SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

19AMB303-FULL STACK AI

M.POORNIMA DEVI,AP/AIML

Searching for solutions

3.3 Searching for solutions

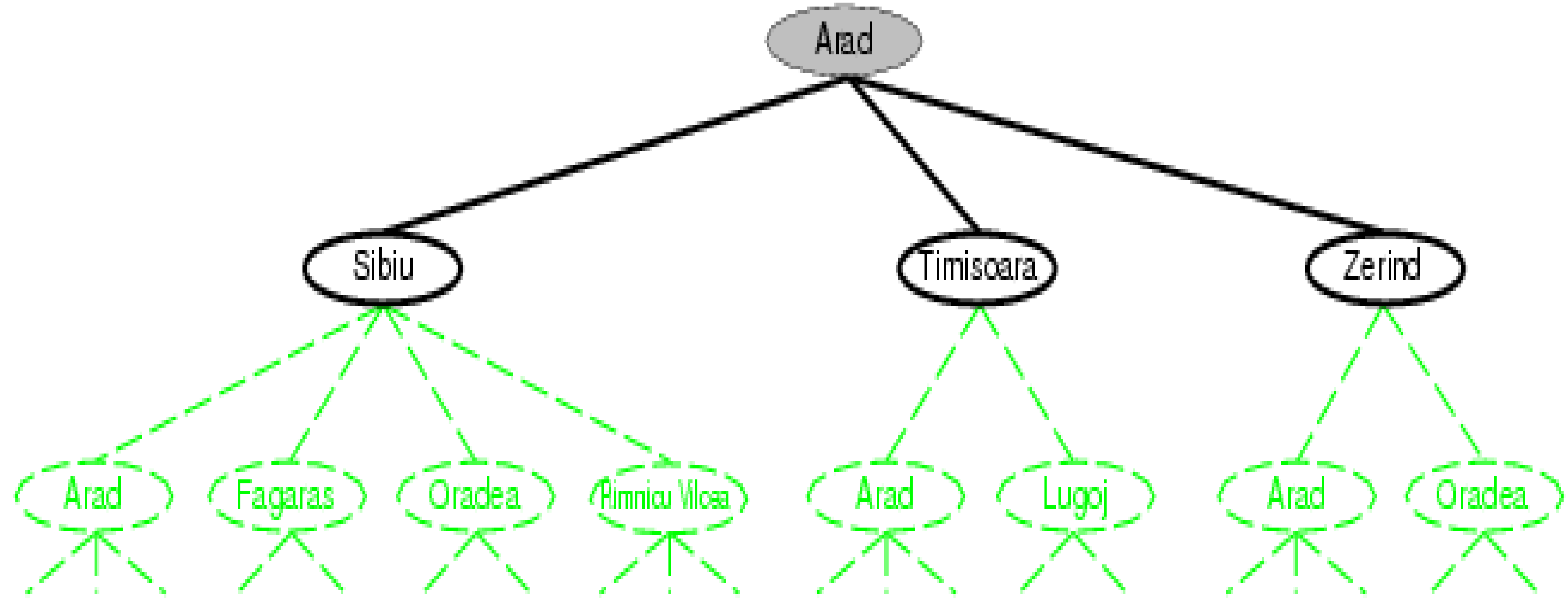
- Finding out a solution is done by
  - searching through the state space
- All problems are transformed
  - as a search tree
  - generated by the initial state and successor function

Search tree

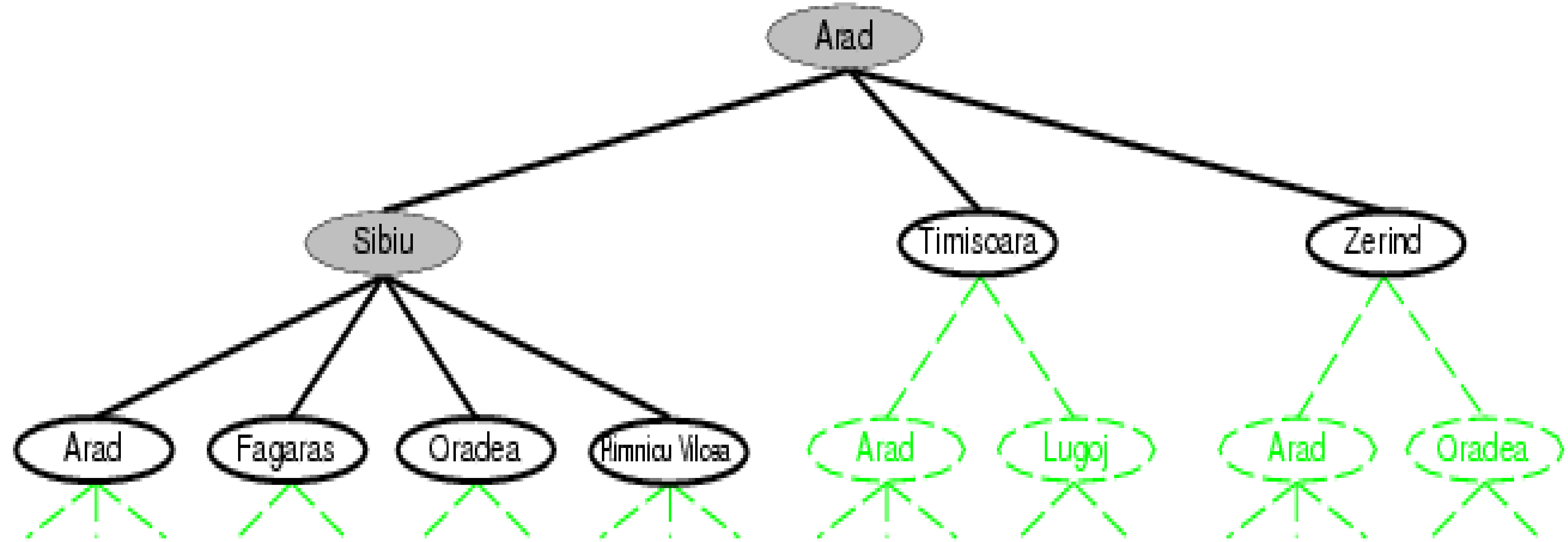
- **Initial state**
  - The root of the search tree is a **search node**
- **Expanding**
  - applying successor function to the current state
  - thereby generating a new set of states
- **leaf nodes**
  - the states having no successors

  Fringe : Set of search nodes that have not been expanded yet.
- Refer to next figure

Tree search example



Tree search example



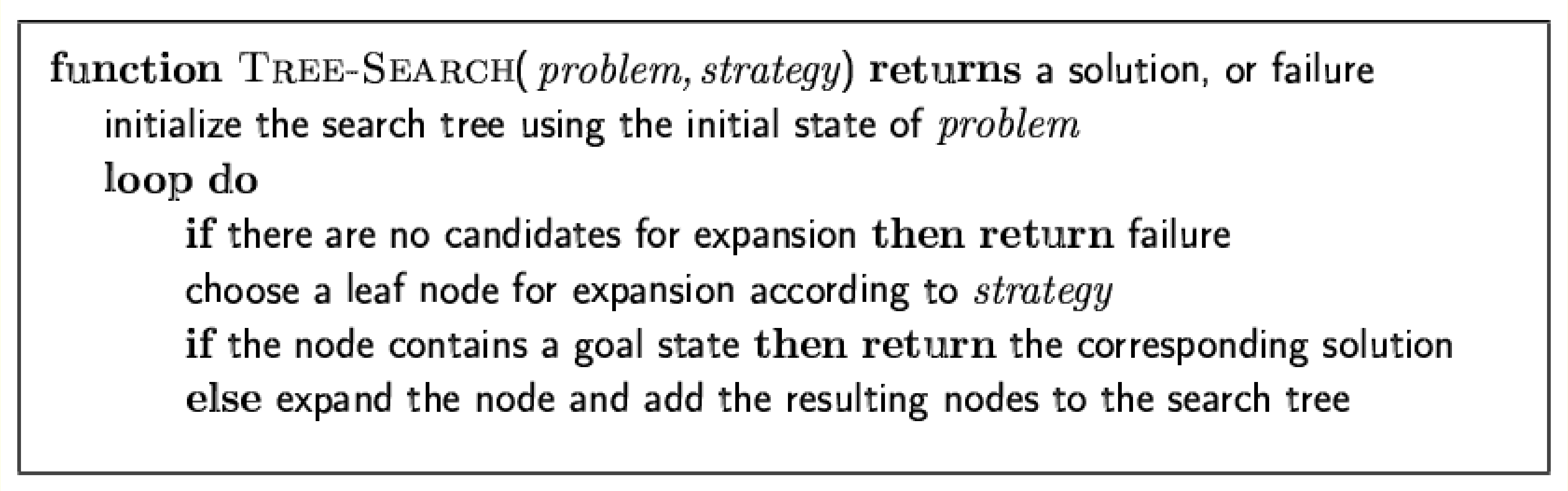
Search tree

- The **essence** of searching
  - in case the first choice is not correct
  - choosing one option and keep others for later inspection
- Hence we have the **search strategy**
  - which determines the choice of which state to expand
  - good choice → fewer work → faster
- Important:
  - state space ≠ search tree

Search tree

- A node is having five components:
  - STATE: which state it is in the state space
  - PARENT-NODE: from which node it is generated
  - ACTION: which action applied to its parent-node to generate it
  - PATH-COST: the cost, g(n), from initial state to the node n itself
  - DEPTH: number of steps along the path from the initial state

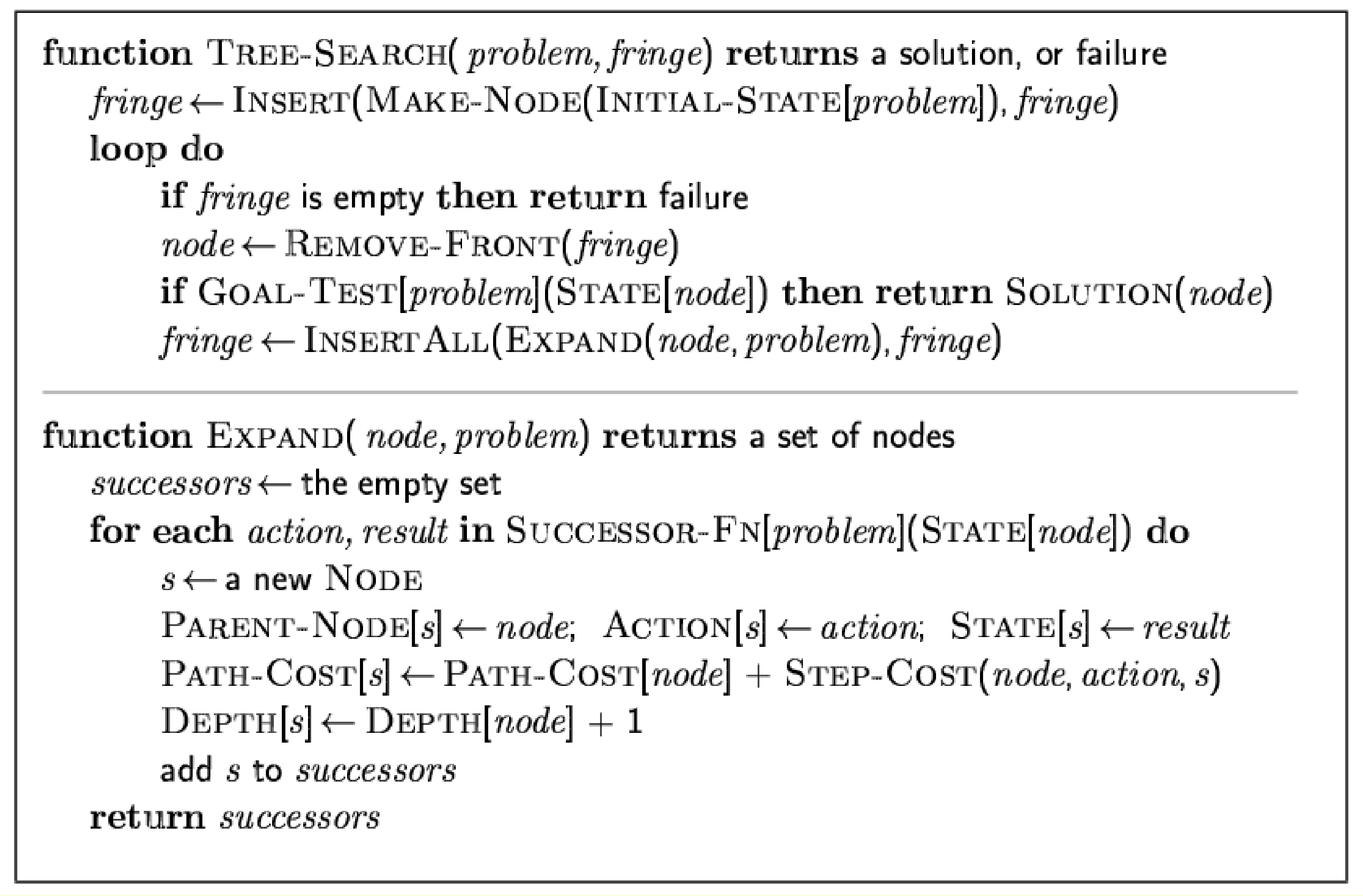
Search tree



```
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
```

Informal Description of Genearl search Algorithm

Search tree



```
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 SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;   ACTION[s] ← action;   STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

easuring problem-solving performance

- The evaluation of a search strategy
  - **Completeness:**
    - is the strategy guaranteed to find a solution when there is one?
  - **Optimality**:
    - does the strategy find the highest-quality solution when there are several different solutions?
  - **Time complexity**:
    - how long does it take to find a solution?
  - **Space complexity**:
    - how much memory is needed to perform the search?

Measuring problem-solving performance

- In AI, complexity is expressed in
  - **b**, branching factor, maximum number of successors of any node
  - **d**, the depth of the shallowest goal node.

  (depth of the least-cost solution)
  - **m**, the maximum length of any path in the state space
- Time and Space is measured in
  - number of nodes generated during the search
  - maximum number of nodes stored in memory

Measuring problem-solving performance

- For effectiveness of a search algorithm
  - we can just consider the total cost
  - **The total cost** = path cost (g) of the solution found **+** search cost
    - search cost = time necessary to find the solution
- Tradeoff:
  - (long time, optimal solution with least g)
  - vs. (shorter time, solution with slightly larger path cost g)

3.4 Uninformed search strategies

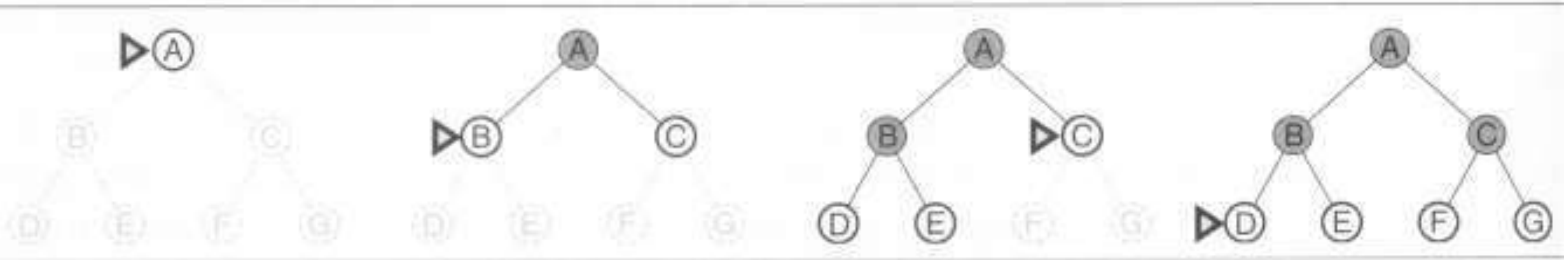
- **Uninformed search**
  - no information about the number of steps
  - or the path cost from the current state to the goal
  - search the state space **<u>blindly</u>**
- **Informed search, or heuristic search**
  - a cleverer strategy that searches toward the goal,
  - based on the information from the current state so far

Uninformed search strategies

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

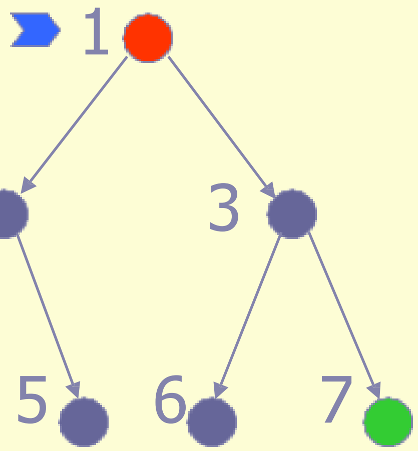
Breadth-first search

- The root node is expanded first (FIFO)
- All the nodes generated by the root node are then expanded
- And then their successors and so on



Breadth-First Strategy

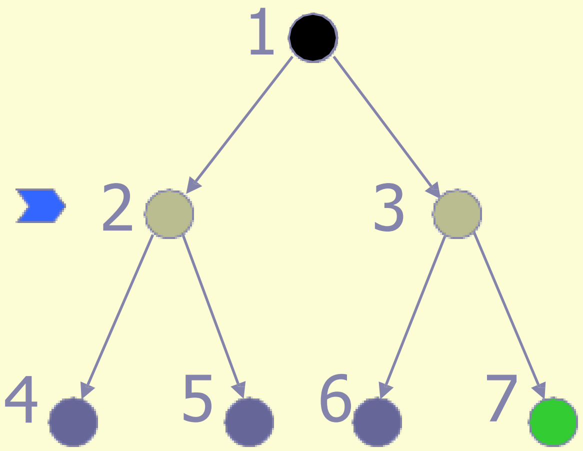
New nodes are inserted at the end of FRINGE



FRINGE = (1)

Breadth-First Strategy

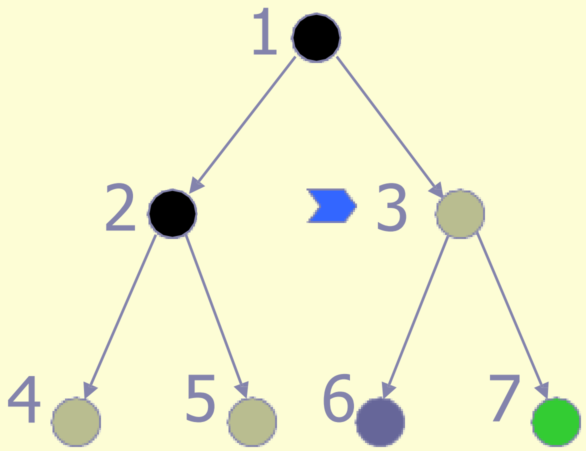
New nodes are inserted at the end of FRINGE



FRINGE = (2, 3)

Breadth-First Strategy

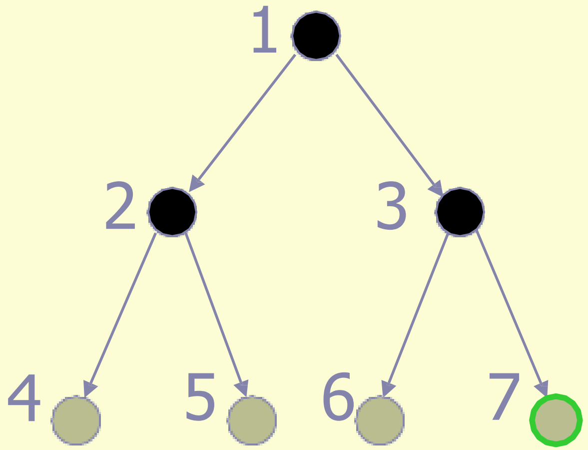
New nodes are inserted at the end of FRINGE



FRINGE = (3, 4, 5)

Breadth-First Strategy

New nodes are inserted at the end of FRINGE



FRINGE = (4, 5, 6, 7)

Breadth-first search (Analysis)

- Breadth-first search
  - Complete – find the solution eventually
  - Optimal, if step cost is 1

The disadvantage

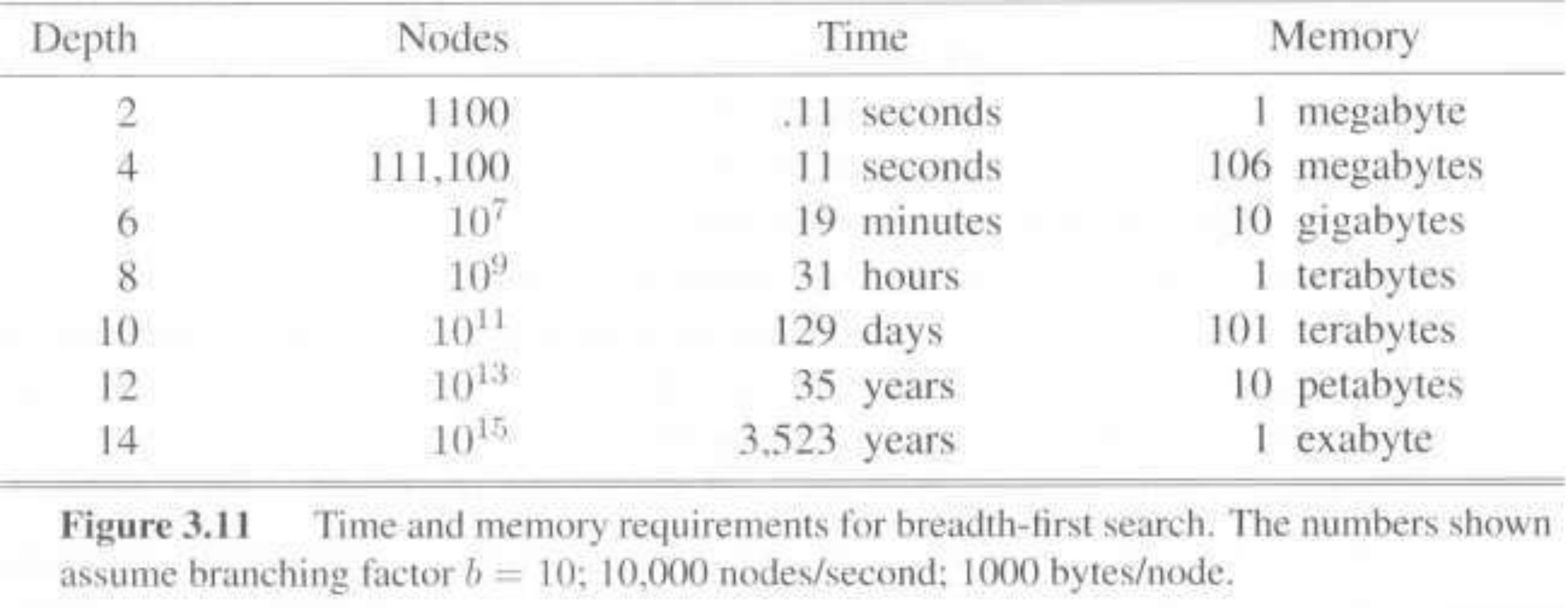
- if the branching factor of a node is large,
- for even small instances (e.g., chess)
  - the ***space complexity*** and the ***time complexity*** are enormous

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d = b(b^d-1) = O(b^{d+1})$
-
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
-
- Space is the bigger problem (more than time)
-

Breadth-first search (Analysis)

- assuming 10000 nodes can be processed per second, each with 1000 bytes of storage



Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

THANKYOU

SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A+’ Grade Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

19AMB303-FULL STACK AI

M.POORNIMA DEVI,AP/AIML

Uniform cost search

- **Breadth-first** finds the shallowest goal state
  - but not necessarily be the least-cost solution
  - work only if all step costs are equal
- **Uniform cost search**
  - modifies breadth-first strategy
    - by always expanding the lowest-cost node
  - The lowest-cost node is measured by the path cost *g(n)*

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? numbr of nodes with *g* ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? Number of nodes with *g* ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of *g(n)*

let

C be the cost of optimal solution.*

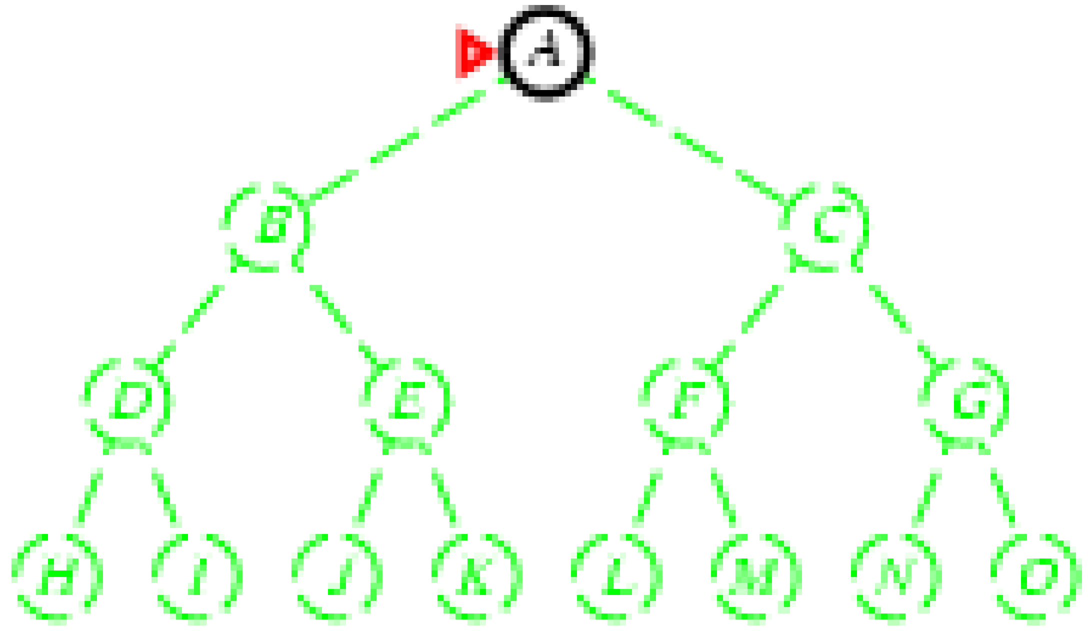
ε is possitive constant (every action cost)

Depth-first search

- Always expands one of the nodes at the ***deepest*** level of the tree
- Only when the search hits a dead end
  - goes back and expands nodes at shallower levels
  - Dead end → leaf nodes but not the goal
- Backtracking search
  - only one successor is generated on expansion
  - rather than all successors
  - fewer memory

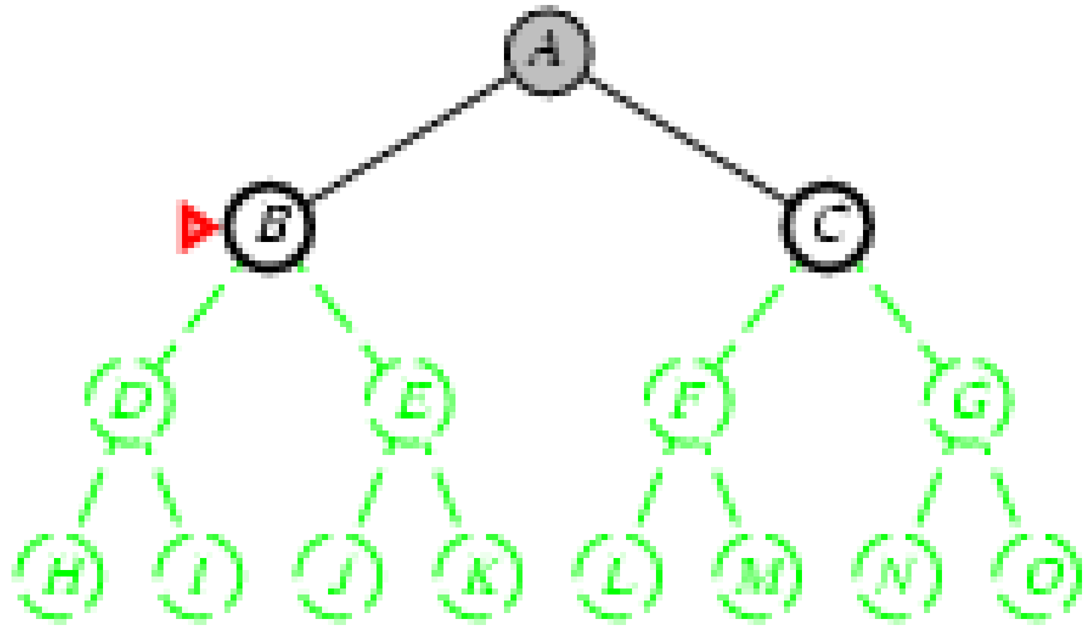
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



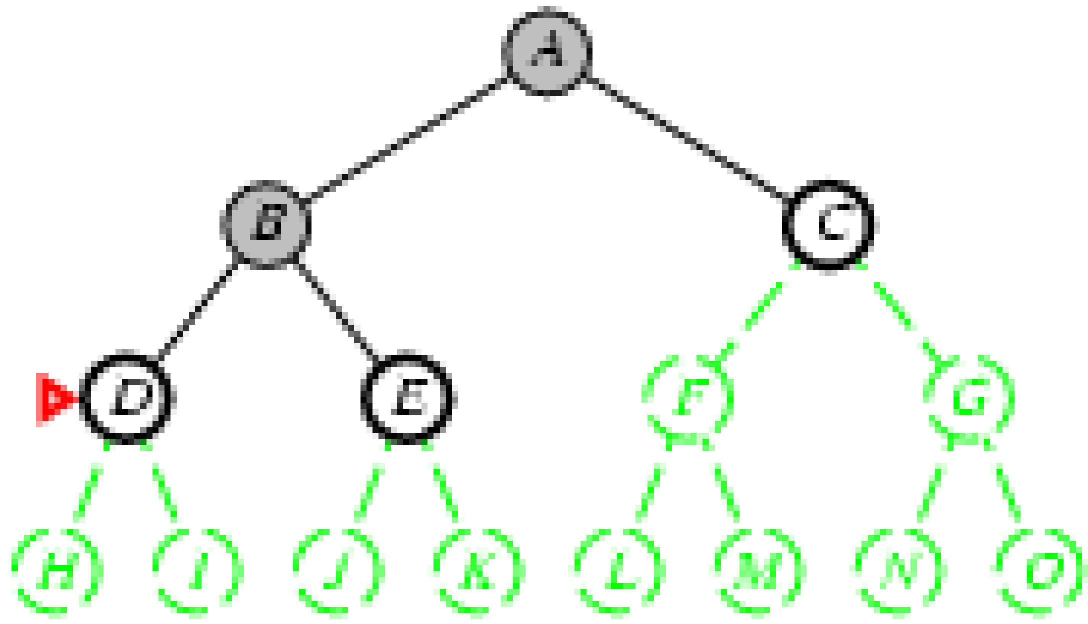
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

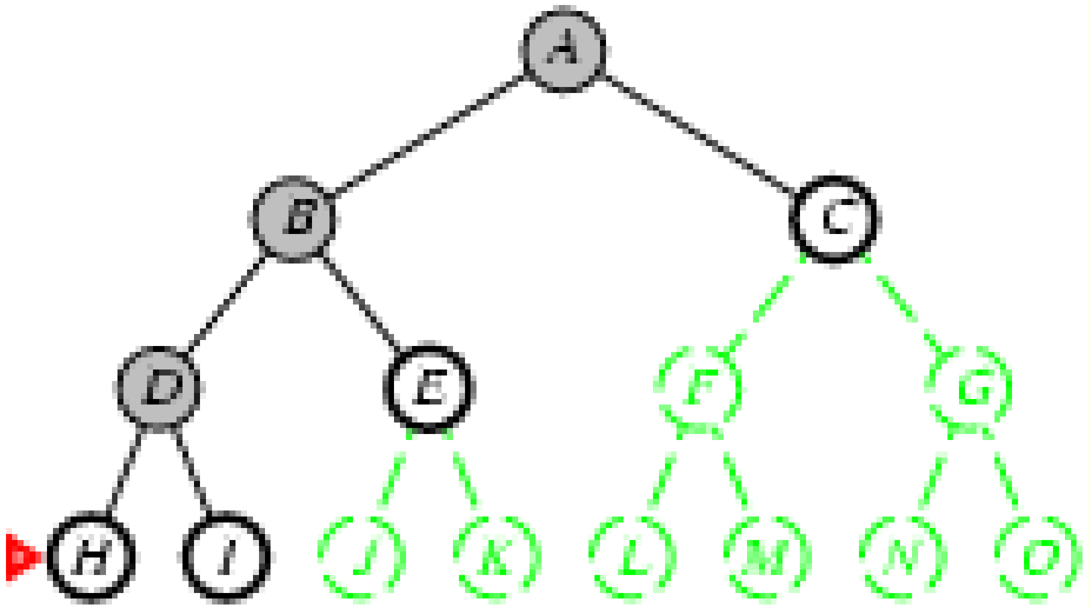


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

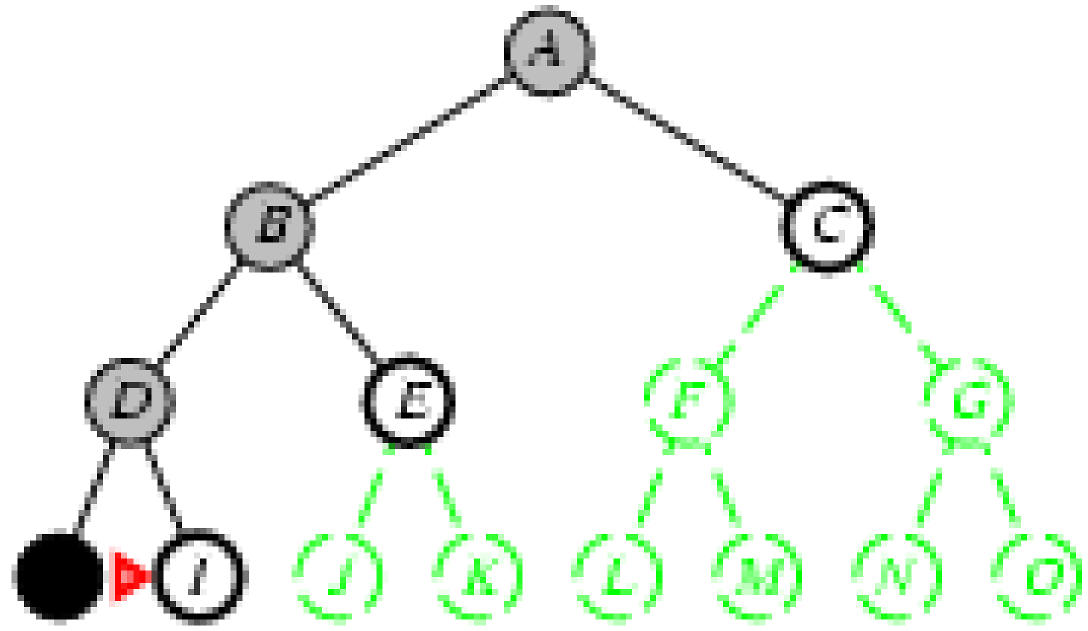


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  - 

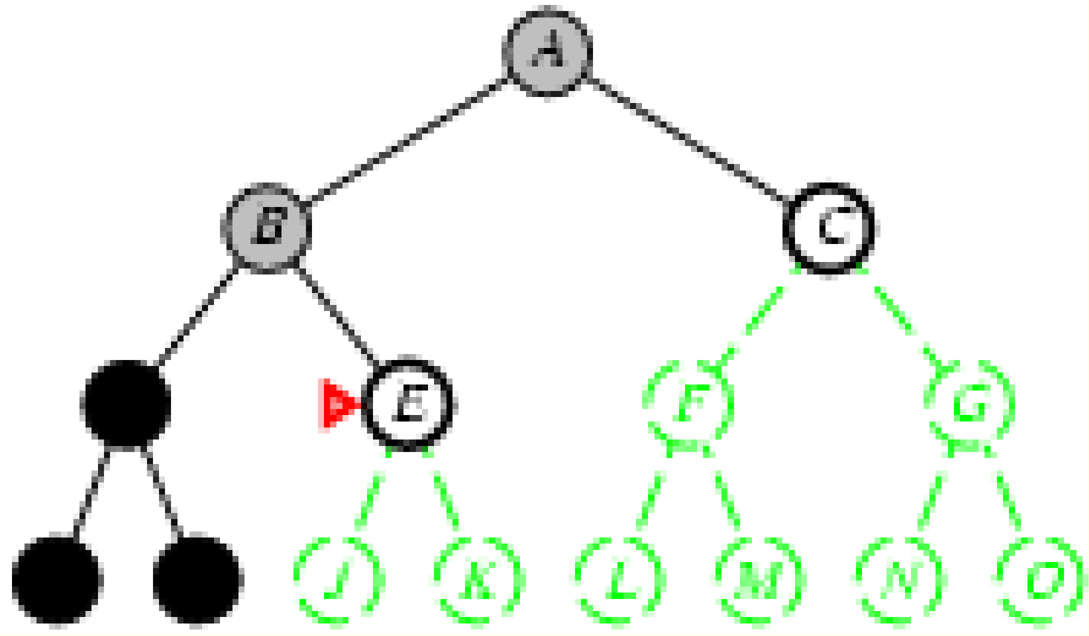
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

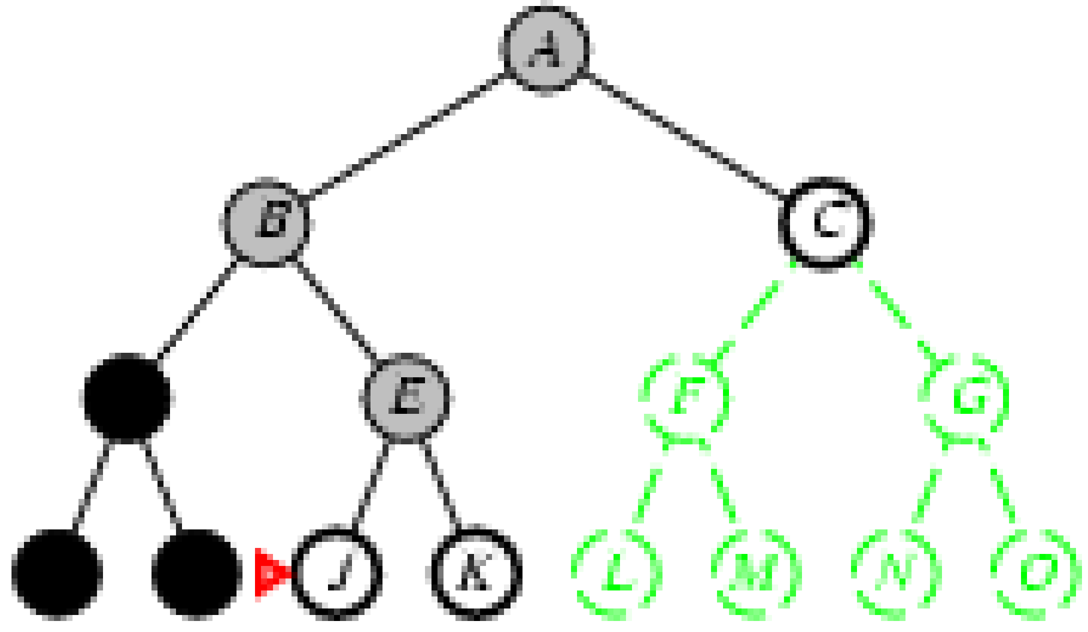


Depth-first search

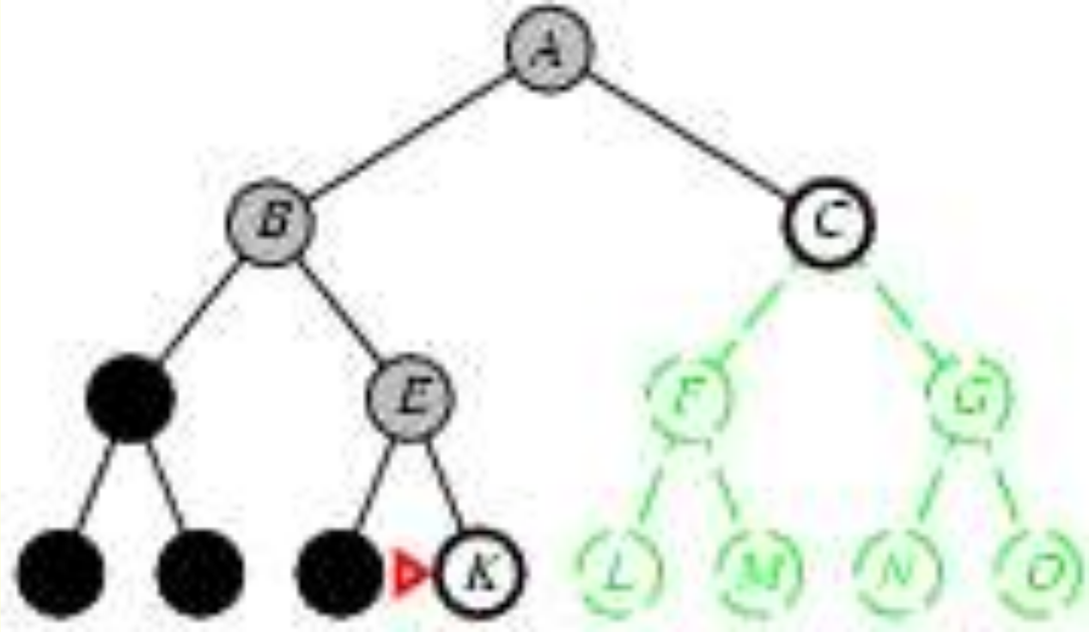
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

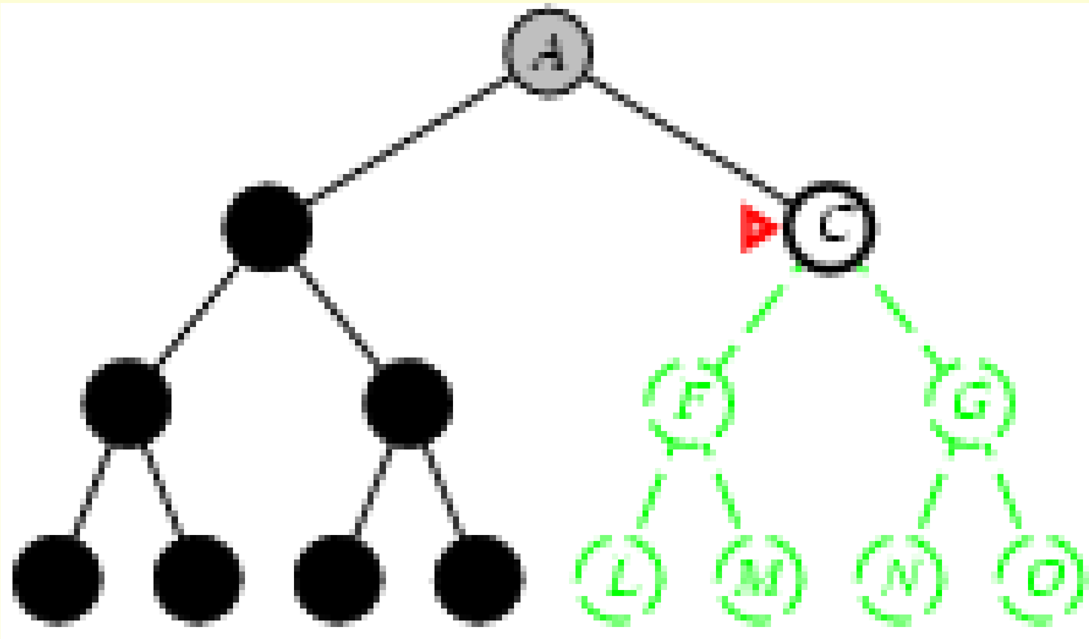
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  - 

Depth-first search

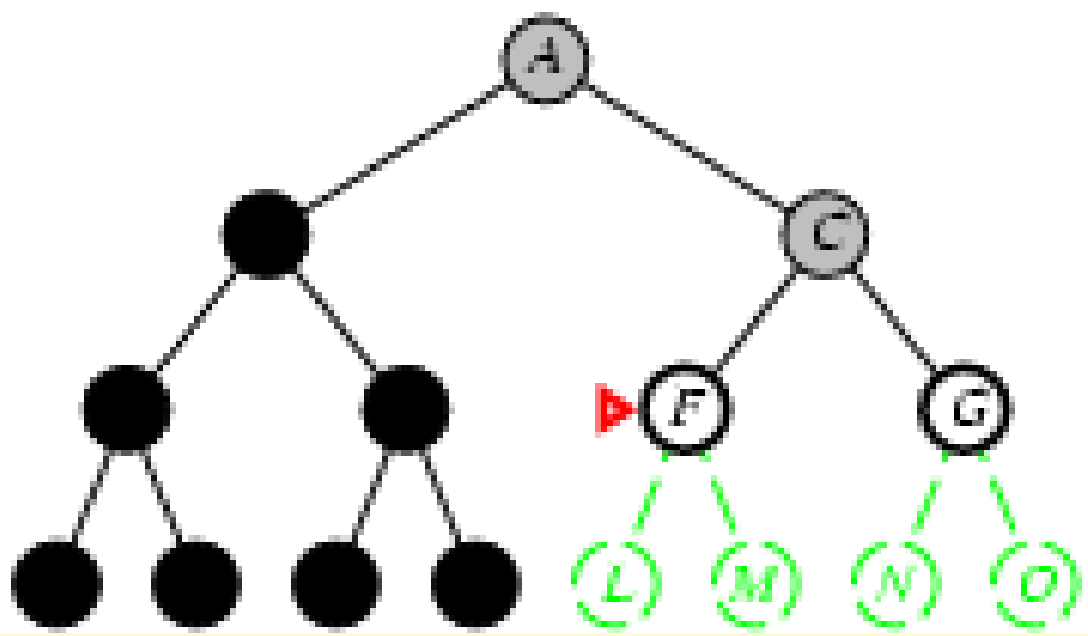
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  - 

Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -

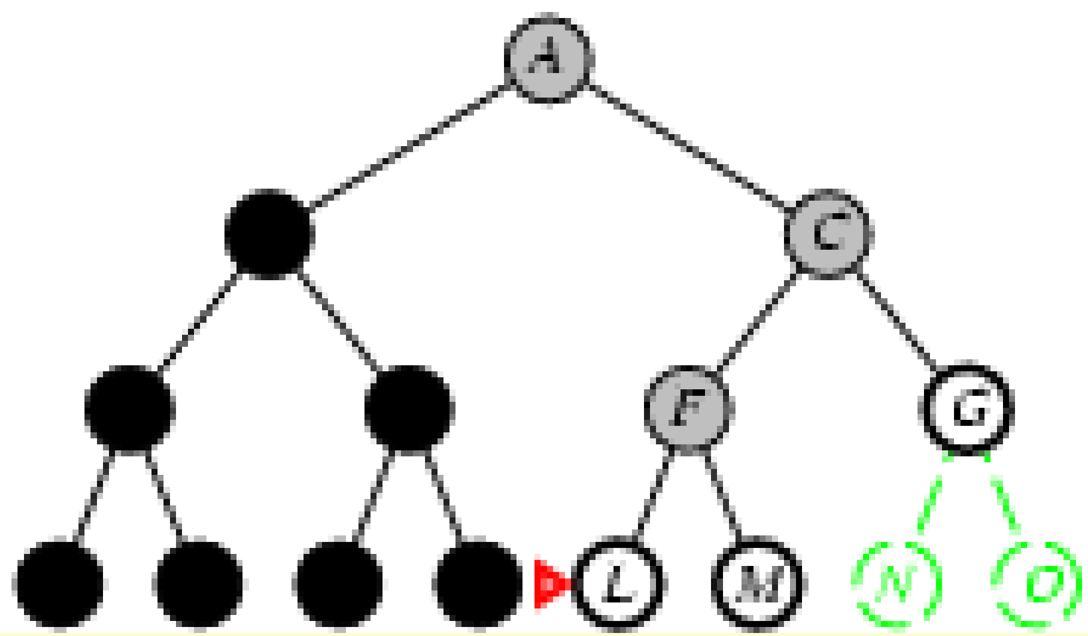


Depth-first search

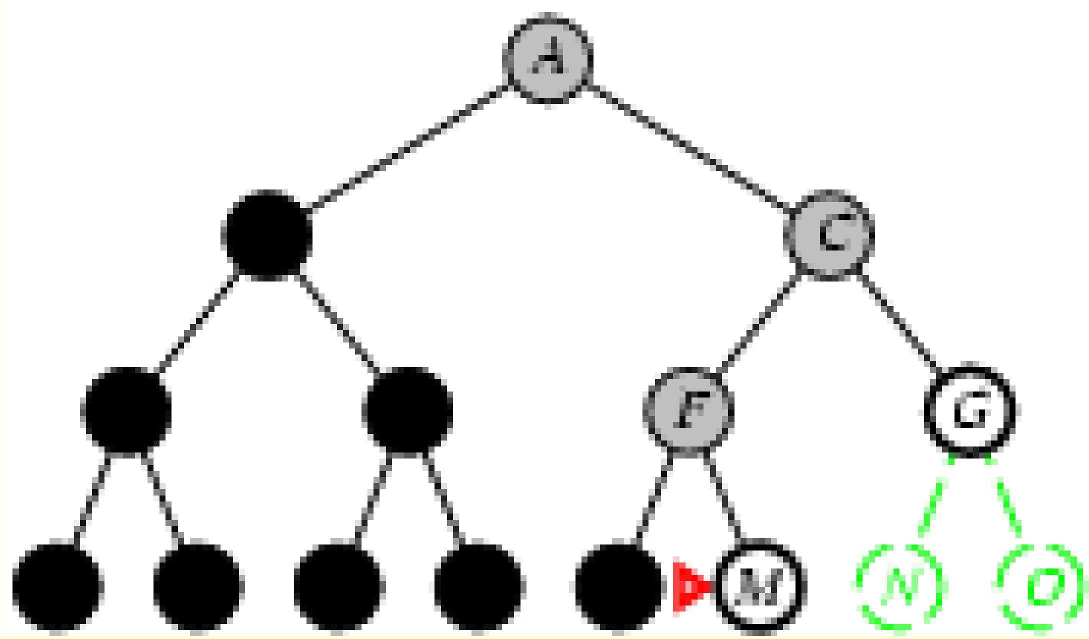
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  - 

Depth-first search

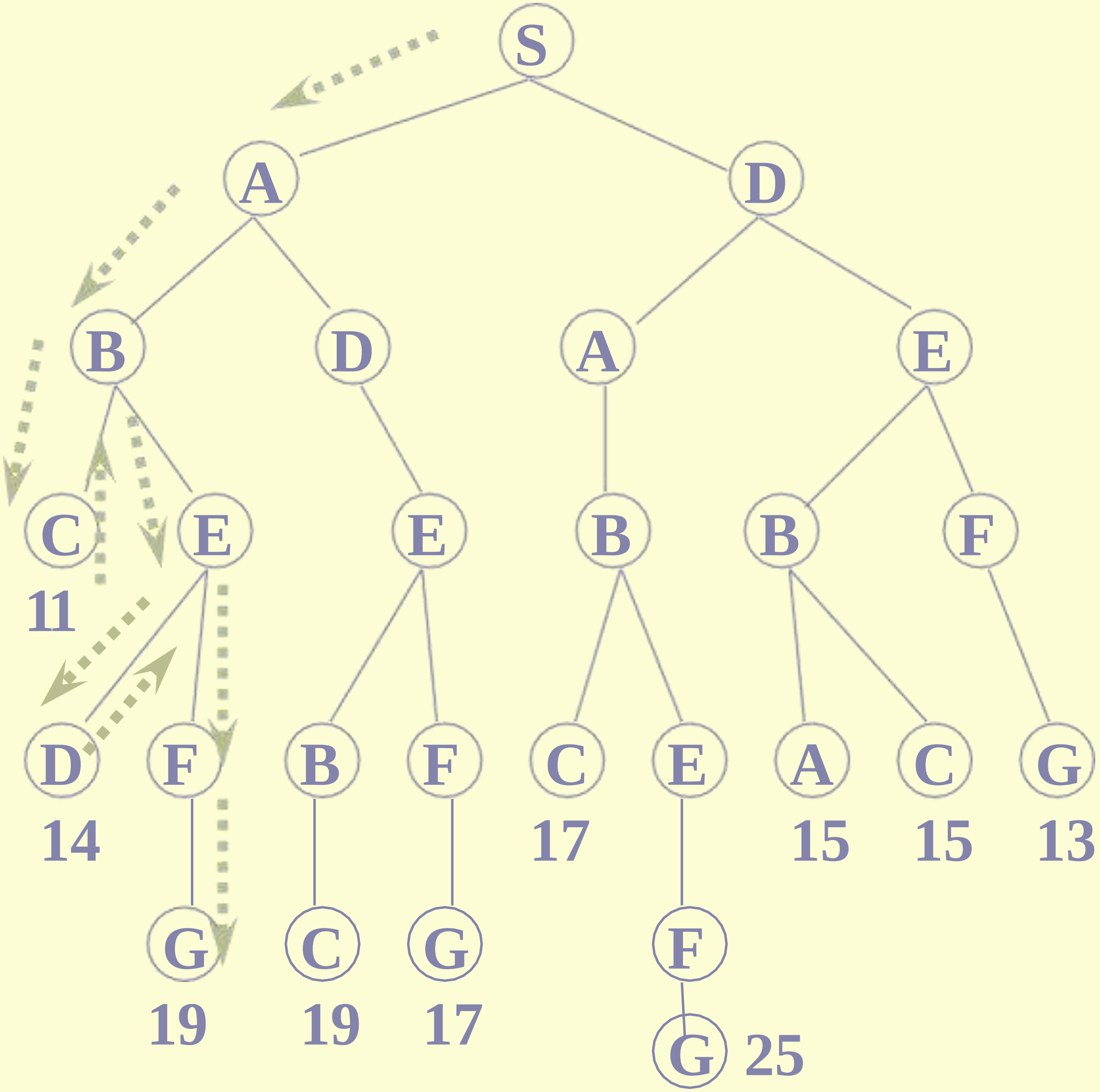
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  - 

Depth-first search



Depth-first search (Analysis)

- Not complete
  - because a path may be infinite or looping
  - then the path will never fail and go back try another option
- Not optimal
  - it doesn't guarantee the best solution
- It overcomes
  - the time and space complexities

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  - → 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

Depth-Limited Strategy

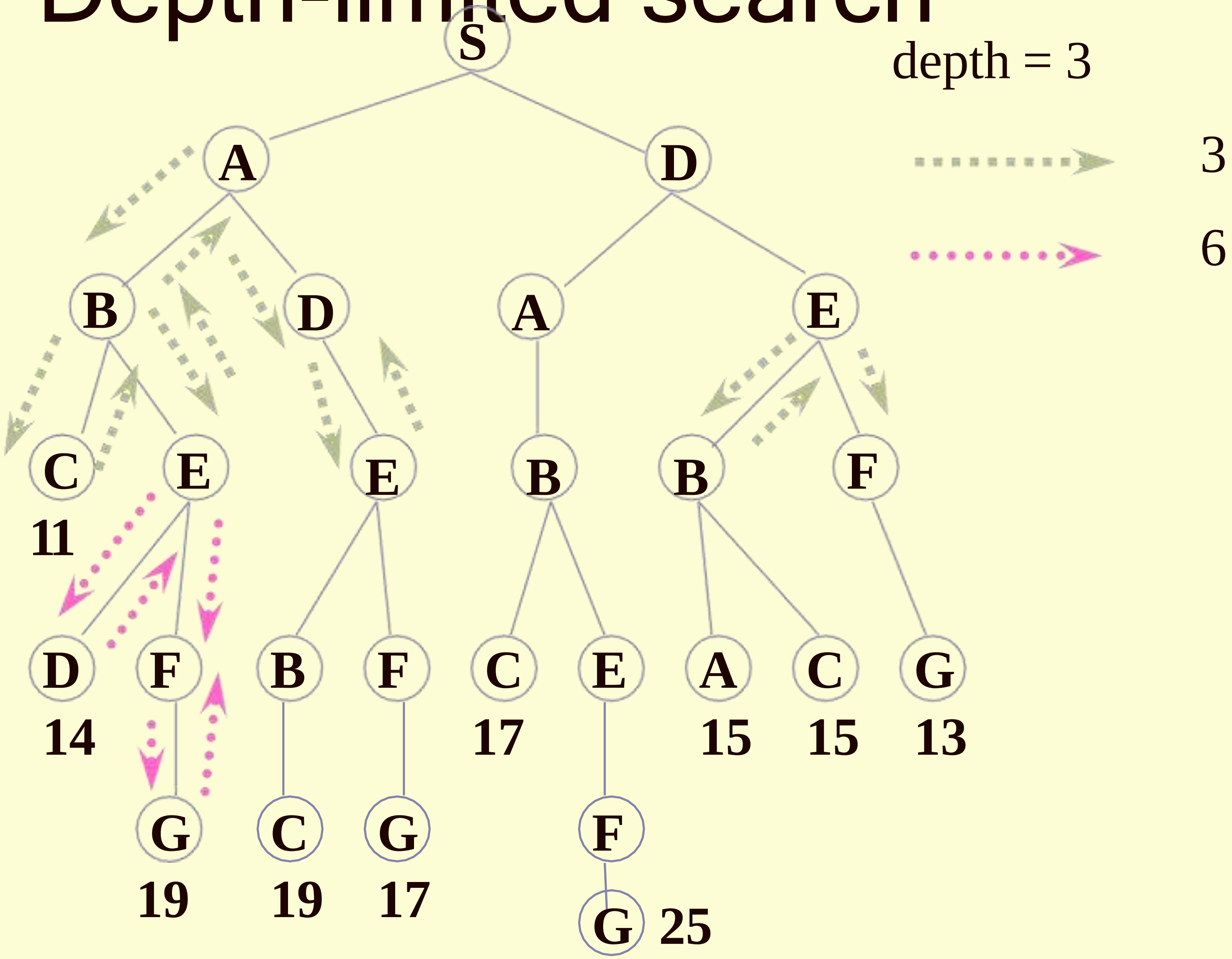
- Depth-first with depth cutoff k (maximal depth below which nodes are not expanded)

- Three possible outcomes:
    - Solution
    - Failure (no solution)
    - Cutoff (no solution within cutoff)

Depth-limited search

- It is depth-first search
  - with a **predefined** maximum depth
  - However, it is usually not easy to define the suitable maximum depth
  - too small → no solution can be found
  - too large → the same problems are suffered from
- Anyway the search is
  - complete
  - but still not optimal

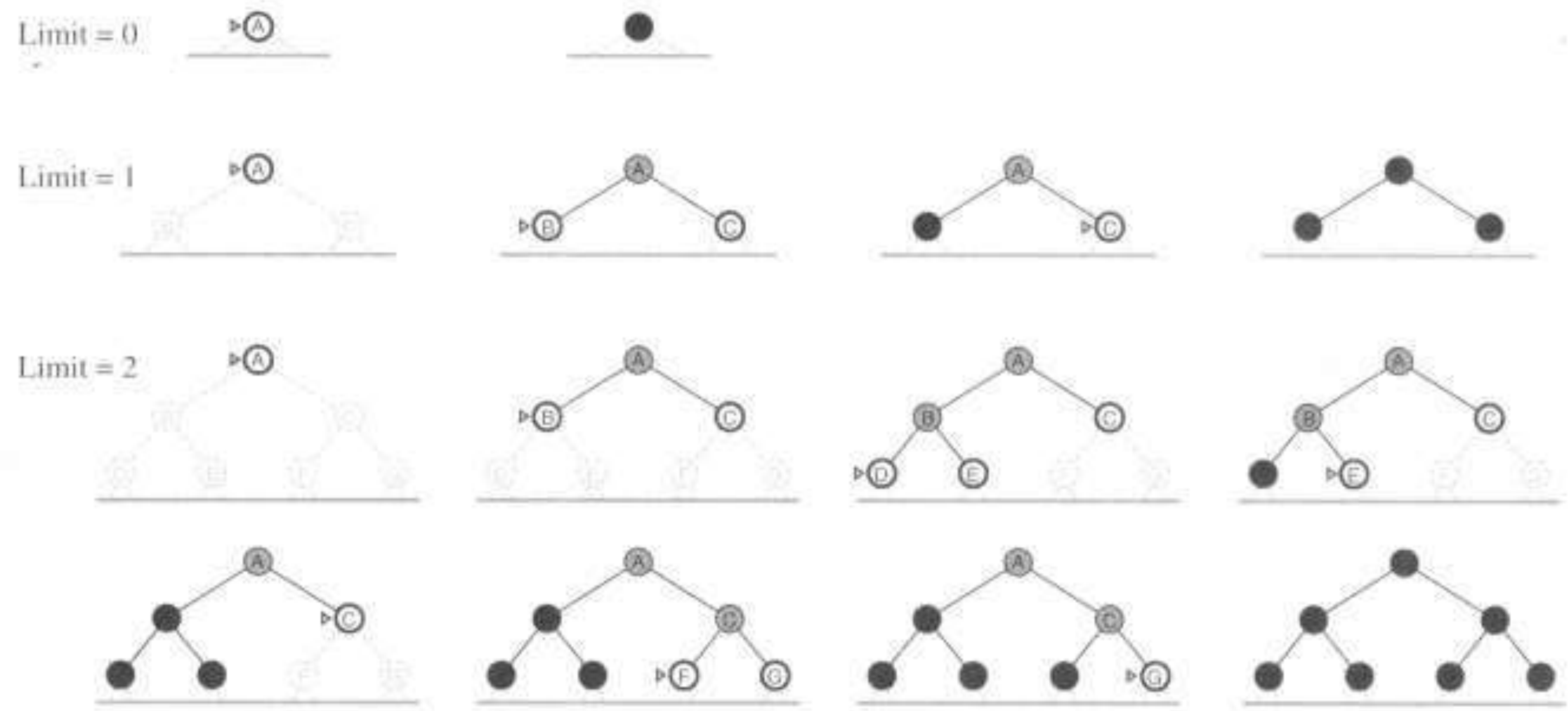
Depth-limited search



Iterative deepening search

- No choosing of the best depth limit
- It tries all possible depth limits:
  - first 0, then 1, 2, and so on
  - combines the benefits of depth-first and breadth-first search

Iterative deepening search



Iterative deepening search (Analysis)

- optimal
- complete
- Time and space complexities
  - reasonable
- suitable for the problem
  - having a large search space
  - and the depth of the solution is not known

Properties of iterative deepening search

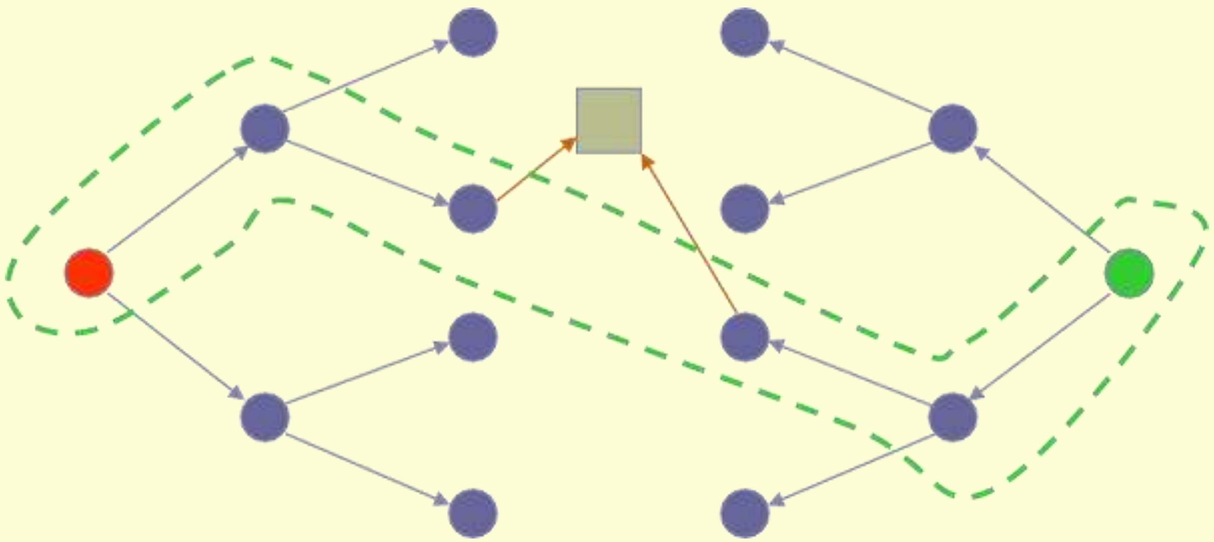
- Complete? Yes
-
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
-
- Space? $O(bd)$
-
- Optimal? Yes, if step cost = 1

Bidirectional search

- Run two simultaneous searches
  - one forward from the initial state another backward from the goal
  - stop when the two searches meet
- However, computing backward is difficult
  - A huge amount of goal states
  - at the goal state, which actions are used to compute it?
  - can the actions be reversible to computer its predecessors?

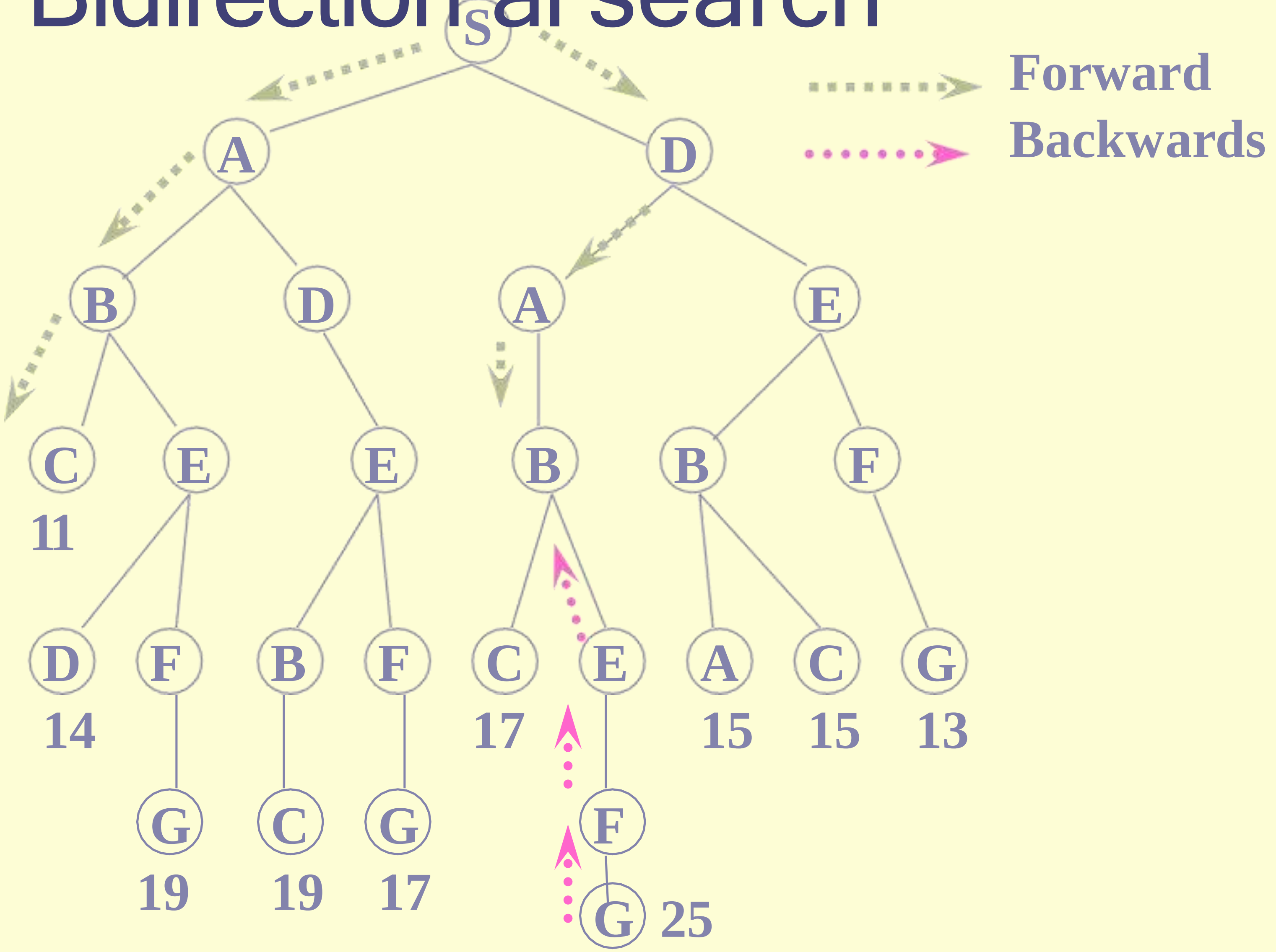
Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2



Time and space complexity = $O(b^{d/2}) << O(b^d)$

Bidirection al search



THANKYOU