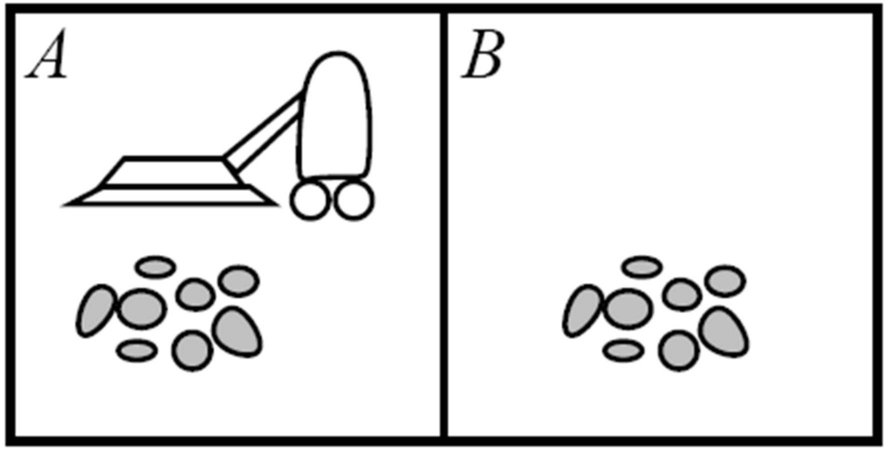
Part 2: Solving Problems by Searching

Univ.-Prof. Dr. Gerhard Widmer
Department of Computational Perception
Johannes Kepler University Linz

gerhard.widmer@jku.at
http://www.cp.jku.at/people/widmer

“Motivation”: The Vacuum Cleaner World



“World”: two rooms A, B that may contain dirt
Agent: a vacuum cleaner
Percepts (observations): location and contents (e.g., [A,dirty])
Actions: GoLeft, GoRight, Suck, (NoOp)
Goal: world should be clean

TASK: *Write an agent program that, starting in some world configuration S, plans action sequences that lead to achievement of the goal*

Overview

Problem solving as search for appropriate action sequences

Definition of search problem

Some simple toy problems

A generic tree search algorithm schema

(Digression: Computational Complexity)

Five uninformed (‘blind’) search strategies:

- Breadth-First Search
- Uniform-Cost Search
- Depth-First Search (Backtracking Search)
- Depth-limited Search
- Iterative Deepening Search

Solving Problems by Planning Action Sequences

Observations:

- Simple Reflex Agents are too limited to solve real problems:
- Direct mapping from percept sequences to actions is too large to store
- Lack of flexibility and autonomy

More promising: Goal-based Problem Solving Agent

"Knowledge Base (KB)"

- Wants to achieve a goal
- Can execute a fixed (finite) set of actions
- Knows about the consequences of its actions
- Tries to find sequences of actions that lead to desirable states (goals)
  ➔ no hard-wired program that prescribes every step

➔ PROBLEM SOLVING as SEARCH for action sequences

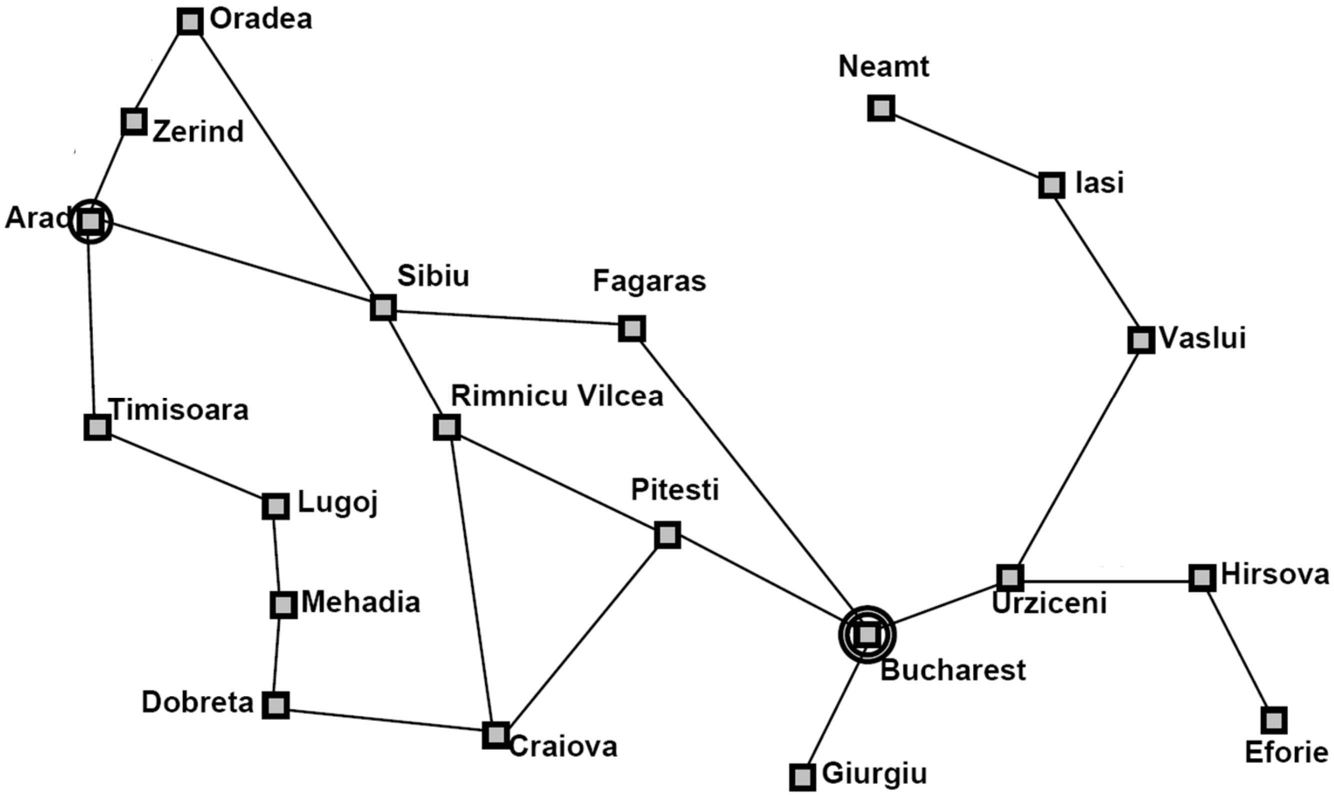
Note: For the moment, assume ***"offline"*** problem solving: solution is computed once, then executed "with eyes closed"; percepts (other than starting state) are ignored. Assumes that the world is deterministic and actions always produce the desired result

Problem Solving as Search

- The “Problem Solving as Search” paradigm may be one of the most fundamental contributions of (early) AI research to computer science
- AI has developed a large number of different search methods – e.g., “uninformed” search, “heuristic” search, “local” search, stochastic/probabilistic search (e.g., genetic/evolutionary algorithms), ...
- Many of these are now considered standard computer science algorithms
- Many tasks can be formulated as search problems
- Crucial: *appropriate formulation* of the task as a search problem
- Search methods are generic, i.e., independent of a particular task
  ➔ widely and directly applicable

Example: Travelling in Romania – Finding a Way from Arad to Bucharest [1)]

Map:



Represented as:

Arad → Zerind
Zerind → Arad
Arad → Sibiu
Arad → Timisoara
Zerind → Oradea
Timisoara → Lugoj
Oradea → Sibiu
….

[1)] from: Russell & Norvig, 2000

Definition: Well-defined Search Problems

A search problem can be formally defined by five components:

- the complete **set of possible world states** S (e.g., {in(Arad), in(Zerind), …})
  (terminology: "world states" = whatever is known (through percepts) about the world at a given point in time)

- for each state s, all possible **actions** that can be taken in s, and their **effects**
  (e.g., in the form of a *successor function* Succ(s) that gives, for each state s, a set of pairs <action, resultstate> :
  Succ(Arad) = {<Arad→Zerind, in(Zerind)> , <Arad→Sibiu, in(Sibiu)> , …}, …

- the **starting (initial) state** (e.g., in(Arad))

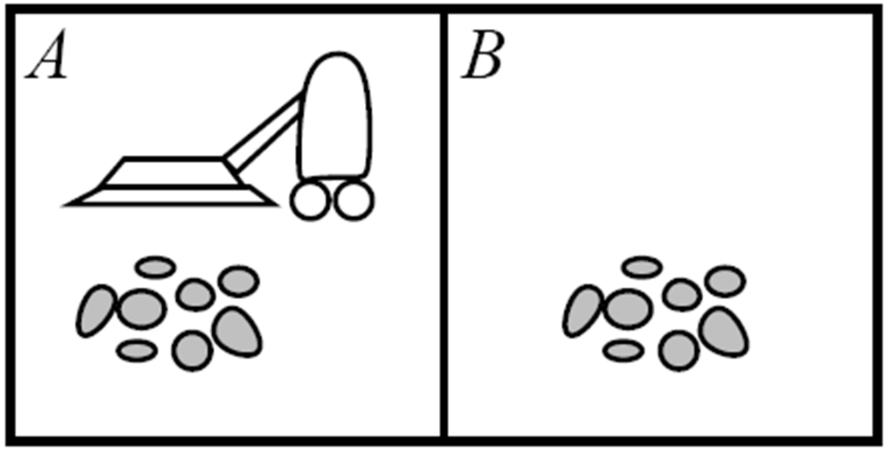
- a **goal test** (a definition of which states constitute goals); can be
  - explicit (direct description of desired state), e.g., x = in(Bucharest)
  - implicit (needs to be computed), e.g., checkmate(x)

- **costs** c(x,a) of taking action a in state x
  (must be non-negative / greater than zero)

Search Problems: Basic Concepts

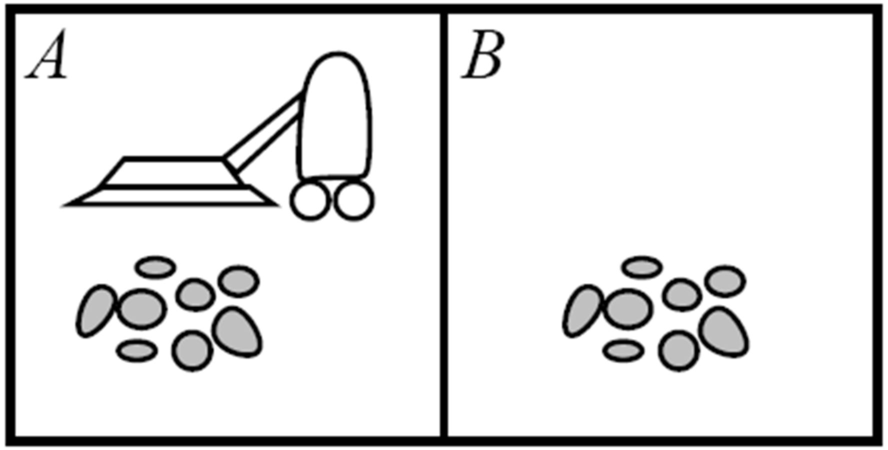
- **State space** = set of all possible states that can be reached from the initial state (defined by initial state and successor function)
- The state space forms a **graph**
- A **path** in the state space = sequence of states connected by actions
- A **solution** to a problem is a path from the initial state to a goal state
- The **cost** of a solution is the sum of the action costs along the solution path
- An **optimal solution** is a solution with minimum cost
- **Search** = process of finding a (possibly optimal) solution in the state space
- Search generally proceeds by creating (explicitly or implicitly) a **search tree**, exploring a (usually very small) subgraph of the state space graph

Some Simple Example Problems ("Toy Problems"):
1. The Vacuum Cleaner World



"World": two rooms A, B that may contain dirt
Agent: a vacuum cleaner
Percepts (observations): location and contents (e.g., [A,dirty])
Actions: GoLeft, GoRight, Suck
Goal: world should be clean

Some Simple Example Problems ("Toy Problems"): 1. The Vacuum Cleaner World

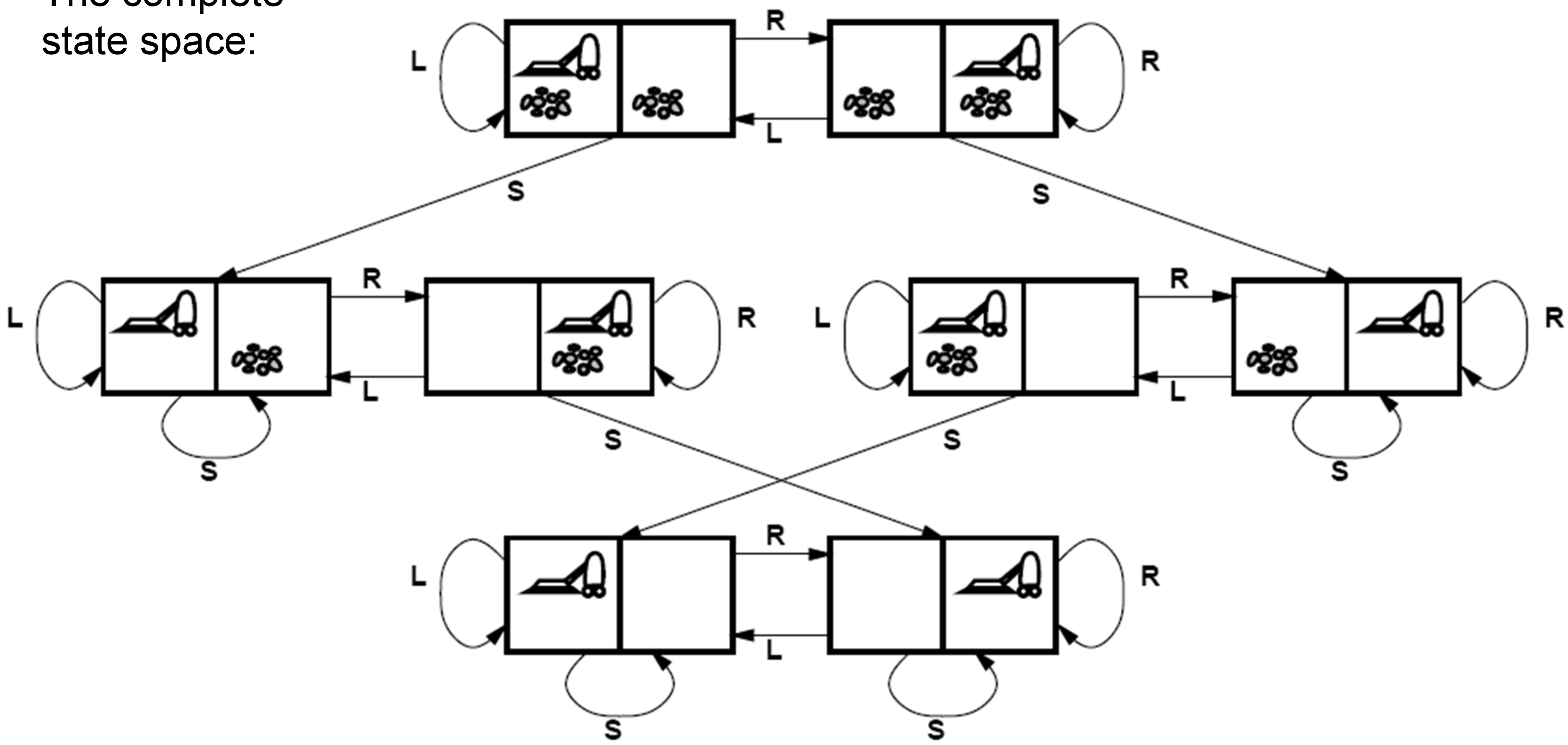


Formalisation as a Search Problem:

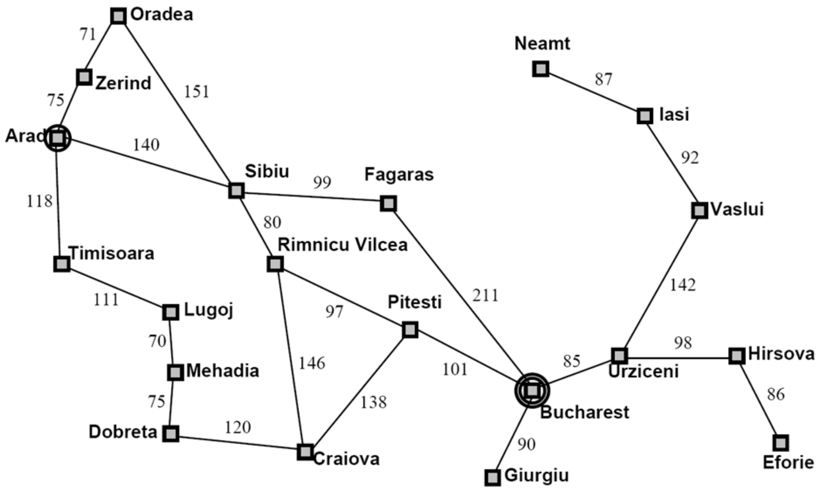
- **States:** triplets <RobotLoc, DirtyOrNotA, DirtyOrNotB>; 2 rooms, each may or may not contain dirt ➔ 2 x 2 x 2 = 8 possible world states
- **Initial state:** any of the states could be the initial state
- **Actions:** Left (L), Right (R), Suck (S)
- **Successor Function:** defines effects of actions (see next slide)
- **Goal Test:** State = <A, clean, clean> or <B, clean, clean>
- **Action costs:** each step costs 1 ➔ optimal solution = shortest path

Some Simple Example Problems ("Toy Problems"): 1. The Vacuum Cleaner World

The complete state space:



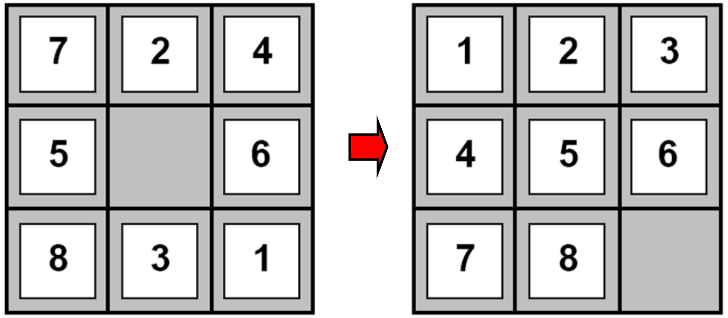
Some Simple Example Problems ("Toy Problems"): 2. The "Traveling in Romania" World



Formalisation as a Search Problem:

- **States:** the agent is in one of N cities ➔ N possible world states $\mathrm{in}(x)$
- **Initial state:** $\mathrm{in}(\mathrm{Arad})$
- **Actions:** $x \rightarrow y$, where x is the current city,
  and y is a city reachable from x via a direct connection
- **Successor Function:** defines all the roads
- **Goal Test:** $\mathrm{State} = \mathrm{in}(\mathrm{Bucharest})$
- **Action costs:** length of road from x to y ➔ path cost = total trip length

Some Simple Example Problems ("Toy Problems"): 3. The 8-Puzzle



Start State Goal State

Formalisation as a Search Problem:

- **States:** a state description specifies the location of each of the eight tiles and the blank in one of the 9 squares
  ➔ size of state space: 9!/2 = 181.440 reachable states!
  (4x4 15-puzzle: 1.300.000.000 …)
- **Initial state:** any of the states could be the initial state
- **Actions:** MoveBlankLeft (L), MBRight (R), MBUp (U), MBDown(D)
- **Goal Test:** true if numbers obey a certain ordering
- **Action costs:** each step costs 1 ➔ want to find shortest solution

Real-World Examples of Search Problems

- Route Finding
- Robot Navigation and Action Planning
- Automatic Assembly Sequencing, Scheduling
- Protein Design
- Generally: Millions of Optimisation/Configuration Problems
- Internet Searching
- Logical Reasoning (!)
- Learning (!)
- …

… any problem that can be formalised as finding a sequence of operations to reach a desired state or produce a desired configuration.

Problem Solving as Search

Basic idea of a general search process (simplified):

1. start at initial state
2. "expand" the current state:
   try (all) possible actions applicable to current state, producing new states
3. check if one of the new states is a goal
4. if yes ➔ solution found
5. if no ➔ select one of the as yet unexpanded states for new search step
6. go to 2.

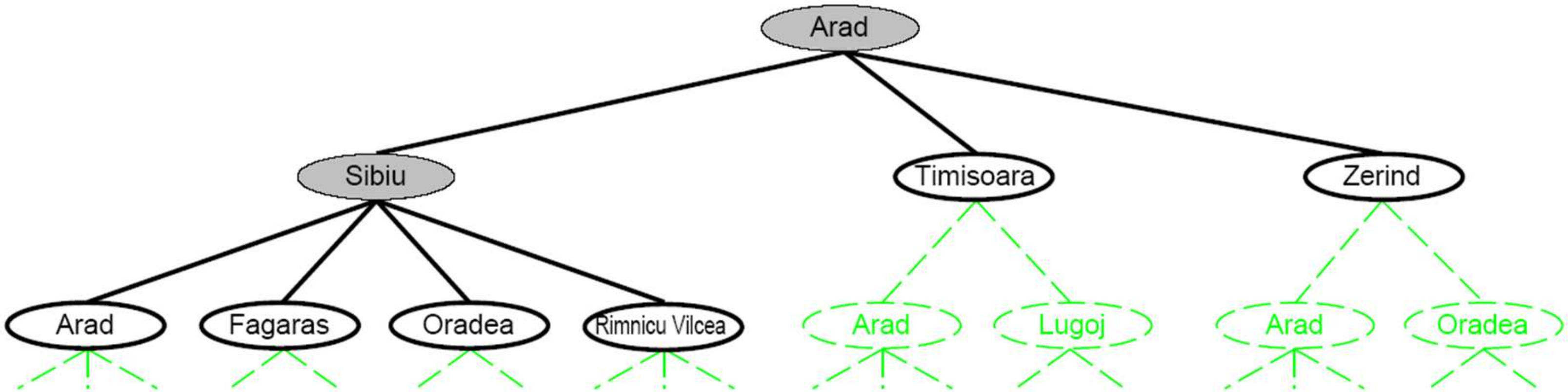
➔ Search process produces a **Search Tree**

- **Search Node:** a node in the search tree (i.e., a state with extra information)
- **Expanding** a node/state = generating a new set of states by applying the successor function to the node (i.e., trying out all actions that could be taken in the current state)
- **Search Strategy:** method for deciding which node to expand next

Search Trees

Two more concepts:

- **Leaf Node** = node that has been generated, but not yet expanded
- **„Fringe“** = set of all current leaf nodes



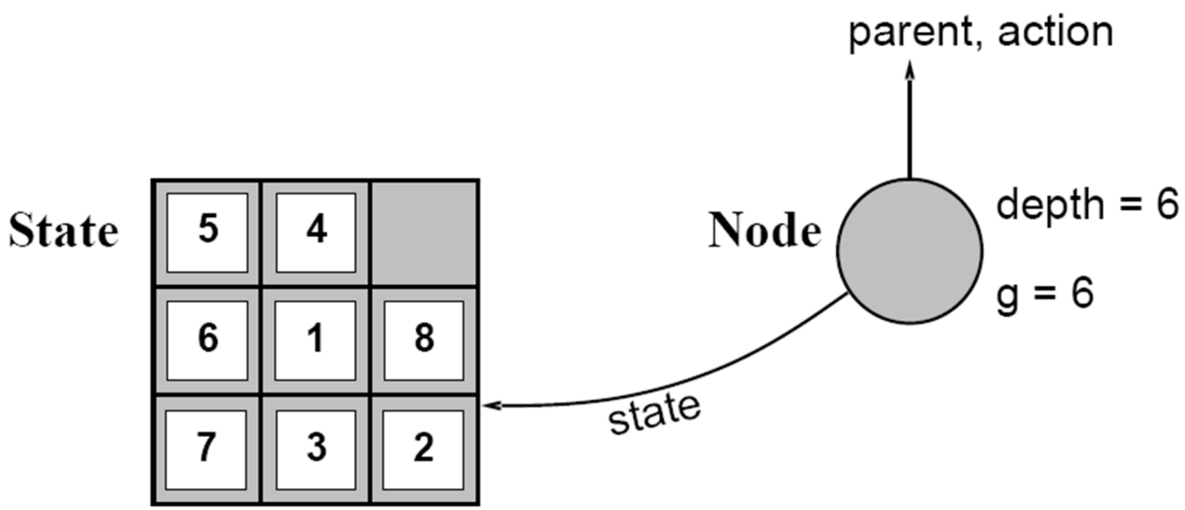
➔ **expanded** = {Arad, Sibiu}

➔ **fringe** = {Arad, Fagaras, Oradea, Rimnicu, Timisoara, Zerind}

Implementation of Search Nodes

Node = data structure with 5 components:

- **State:** the state in state space to which the node corresponds
- **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 $g(n)$ of the path from the initial state to the node n (the path is given by the parent pointers)
- **Depth:** the number of steps along the path from the initial node



Search Strategy =
Specific way of selecting next node to expand (from fringe)

Implement fringe as a **queue** data structure

Operations on a queue:

- Make-Queue(element,...)
- Empty?(queue)
- First(queue)
- Remove-First(queue)
- Insert(element, queue)
- Insert-All(element, ..., queue)

this is the operation that distinguishes different types of queues ..

A Generic Tree Search Algorithm

```
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-FRONT(fringe)
       if GOAL-TEST(problem, STATE(node)) then return node
       fringe ← INSERTALL(EXPAND(node, problem), fringe)
```


Evaluation Criteria for Search Strategies

1. **Completeness:**
   Is the algorithm guaranteed to find a solution if one exists?

2. **Optimality:**
   Is the strategy guaranteed to find the *optimal* solution?

3. **Time Complexity:**
   How long does it take to find a solution?

4. **Space Complexity:**
   How much memory is needed to perform the search?

A Short Digression: Computational Complexity Analysis, O(·) Notation, NP Hard Problems

Complexity Analysis of Algorithms:

Most important questions:

1. How long does algorithm A take to solve problem P?
   („time complexity“)

2. How much space (memory) does algorithm A require to solve P?
   („space complexity“)

Complexity Analysis of Algorithms

Two approaches to determining time and space complexity:

1. **Benchmarking** – run the algorithm on problem P (a specific implementation, on a specific machine, given a specific input) and measure the time taken (ms) and space used ($bytes$)

   ➔ only very specific information

2. **Asymptotic Analysis** – abstract mathematical analysis of how many „computation steps“ (time) and „storage units“ (space) the algorithm will (roughly) need, given not a specific input, but any input of „size“ n

   ➔ independent of implementation details
   ➔ independent of specific input – tells us how algorithm behaves on a whole *class* of problems (e.g., „sorting lists of length n“)
   ➔ tells us how computation costs grow with problem size

Asymptotic Analysis

function SUMMATION(*sequence*) **return** a number
    $sum \leftarrow 0$
    for $i \leftarrow 1$ **to** LENGTH(*sequence*)
        $sum \leftarrow sum + sequence[i]$
    return *sum*

Step 1: abstract over the input:
find a parameter that characterises the size of the input
Example: n = length of the sequence

Step 2: abstract over the algorithm:
find some (implementation-independent) measure that reflects the running time of the algorithm
Example: number of lines of code executed
(or number of additions performed)

Step 3: analyse the algorithm to find a formula T that computes the total number of steps taken by the algorithm as a function of the size of the input
Example: $T(n) = 2n + 2$ (lines of code)

Asymptotic Analysis

Problems with exact analysis:

1. precise number of steps may be different for different inputs
   ➔ usually only possible to compute worst case (T_{worst}) or average case complexity (T_{avg})
2. often impossible to perform exact analysis of the algorithm
   ➔ use **„order approximation“:** „SUMMATION algorithm is $O(n)$“
   Meaning: „its measure $T(n)$ is at most some constant factor times n (with the possible exception of a few small values of n)“

Def.: Order of a function (complexity in „Big O Notation“):

$T(n)$ is $O(f(n))$ if $T(n) \leq k \cdot f(n)$ for some constant k, for all $n > n_0$

Ex.: QUICKSORT is $O(n^2)$ in the worst case, and $O(n \log n)$ in the average case

This is called **asymptotic analysis** (tells us how T grows as n grows)

Complexity Analysis of Problems

Problem complexity analysis:

- analyses the complexity of problems / problem classes (rather than specific algorithms)
- independent of a specific algorithm (i.e., no algorithm can do better in the worst case)

Examples:

- computing a random number between 0 and n is $O(1)$
- finding a word in a dictionary of n words is $O(\log n)$
- computing the maximum in a list of n numbers is $O(n)$
- sorting a list of n numbers is $O(n^2)$
- determining whether two graphs with n vertices are isomorphic is $O(c^n)$ (for some constant c)

Commonly Encountered Orders of Functions / Complexities

notation	name
$O(1)$	constant
$O(\log n)$	logarithmic
$O([\log n]^c)$	polylogarithmic
$O(n)$	linear
$O(n \log n)$	„linearithmic“, quasilinear, supralinear
$O(n^2)$	quadratic
$O(n^c), c > 1$	polynomial, sometimes called „algebraic“
$O(c^n)$	exponential, sometimes called „geometric“
$O(n!)$	factorial

„tractable“

„intractable“

Problem Classes in Algorithmic Complexity Theory

P – The class of polynomial problems:

- solvable in time polynomial in n – $O(n^c)$ for some constant c
- considered „easy“ or „solvable“ („tractable“)
  -- contains those problems with running times like $O(n)$ or $O(n \log n)$
  -- but also those with complexity $O(n^{1000})$ ...

NP – The class of non-deterministic polynomial problems:

- definition: a problem is in NP if there is some algorithm that can guess a solution and then verify, in polynomial time, whether the solution is correct
- general belief in complexity theory (though still not proved):
  - $NP \neq P$
  - NP contains many problems with exponential complexity ($\geq O(c^n)$)
- exponential problems are intractable for even medium-sized inputs

Unfortunately, many of the interesting problems are in NP ...

Evaluation Criteria for Search Strategies

1. **Completeness:**
   Is the algorithm guaranteed to find a solution when one exists?

2. **Optimality:**
   Is the strategy guaranteed to find the *optimal* solution?

3. **Time Complexity:**
   How long does it take to find a solution?

4. **Space Complexity:**
   How much memory is needed to perform the search?

➔ *Problem Size* is expressed in terms of 3 quantities:

b ... branching factor (max. number of successors of any node)
d ... depth of the shallowest goal node (= length of minimum length solution)
m ... maximum length of any path in the state space (may be infinite)

Time and space are measured in terms of the number of nodes generated

Five Uninformed (Blind) Search Strategies

- Breadth-First Search
- Uniform-Cost Search
- Depth-First Search (Backtracking Search)
- Depth-limited Search
- Iterative Deepening Search

“uninformed” / “blind”:

- Search strategy has no understanding of the “meaning” of states
- All it can do is generate successors and recognise goal states

Alternative: **“informed” / “heuristic”** search:

- Search strategy knows (can estimate) which non-goal states are “more promising” than others
- Will be covered in next lecture …

Five Uninformed (Blind) Search Strategies

… differ in how they instantiate the generic tree search algorithm:

➔ which node is expanded next?

= how is the INSERTALL function realised?

```
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-FRONT(fringe)
        if GOAL-TEST(~~problem, STATE(node)) then return~~ node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Question: Why not test for goal immediately after generating a node?

➔ Will understand later: because a better solution might still be found later (before node itself is due for expansion) – cf. uniform cost search etc.; makes no big difference in terms of asymptotic complexity

Breadth-First Search (BFS)

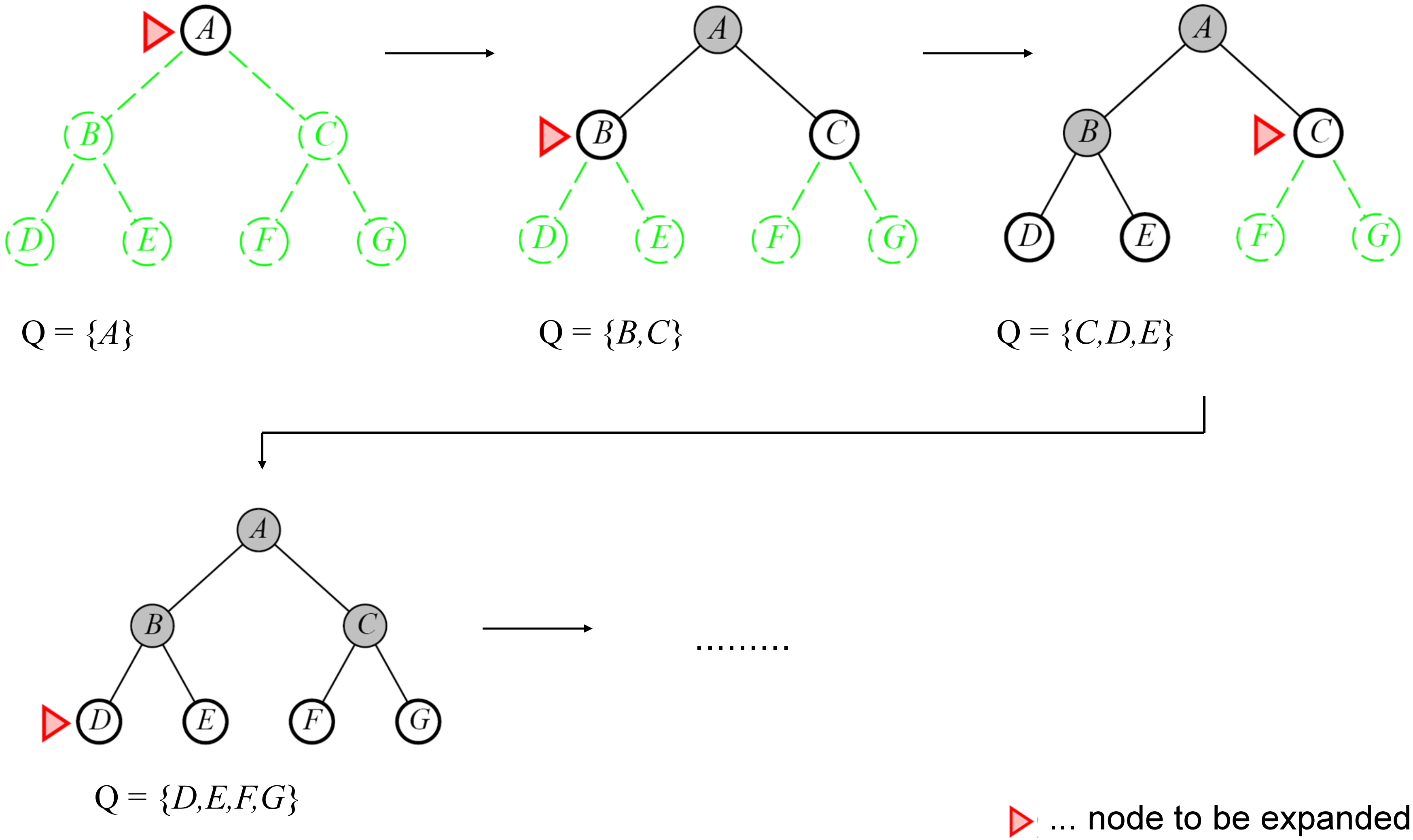
Basic strategy:

- Expand root node (i.e., initial state)
- then expand all successors of the root node
- then expand all their successors, etc.
- In general: expand all nodes at a given depth before expanding any nodes at the next level

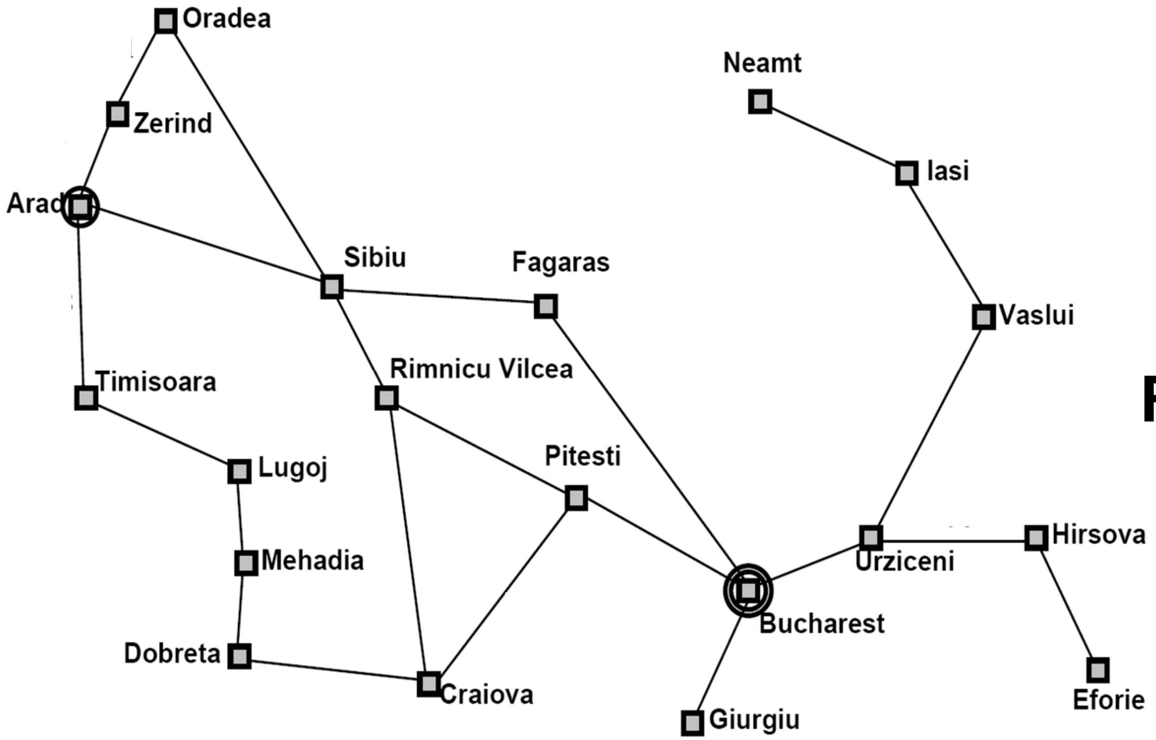
Realisation:

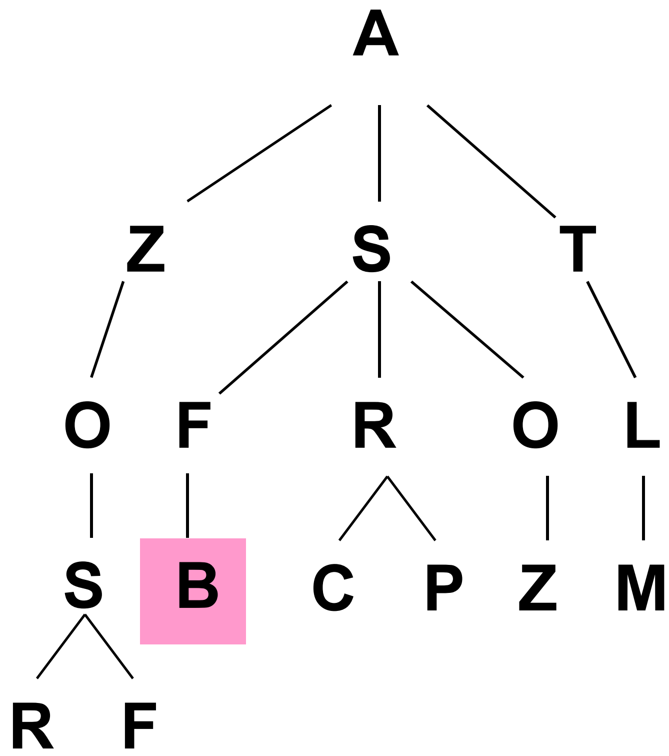
- Call TREE-SEARCH with an empty fringe that is a **first-in-first-out (FIFO) queue**
- FIFO puts all newly generated successors at the end of the queue
  ➔ nodes that are generated first will be expanded first

Breadth-First Search (BFS)

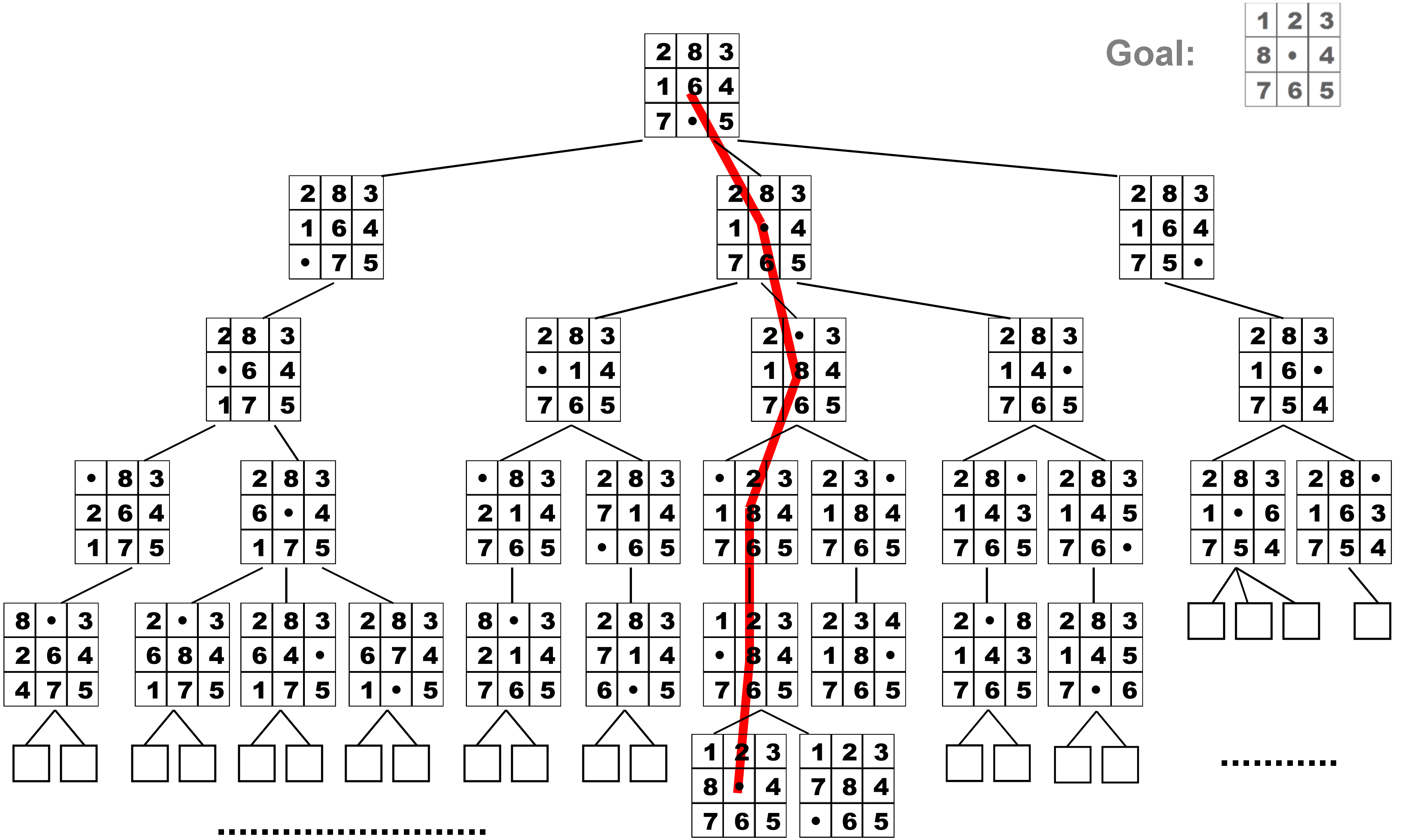


Breadth-First Search in Romania





Breadth-First Search: Solving the 8-Puzzle



Properties of Breadth-First Search

Completeness:

BFS is complete – if the shallowest goal node is at depth d, BFS will eventually find it (after expanding all nodes of depth ≤ d)

Optimality:

BFS is optimal if cost is constant (it first finds the solution with the shortest path); not optimal in general

Time Complexity (in terms of nodes generated):

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

Space Complexity:

Fringe must be kept in memory; at depth $d+1$, fringe size is $\leq b^{d+1}$

➔ space complexity is $O(b^{d+1})$

b ... branching factor (max. number of successors of any node)
d ... depth of the shallowest goal node (i.e., goal with minimum length path)
m ... maximum length of any path in the state space (may be infinite)

Time and Memory Requirements of BFS

Assumptions:

- branching factor $b = 10$
- compute $10{,}000$ nodes/sec
- 1000 bytes / node

$$\left(\sum_{l=1}^{d+1} b^l\right) - b$$

Solution depth d	Nodes generated	Time	Memory
2	1,100	0.11 seconds	1 megabyte
4	111,100	11 seconds	100 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	100 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Breadth-First with Costs: Uniform-Cost Search (UCS)

Generalisation of Breadth-First Search to deal with different action costs

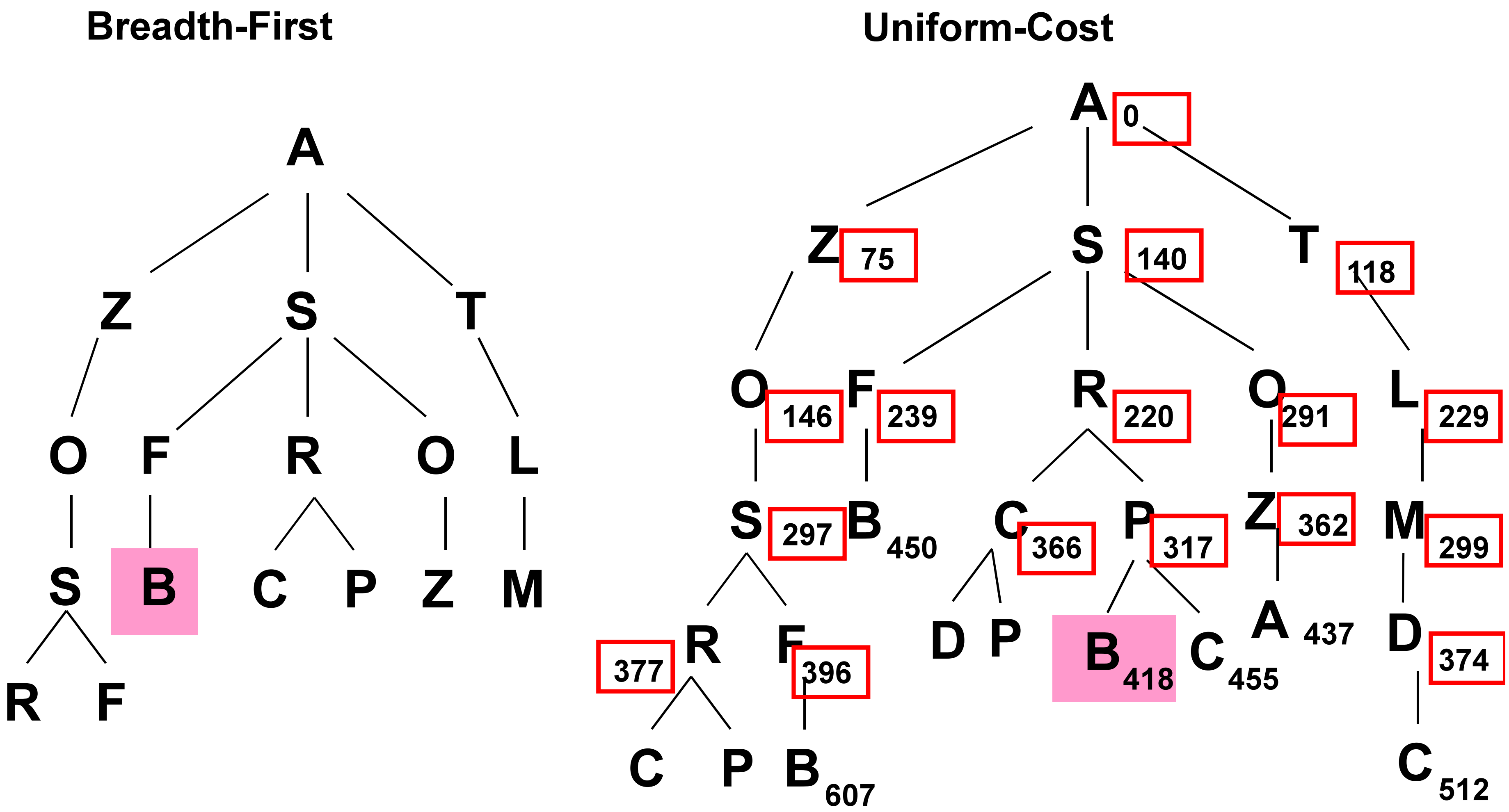
Basic strategy:

- Expand node with ***lowest path cost*** so far (i.e., cost from root to node) (instead of node with *shortest* path, as in BFS)
- Effect: finds optimal-cost solution first (instead of shortest-path solution)
- Equivalent to BFS if all costs are equal

Realisation:

- Fringe = ***priority queue*** sorted by **path cost** (lowest first)

Uniform-Cost Search in Romania



Properties of Uniform-Cost Search

Completeness: is complete ***if*** all actions have cost > 0
(a zero-cost action leading back to the same state (e.g., no-op) would lead to an infinite loop)

Optimality: optimal!

Time and Space Complexity:

- difficult to analyse in terms of d, because search is guided by cost rather than depths
- let C^* = cost of the optimal solution; assume each action costs at least ε

➔ worst-case time and space complexity is $O(b^{\lceil C^*/\varepsilon \rceil})$

➔ can be much greater than b^d

(because UCS can (and often does) explore large trees of cheap steps before exploring expensive and perhaps useful steps)

➔ when all steps have equal costs, then $O(b^{\lceil C^*/\varepsilon \rceil}) = O(b^d)$, of course...

Depth-First Search (DFS)

Basic strategy:

- Always expand the *deepest* node in the current fringe
- Expand a branch until either a solution is found or no more actions possible
- If no more successors (no more actions possible)
  ➔ backtrack to last choice point and expand alternative node

Realisation:

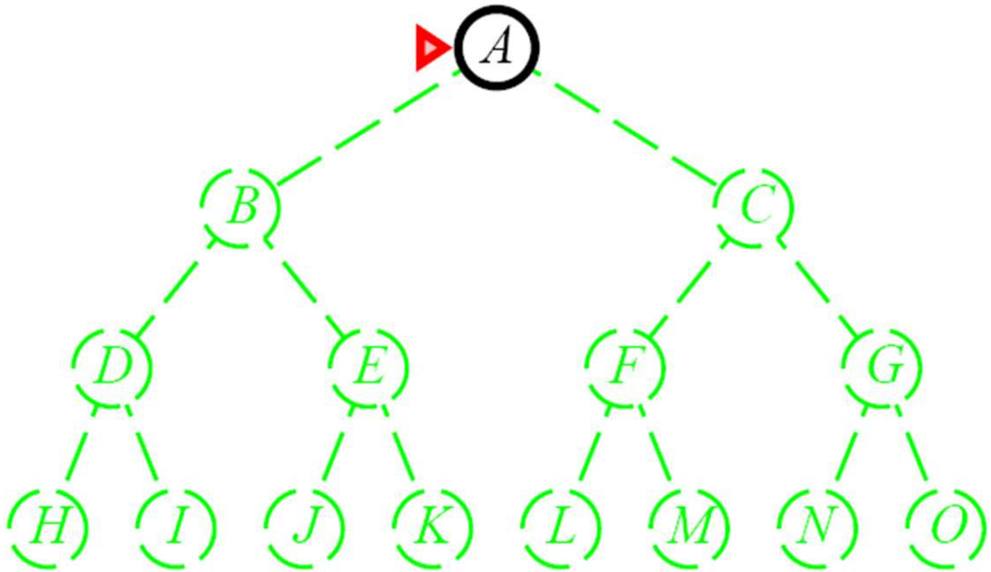
- Fringe = **LIFO (last-in-first-out) queue („stack“)**
  (i.e., newly expanded nodes are added to the beginning of the queue)

- Alternative implementation: recursive function that calls itself on each of its children in turn

Advantage:

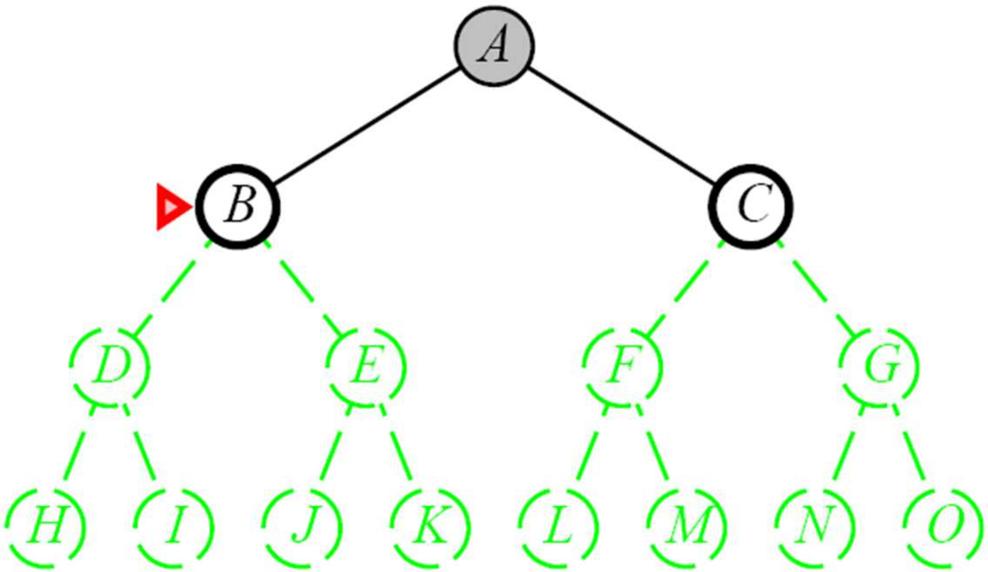
- Low memory requirements: needs to store only a single path at each moment, plus unexpanded siblings for each node on the path
- In other words: the fringe never ‚explodes‘ to exponential size

Depth-First Search



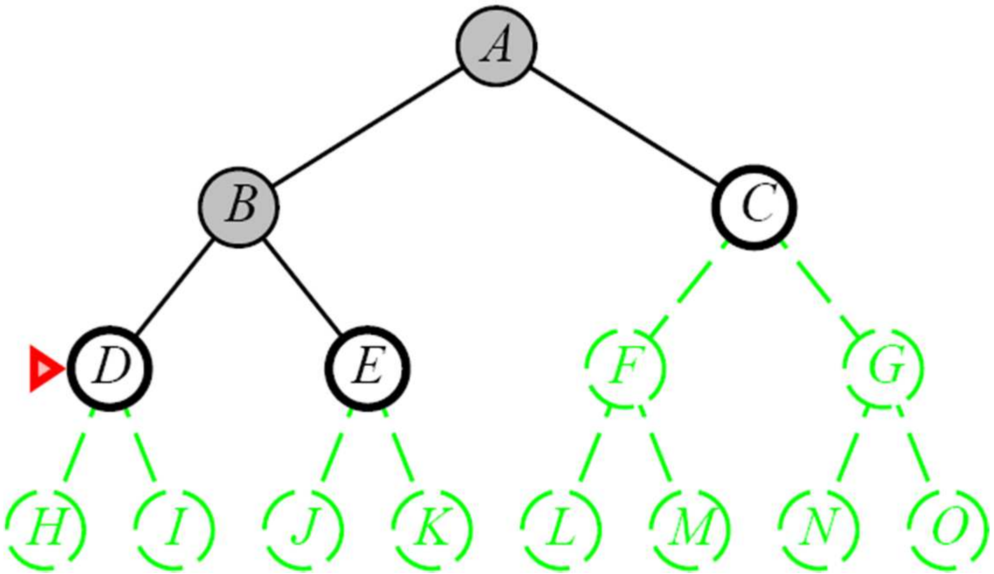
$Q = \{A\}$

Depth-First Search



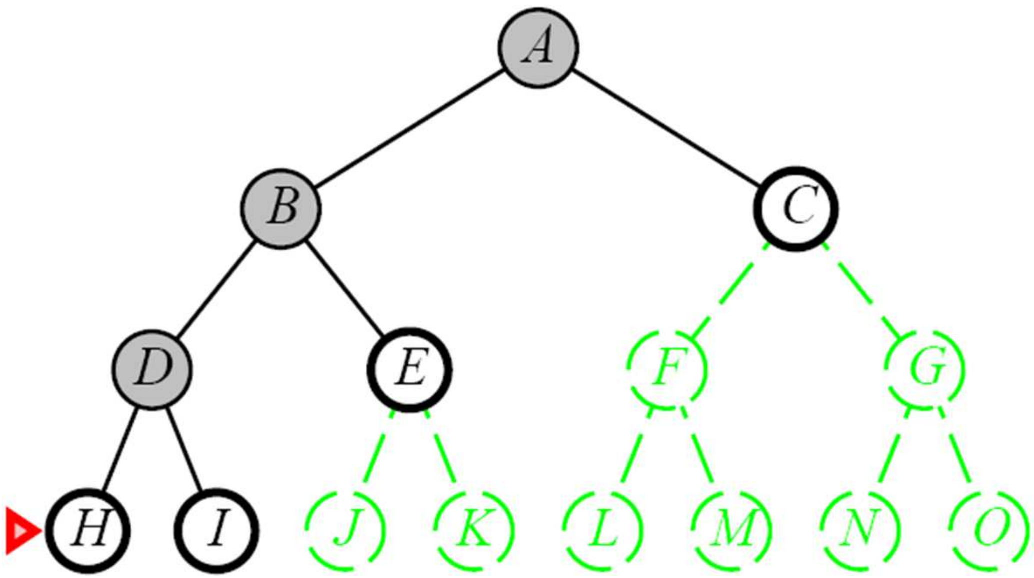
$Q = \{A\}$
$Q = \{B, C\}$

Depth-First Search



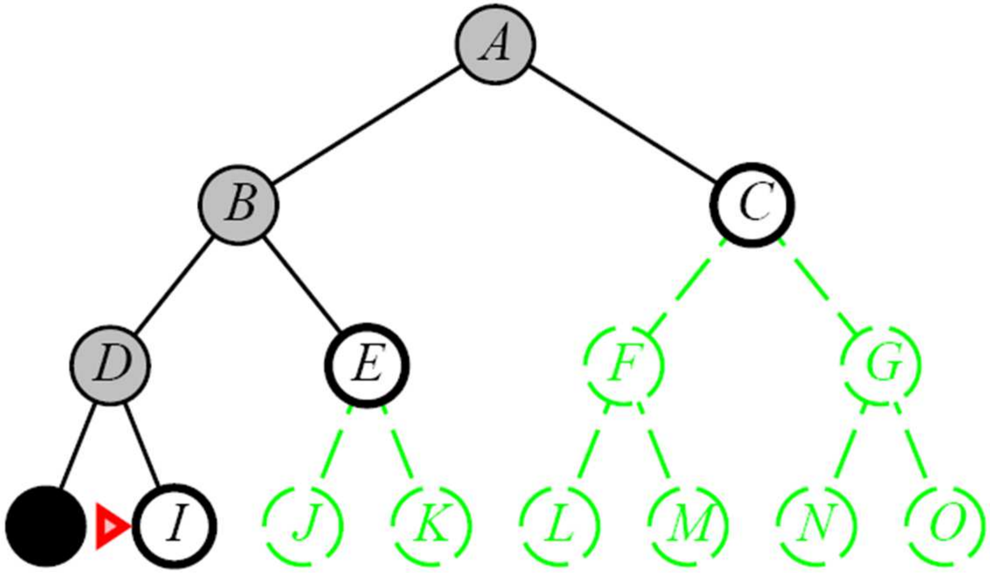
$Q = \{A\}$
$Q = \{B,C\}$
$Q = \{D,E,C\}$

Depth-First Search



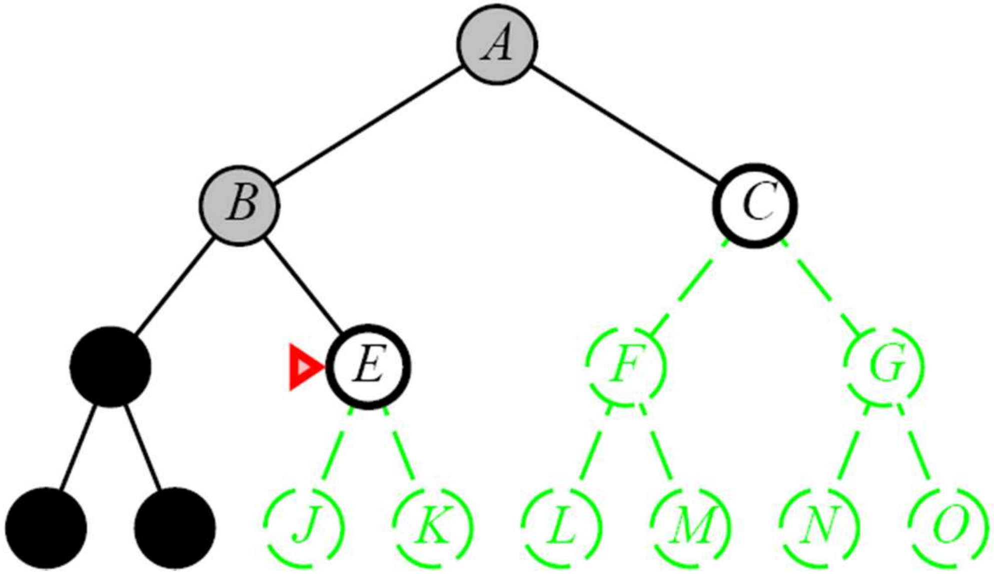
$Q = \{A\}$
$Q = \{B, C\}$
$Q = \{D, E, C\}$
$Q = \{H, I, E, C\}$

Depth-First Search



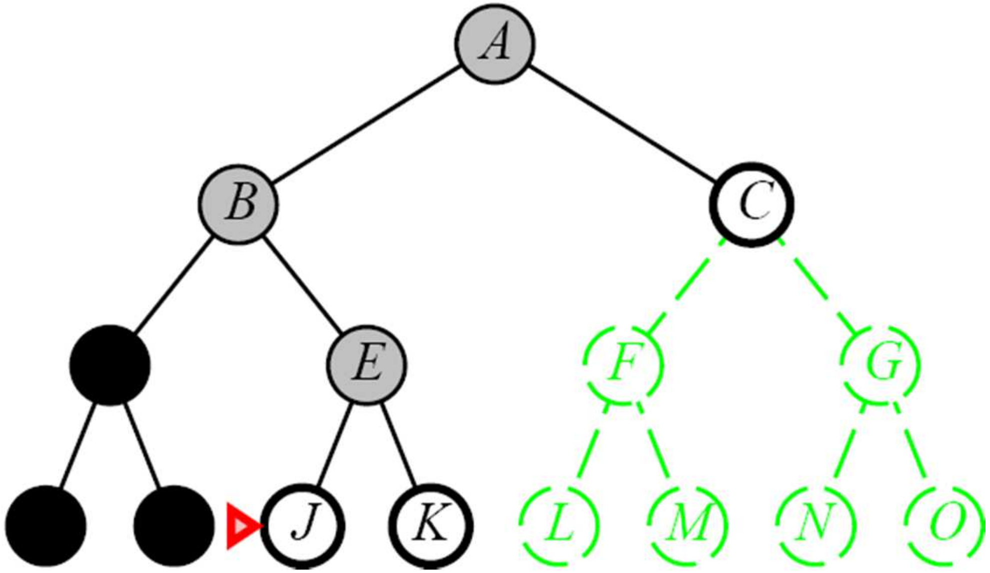
$Q = \{A\}$
$Q = \{B,C\}$
$Q = \{D,E,C\}$
$Q = \{H,I,E,C\}$
$Q = \{I,E,C\}$

Depth-First Search



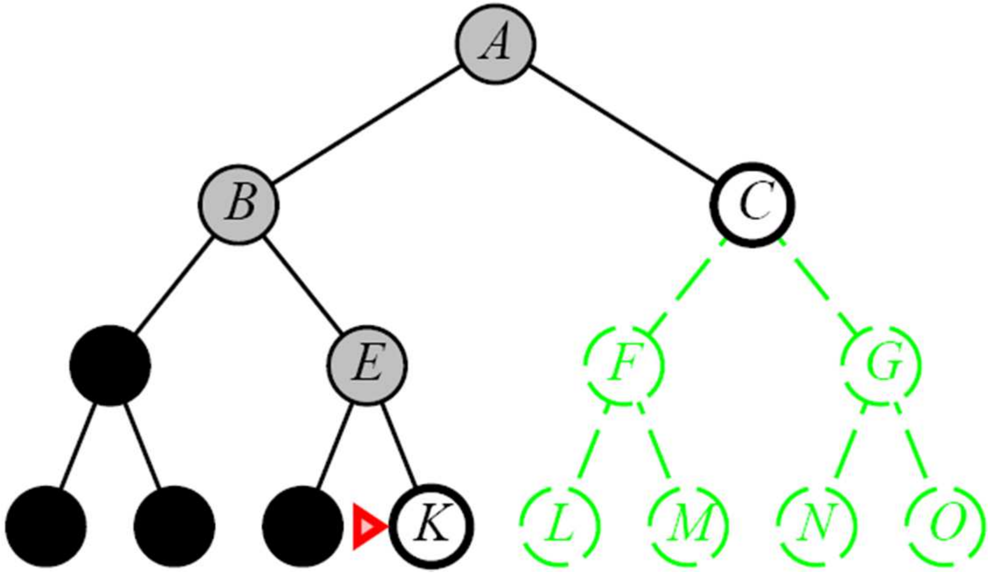
$Q = \{A\}$
$Q = \{B,C\}$
$Q = \{D,E,C\}$
$Q = \{H,I,E,C\}$
$Q = \{I,E,C\}$
$Q = \{E,C\}$

Depth-First Search



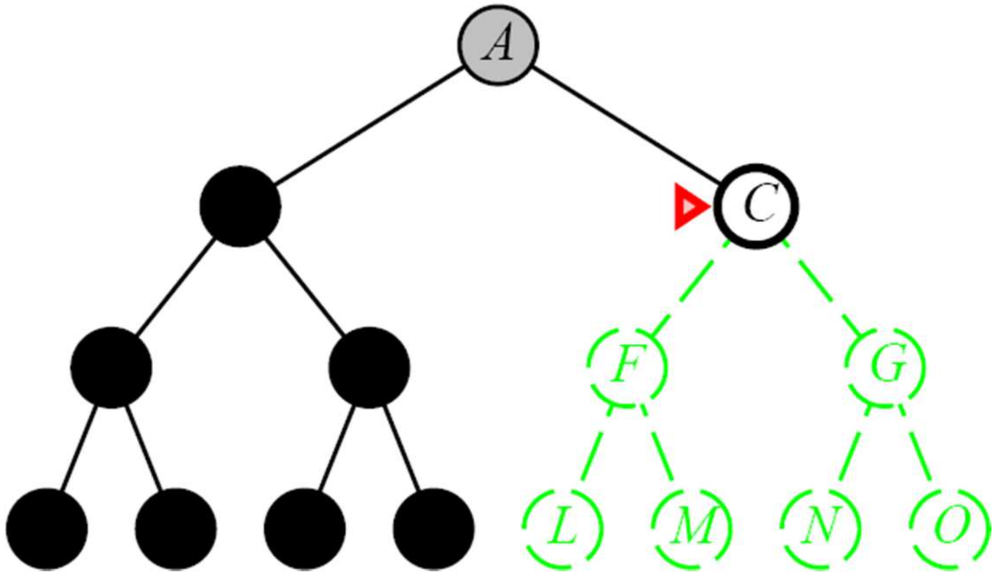
$Q = \{A\}$
$Q = \{B,C\}$
$Q = \{D,E,C\}$
$Q = \{H,I,E,C\}$
$Q = \{I,E,C\}$
$Q = \{E,C\}$
$Q = \{J,K,C\}$

Depth-First Search



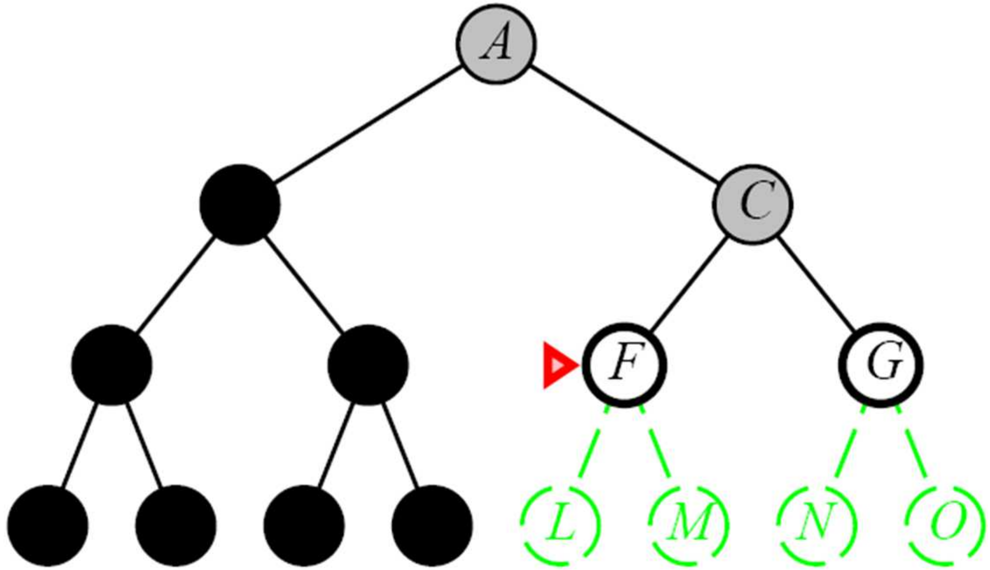
Q = $\{A\}$
Q = $\{B,C\}$
Q = $\{D,E,C\}$
Q = $\{H,I,E,C\}$
Q = $\{I,E,C\}$
Q = $\{E,C\}$
Q = $\{J,K,C\}$
Q = $\{K,C\}$

Depth-First Search



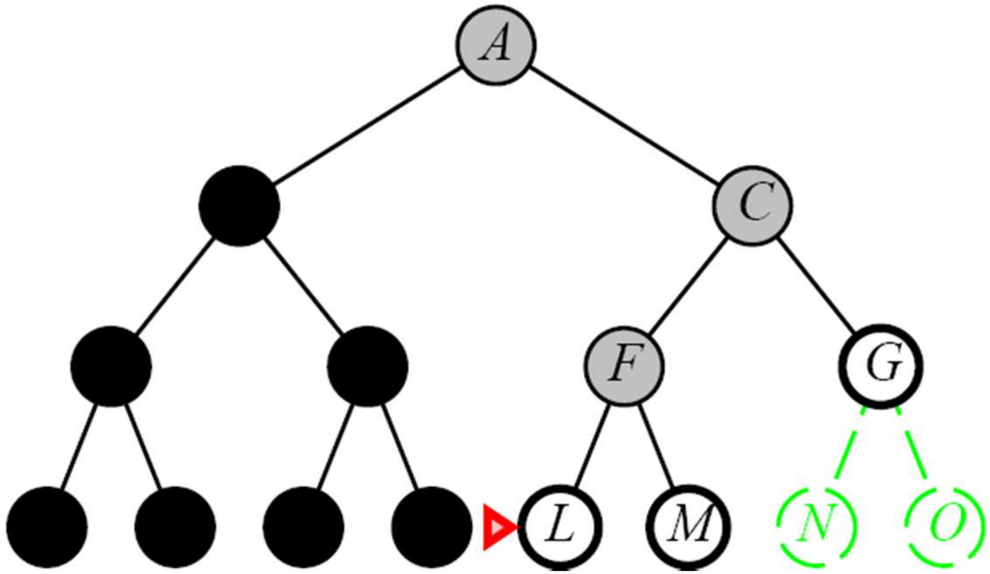
$Q = \{A\}$
$Q = \{B,C\}$
$Q = \{D,E,C\}$
$Q = \{H,I,E,C\}$
$Q = \{I,E,C\}$
$Q = \{E,C\}$
$Q = \{J,K,C\}$
$Q = \{K,C\}$
$Q = \{C\}$

Depth-First Search



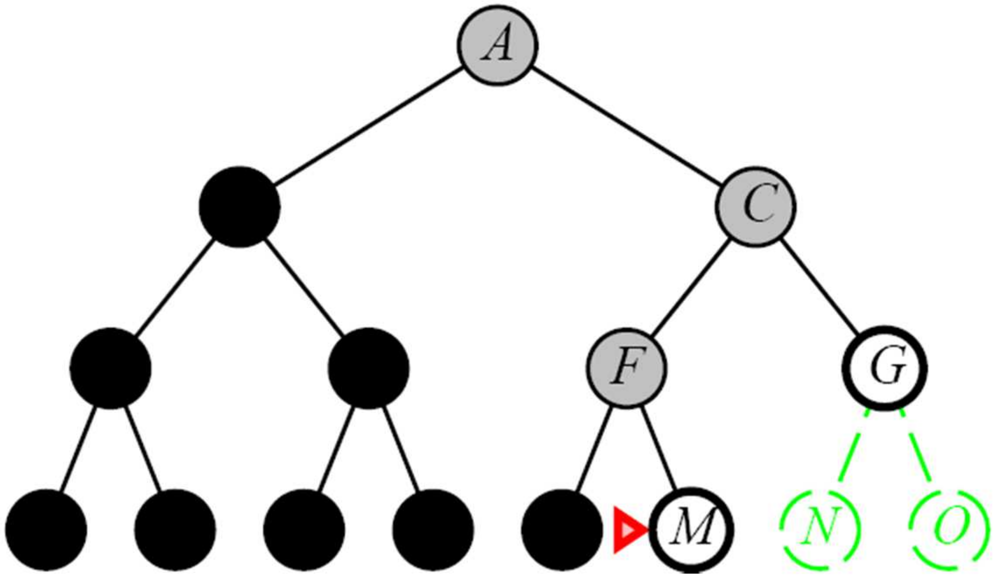
Q = $\{A\}$
Q = $\{B,C\}$
Q = $\{D,E,C\}$
Q = $\{H,I,E,C\}$
Q = $\{I,E,C\}$
Q = $\{E,C\}$
Q = $\{J,K,C\}$
Q = $\{K,C\}$
Q = $\{C\}$
Q = $\{F,G\}$

Depth-First Search



$Q = \{A\}$
$Q = \{B,C\}$
$Q = \{D,E,C\}$
$Q = \{H,I,E,C\}$
$Q = \{I,E,C\}$
$Q = \{E,C\}$
$Q = \{J,K,C\}$
$Q = \{K,C\}$
$Q = \{C\}$
$Q = \{F,G\}$
$Q = \{L,M,G\}$

Depth-First Search



$Q = \{A\}$
$Q = \{B,C\}$
$Q = \{D,E,C\}$
$Q = \{H,I,E,C\}$
$Q = \{I,E,C\}$
$Q = \{E,C\}$
$Q = \{J,K,C\}$
$Q = \{K,C\}$
$Q = \{C\}$
$Q = \{F,G\}$
$Q = \{L,M,G\}$
$Q = \{M,G\}$

Properties of Depth-First Search

Completeness:

DFS is ***not*** complete – can follow infinite paths (if $\boldsymbol{m} = \infty$) and miss solution (even if solution might be very close to root)

Optimality:

DFS is ***not*** optimal – will find „left-most“ solution, not shortest one

Time Complexity (in terms of nodes generated):

in the worst case, DFS generates entire search tree (if goal is in right-most path)

➔ time complexity is $O(b^m) >> O(b^d)$!!

Space Complexity:

only current branch + unexpanded nodes along branch need to be kept

➔ space complexity is $bm+1 = O(bm)$ ➔ linear space complexity!!

$\boldsymbol{b}$ ... branching factor (max. number of successors of any node)
$\boldsymbol{d}$ ... depth of the shallowest goal node (i.e., goal with minimum length path)
$\boldsymbol{m}$ ... maximum length of any path in the state space (may be infinite)

Depth-Limited Search

Goal:

- Fix DFS´s problem with unbounded trees (infinite paths)

Basic strategy:

- Use DFS, but with a **fixed depth limit** l
- All nodes at depth l are treated as if they had no successors
- Problem: how to choose an appropriate l ?

Effects:

- l is additional source of incompleteness (if we choose an $l < d$)
- Just like DFS, DLS is non-optimal (even if we choose $l > d$)
- Time complexity is $O(b^l)$
- Space complexity is $O(bl)$

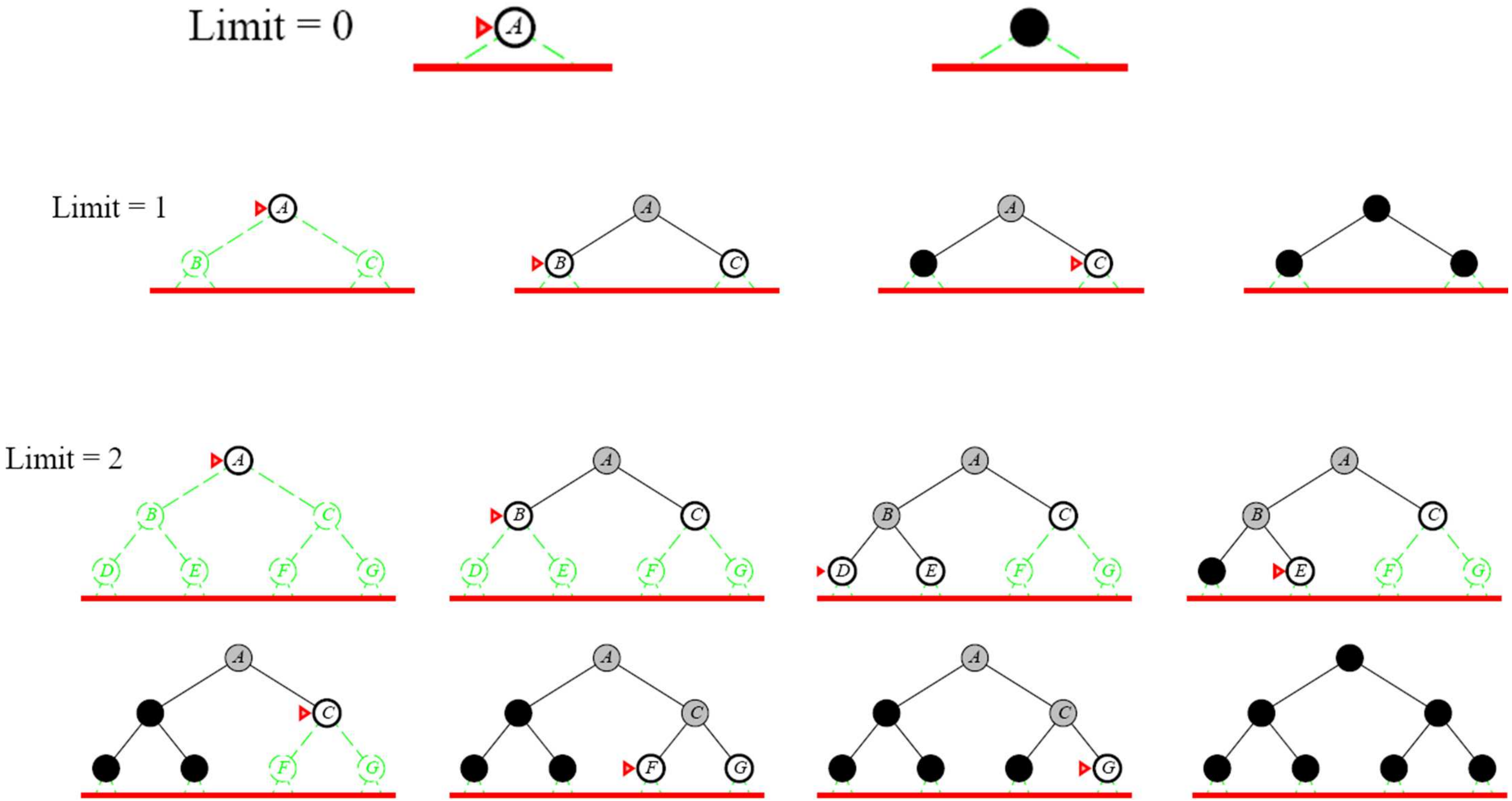
Iterative Deepening Depth-First Search

Basic idea:

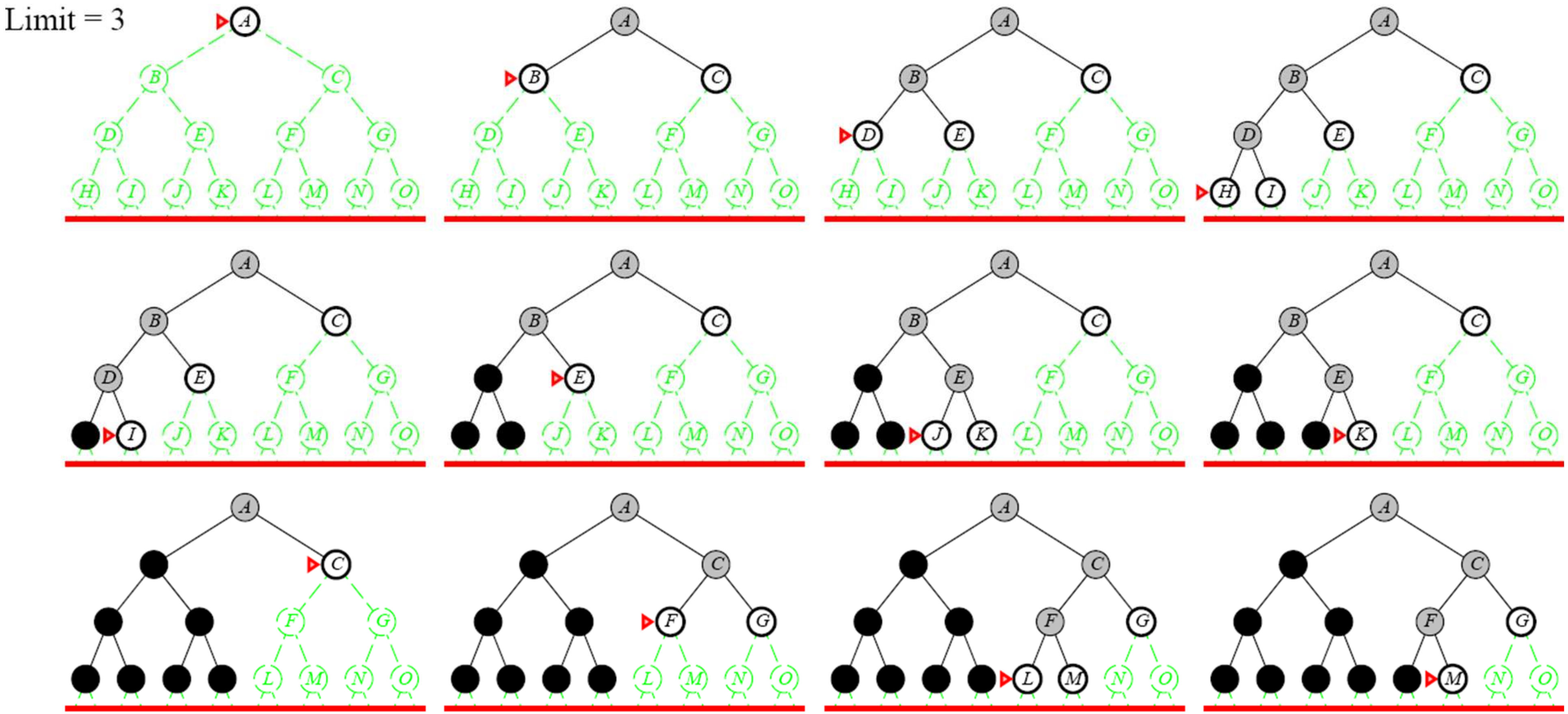
- Iterative strategy for finding the best depth limit l
- Repeatedly call Depth-limited DFS with depth limit $l = 0, 1, 2, \ldots,$ until a goal is found (when depth limit l reaches d)

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

Iterative Deepening Depth-First Search



Iterative Deepening Depth-First Search



Properties of Iterative Deepening Search

Iterative Deepening combines advantages of DFS and BFS:

Completeness: complete (like breadth-first search)

Optimality: optimal if costs are uniform (like breadth-first search)

Time Complexity (in terms of nodes generated):

$O(b^d)$ (analogous to breadth-first search)

Space Complexity:

$O(bd)$ (analogous to depth-first search)

b ... branching factor (max. number of successors of any node)
d ... depth of the shallowest goal node (i.e., goal with minimum length path)
m ... maximum length of any path in the state space (may be infinite)

Isn't Iterative Deepening Wasteful?

Apparent problem: All tree levels < d are generated multiple times!

But:

- Most of the nodes in a search tree (with a fixed branching factor b) are in the bottom level ($\sum_{i=1}^{d-1} b^i < b^d$)
- So it does not matter much if upper levels are generated several times

More precisely:

- In IDS, nodes at bottom level (depth d) are generated 1x, nodes at next-to-bottom level are generated 2x, ..., children of root are generated d x

➔ # nodes generated: $(d)b + (d-1)b^2 + \cdots + (1)b^d = O(b^d)$

➔ Time complexity of IDS is still $O(b^d)$

➔ In general, Iterative Deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is unknown

Complexity of Uninformed Search Methods: Summary

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

b ... branching factor (max. number of successors of any node)
d ... depth of the shallowest goal node (i.e., goal with minimum length path)
m ... maximum length of any path in the state space (may be infinite)
l ... depth limit

Avoiding Repeated States

Problem:

- In many applications, some states can be reached via different action sequences, or actions are reversible (e.g., route finding, 8-puzzle, ...)
  - that is, the state space is a *graph*, not a tree
- Consequence: duplication of nodes and subtrees in the search tree, cycles, infinite paths!

Solution:

- Keep track of all nodes already expanded („CLOSED list“)
- Compare newly generated node to nodes in CLOSED list
- Expand new node only if not equal to a node in CLOSED
  ➔ General graph search algorithm

Avoiding Repeated States: The General Graph-Search Algorithm

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

The General Graph-Search Algorithm

Problems:

- **Optimality** may be lost by simply discarding new, duplicated states (may have lower cost than previously found node with same state)
  - ➔ need to compare path costs
  - ➔ may need to revise path costs of descendants of kept node
  - ➔ things become more complex

- GRAPH-SEARCH must keep every node in memory (in *fringe* or *closed*)
  - ➔ depth-first and iterative deepening search no longer have linear space costs
  - ➔ back to **$O(b^d)$ space complexity** ...
  - ➔ many searches will be impossible because of memory limitations

Problems can sometimes be avoided by clever design of the search problem representation and state space!

