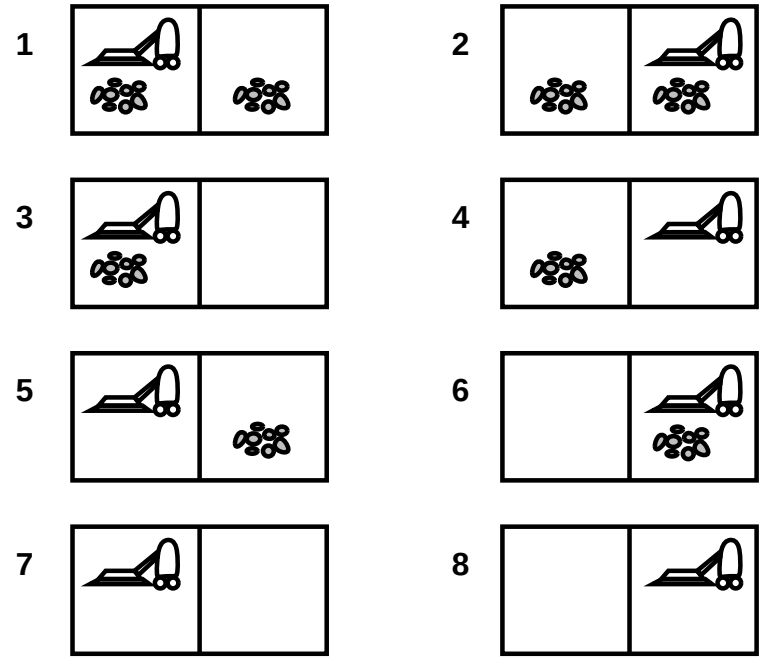
**Solution:** [***right***, ***suck***, ***left***, ***suck***]

***right*** $\rightarrow \{2, 4, 6, 8\}$

***suck*** $\rightarrow \{4, 8\}$

***left*** $\rightarrow \{3, 7\}$

***suck*** $\rightarrow \{7\}$



# Example: vacuum-cleaner world

**Contingency**

**Murphy's Law:**
***suck*** **can dirty a clean carpet**

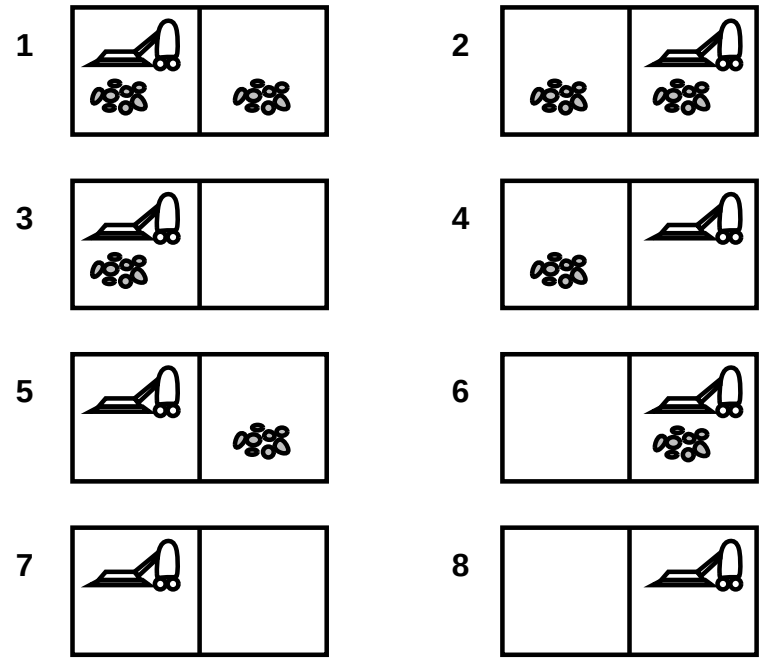
**Local sensing:**
***dirty/not dirty*** **at location only**

**Start in:** $\{1, 3\}$

**Solution:** **[*suck*, *right*, *suck*]**

$$\textit{suck} \rightarrow \{5, 7\}$$
$$\textit{right} \rightarrow \{6, 8\}$$
$$\textit{suck} \rightarrow \{6, 8\}$$

**Improvement:** **[*suck*, *right*, if *dirt* then *suck*]**
**(decide whether in $6$ or $8$ using local sensing)**



# Single-state problem formulation

**Defined by the following four items**

1. **Initial state**

   **Example:** ***Arad***

2. **Successor function** $S$

   **Example:** $S(\textbf{\textit{Arad}}) = \{ \langle \textbf{\textit{goZerind}}, \textbf{\textit{Zerind}} \rangle, \langle \textbf{\textit{goSibiu}}, \textbf{\textit{Sibiu}} \rangle, \ldots \}$

3. **Goal test**

   **Example:** $x = \textbf{\textit{Bucharest}}$ **(explicit test)**

   $\textbf{\textit{noDirt}}(x)$ **(implicit test)**

4. **Path cost** **(optional)**

   **Example:** **sum of distances, number of operators executed, etc.**

# Single-state problem formulation

## Solution

**A sequence of operators**
**leading from the initial state to a goal state**

# Selecting a state space

## Abstraction

**Real world is absurdly complex**

**State space must be abstracted for problem solving**

## (Abstract) state

**Set of real states**

## (Abstract) operator

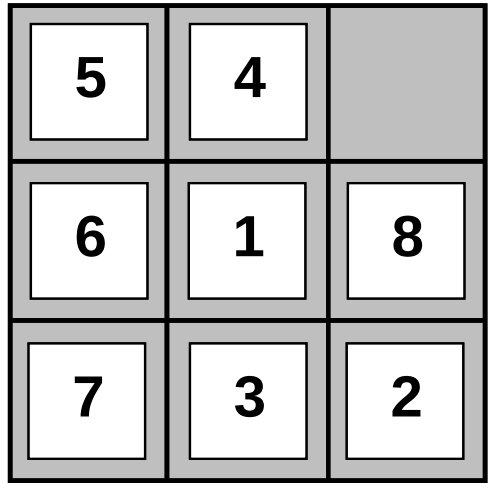
**Complex combination of real actions**

**Example: *Arad* $\longrightarrow$ *Zerind* represents complex set of possible routes**

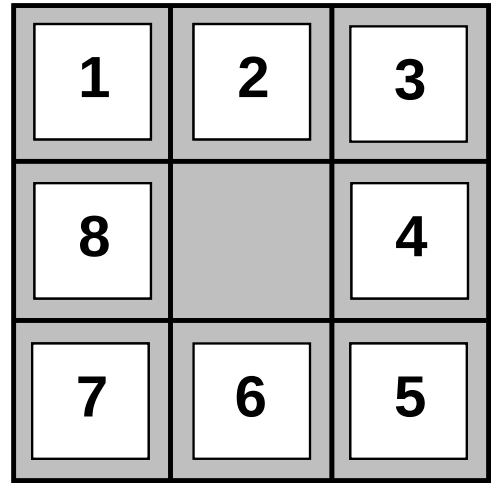
## (Abstract) solution

**Set of real paths that are solutions in the real world**

# Example: The 8-puzzle




Start State




Goal State

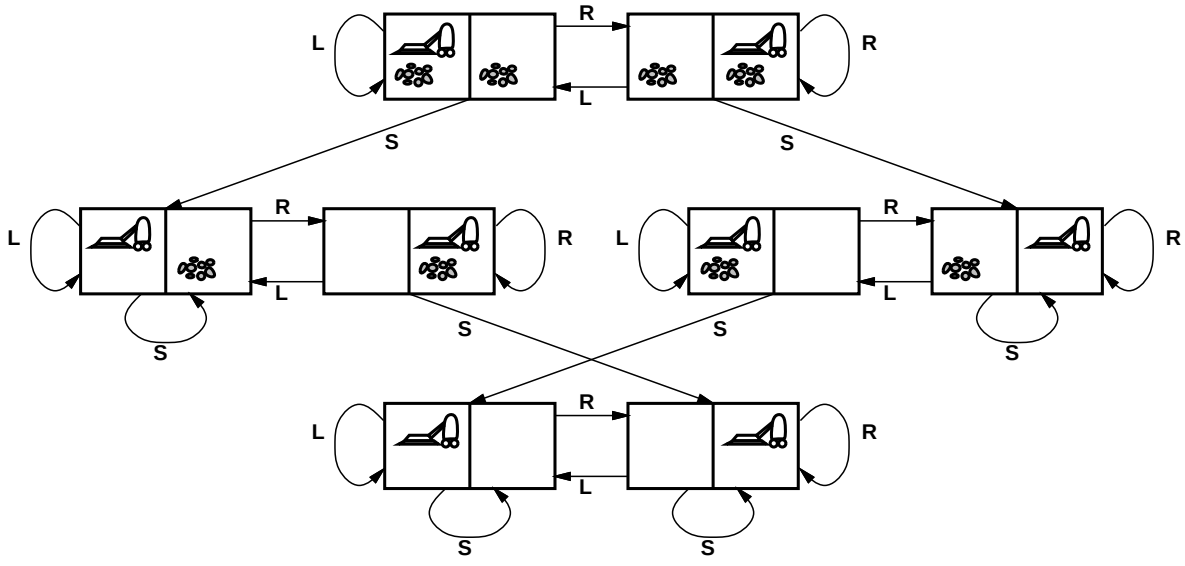
**States** integer locations of tiles

**Actions** *left*, *right*, *up*, *down*

**Goal test** = goal state?

**Path cost** 1 per move

# Example: Vacuum-cleaner



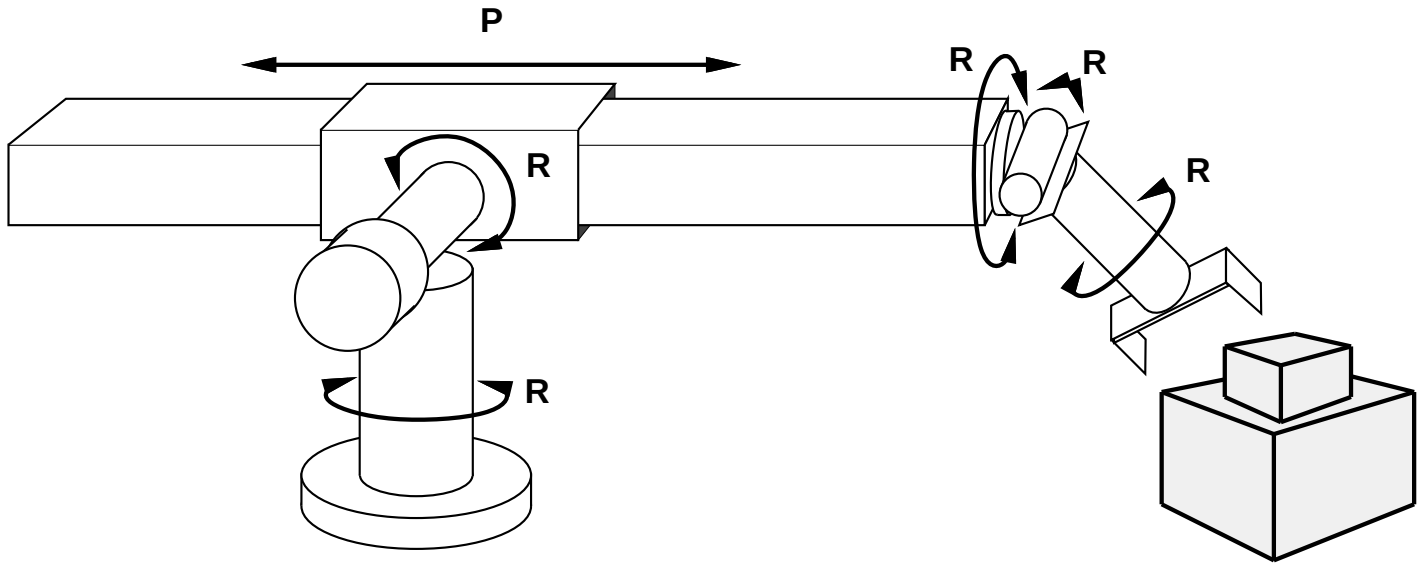

**States** integer dirt and robot locations

**Actions** *left*, *right*, *suck*, *noOp*

**Goal test** *not dirty?*

**Path cost** 1 per operation (0 for *noOp*)

# Example: Robotic assembly




**States** real-valued coordinates of

robot joint angles and parts of the object to be assembled

**Actions** continuous motions of robot joints

**Goal test** assembly complete?

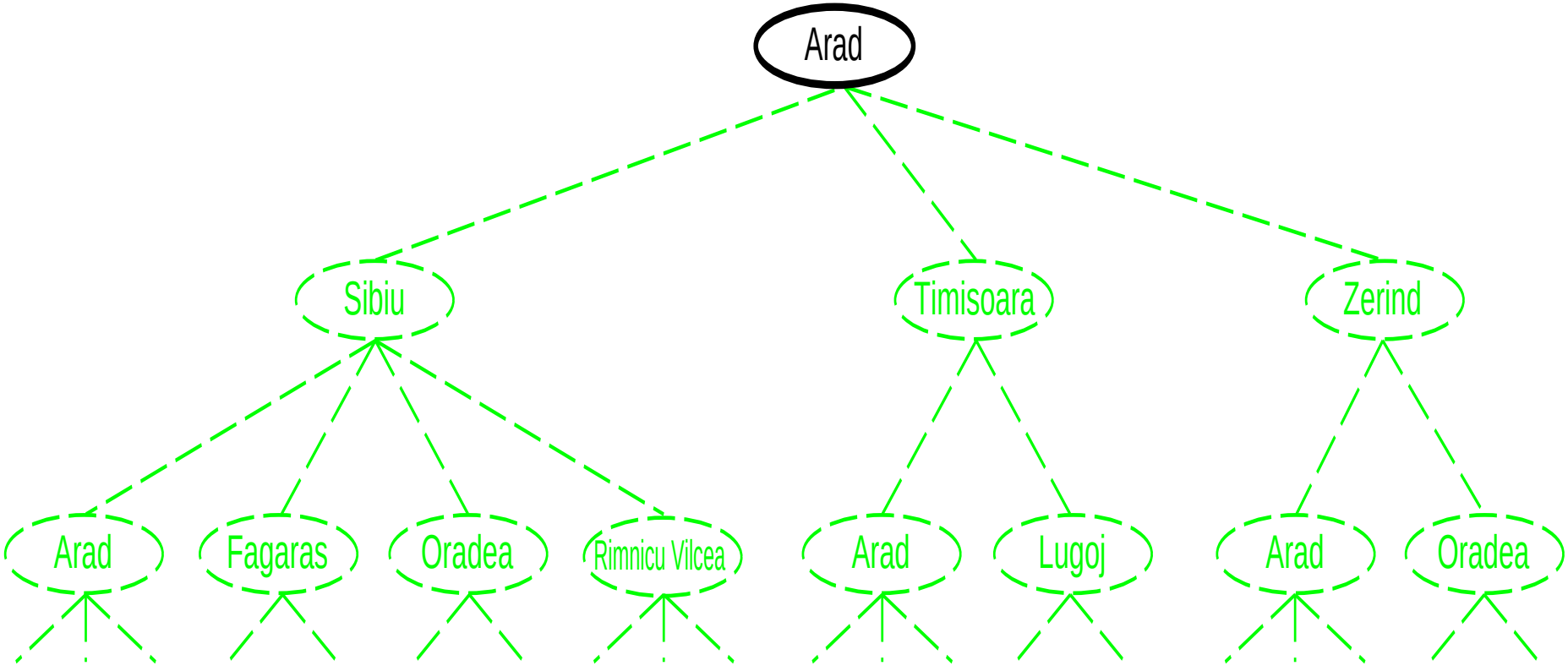
**Path cost** time to execute

# Tree search algorithms

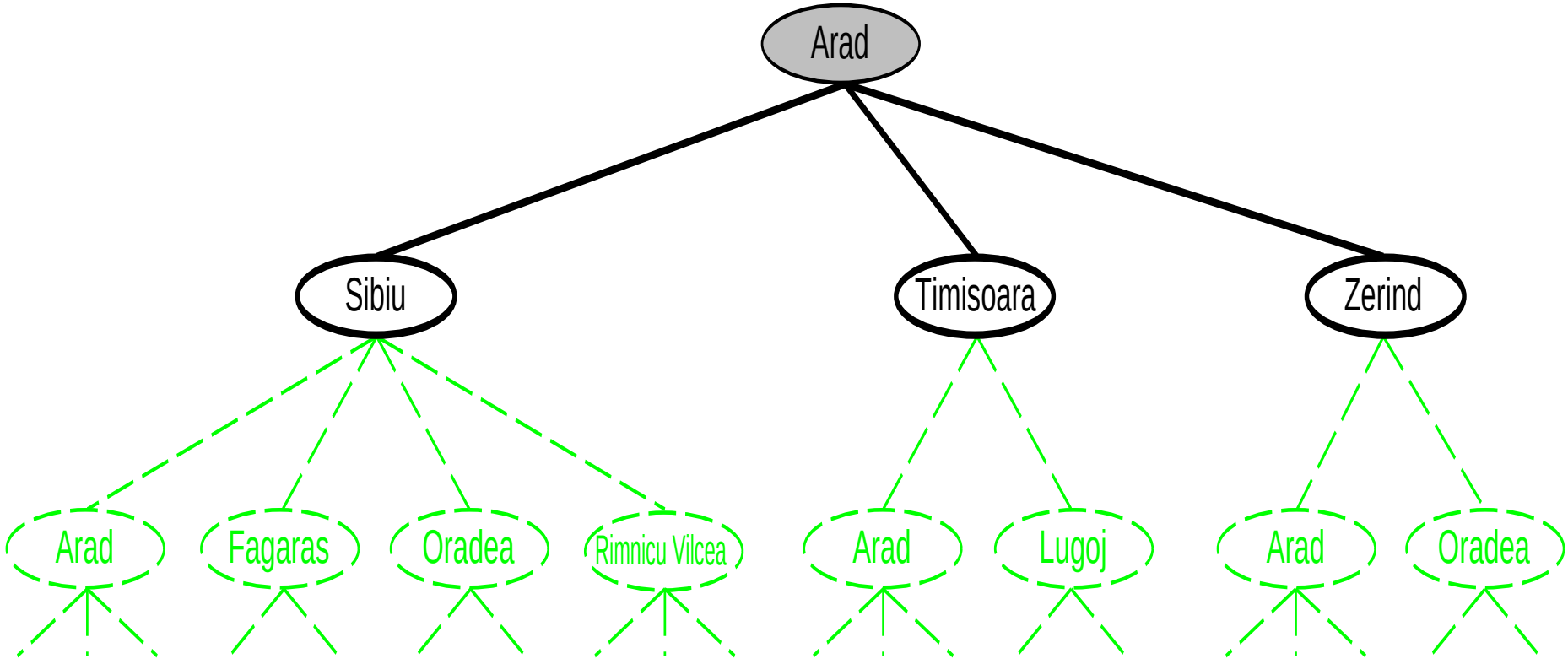
- **Offline**
- **Simulated exploration of state space in a search tree by generating successors of already-explored states**

```
function TREE-SEARCH(problem, strategy) returns a solution or failure
    initialize the search tree using the initial state of 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 the resulting nodes to the search tree
    end
```

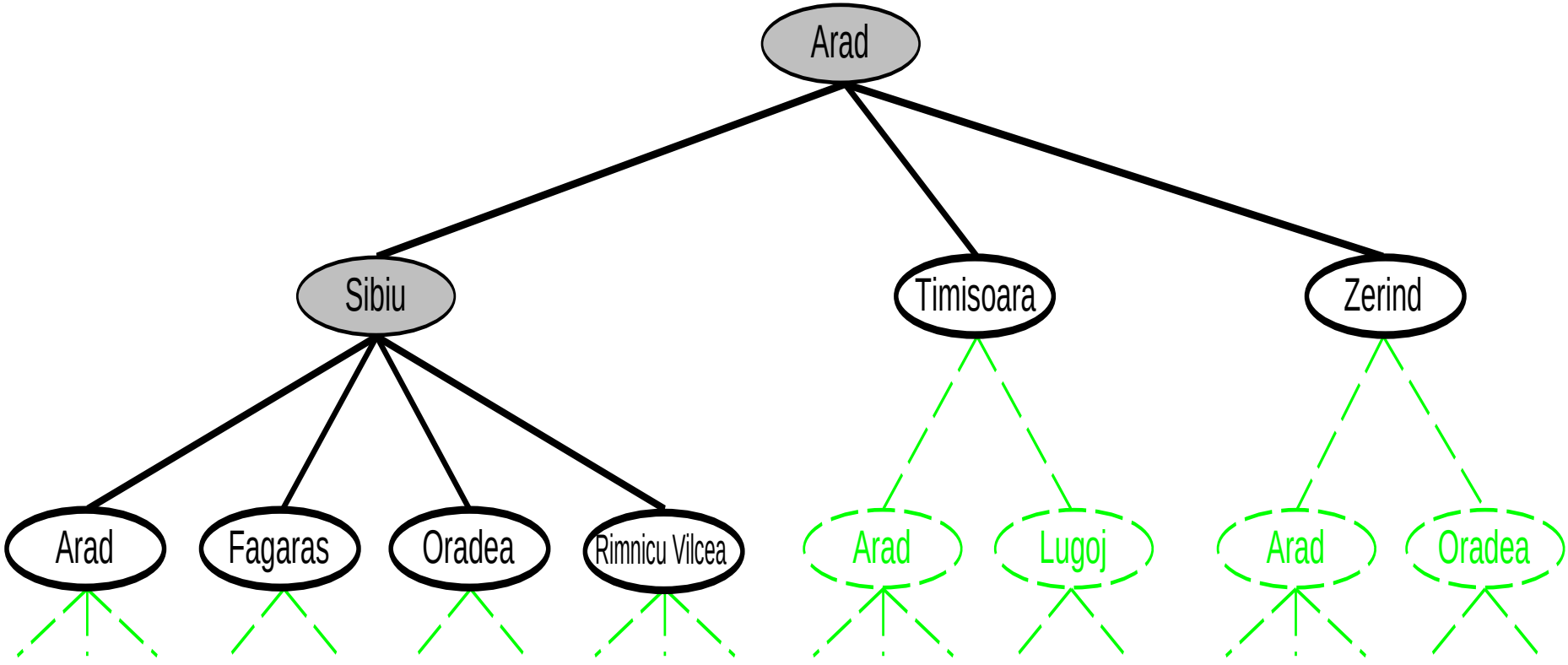
# Tree search: Example



# Tree search: Example



# Tree search: Example



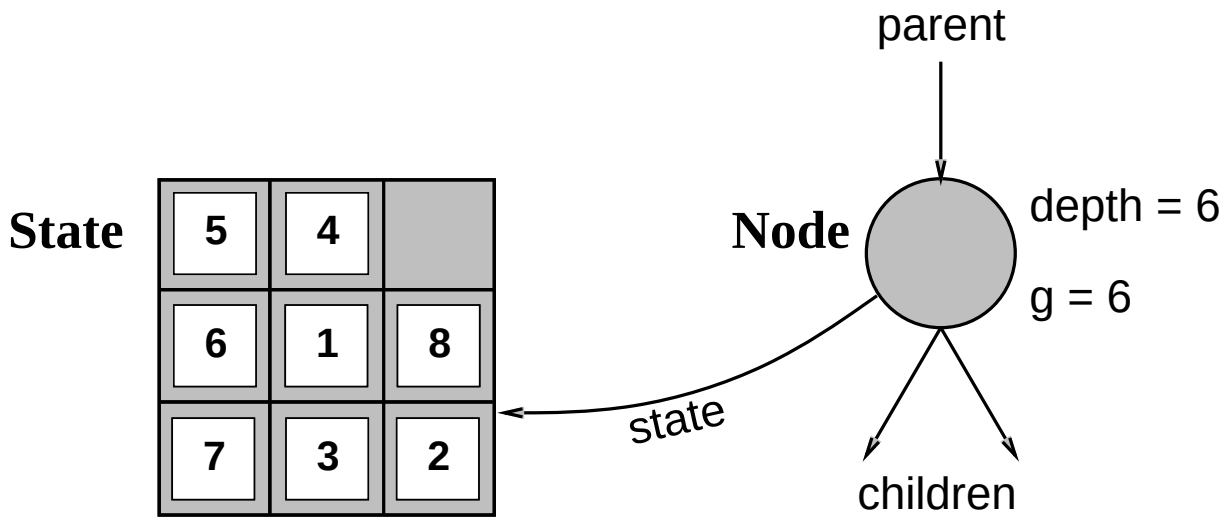
# Implementation: States vs. nodes

## State

**A (representation of) a physical configuration**

## Node

**A data structure constituting part of a search tree**
**(includes *parent*, *children*, *depth*, *path cost*, etc.)**



# Implementation of search algorithms

```
function TREE-SEARCH(problem, fringe) returns a solution or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]),fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then
      return node
    else
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
  end
```

| | |
|---|---|
| **fringe** | **queue of nodes not yet considered** |
| **State** | **gives the state that is represented by *node*** |
| **Expand** | **creates new nodes by applying possible actions to *node*** |

# Search strategies

## Strategy

**Defines the order of node expansion**

## Important properties of strategies

| | |
|---|---|
| **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 measured in terms of

| | |
|---|---|
| $b$ | **maximum branching factor of the search tree** |
| $d$ | **depth of a solution with minimal distance to root** |
| $m$ | **maximum depth of the state space (may be $\infty$)** |

# Uninformed search strategies

## Uninformed search

**Use only the information available in the problem definition**

## Frequently used strategies

- **Breadth-first search**
- **Uniform-cost search**
- **Depth-first search**
- **Depth-limited search**
- **Iterative deepening search**

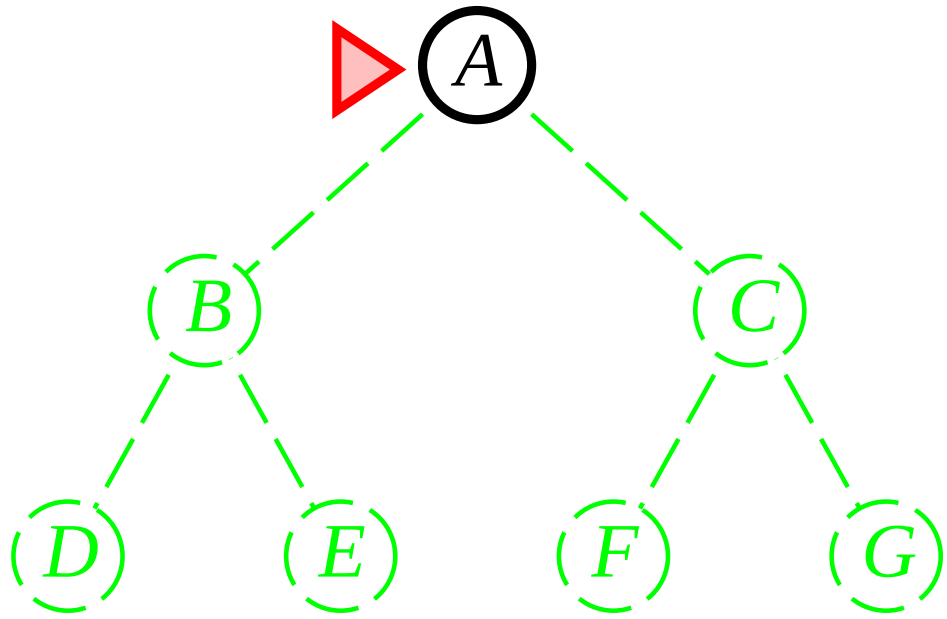
# Breadth-first search

## Idea

**Expand shallowest unexpanded node**

## Implementation

***fringe* is a FIFO queue, i.e. successors go in at the end of the queue**



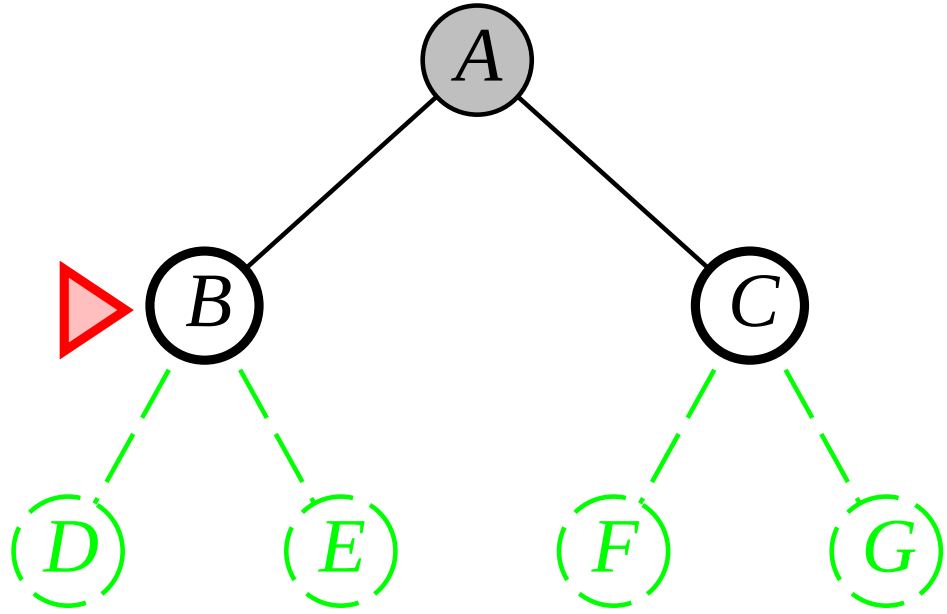
# Breadth-first search

## Idea

**Expand shallowest unexpanded node**

## Implementation

***fringe* is a FIFO queue, i.e. successors go in at the end of the queue**



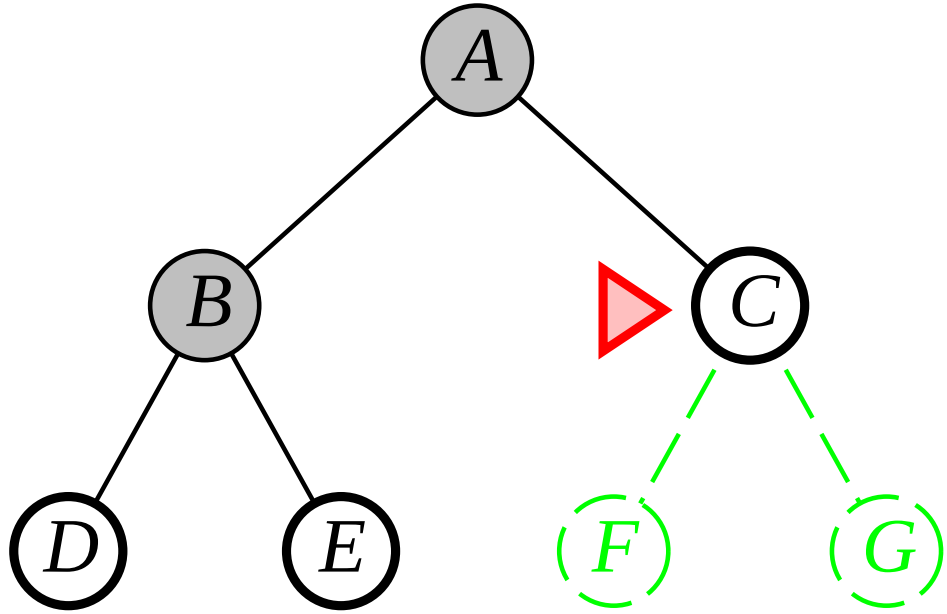
# Breadth-first search

## Idea

**Expand shallowest unexpanded node**

## Implementation

***fringe* is a FIFO queue, i.e. successors go in at the end of the queue**



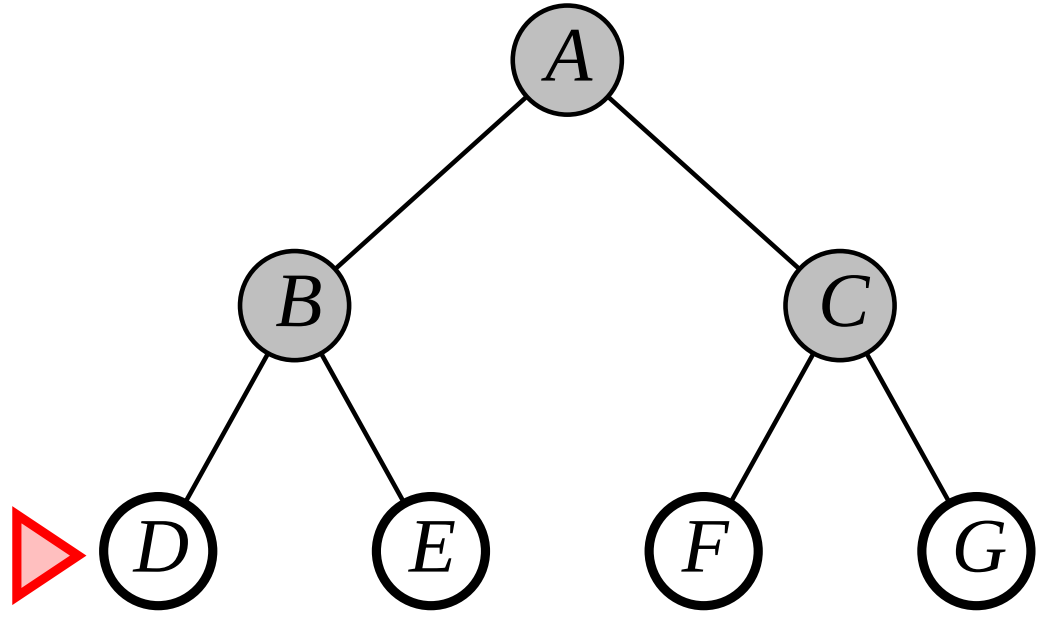
# Breadth-first search

## Idea

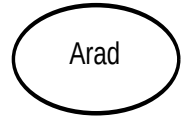
**Expand shallowest unexpanded node**

## Implementation

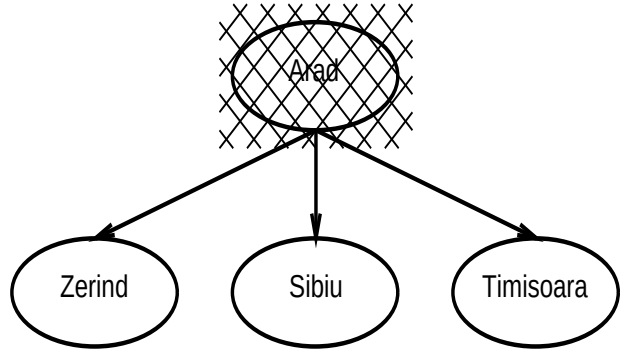
***fringe* is a FIFO queue, i.e. successors go in at the end of the queue**



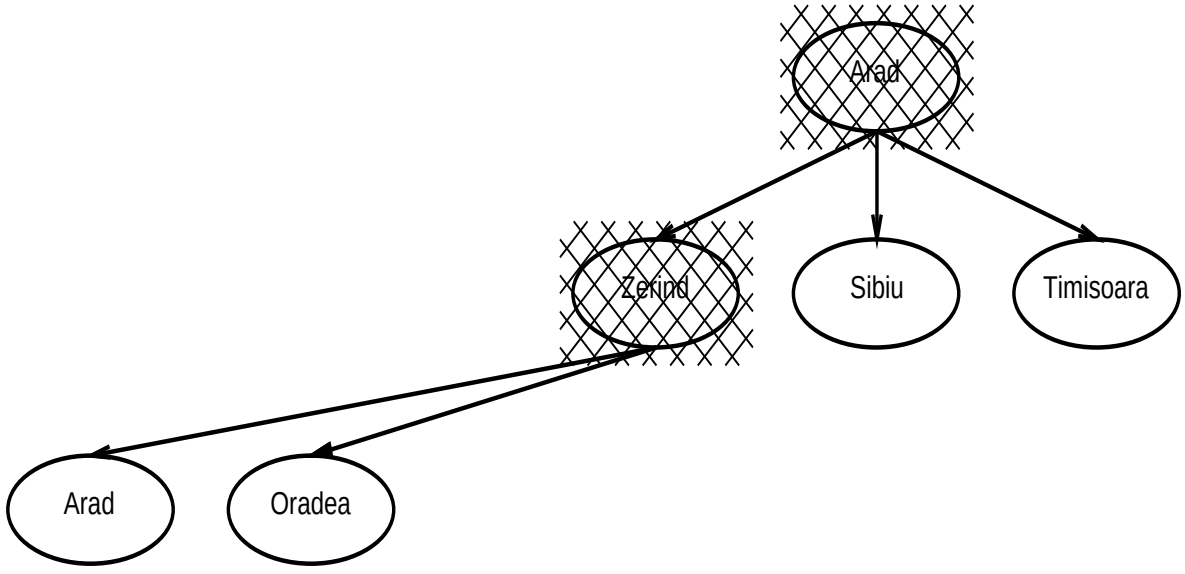
# Breadth-first search: Example Romania



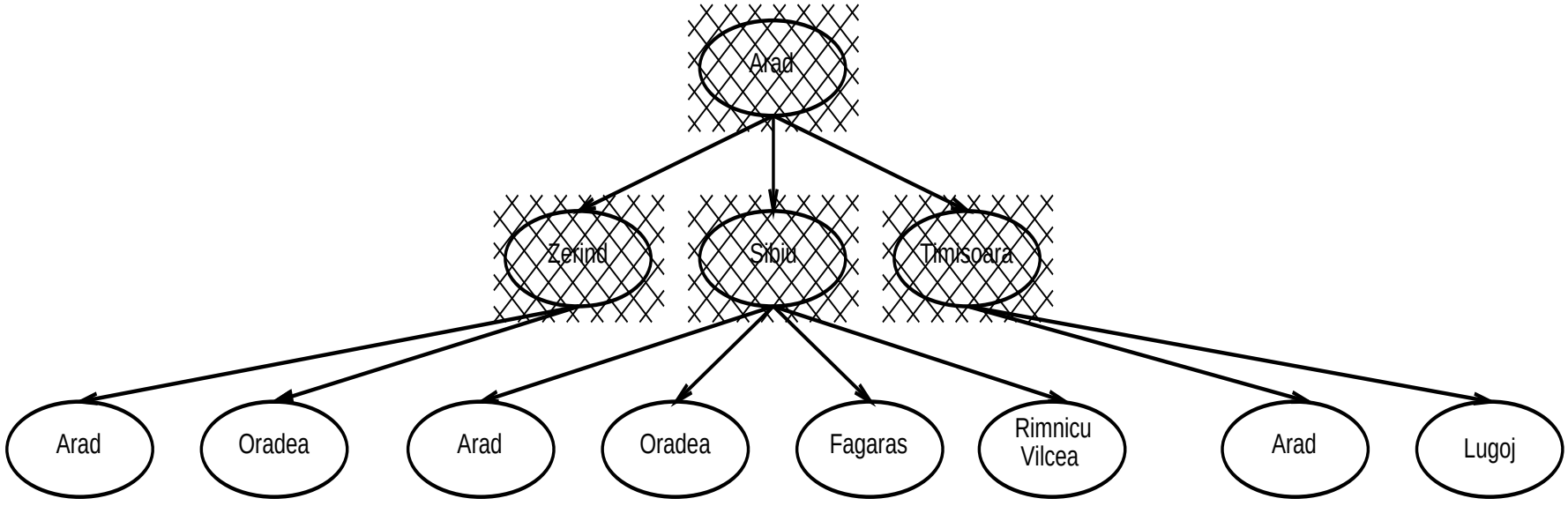
# Breadth-first search: Example Romania



# Breadth-first search: Example Romania



# Breadth-first search: Example Romania



# Breadth-first search: Properties

**Complete** **Yes (if $b$ is finite)**

**Time** $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) \in O(b^{d+1})$

**i.e. exponential in $d$**

**Space** $O(b^{d+1})$

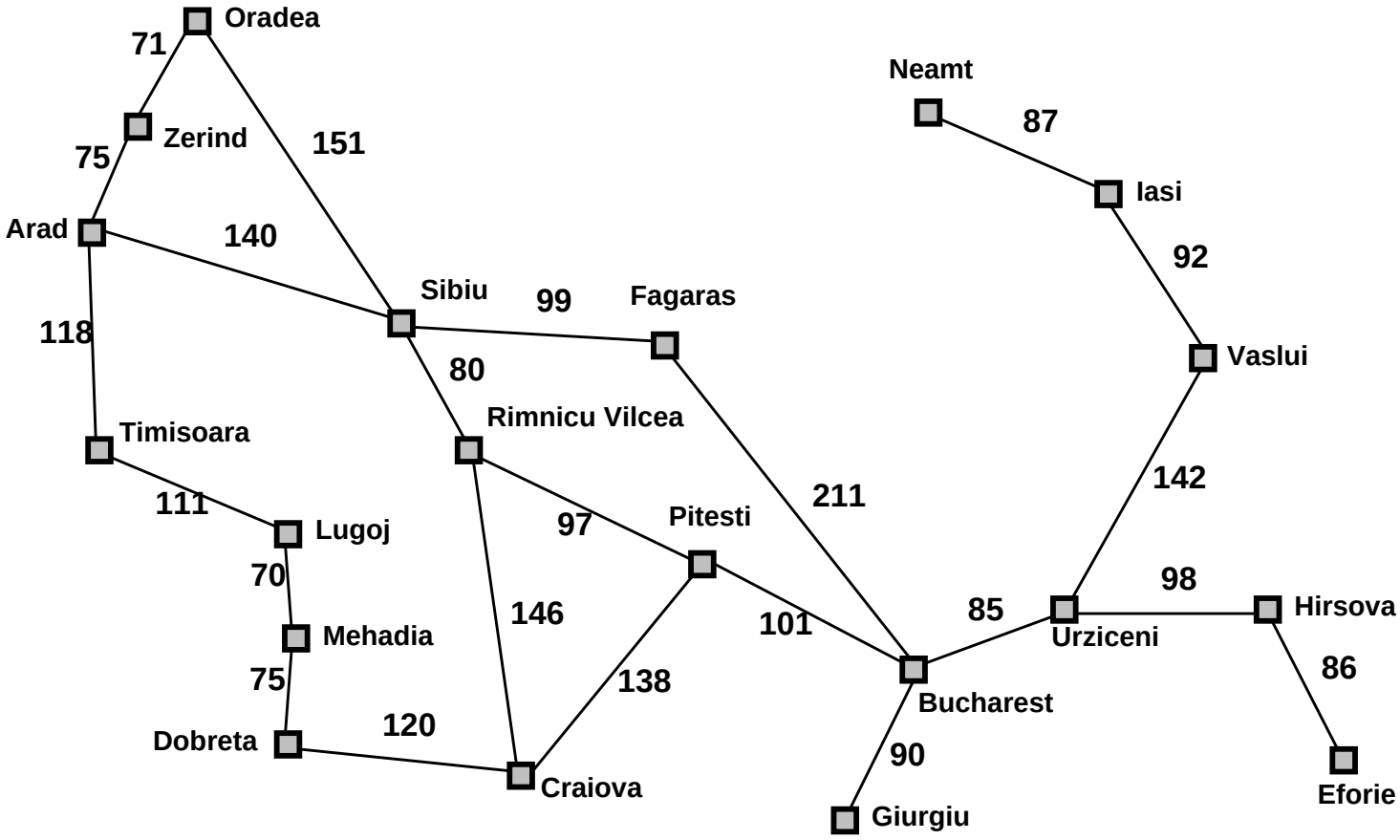
**keeps every node in memory**

**Optimal** **Yes (if cost = 1 per step), not optimal in general**

**Disadvantage**

**Space is the big problem**
**(can easily generate nodes at 5MB/sec so 24hrs = 430GB)**

# Romania with step costs in km




| Straight−line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 178 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 98 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Uniform-cost search

## Idea

**Expand least-cost unexpanded node**
**(costs added up over paths from root to leafs)**

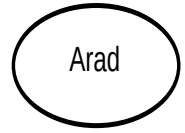
## Implementation

***fringe* is queue ordered by increasing path cost**

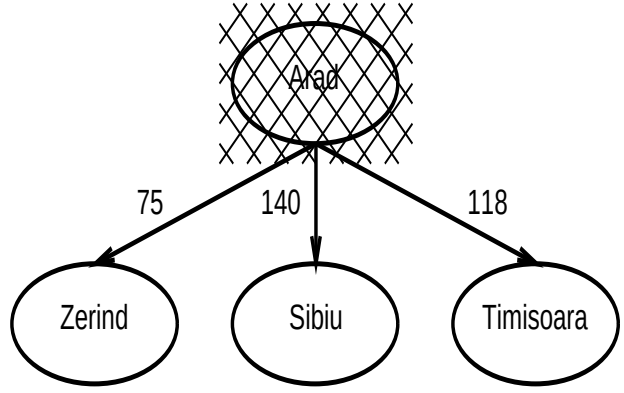
## Note

**Equivalent to depth-first search if all step costs are equal**

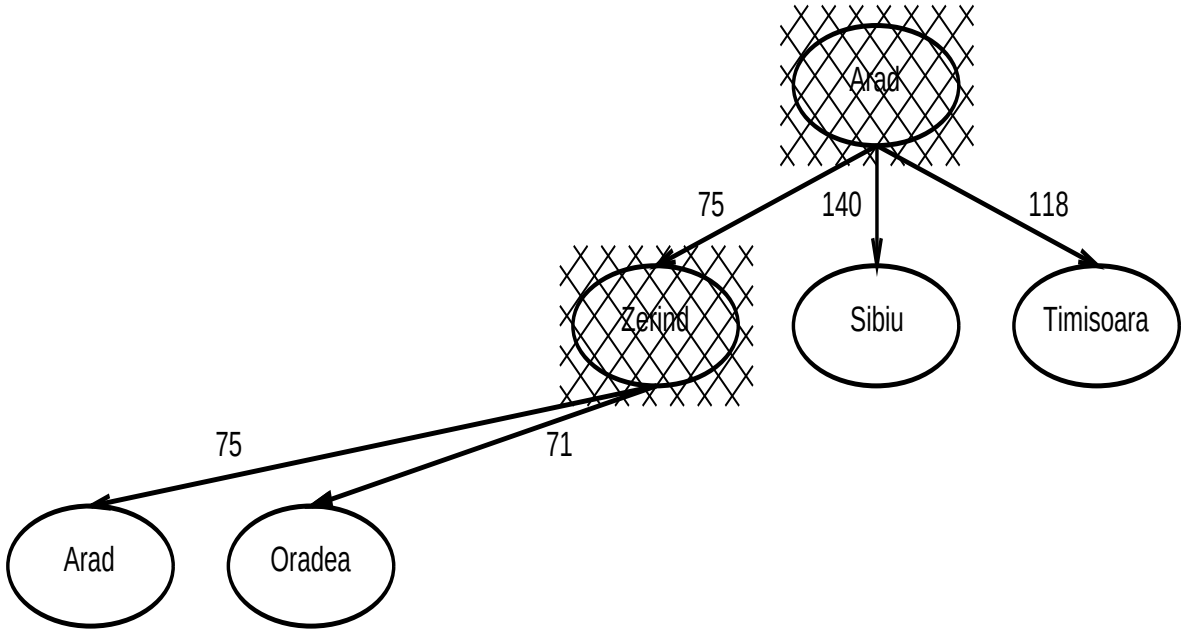
# Uniform-cost search



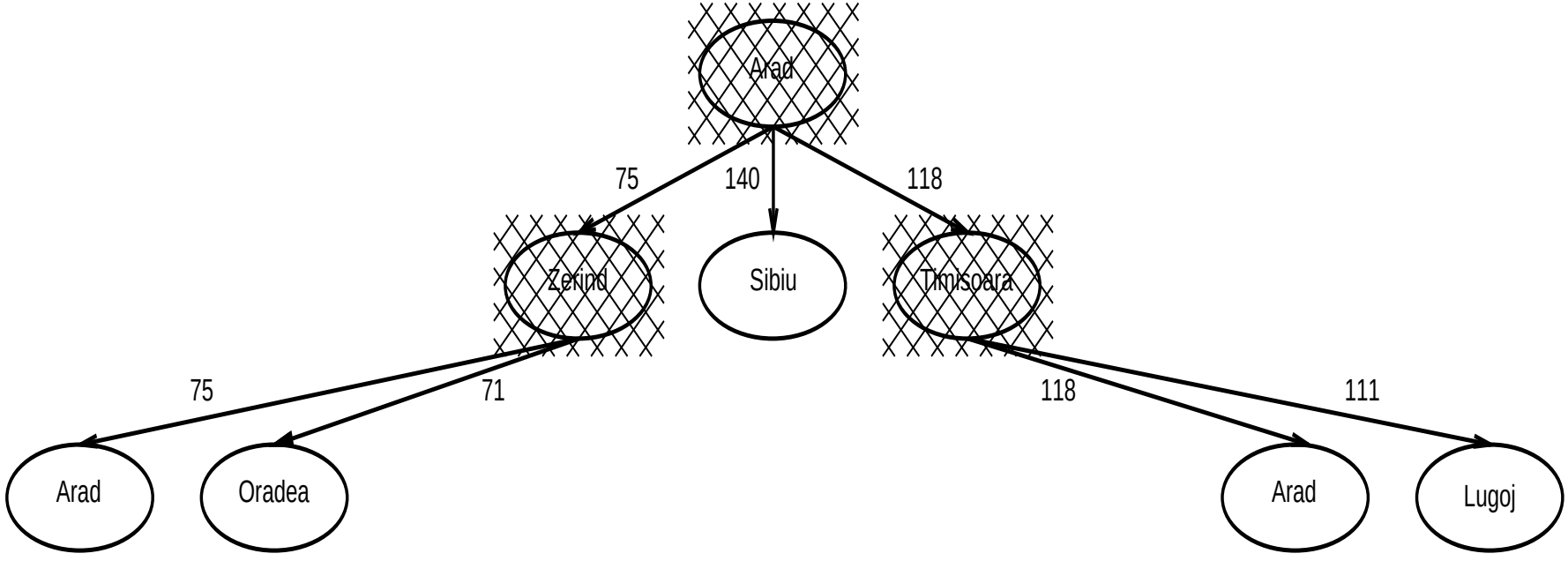
# Uniform-cost search



# Uniform-cost search



# Uniform-cost search



# Uniform-cost search: Properties

| | |
|---|---|
| **Complete** | **Yes (if step costs positive)** |
| **Time** | **# of nodes with past-cost less than that of optimal solution** |
| **Space** | **# of nodes with past-cost less than that of optimal solution** |
| **Optimal** | **Yes** |

# Depth-first search

## Idea

**Expand deepest unexpanded node**

## Implementation

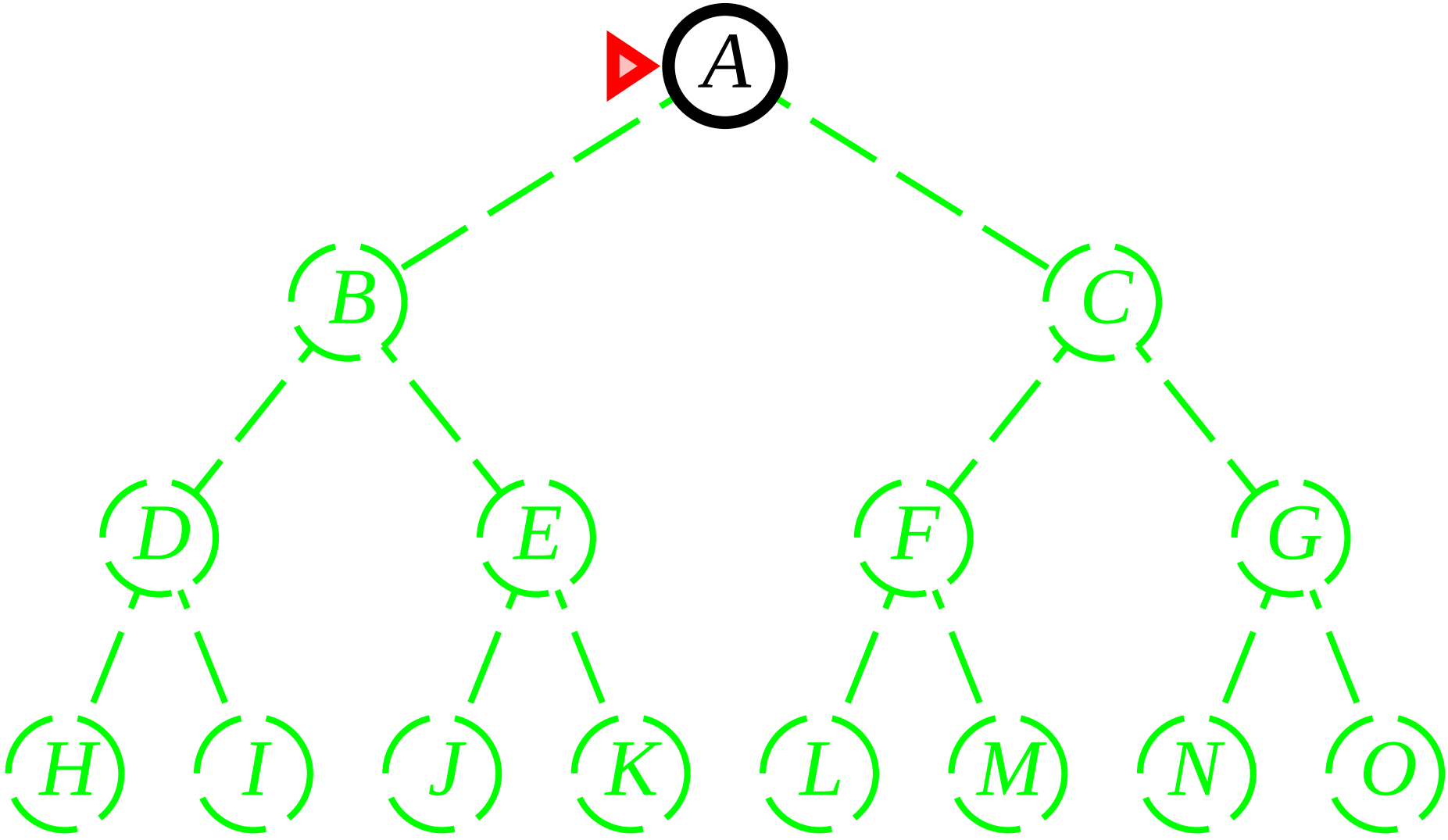
***fringe* is a LIFO queue (a stack), i.e. successors go in at front of queue**

## Note

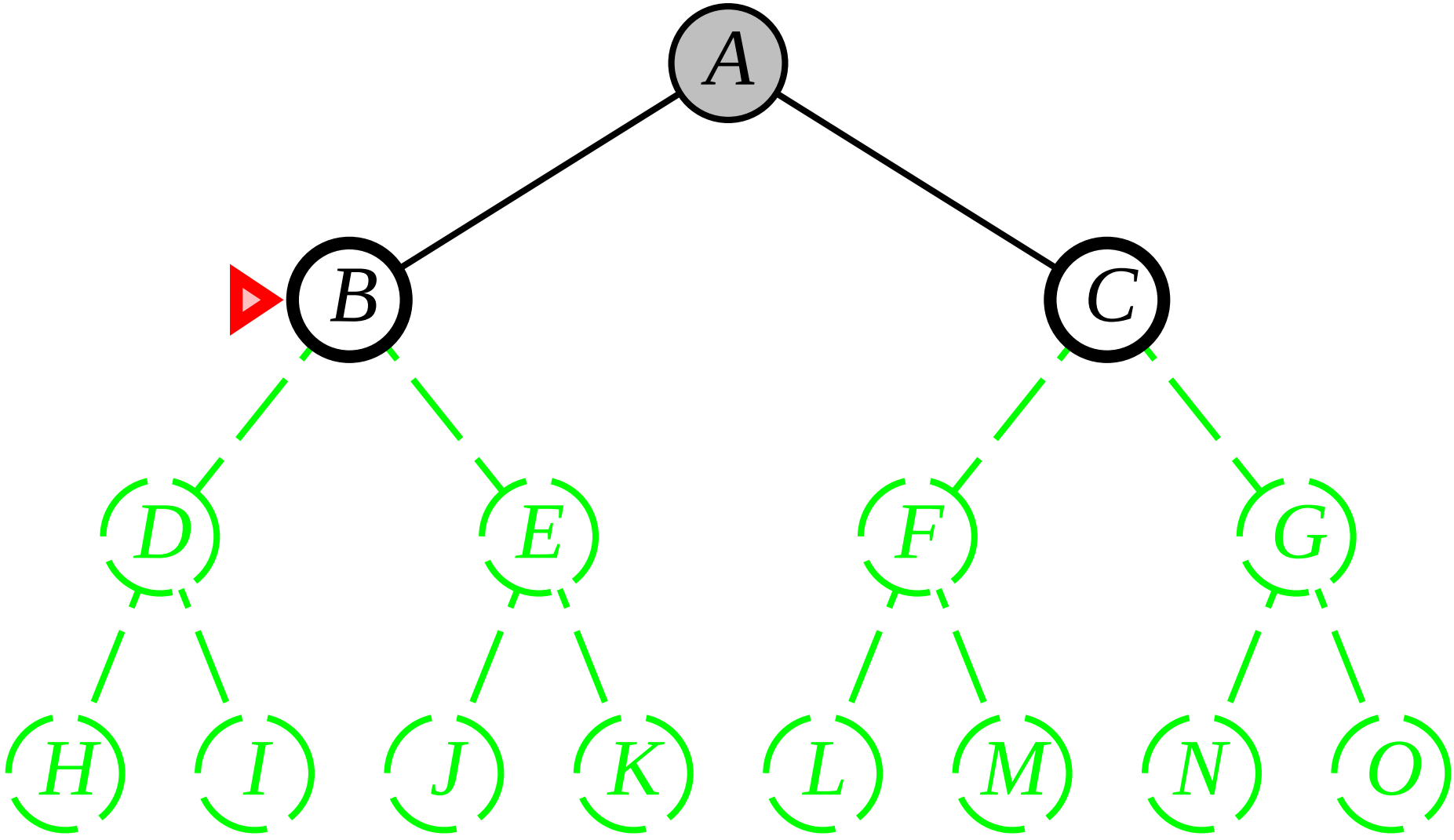
**Depth-first search can perform infinite cyclic excursions**

**Need a finite, non-cyclic search space (or repeated-state checking)**

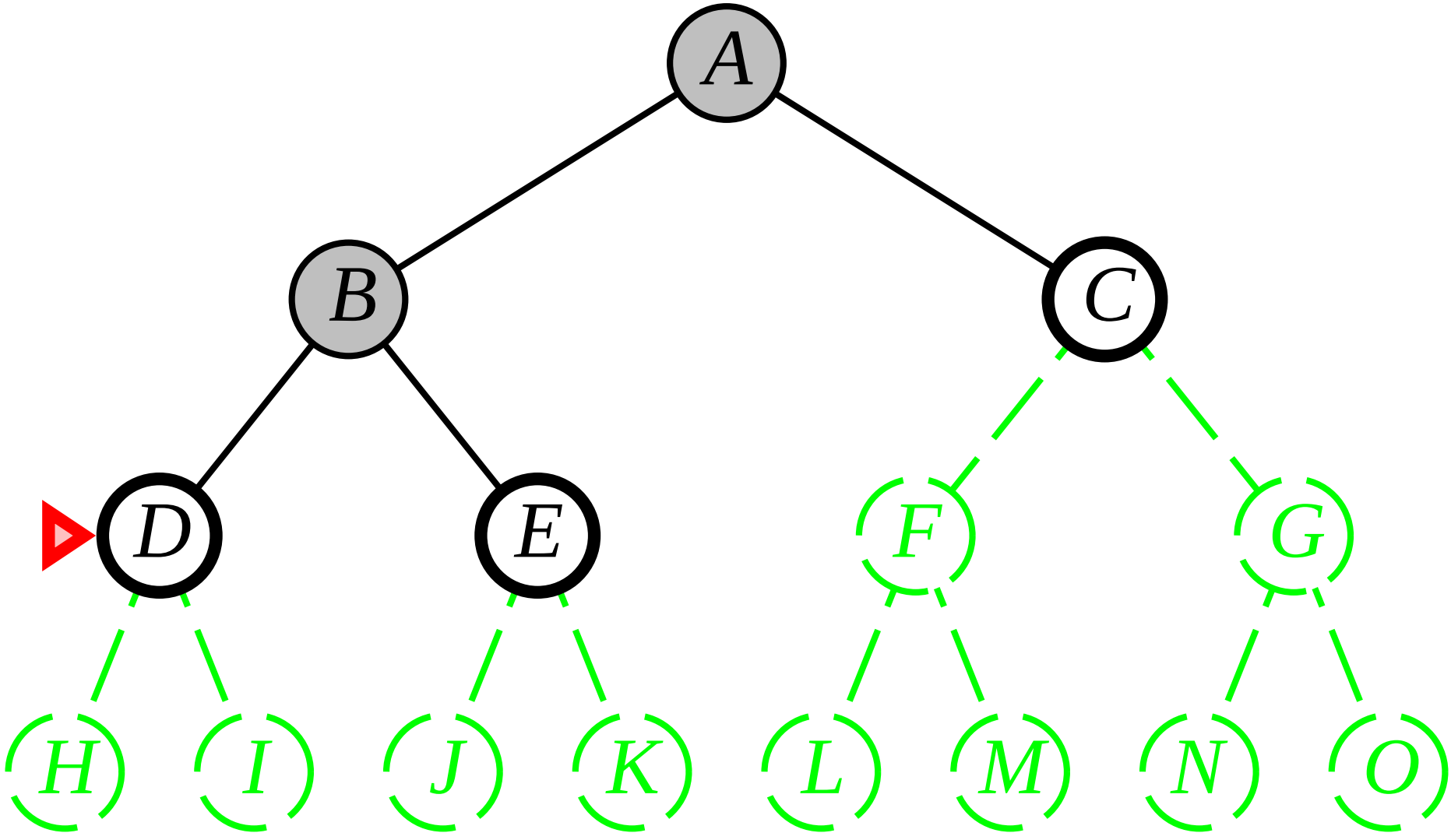
# Depth-first search



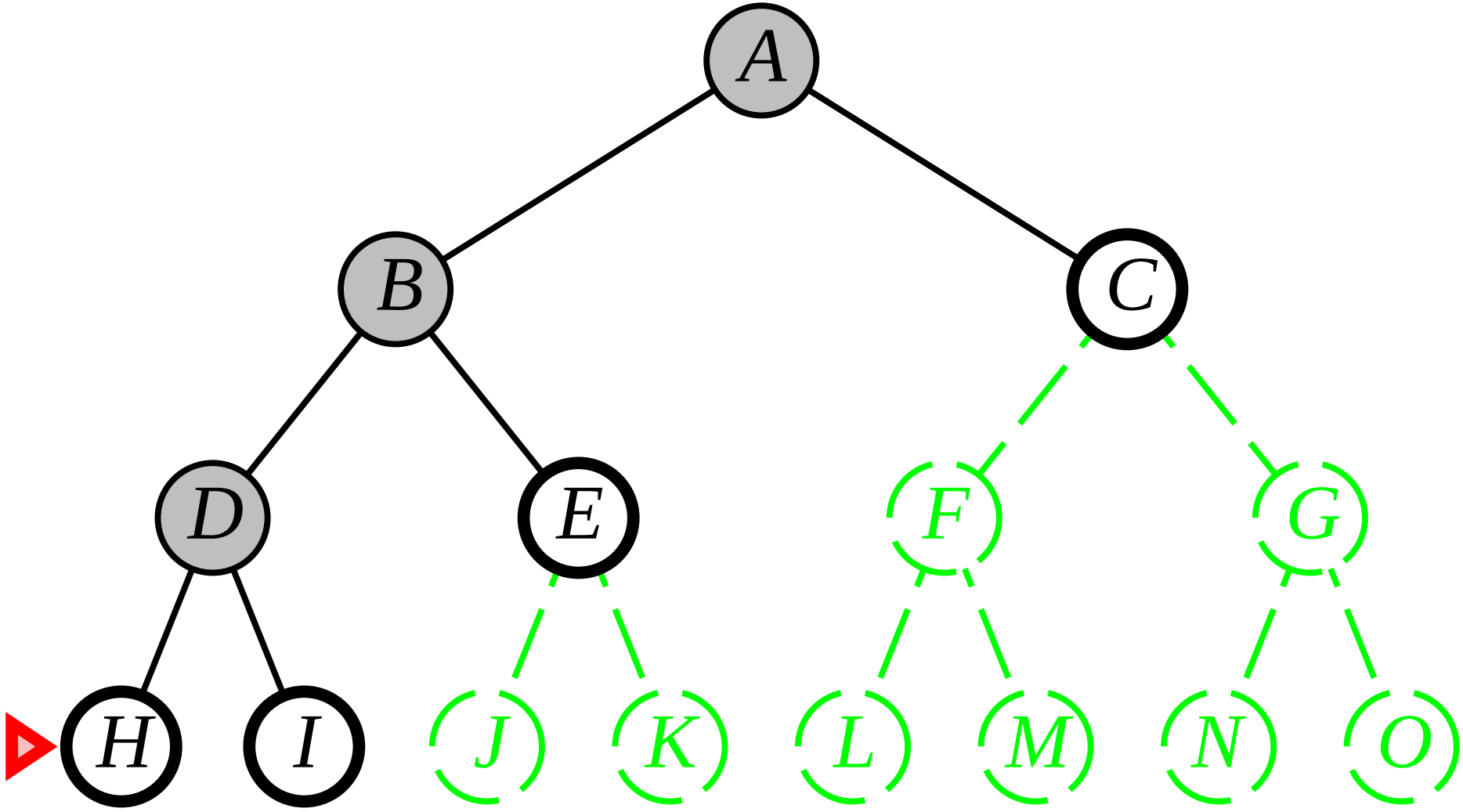
# Depth-first search



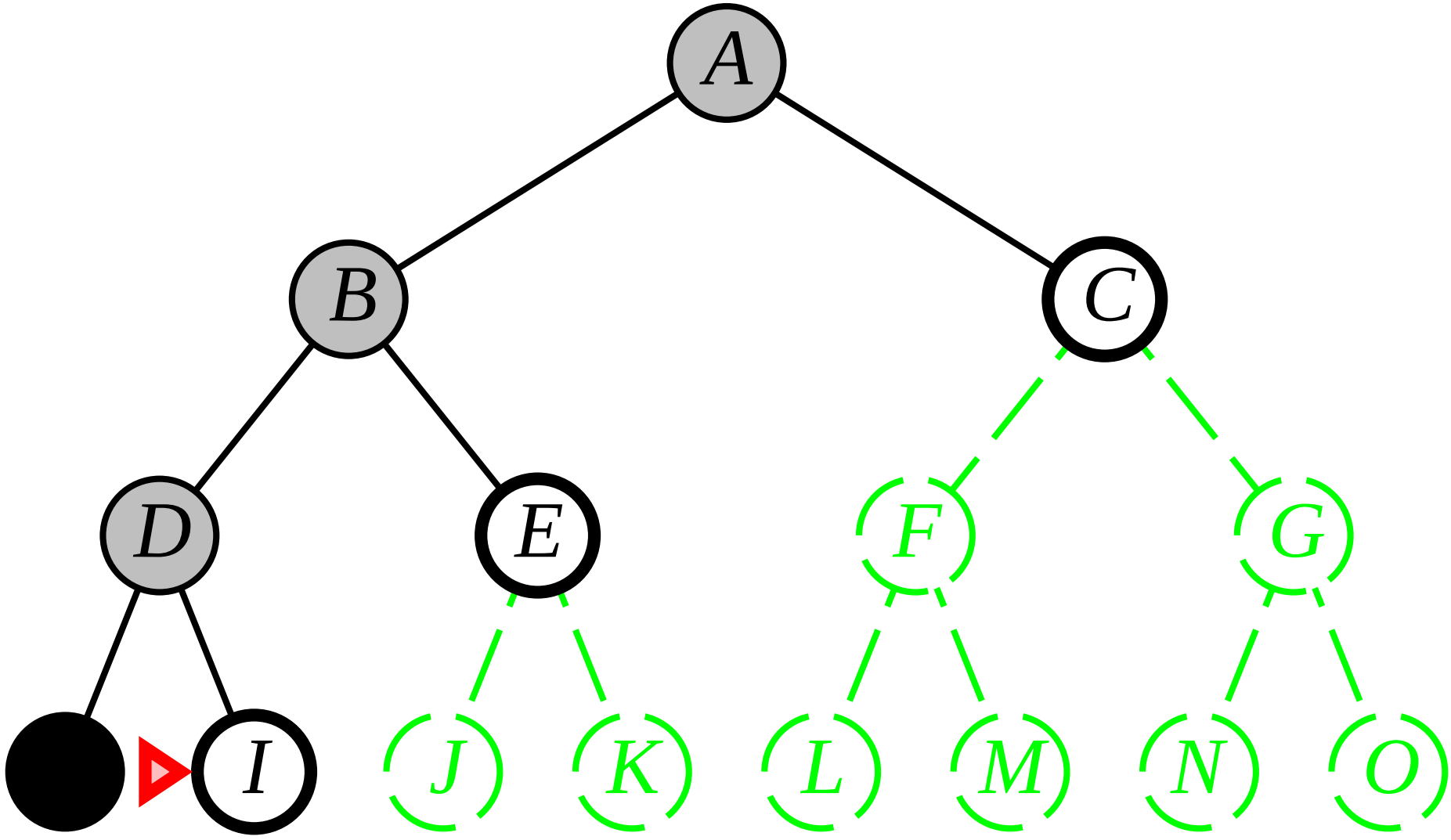
# Depth-first search



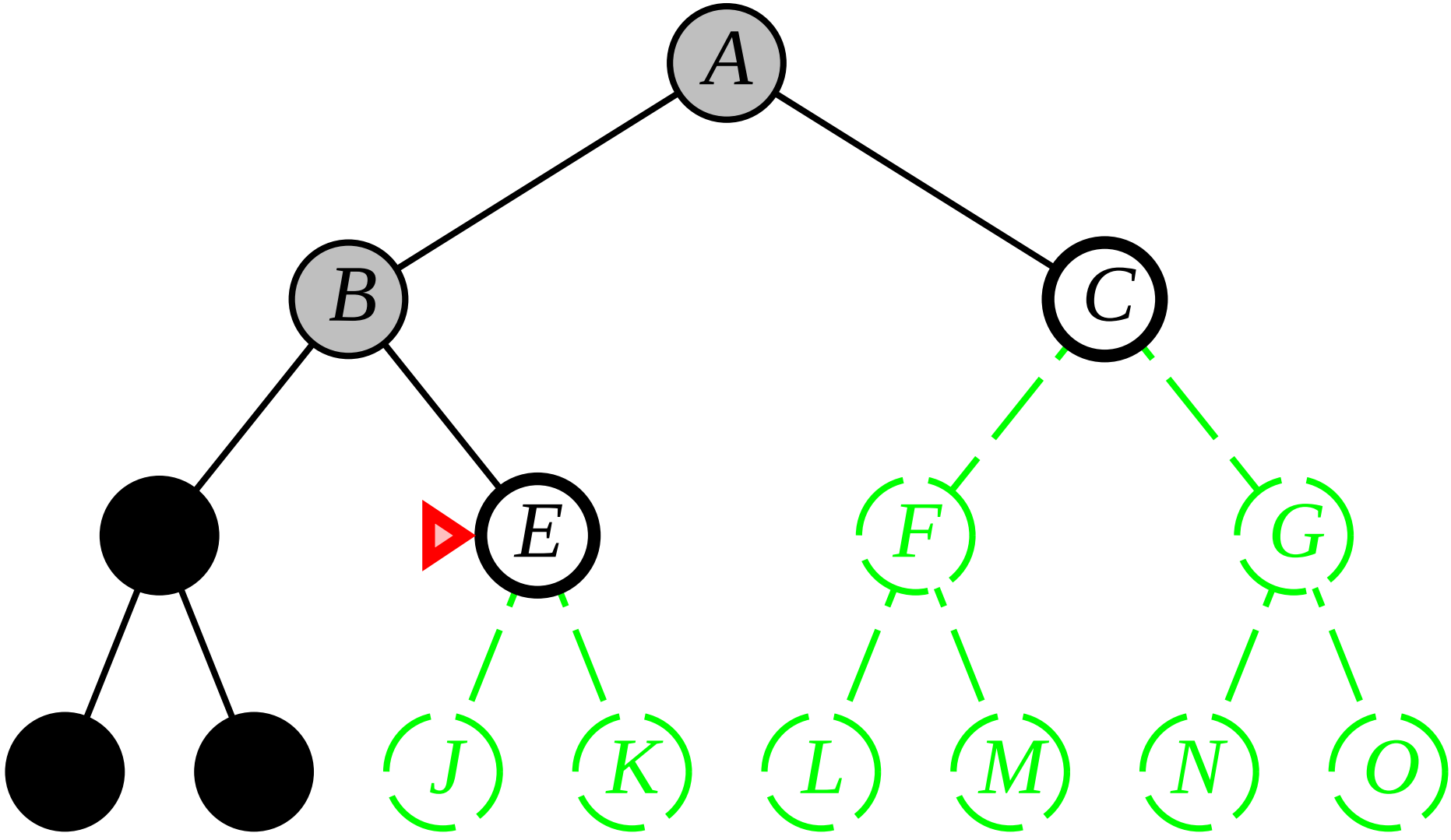
# Depth-first search



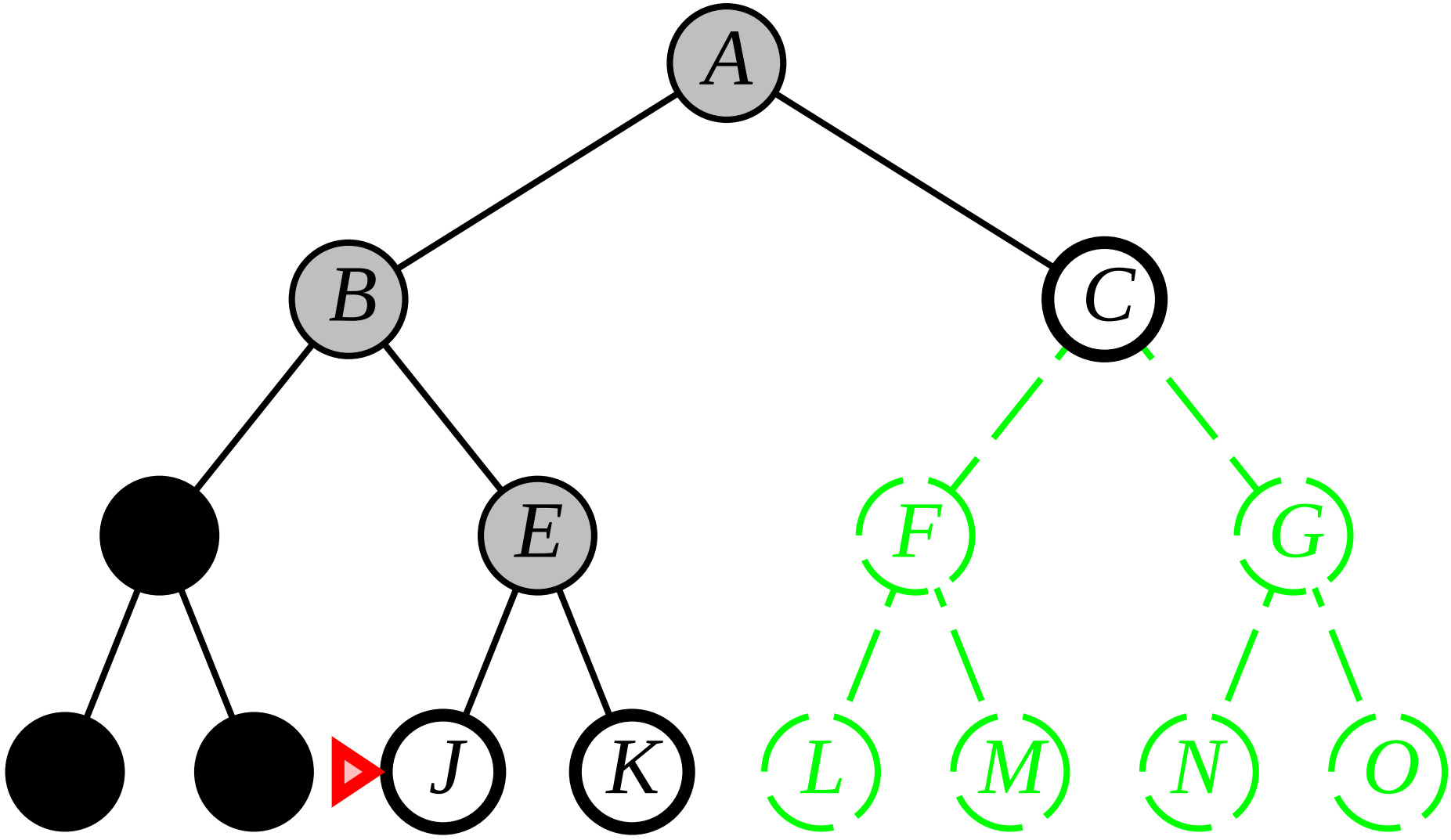
# Depth-first search



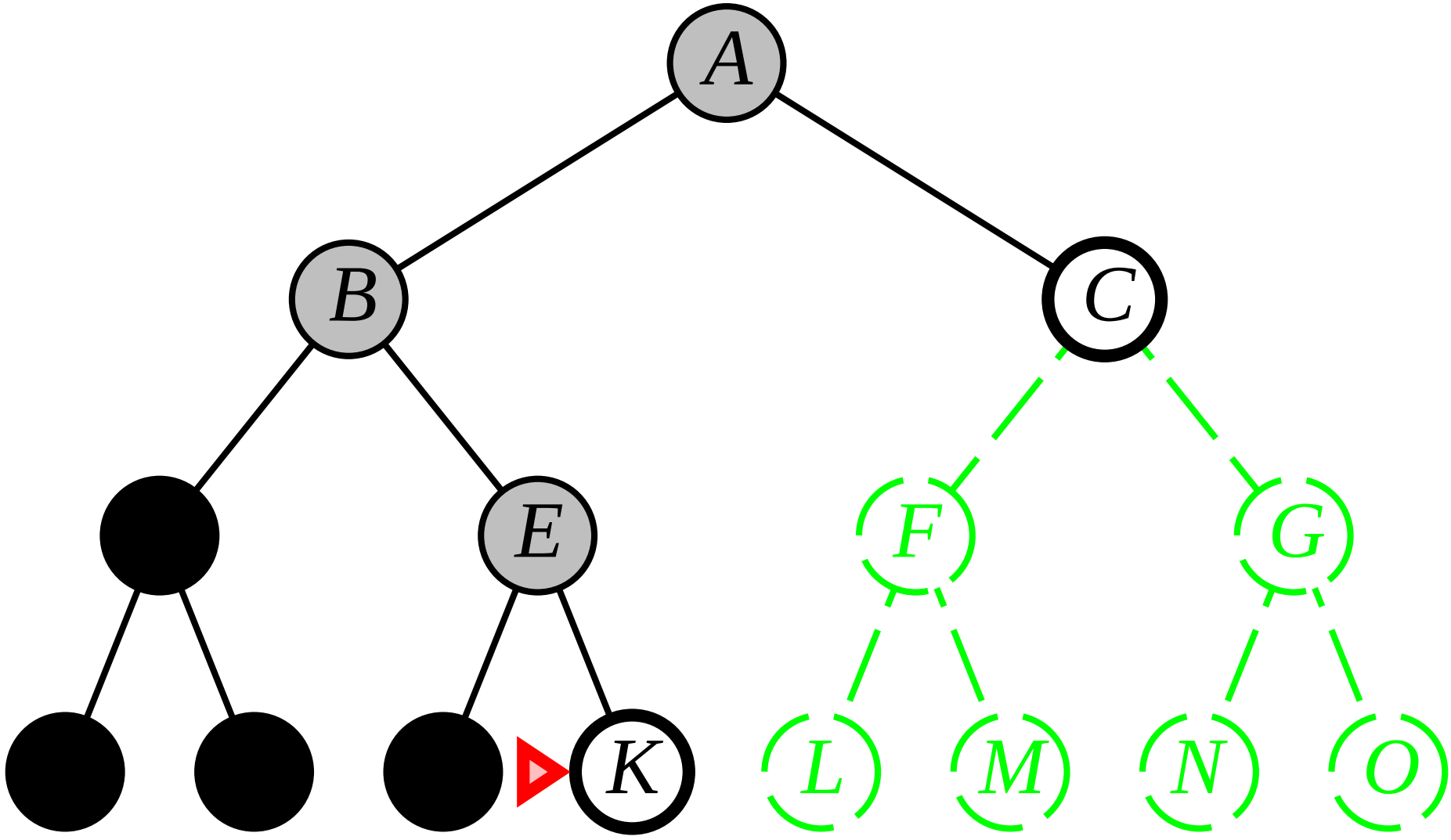
# Depth-first search



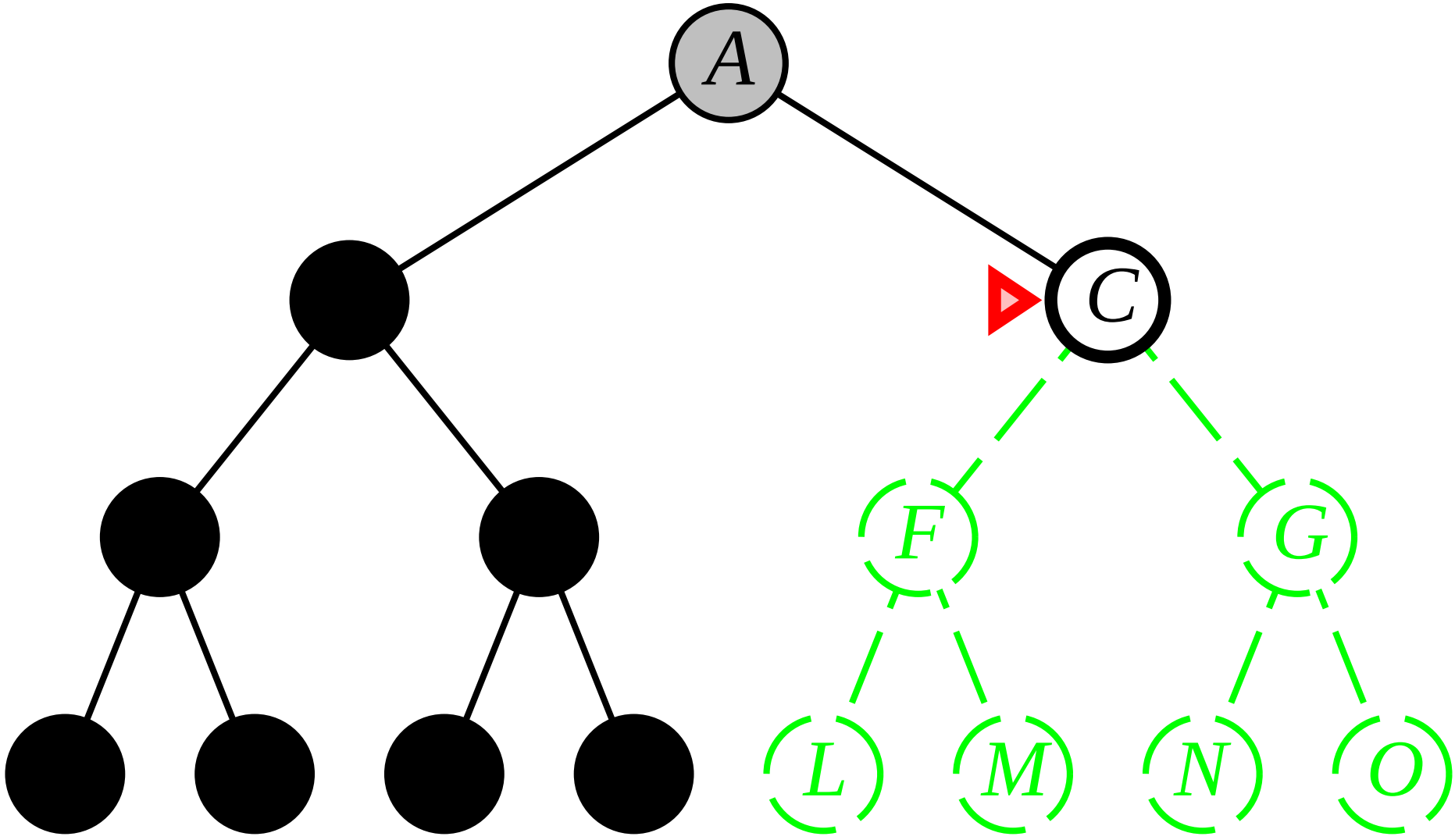
# Depth-first search



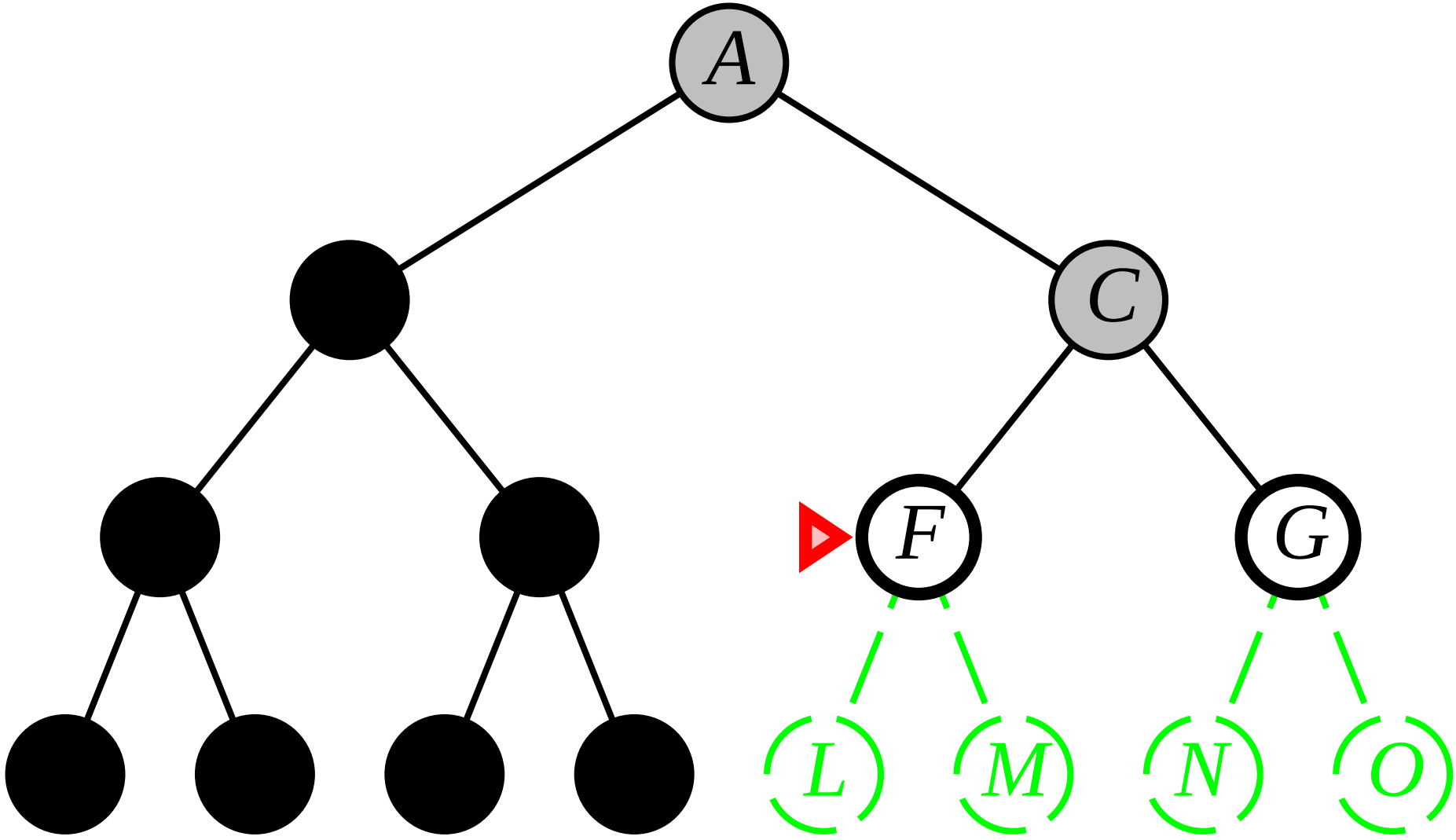
# Depth-first search



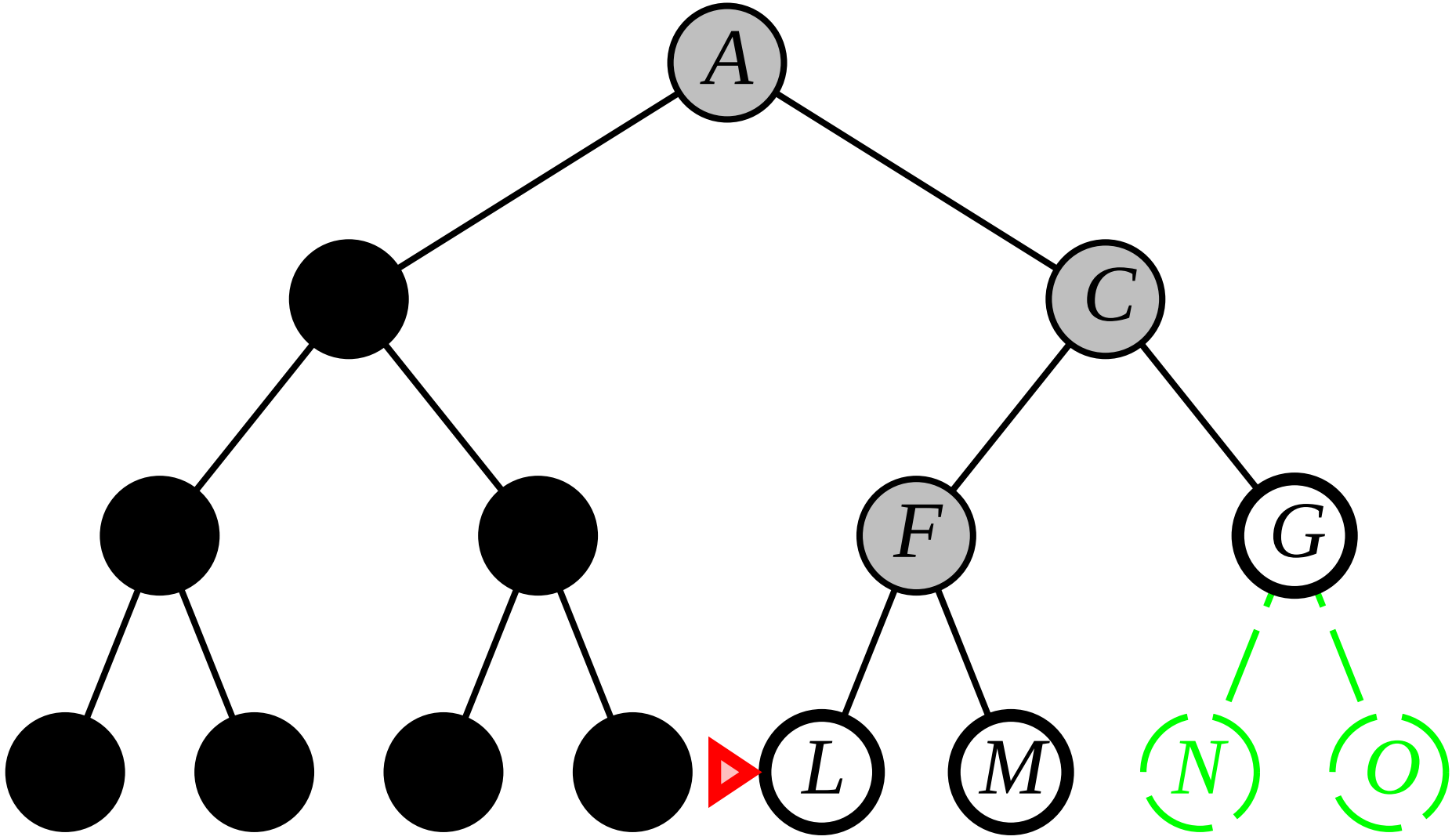
# Depth-first search



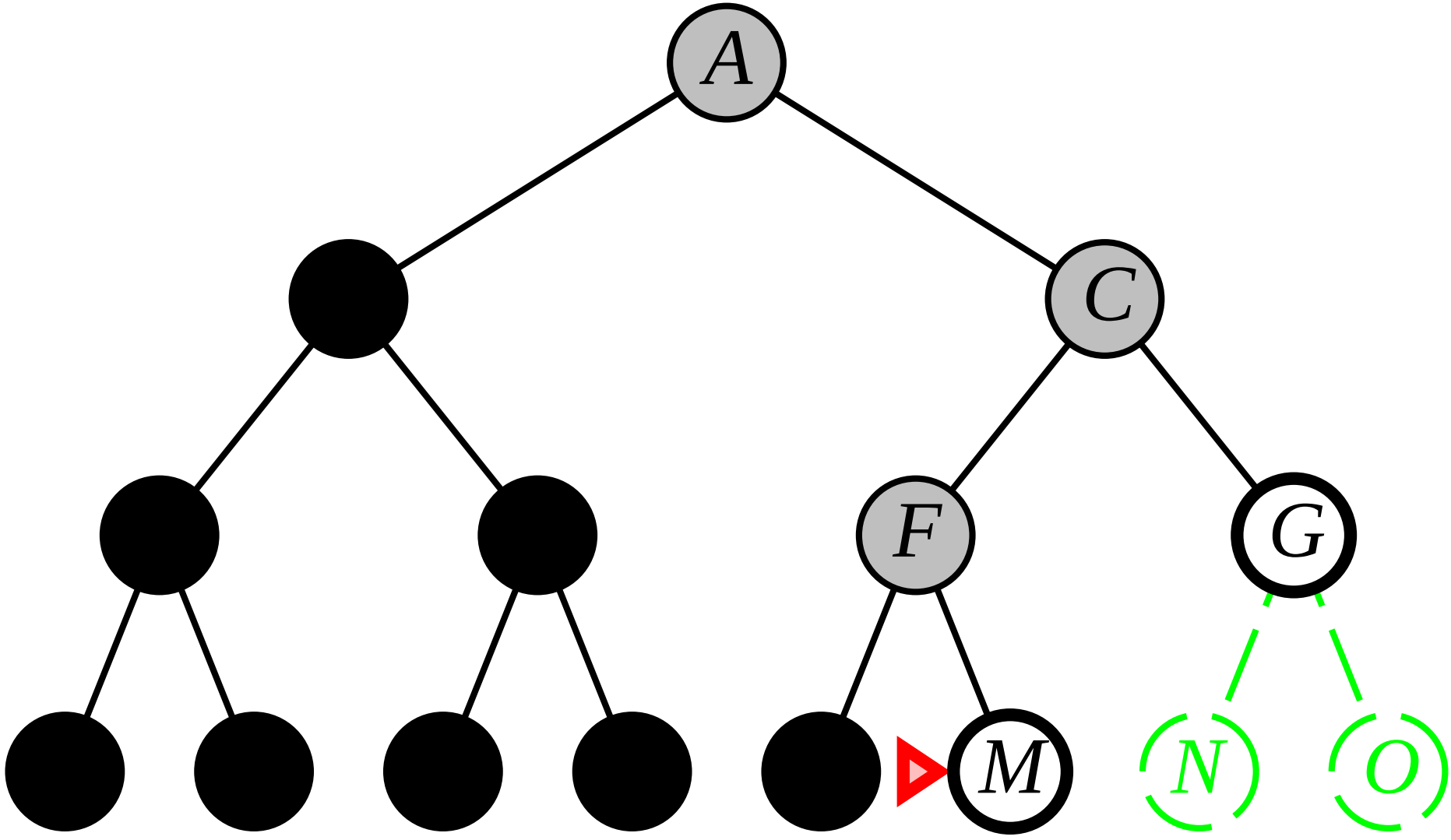
# Depth-first search



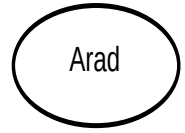
# Depth-first search



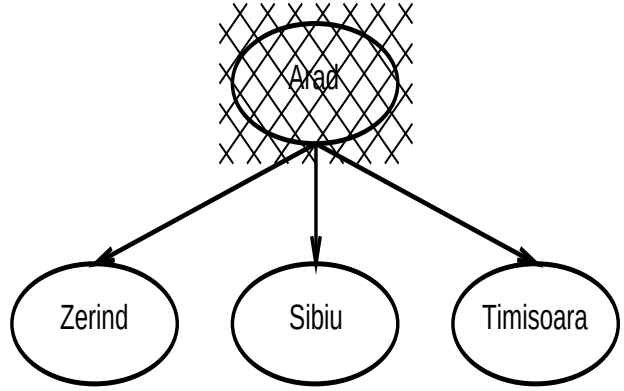
# Depth-first search



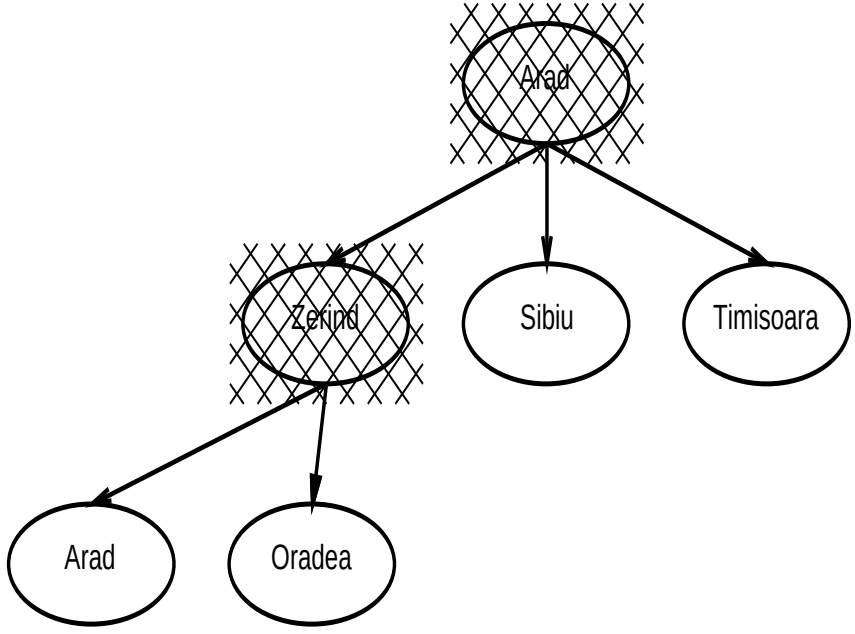
# Depth-first search: Example Romania



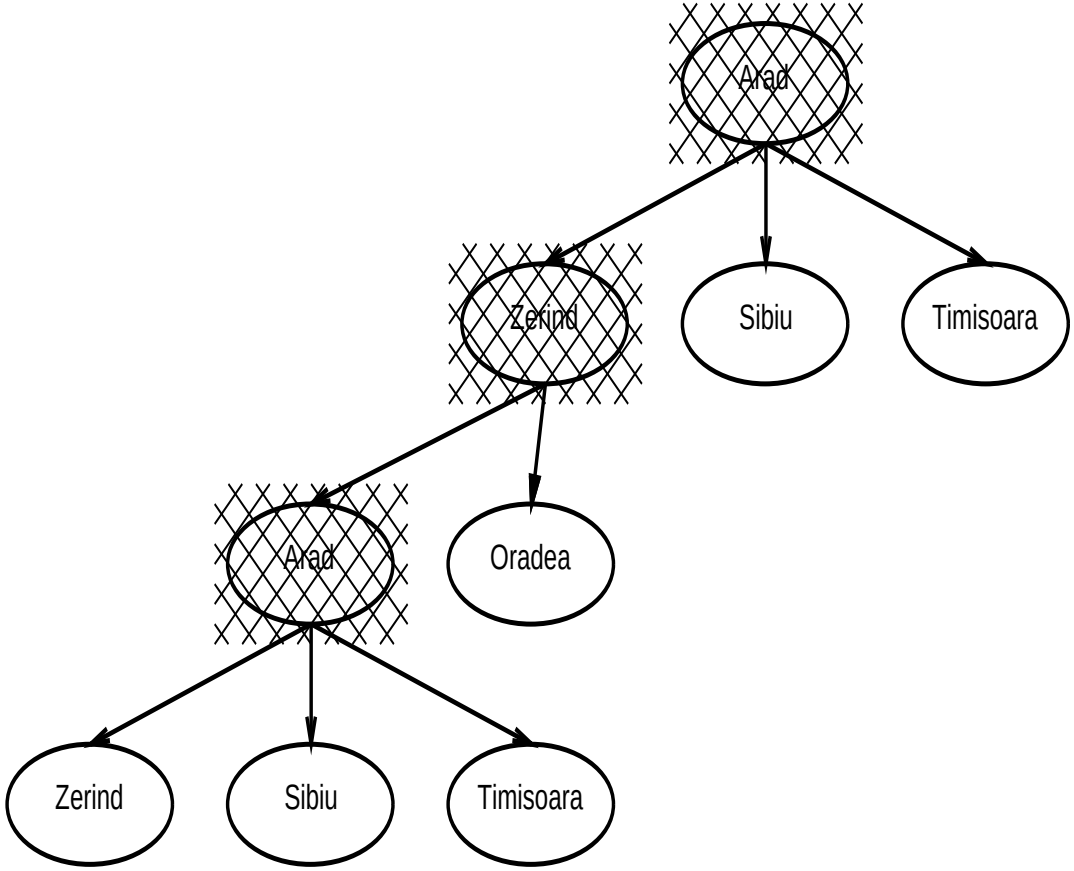
# Depth-first search: Example Romania



# Depth-first search: Example Romania



# Depth-first search: Example Romania



# Depth-first search: Properties

**Complete** **Yes: if state space finite**

**No: if state contains infinite paths or loops**

**Time** $O(b^m)$

**Space** $O(bm)$ **(i.e. linear space)**

**Optimal** **No**

**Disadvantage**

**Time terrible if $m$ much larger than $d$**

**Advantage**

**Time may be much less than breadth-first search if solutions are dense**

# Iterative deepening search

## Depth-limited search

**Depth-first search with depth limit**

## Iterative deepening search

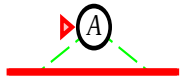
**Depth-limit search with ever increasing limits**

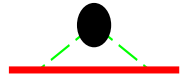
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution or failure
    inputs: problem     /* a problem */

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```

# Iterative deepening search with depth limit 0

Limit = 0

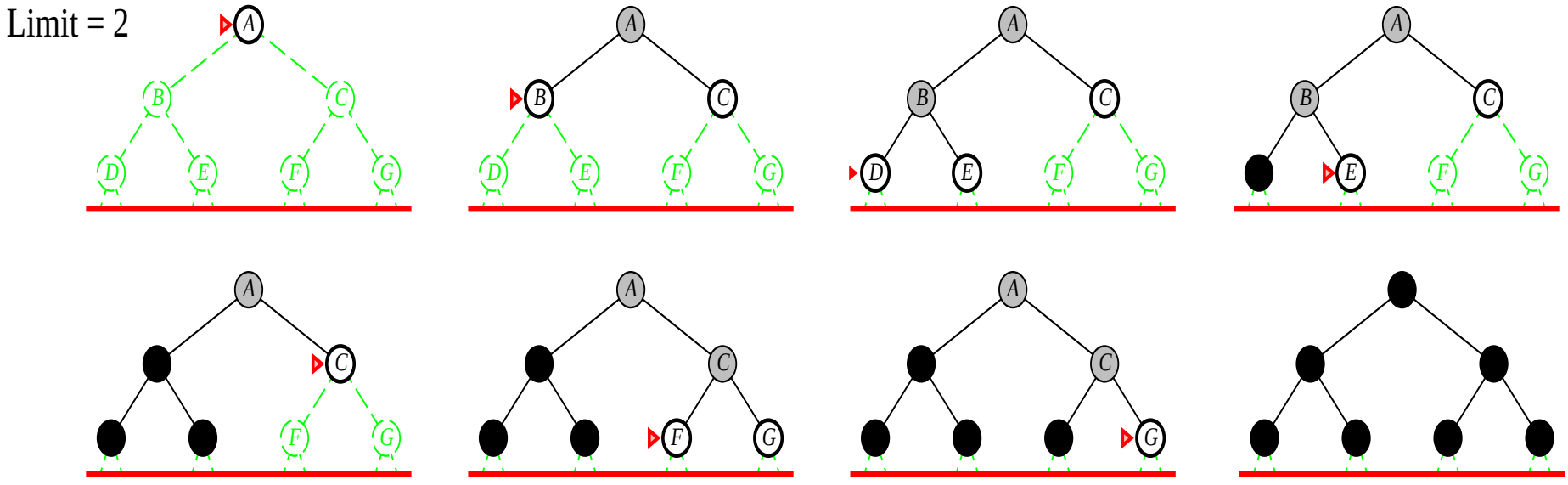


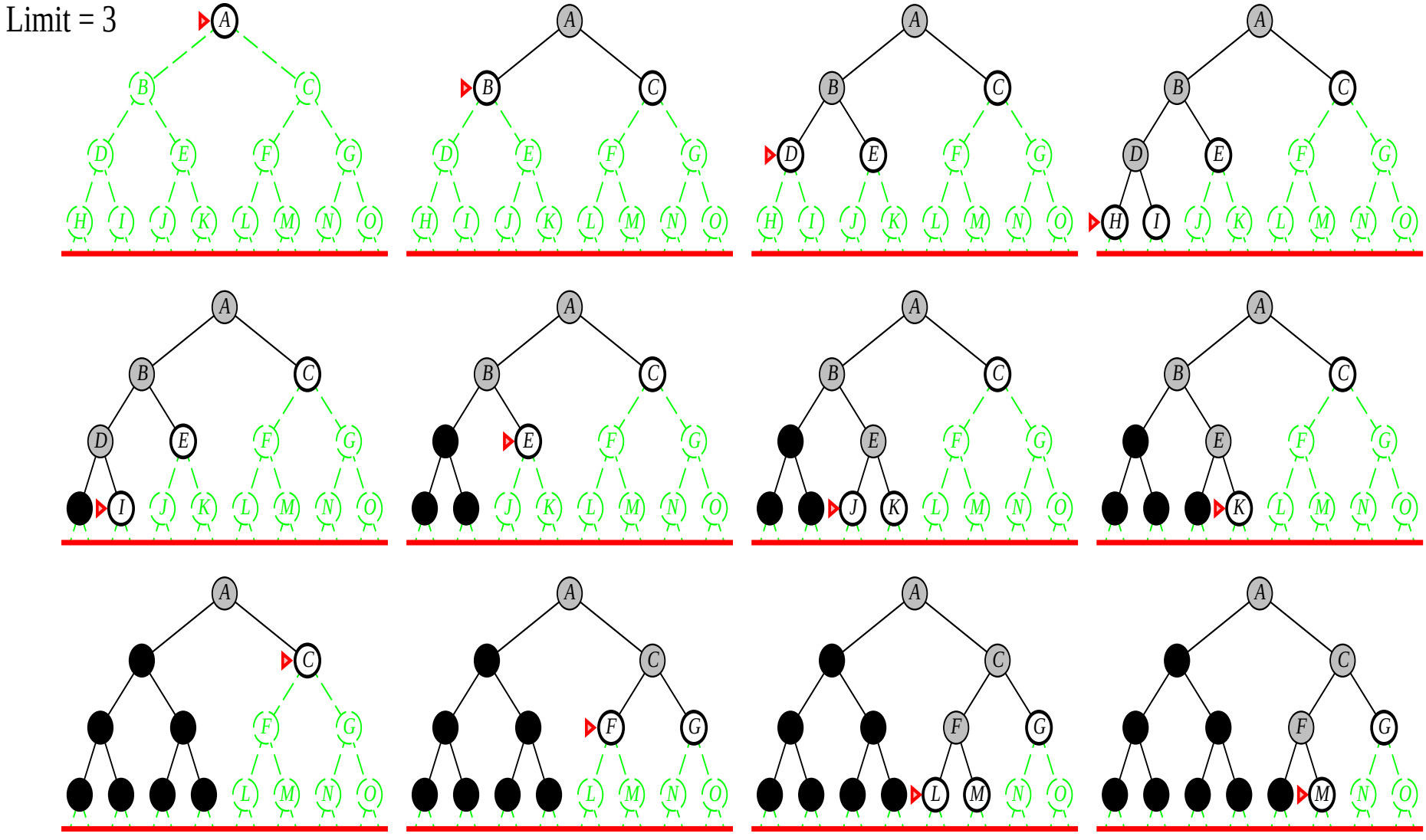
# Iterative deepening search with depth limit 1

Limit = 1

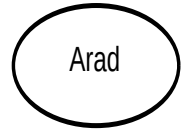
# Iterative deepening search with depth limit 2



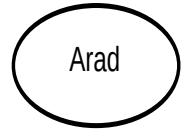
# Iterative deepening search with depth limit 3



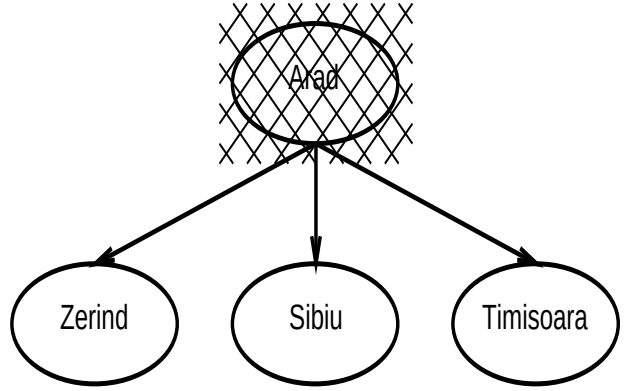
# Iterative deepening search: Example Romania with $l = 0$



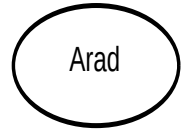
# Iterative deepening search: Example Romania with $l = 1$



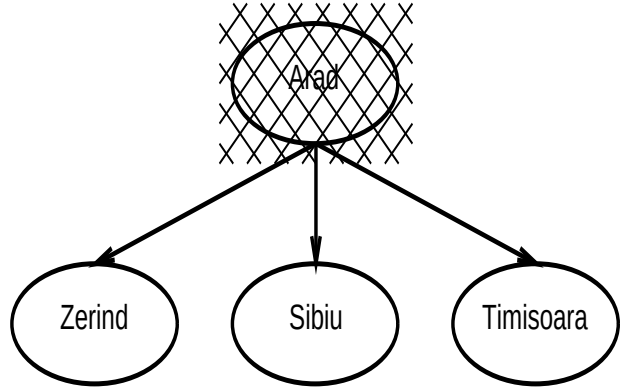
# Iterative deepening search: Example Romania with $l = 1$



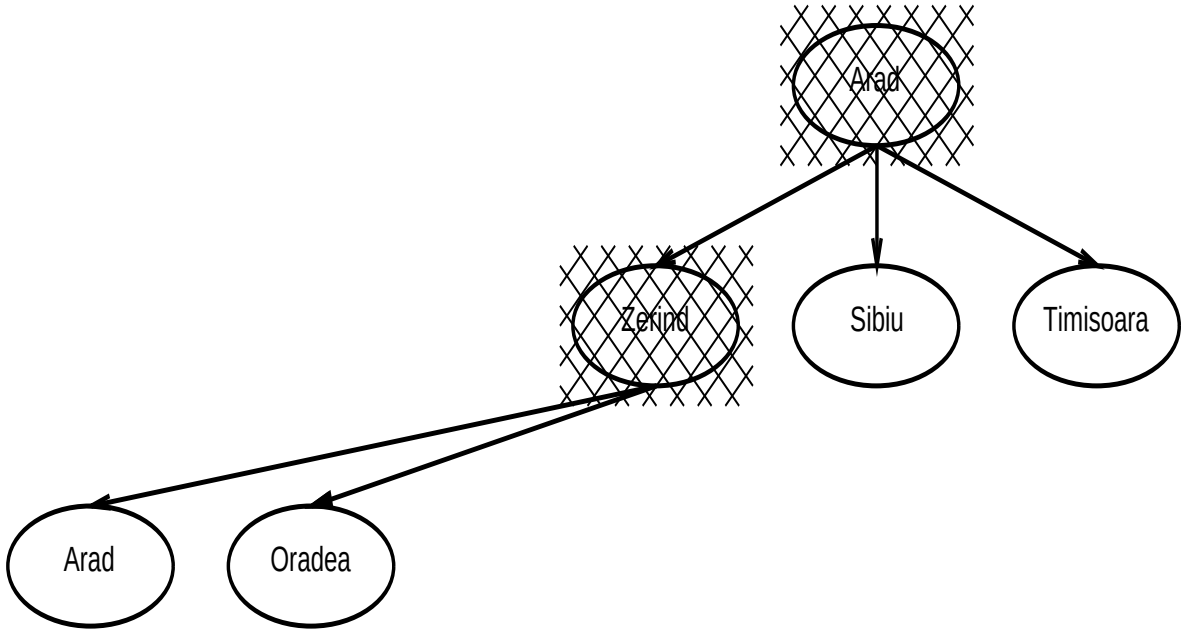
# Iterative deepening search: Example Romania with $l = 2$



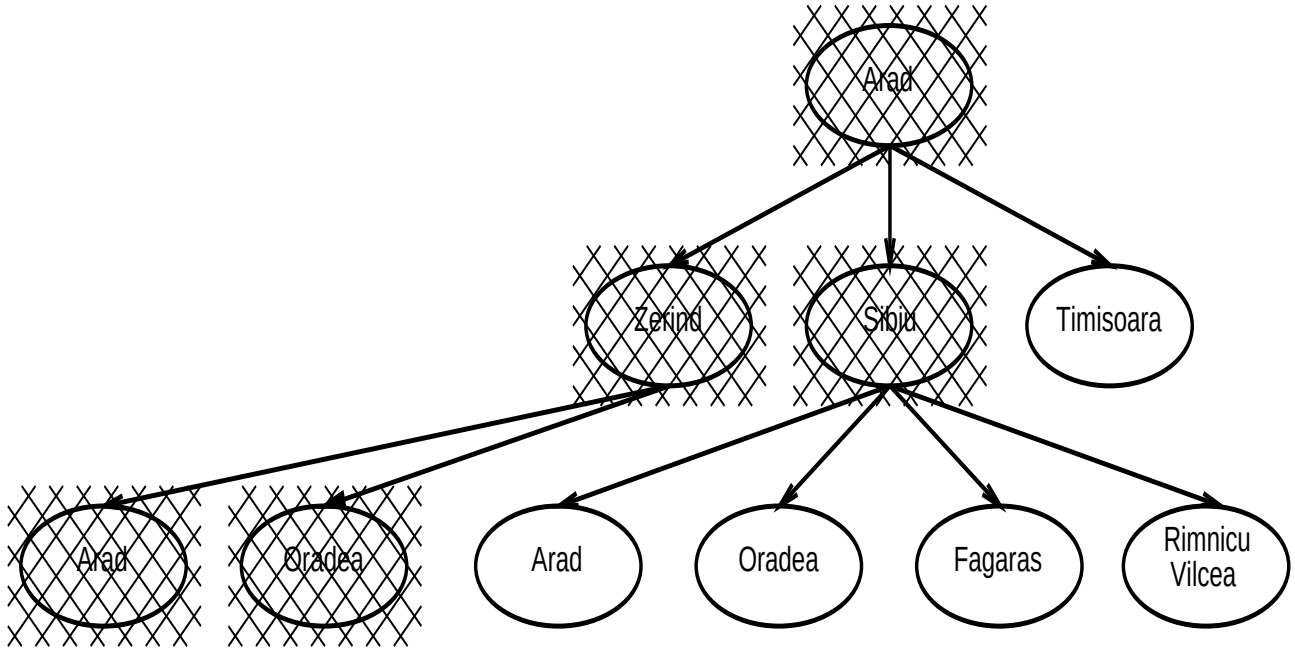
# Iterative deepening search: Example Romania with $l = 2$



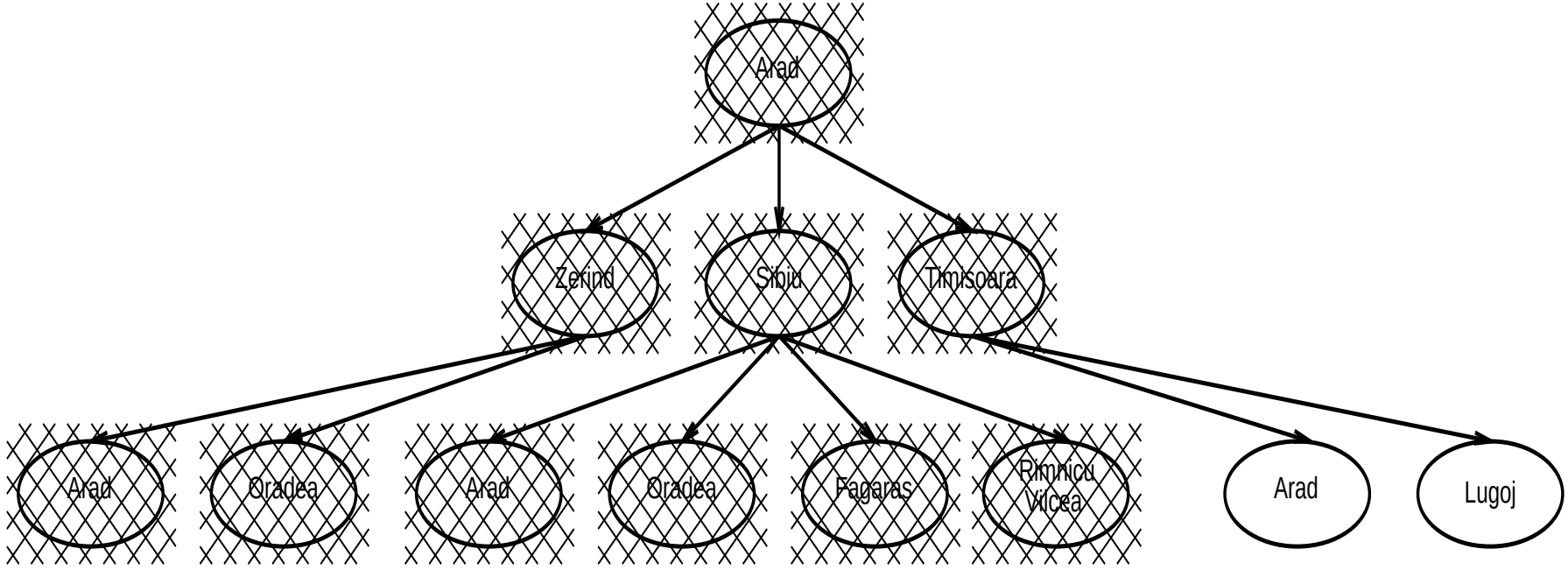
# Iterative deepening search: Example Romania with $l = 2$



# Iterative deepening search: Example Romania with $l = 2$



# Iterative deepening search: Example Romania with $l = 2$



# Iterative deepening search: Properties

| | |
|---|---|
| **Complete** | **Yes** |
| **Time** | $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d \in O(b^{d+1})$ |
| **Space** | $O(bd)$ |
| **Optimal** | **Yes (if step cost = 1)** |

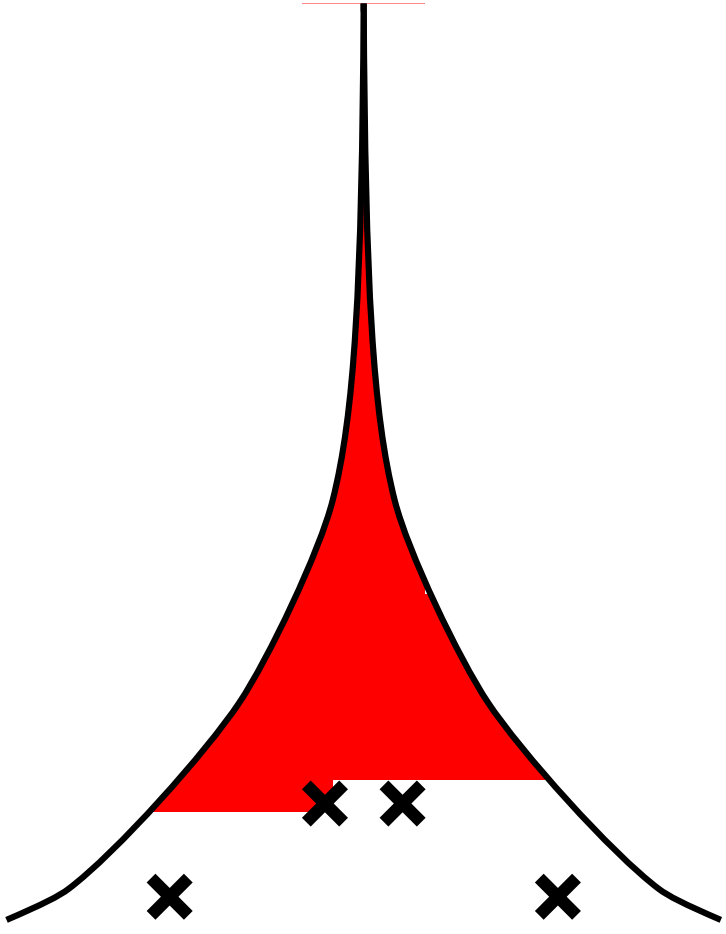
**(Depth-First) Iterative-Deepening Search often used in practice for search spaces of large, infinite, or unknown depth.**

# Comparison

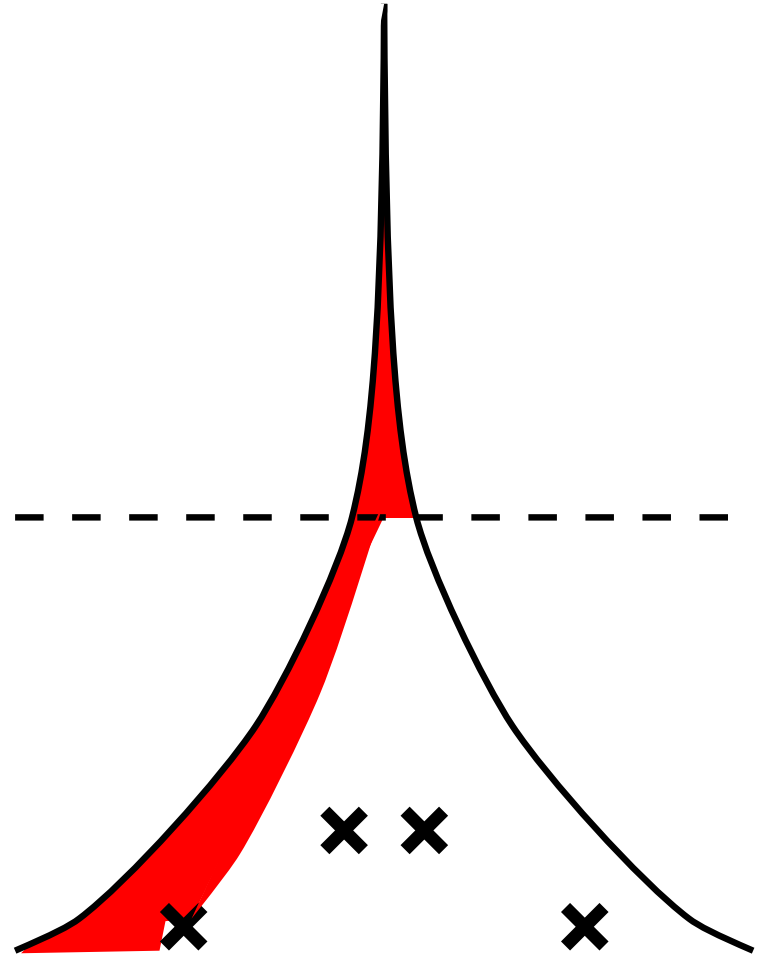
| Criterion | Breadth-first | Uniform-cost | Depth-first | Iterative deepening |
|---|---|---|---|---|
| Complete? | Yes$^*$ | Yes$^*$ | No | Yes |
| Time | $b^{d+1}$ | $\approx b^d$ | $b^m$ | $b^d$ |
| Space | $b^{d+1}$ | $\approx b^d$ | $bm$ | $bd$ |
| Optimal? | Yes$^*$ | Yes | No | Yes |

# Comparison

## Breadth-first search



## Iterative deepening search



# Summary

- **Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored**
- **Variety of uninformed search strategies**
- **Iterative deepening search uses only linear space and not much more time than other uninformed algorithms**